

Chapter 6: Using the Debugger

Time 1 Hour

At the end of this chapter you should understand how to use J-Link and Olimex JTAG debugging hardware with a WICED Bluetooth kit. You will also understand how to use the BTSPy application to look at details of Bluetooth HCI traffic.

6.1	BTSPY.....	2
6.1.1	INTRODUCTION	2
6.1.2	PROJECT CONFIGURATION	3
6.2	DEBUGGING HARDWARE	5
6.3	PROJECT CONFIGURATION.....	6
6.3.1	EDITS TO PIN CONFIGURATION	6
6.3.2	EDITS TO MAKEFILE.MK.....	8
6.3.3	EDITS TO THE PROJECT SOURCE CODE.....	8
6.3.4	EDITS TO MAKE TARGET	9
6.3.5	PROGRAMMING THE PROJECT	9
6.4	HOST DEBUG ENVIRONMENT SETUP.....	10
6.4.1	SEGGER J-LINK	10
6.4.2	OLIMEX ARM-USB-TINY-H.....	16
6.5	USING THE DEBUGGER.....	26
6.6	EXERCISES.....	30
	EXERCISE - 6.1 RUN BTSPY	30
	EXERCISE - 6.2 (ADVANCED) RUN THE DEBUGGER	31

6.1 BTSpy

6.1.1 Introduction

As you learned earlier, application debug trace messages can be printed to the PUART or HCI UART by using the `WICED_BT_TRACE()` macro. In addition to messages from the application, a tool called *BTSpy* can be used to print Bluetooth protocol trace messages to be viewed alongside application traces.

By routing application traces with the `wiced_set_debug_uart(WICED_ROUTE_DEBUG_TO_WICED_UART)` API call, you can view traces using the separate *BTSpy* and *ClientControl* utilities. These Windows applications are supplied with WICED Studio and are available in the `wiced_tools` and `apps\host\client_control` folders respectively.

The *ClientControl* utility connects to the HCI UART port and receives trace message packets. These message packets are then sent through a socket connection to the *BTSpy* utility, where they can be displayed and logged.

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 stack 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.

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 controlled by the `ENABLE_HCI_TRACE` compile flag.

6.1.2 Project Configuration

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

1. Define `ENABLE_HCI_TRACE` and `WICED_BT_TRACE_ENABLE` in the application *makefile.mk* file.

```
C_FLAGS += -DWICED_BT_TRACE_ENABLE
C_FLAGS += -DHCI_TRACE_OVER_TRANSPORT
```

Note: If you used WICED BT Designer to create the project, this is done by default.

2. Make the following changes to the C source file. Note: You can use `#ifdef` and `#endif` around the C source file edits if you want to conditionally include them in the project based on `ENABLE_HCI_TRACE` being defined in the makefile.

Note: If you used WICED BT Designer to create the project, the steps below are done by default but make sure to check the debug pin routing since you may have changed it to use the PUART.

- a. Include the transport header file if it isn't already done:

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

- b. Setup a global HCI UART transport structure as follows:

```
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. */
};
```

- c. Define a trace callback function as follows where <appname> is the name of your application (don't forget to add a prototype if the function definition is after `application_start`):

```
#ifdef HCI_TRACE_OVER_TRANSPORT
/* Handle Sending of Trace over the Transport */
void <appname>_trace_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 );
}
#endif
```

- d. Register the callback function:

```
#ifndef HCI_TRACE_OVER_TRANSPORT
// There is a virtual HCI interface between upper layers of the stack and
// the controller portion of the chip with lower layers of the BT stack.
// Register with the stack to receive all HCI commands, events and data.
wiced_bt_dev_register_hci_trace(<appname>_trace_callback);
#endif
```

- e. In the application initialization, setup the debug print pins, and initialize the HCI UART:

```
#if ((defined WICED_BT_TRACE_ENABLE) || (defined HCI_TRACE_OVER_TRANSPORT))
/* Set the Debug UART as WICED_ROUTE_DEBUG_NONE to get rid of prints */
// wiced_set_debug_uart( WICED_ROUTE_DEBUG_NONE );

/* Set Debug UART as WICED_ROUTE_DEBUG_TO_PUART to see debug traces on Peripheral UART
(PUART) */
//wiced_set_debug_uart( WICED_ROUTE_DEBUG_TO_PUART );

/* Set the Debug UART as WICED_ROUTE_DEBUG_TO_WICED_UART to send debug strings over
the WICED debug interface */
wiced_set_debug_uart( WICED_ROUTE_DEBUG_TO_WICED_UART );
#endif
```

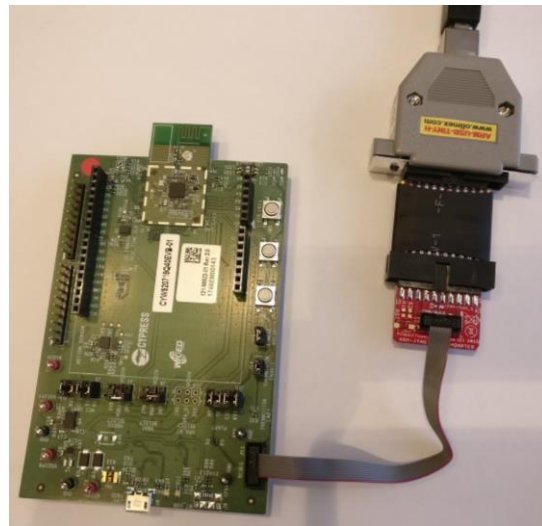
3. Build and download the application to the evaluation board.
4. Press **Reset** (SW2) on the WICED evaluation board to reset the HCI UART after downloading and before opening the port with *ClientControl*. This is necessary because the programmer and HCI UART use the same interface.
5. Run the *ClientControl* utility from *apps\host\client_control\Windows*. To open the utility from inside WICED Studio, right-click the utility and select **Open With > System Editor**.
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 from *wiced_tools\BTSpy\Win32*. To open the utility from inside WICED Studio, right-click the utility and select **Open With > System Editor**.
8. In the *ClientControl* utility, open the HCI UART COM port by clicking on "Open Port".
9. 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.
 - b. Note: The HCI UART is the same port used for programming so you must disconnect each time you want to re-program
 - c. 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.

6.2 Debugging Hardware

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:



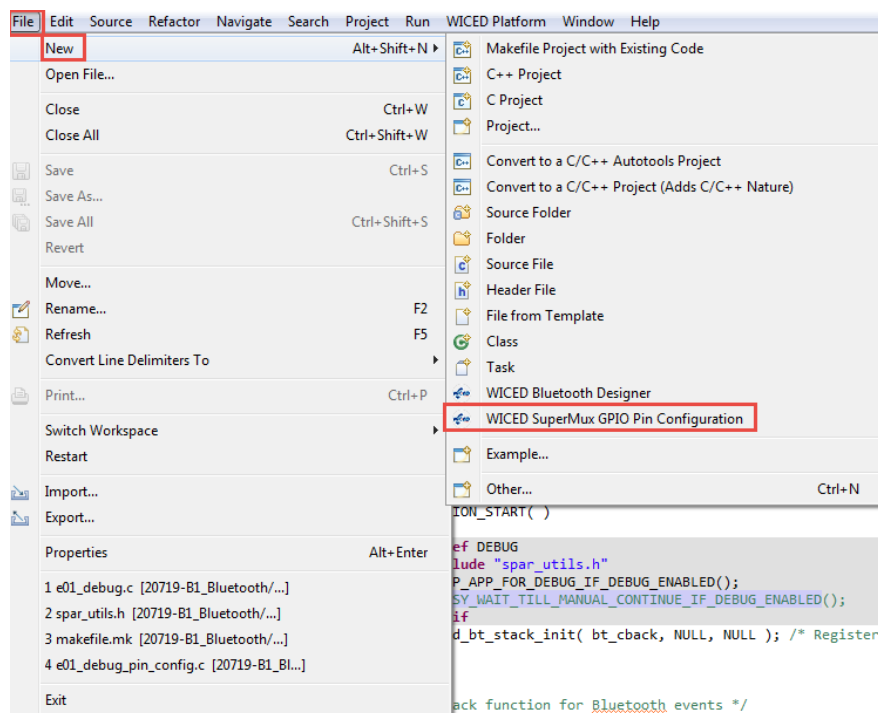
6.3 Project Configuration

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:

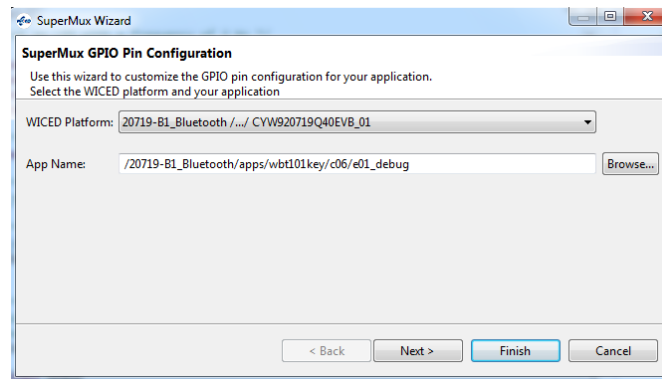
6.3.1 Edits to Pin Configuration

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*.



