

Please note that Cypress is an Infineon Technologies Company.

The document following this cover page is marked as “Cypress” document as this is the company that originally developed the product. Please note that Infineon will continue to offer the product to new and existing customers as part of the Infineon product portfolio.

Continuity of document content

The fact that Infineon offers the following product as part of the Infineon product portfolio does not lead to any changes to this document. Future revisions will occur when appropriate, and any changes will be set out on the document history page.

Continuity of ordering part numbers

Infineon continues to support existing part numbers. Please continue to use the ordering part numbers listed in the datasheet for ordering.



ModusToolbox™



Hardware Debugging for CYW207xx and CYW208xx

Document Number: 002-20504 Rev. *1

Cypress Semiconductor
An Infineon Technologies Company
198 Champion Court
San Jose, CA 95134-1709
www.cypress.com, www.infineon.com

Contents

1	Introduction.....	3
2	Debug Probe Software Setup	4
2.1	Debug Probe	4
2.2	Cypress MiniProg4	4
2.3	SEGGER J-Link.....	5
3	Kit Setup.....	6
4	Preparing the Embedded Application for Debug.....	7
5	Using the Hardware Debugger	9
5.1	Validate the Hardware Setup	9
5.2	Using the Eclipse IDE for ModusToolbox	11
5.3	Using the Visual Studio Code IDE	13
5.4	Use VS Code for Hardware Debugging	15
5.5	Hardware Debugging from the Command Line	17
6	Troubleshooting	20
	Document History	21
	Worldwide Sales and Design Support.....	22

1 Introduction

ModusToolbox™ provides support for source-code-level debugging of applications running on CYW207xx and CYW208xx devices. Although this debugging technique often conflicts with the real-time requirements of IoT, it can provide valuable insight by stopping the CPU at breakpoints or watchpoints and providing the ability to check on the state of code and data using source code symbols.

The ModusToolbox kit hardware supports source-code-level debugging via a Serial Wire Debug (SWD) interface that provides a means for a development PC to control the execution of the CYW207xx and CYW208xx Arm® CPUs. Third-party JTAG/SWD debug probes are used to interface between the development PC and the debug interface hardware. The PC connected to the debug probe uses associated GDB server software. This server provides a socket interface to the GNU debugger (GDB), allowing it to control the debug interface and Arm CPU. ModusToolbox coordinates the interfaces between these entities: the CYW207xx and CYW208xx Arm CPUs, the debug probe and GDB server, GDB, and the symbols GDB needs to translate between source code symbols and hardware memory addresses, registers, and so on.

This document provides a description of hardware debugging support for ModusToolbox software and CYW207xx and CYW208xx devices. This document describes the generic tools, setup, and techniques. Other kit-specific documents are provided to describe those exact implementations.

The hardware interface for the Debug Access Port is provided to the Arm CPU cores within the CYW207xx and CYW208xx devices. These Arm architecture devices use a 2-pin SWD interface. The SWD pins are SWDIO and SWDCK. Many debug probes support SWD.

Current hardware debug support for CYW207xx and CYW208xx devices requires either a Segger J-Link probe or OpenOCD supported probe.

Hardware and Software Requirements

The following items are required to debug an application developed for Cypress WICED SoC.

Item		Description
Reference design board		A hardware reference design board based on a Cypress CYW207xx and CYW208xx SoC.
One of these:	Segger J-Link	Segger provides several models of JTAG-SWD debug probes that are compatible for hardware debugging. See www.segger.com/downloads/jlink .
	Cypress MiniProg4 Programmer/Debugger	See https://www.cypress.com/documentation/development-kitsboards/cy8ckit-005-miniprogram-and-debug-kit
PC		A computer (Windows, MacOS, or Linux) that hosts the following software items: <ul style="list-style-type: none"> • ModusToolbox • A GDB server and debug probe interface software. For this document we consider the SEGGER GDB server or OpenOCD.
Debug probe USB cable		Used for PC interface to debug hardware.
GDB Server software		OpenOCD software is included with ModusToolbox. Segger GDB Server is available at www.segger.com/downloads/jlink .
Debug probe 10-pin connector/adaptor		An adaptor may be needed to match from a 20-pin JTAG interface on the debug probe to a 10-pin SWD interface on the development kit.

Table 1-1. Hardware Reference

2 Debug Probe Software Setup

2.1 Debug Probe

Purchase debug probes and software separately. SEGGER provides the J-Link probe that has been demonstrated to work well with CYW207xx and CYW208xx devices. Other debug probes, such as Cypress MiniProg4, are also compatible.

The instructions and screen shots are similar for Windows, Linux, and Mac operating systems unless otherwise noted.

2.2 Cypress MiniProg4

Follow the instructions provided with MiniProg4 for installation. No software or driver installation beyond plug-and-play is required for operation.

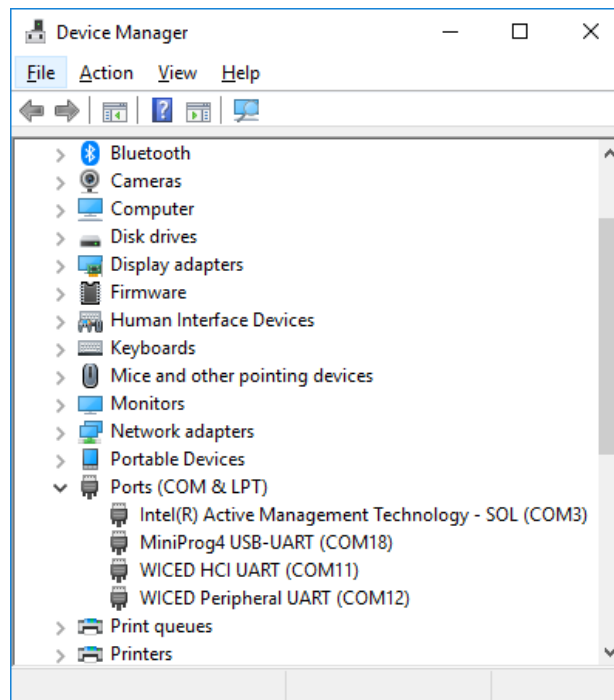


Figure 2-1. Device Manager with MiniProg4 and WICED BT Kit

2.3 SEGGER J-Link

1. Install the Segger J-Link GDB Server. Download the software from www.segger.com/downloads/jlink. Look for “J-Link Software and Documentation Pack”.
2. Once the software is installed, connect 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 on a Windows PC in the Device Manager as [Figure 2-2](#) shows.