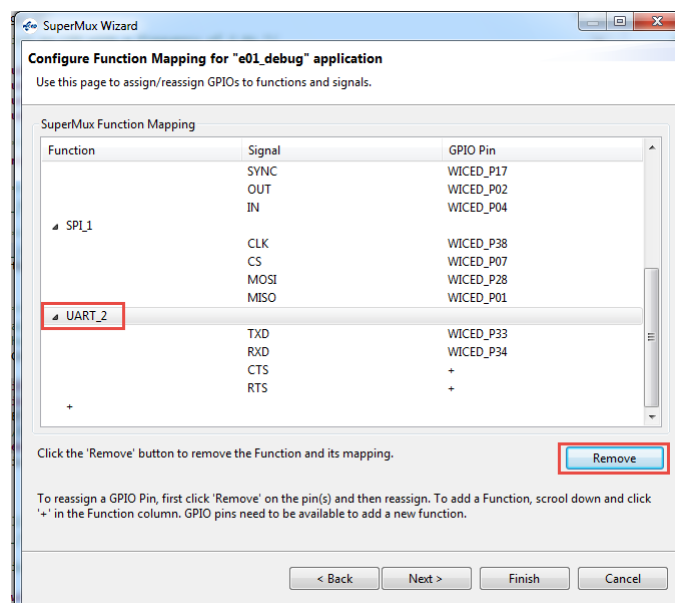
The first pin configuration window will look like this:



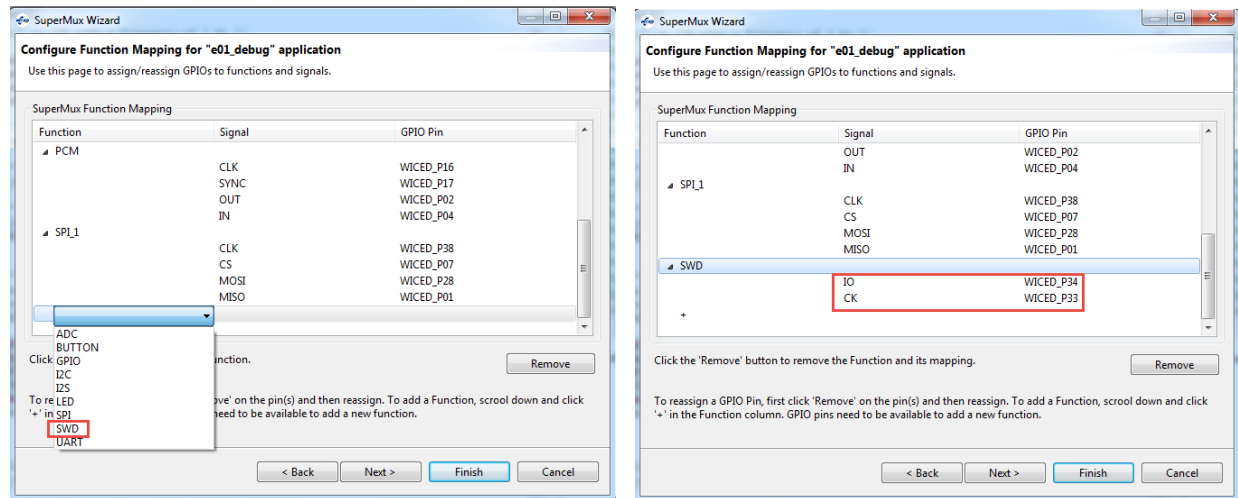
Select the WICED Platform you are using (e.g. WBT101_2_CYW920719Q40EVB_01) and that application name is correct and then click "Next". If you don't select the correct platform name you will have to cancel and start over. Leave the selected pins as-is and click on "Next" again.

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 can be used to re-run the wizard again if necessary. The makefile.mk is also automatically updated to include the new pin configuration file in the project.

6.3.2 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 += ex01_debug_pin_config.c
endif
```

6.3.3 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
```

This 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.

6.3.4 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.ch06.ex01_debug for the CYW920719Q40EVB-01 and PSoC Analog Front End Shield combination , the Make Target would be:

wbt101.ch06.ex01_debug-WBT101_2_CYW920719Q40EVB-01 **DEBUG=1** download

6.3.5 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.

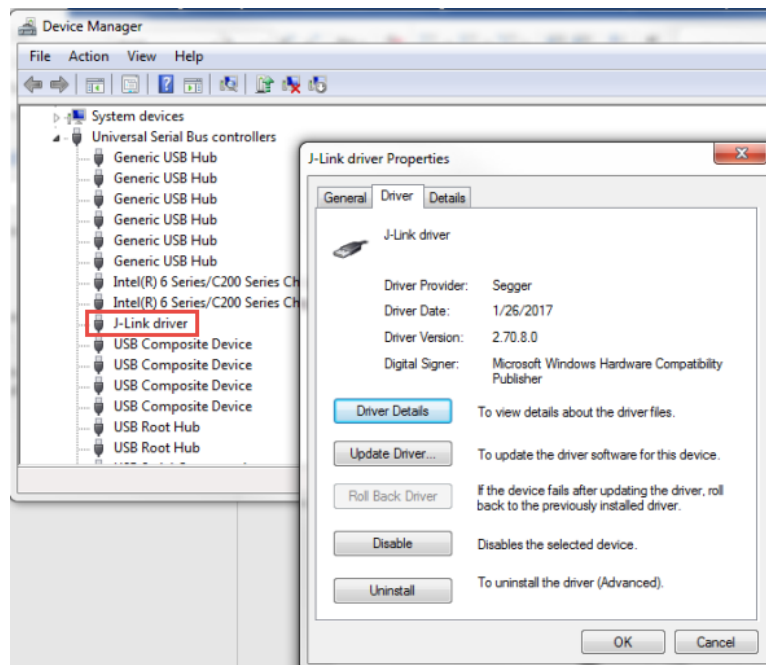
6.4 Host Debug Environment Setup

6.4.1 Segger J-Link

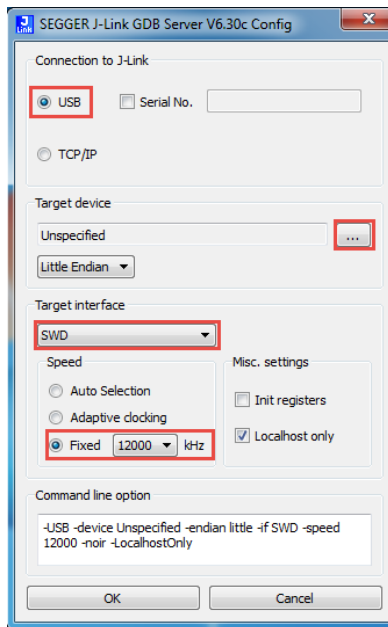
Segger J-Link Software and Hardware Setup

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".

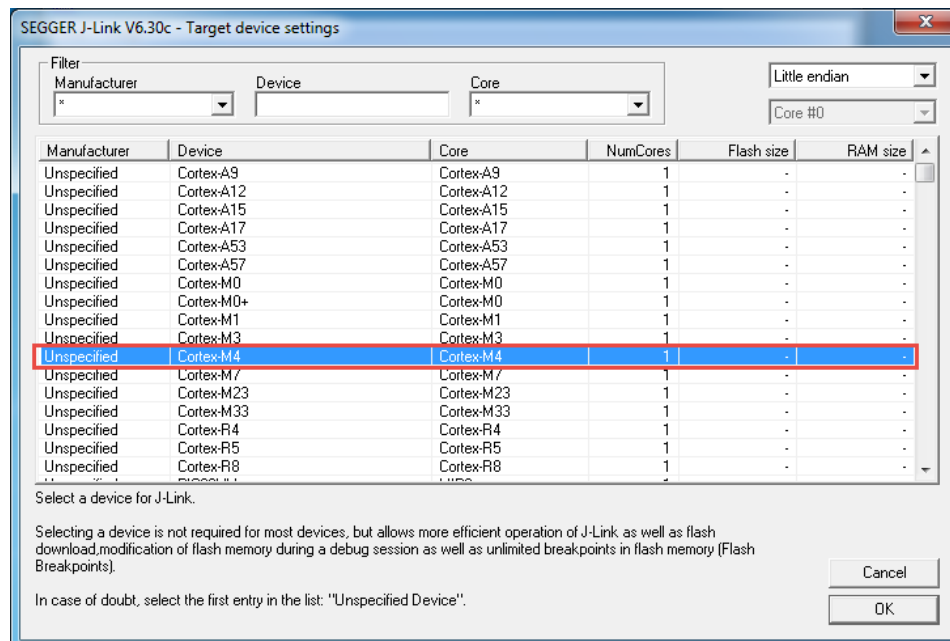
Once the software is installed, connect the J-Link to the kit's debug connector and then plug the J-Link into a USB port on the computer. When USB enumeration completes, the J-Link device should appear in the Device Manager as shown here:



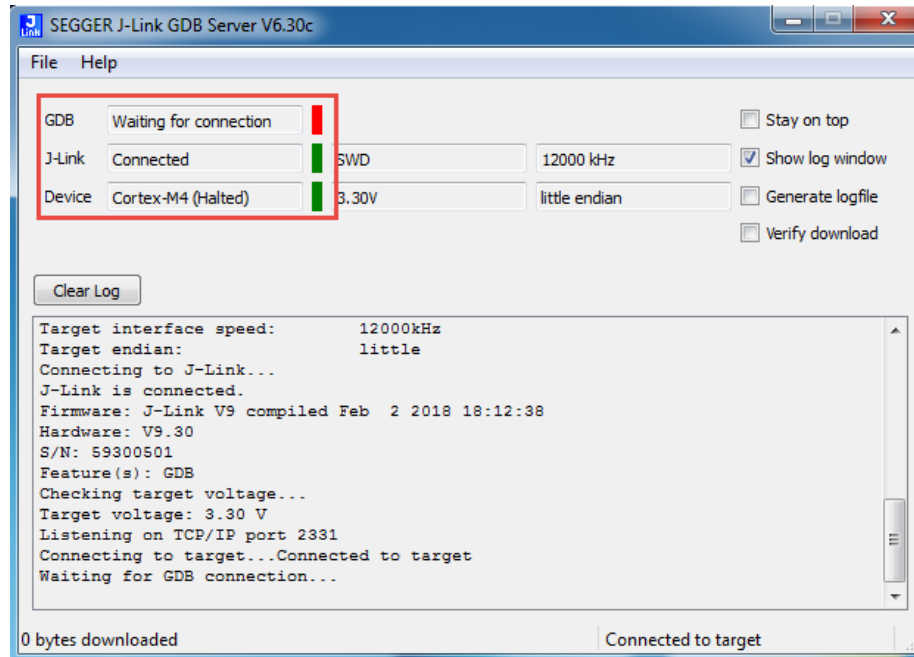
Run the Segger J-Link GDB Server program and select the USB and SWD interfaces. Set the speed to 12000 kHz.