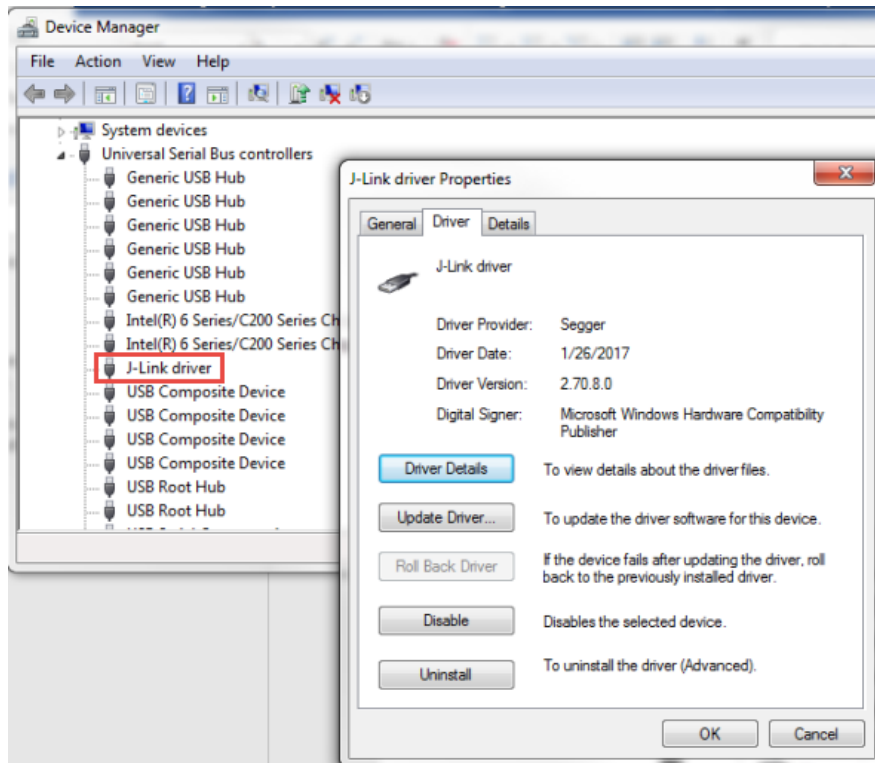


Figure 2-2. Device Manager Listing J-Link Device

3. You can check the SEGGER J-Link hardware connection on a Linux PC using `lsusb` on a terminal to list USB devices. On a Mac operating system, you can use the System Information utility.

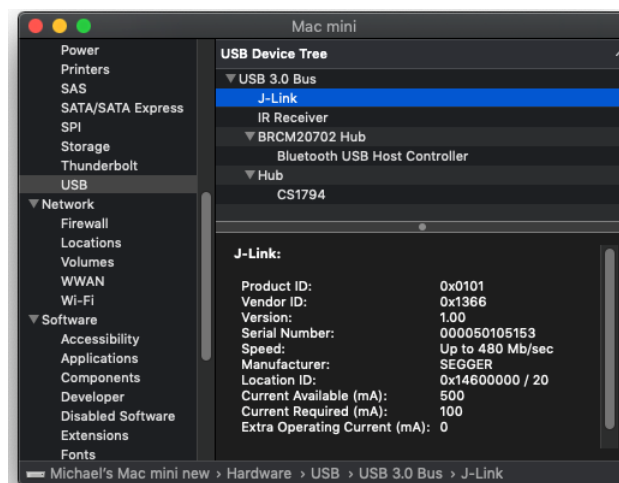


Figure 2-3. Using System Information Utility

3 Kit Setup

Each Cypress IoT development kit has particular setup requirements for the debug interface. Some have a dedicated debug probe socket, either 10-pin or 20-pin. A few kits do not have a dedicated debug socket, but will have pin headers that can be “fly-wired” to a debug probe connector. Besides the physical connection, each board may have some switch or jumper settings necessary to support hardware debugging. Kit-specific documents (kit user guides, for example) are available to describe the exact requirements.



Figure 3-1. Typical Kit with Hardware Debug Probes

4 Preparing the Embedded Application for Debug

By default, CYW207xx and CYW208xx devices do not enable SWD interface pins. These devices boot from the ROM code and do not have an opportunity to set up SWD pins until the ModusToolbox embedded application is loaded and executed. This means that SWD support must be enabled as a build option so that code will be compiled in to configure pins during the application startup sequence to support SWD. The code necessary to configure GPIOs for debugger support is defined in a macro called `SETUP_APP_FOR_DEBUG_IF_DEBUG_ENABLED()`.

Because the pins must be configured prior to attaching the debugger, the application is built to run in a software loop immediately after configuring the GPIO for debug support so that you can attach to the debugger before the main application code begins to execute. The code for this loop is defined in a macro called `BUSY_WAIT_TILL_MANUAL_CONTINUE_IF_DEBUG_ENABLED()`.

These macros are defined in the source code file `<ModusToolbox workspace dir>/<project>/wiced_btsdk/baselib/<device>/WICED/common/spar_utils.h`.

By default, these macros are enabled in an early part of application initialization, just after pins are configured in `wiced_platform_init()`. Depending on the application, it may make sense to move them to another location. Some critical points to consider are whether any GPIO configuration occurs in the application that would be executed subsequent to the debug setup. If the GPIO used for debug have their configuration overwritten by other code, the debug session will not work. Also, if the wait loop is executed in a time-critical portion of the application, then the time-critical functionality could be broken.