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:

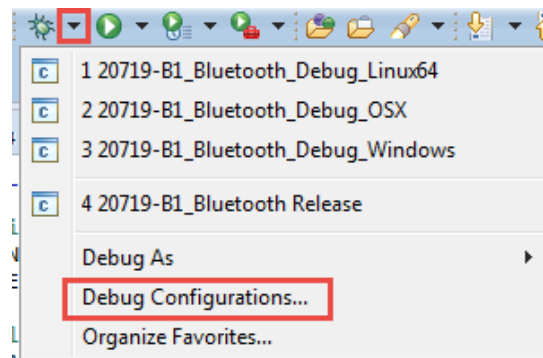


If the window does not appear as shown above or if the window closes after a few seconds, double check that the WICED kit has the correct firmware loaded and that everything is connected properly.

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.

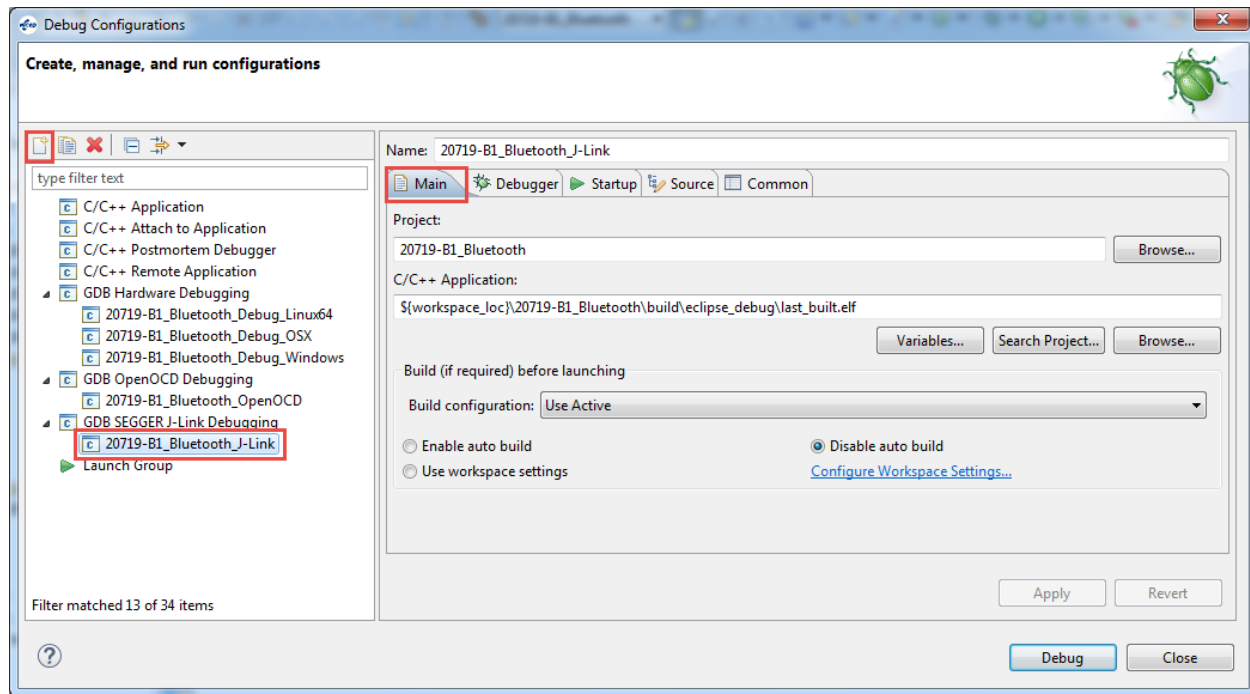
WICED Studio setup for J-Link

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:

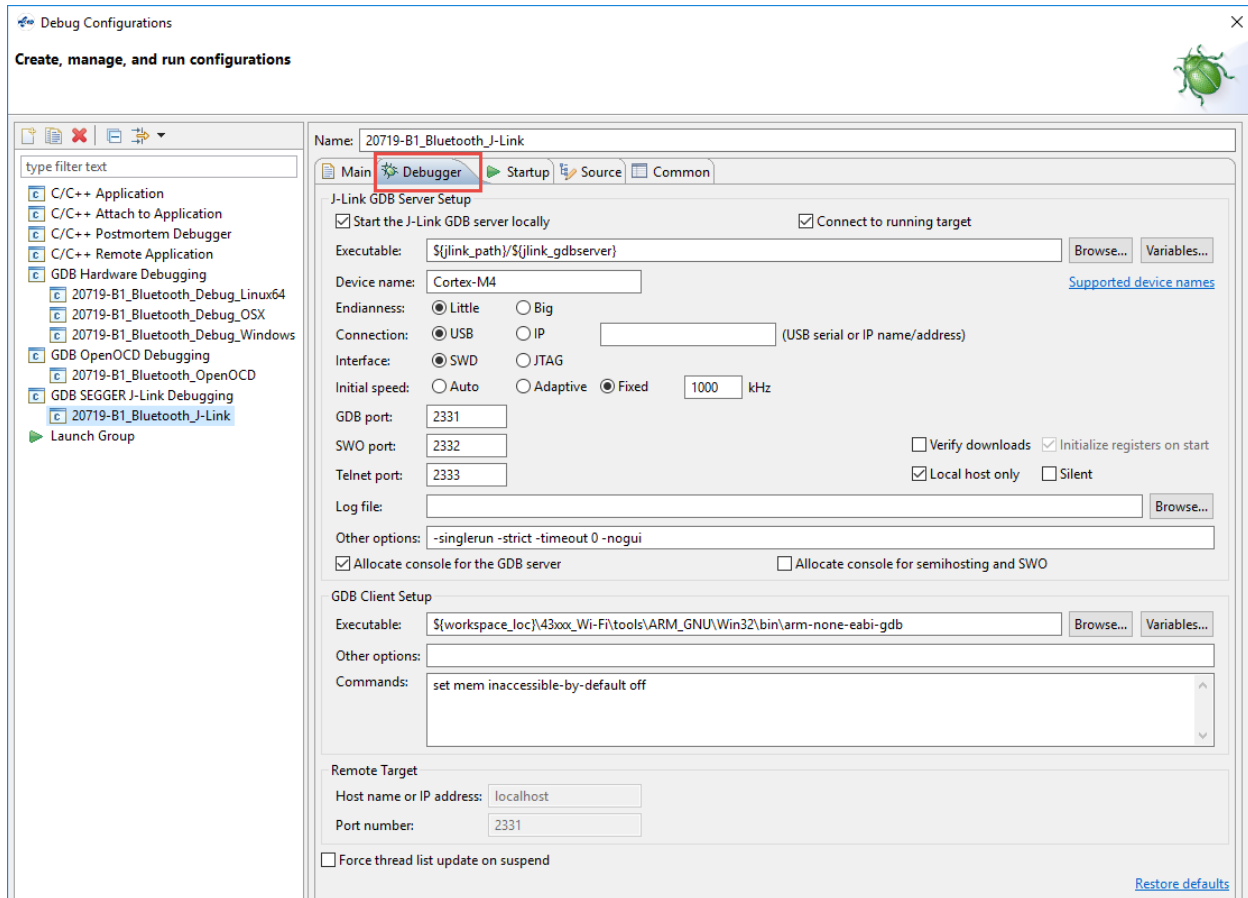


The *Project* and *C/C++ Application* string shown above are:

20719-B1_Bluetooth

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

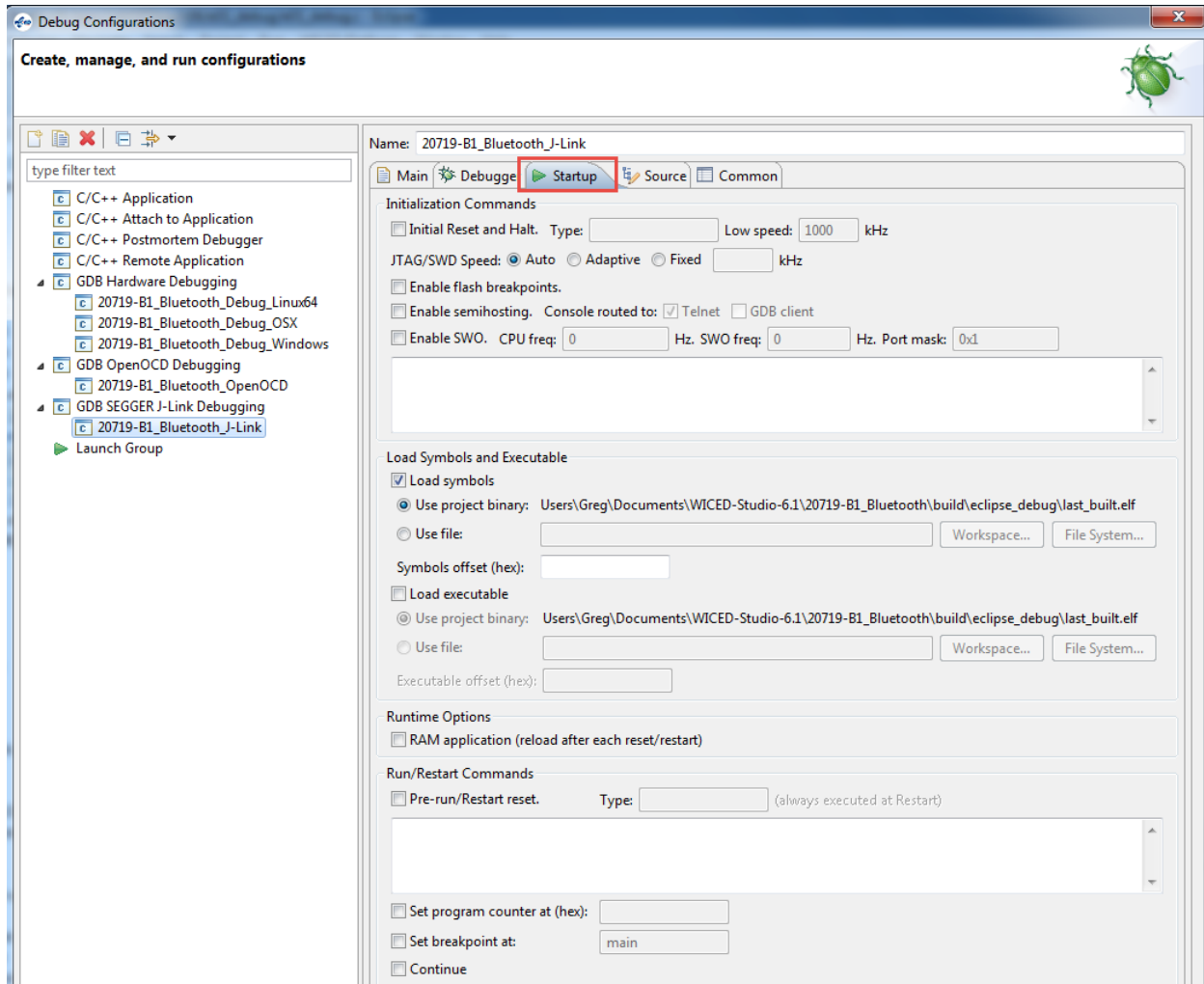
Hint: Click Apply on each tab as you complete the tab since it will affect what you see on the other tabs.



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 6.5 to learn how to use the debugger functions.

6.4.2 Olimex ARM-USB-TINY-H

Olimex ARM-USB-TINY-H Software and Hardware Setup

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

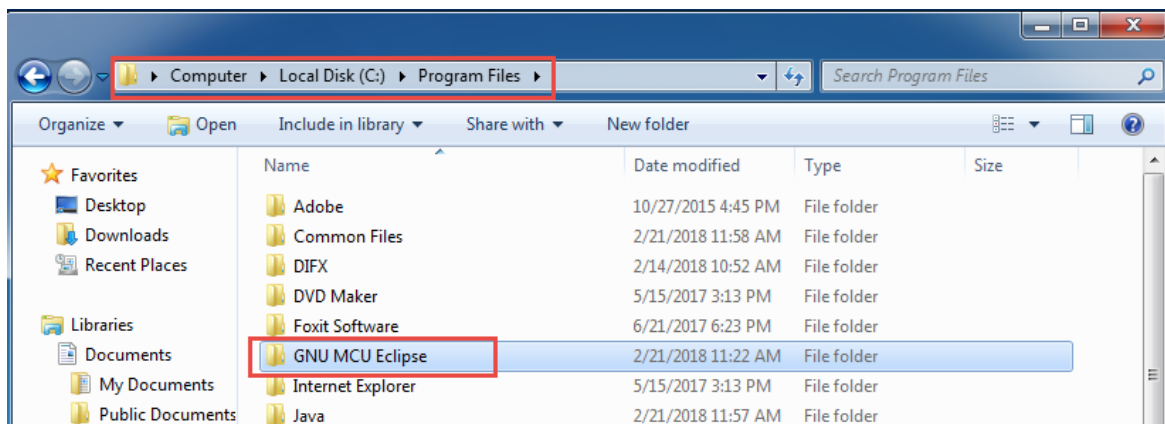
Download the version of Open OCD for your system from:

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

Once the file is downloaded, unzip it and move the *GNU MCU Eclipse* folder to:

C:\Program Files

An example of the resulting path can be seen here:

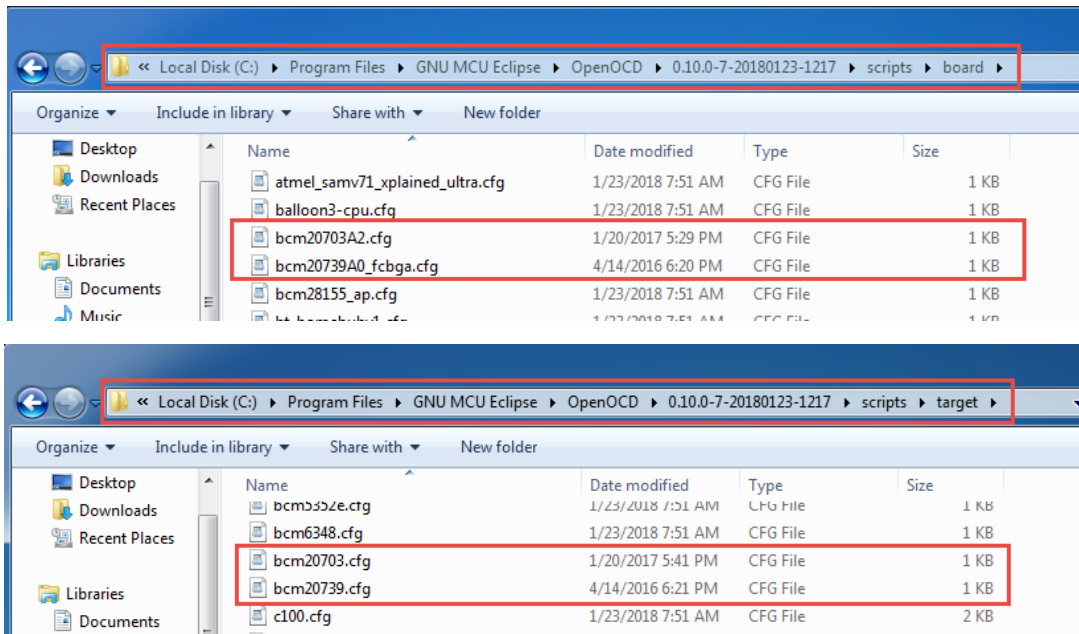


More detailed information and instructions on OpenOCD can be found at:

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

Platform Specific Configuration Files

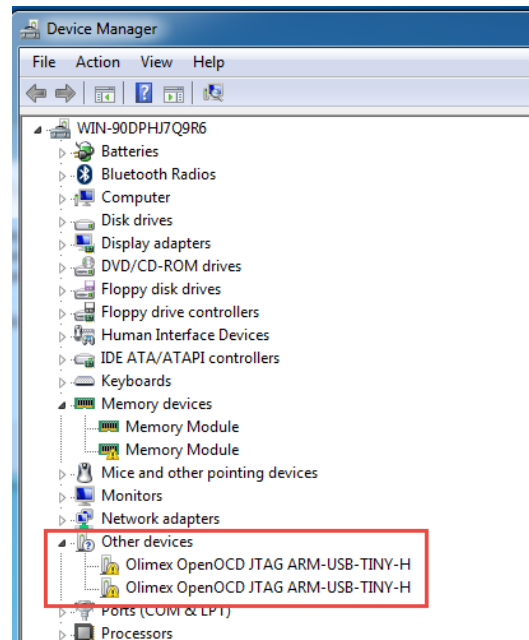
Inside the folder *GNU MCU Eclipse\OpenOCD<version>\scripts* there are folders called *board* and *target*. You must copy the board and target files for the kit you are using into the appropriate folders. The files are as shown here:



You can find these files in the class folder under *Debug_Config*.

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:



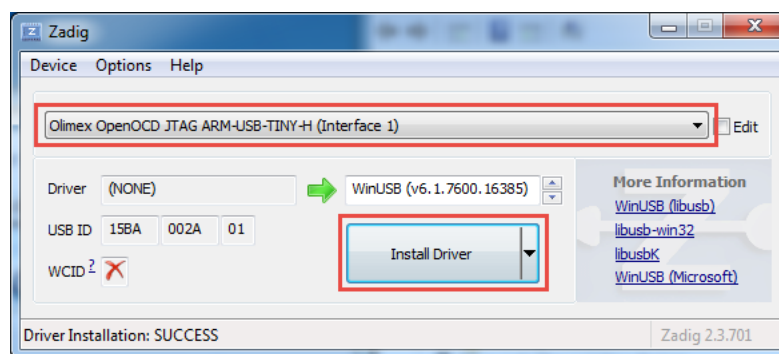
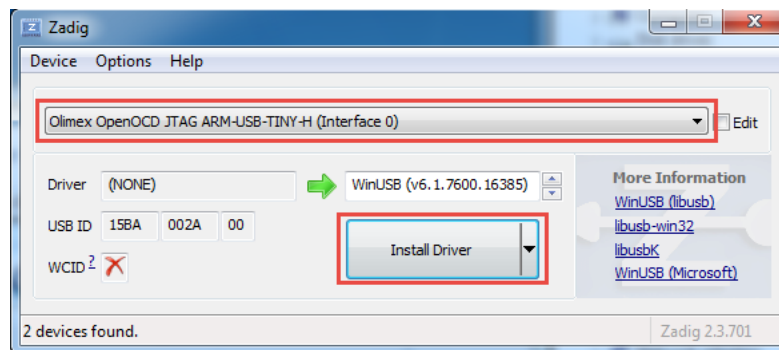
If the driver is not installed properly, download the driver update tool from:

<http://zadig.akeo.ie/>

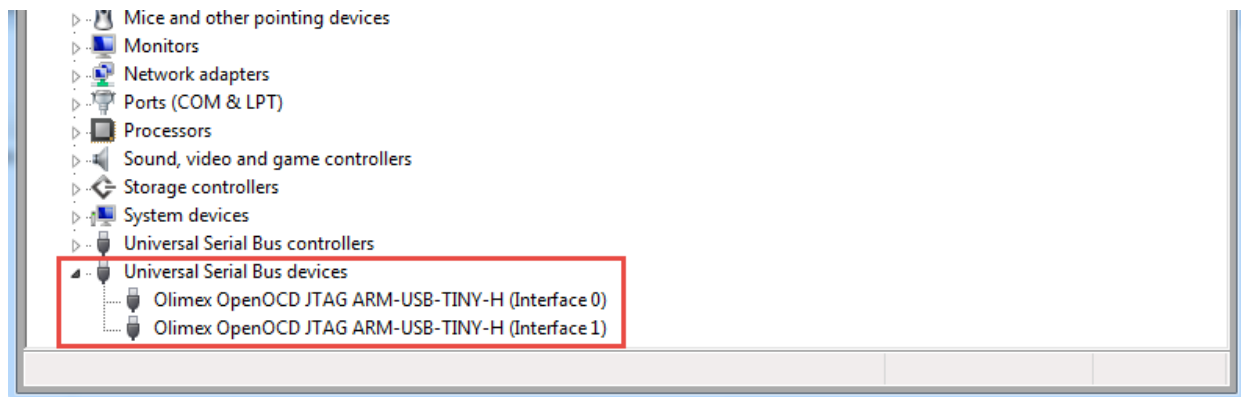
From the website, download Zadig from the link as shown here:



Once you have download Zadig, run the executable. 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:



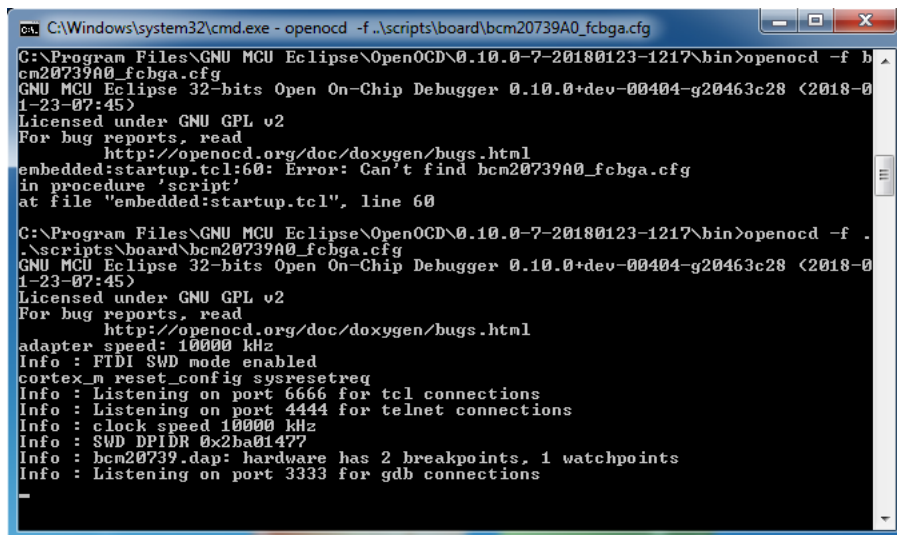
OpenOCD Verification

To verify that the files are installed correctly and the drivers are working, open a command window or power shell window in `C:\Program Files\GNU MCU Eclipse\OpenOCD\<version>\bin` and enter the following command:

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

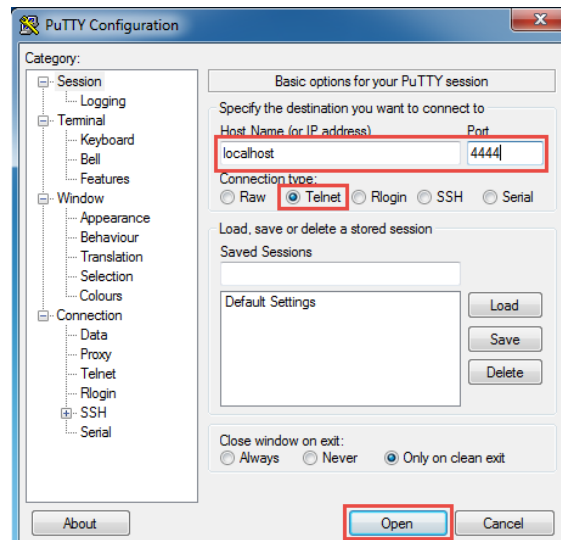
Note: To open a command window or power shell window, you can go to the folder just above in Windows explorer, hold Shift, right-click on an open area in the window, and then select "Open Command Window Here" or "Open Powershell Window Here".

You should see a result like this:

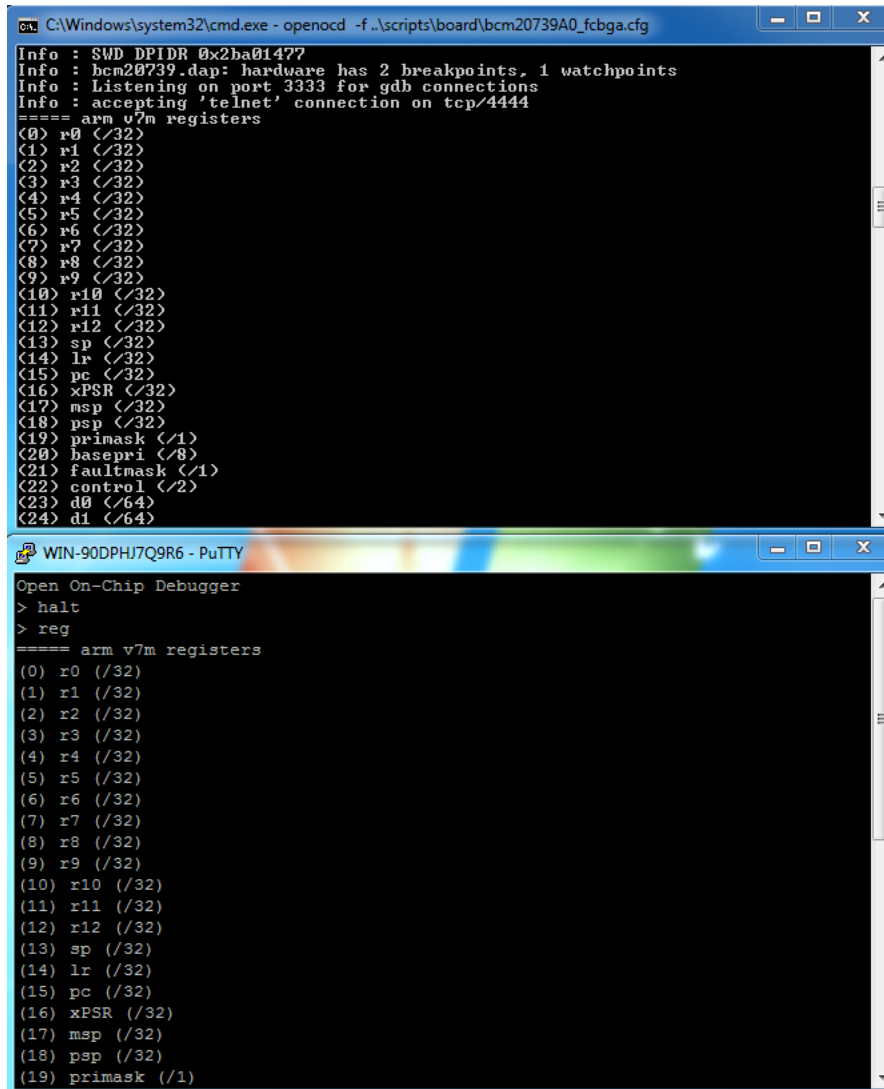


```
C:\Windows\system32\cmd.exe - openocd -f ..\scripts\board\bcm20739A0_fcbga.cfg
C:\Program Files\GNU MCU Eclipse\OpenOCD\0.10.0-7-20180123-1217\bin>openocd -f b
cm20739A0_fcbga.cfg
GNU MCU Eclipse 32-bits Open On-Chip Debugger 0.10.0+dev-00404-g20463c28 (2018-0
1-23-07:45)
Licensed under GNU GPL v2
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 tcl 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
```

Next, 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 Windows command prompt running OpenOCD. The bottom window is a PuTTY terminal running an Open-On-Chip Debugger.

```

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)

WIN-90DPHJ7Q9R6 - PuTTY
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)
  
```

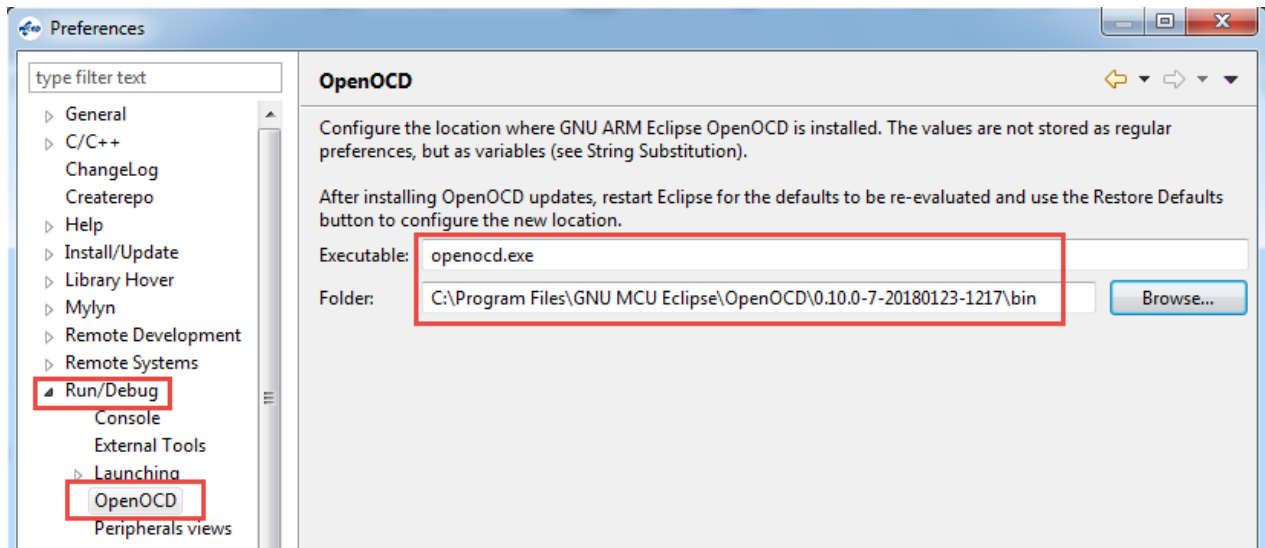
Enter the command "exit" to quit out of Telnet and hit Ctrl-C to stop the OpenOCD GDB Server.

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.

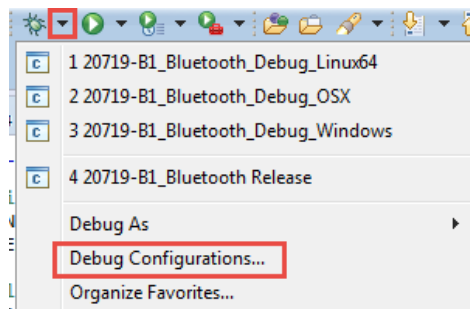


The *Folder* shown above is:

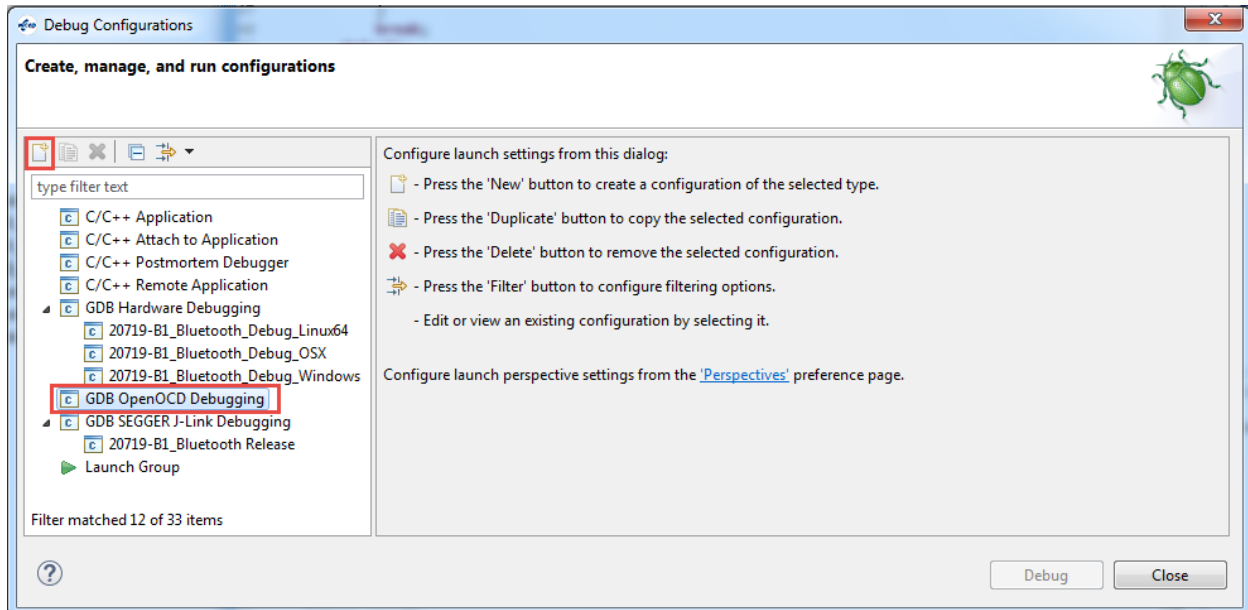
`C:\Program Files\GNU MCU Eclipse\OpenOCD\0.10.0-7-20180123-1217\bin` but it may be different in your case.

Debug Configuration

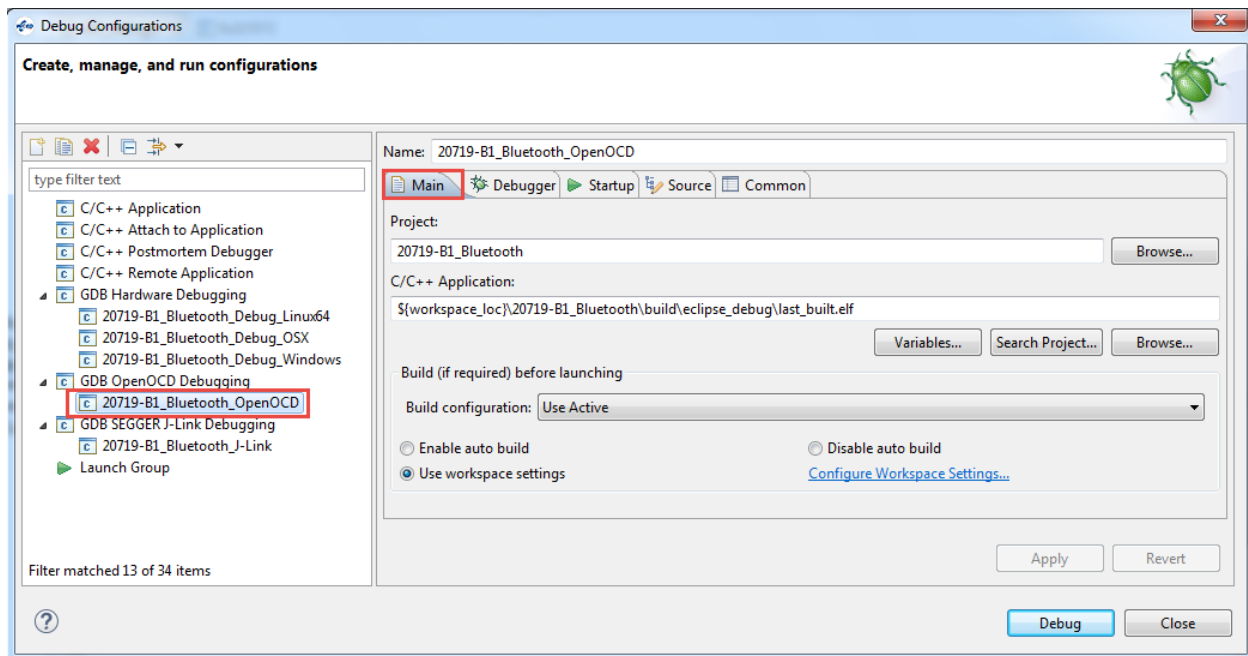
From WICED Studio, click the arrow next to the Bug icon and select "Debug Configurations...".



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



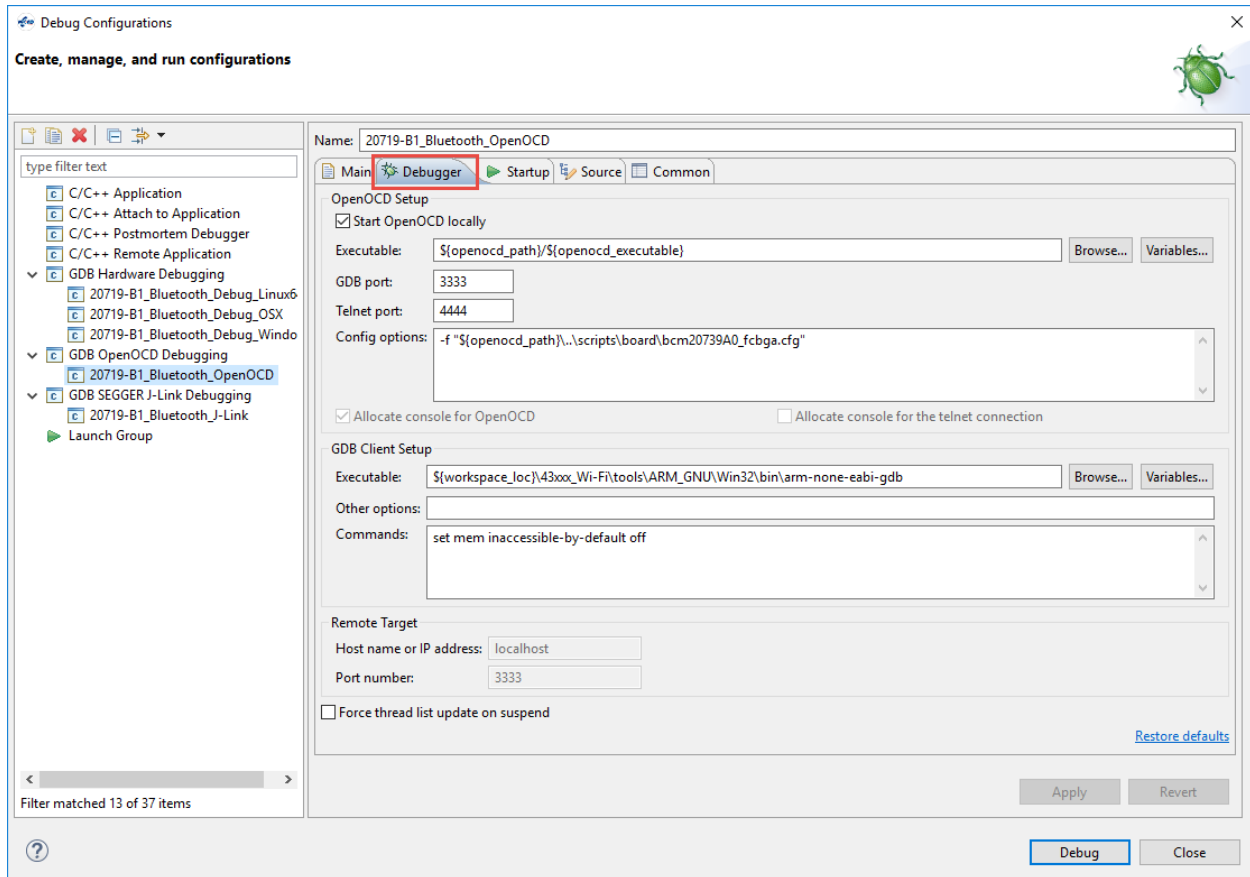
Change the name of the configuration if desired (the pictures below use the name "20719-B1_Bluetooth_OpenOCD") and set the options on each tab as shown in the following pictures.



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

20719-B1_Bluetooth

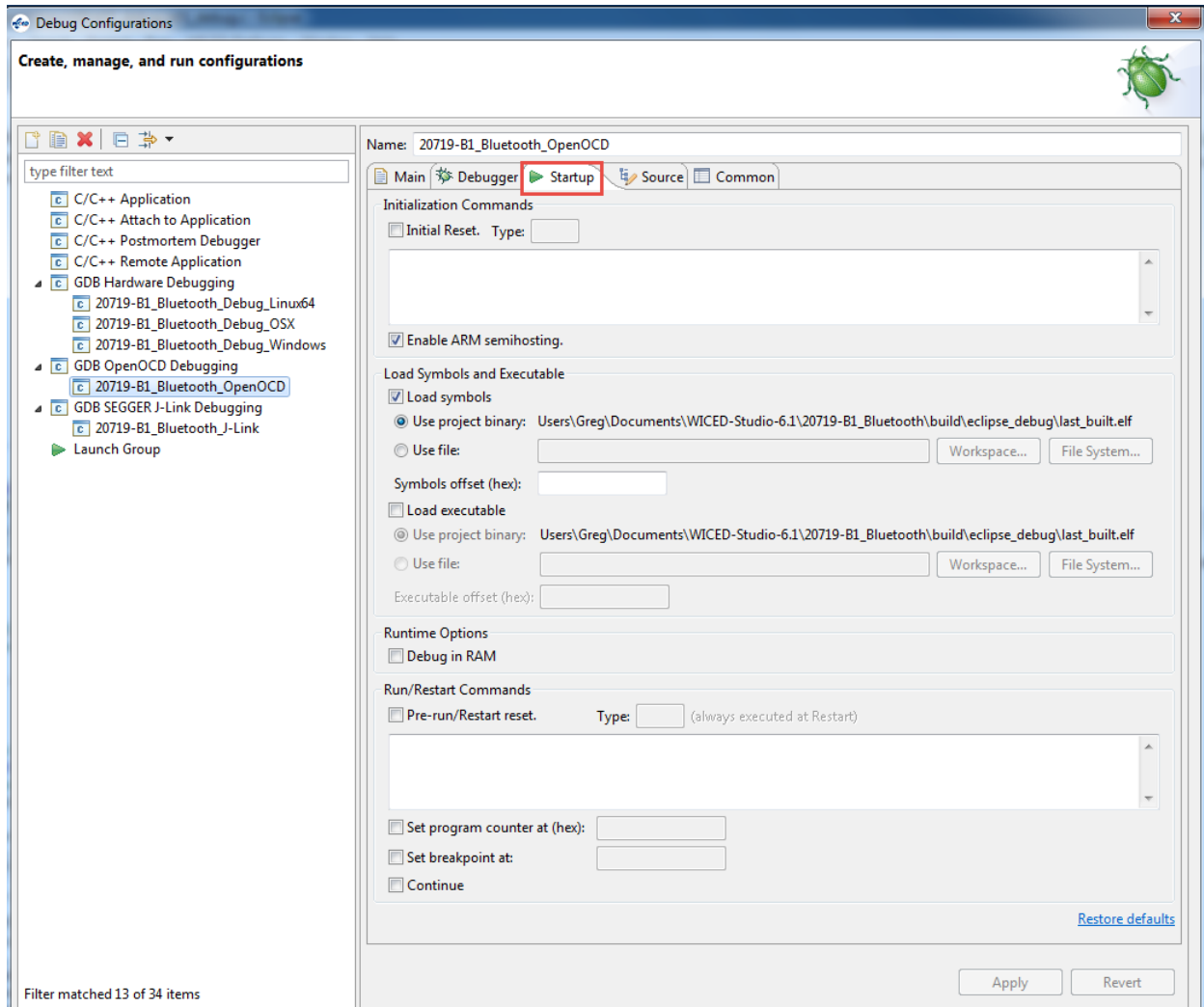
\$(workspace_loc)\20719-B1_Bluetooth\build\eclipse_debug\last_built.elf



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

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

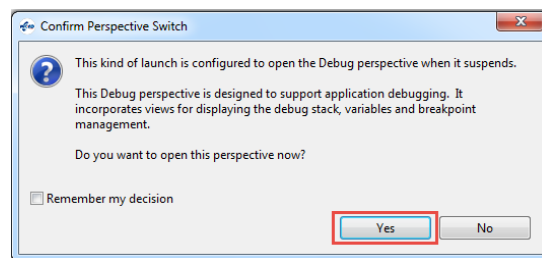
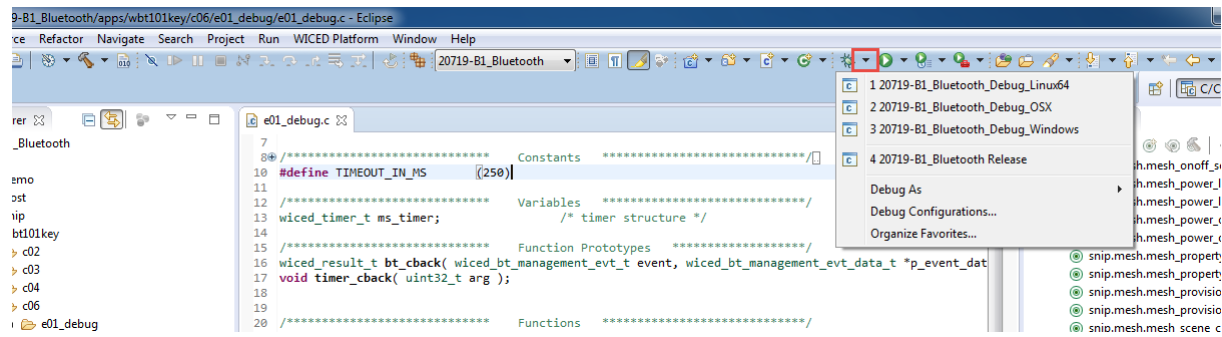
`${workspace_loc}/43xxx_Wi-Fi/tools/ARM_GNU/Win32/bin/arm-none-eabi-gdb`



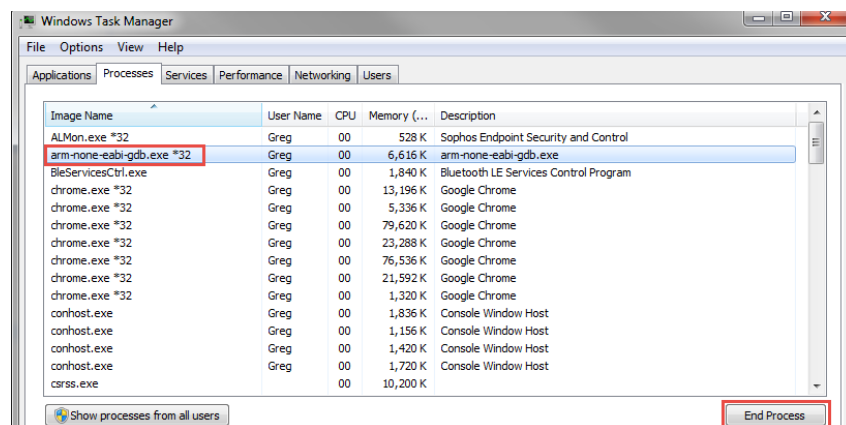
Click "Apply" as you make changes. Once you are done, reset the kit to make sure it is in the busy wait loop and click on "Debug". When the debugger has started running, go onto section 0 to learn how to use the debugger functions.

6.5 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.



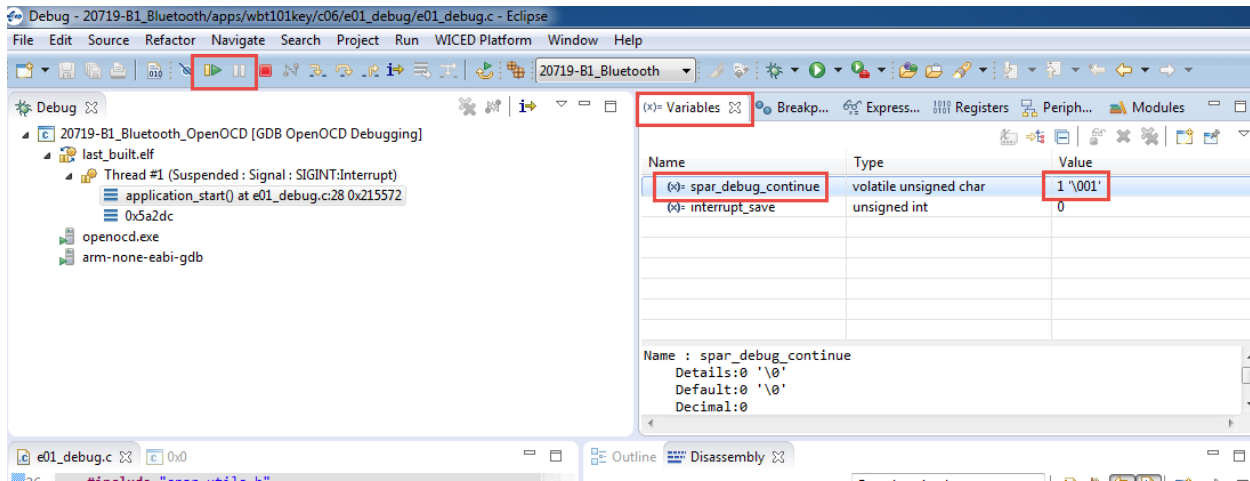
If you get an error when trying to launch the debugger you may need to terminate an existing debug process. Open the Windows Task Manager, select the Process tab, click on "Image Name" to sort by the process name and terminate all "arm-none-eabi-gdb" processes.



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, change the value of *spar_debug_contine* from 0 to 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.

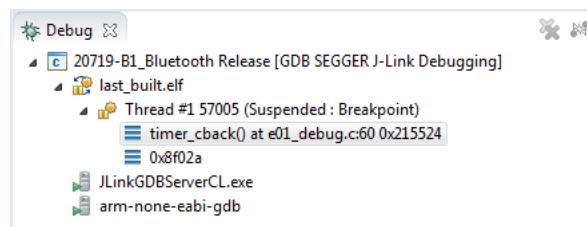
Click the "Resume" button (shown in the figure above) to resume execution. The program will halt once it reaches the breakpoint.

```

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

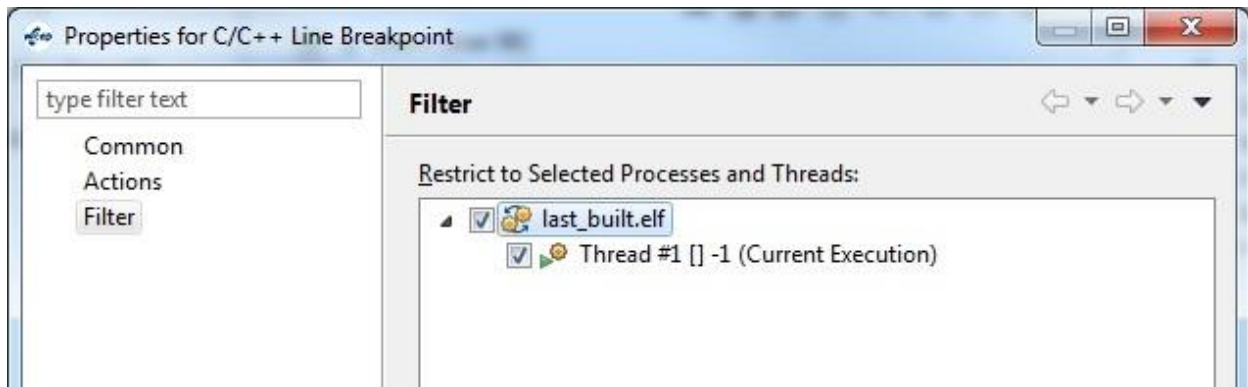
```

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.

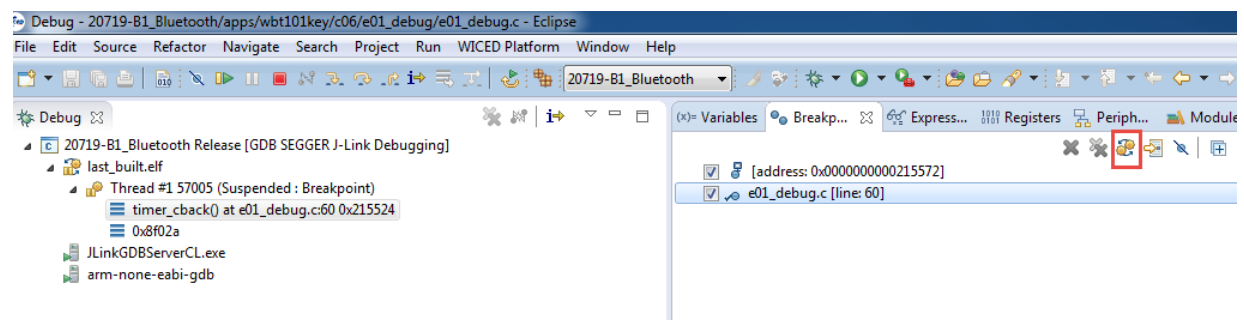


You can enable or disable breakpoints by double clicking on the green circle next to the line in the source code or from the "Breakpoints" window. If you don't see the Breakpoints tab, use the menu item "Window > Show View > Breakpoints".

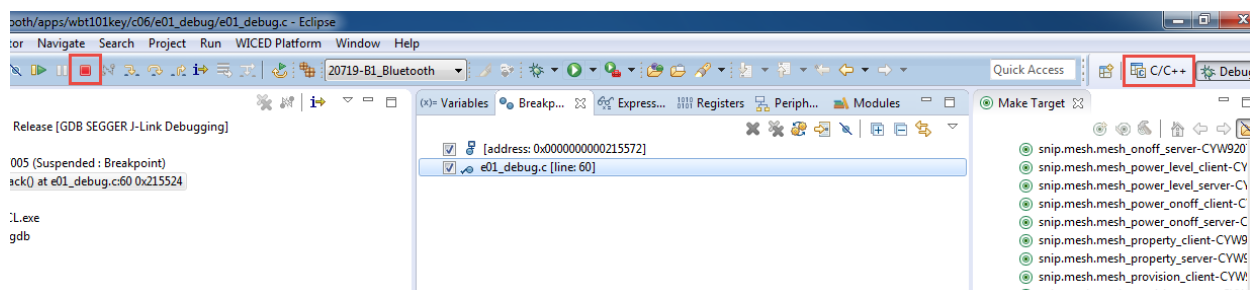
If breakpoints are created prior to starting the current debug session, they will not be associated with the current thread and will be indicated with a blue circle without a check mark. To enable the breakpoints in the current thread, right-click the desired breakpoint and select "Breakpoint Properties..." 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, click 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.

6.6 Exercises

Exercise - 6.1 Run BTSPy

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

Project Creation

1. Copy the project `ch04b/ex02_ble_pair` to `cho6/ex02_btspy`. Make all the necessary name changes and create a Make Target. Use "`<inits>_spy`" for the device name.
2. In the makefile, verify that the `ENABLE_HCI_TRACE` 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

5. Build and download the application to the kit.
6. 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.
7. Run the *ClientControl* utility from `apps\host\client_control\Windows`. To open the utility from inside WICED Studio, right-click the utility and select **Open With > System Editor**.
8. 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).
9. Run the *BTSPy* utility from `wiced_tools\BTSPy\Win32`. To open the utility from inside WICED Studio, right-click the utility and select **Open With > System Editor**.
10. In the *ClientControl* utility, open the HCI UART COM port by clicking on "Open Port".

Testing

11. 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.
12. Use CySmart to connect to the device and observe the messages in the BTSPy window.
13. 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.
14. Read the CapSense button Characteristic and observe the messages.
15. Add a note that says " Button Characteristic Read Complete"
16. Disconnect from the device.

17. Once you are done logging, click on the "Save" button, click on "Stop Logging", and then click on "OK".
18. If you need to re-program the kit, you need to close the port from WICED Client Control first.

Exercise - 6.2 (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 ex01_led_control from the templates folder.
2. Run the 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 "ledState" 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.