To set up the build to enable the desired pins for SWD using an IDE, navigate to the application source files, open the application “makefile” for editing, and set `ENABLE_DEBUG` to ‘1’. Note that the SWD pins are defined for each kit and are configured by the macro definitions in `spar_utils.h`. The macro definitions may need to be modified to change the pins that will be used for SWD. The pin configurator is not used because the pin functionality depends on setting or unsetting `ENABLE_DEBUG`. After editing the makefile, be sure to “make clean” to force a rebuild of all sources if they were previously compiled.

Note that source-code-level debugging depends on source code and debug symbol availability. You can use the embedded application and source code libraries supplied for ModusToolbox for source code debugging. Debug symbols for code and data defined in the device ROM or pre-compiled libraries are not provided. Hardware debugging through these portions will show disassembled machine code.

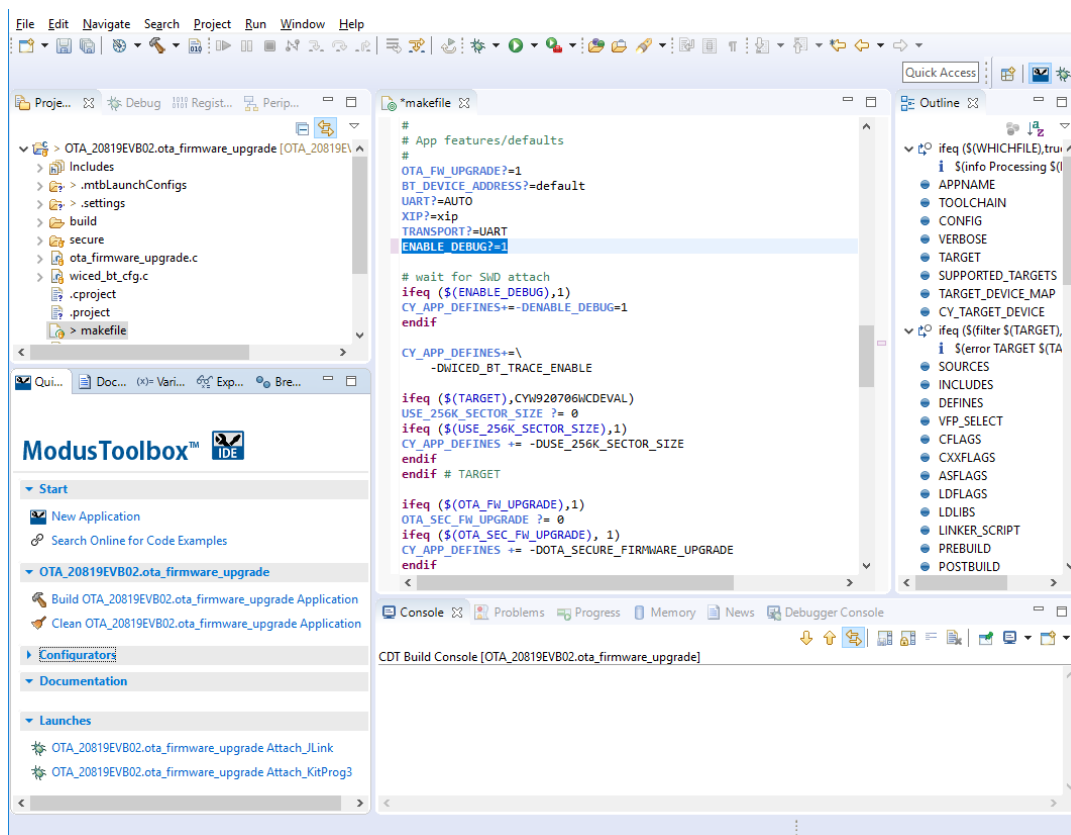


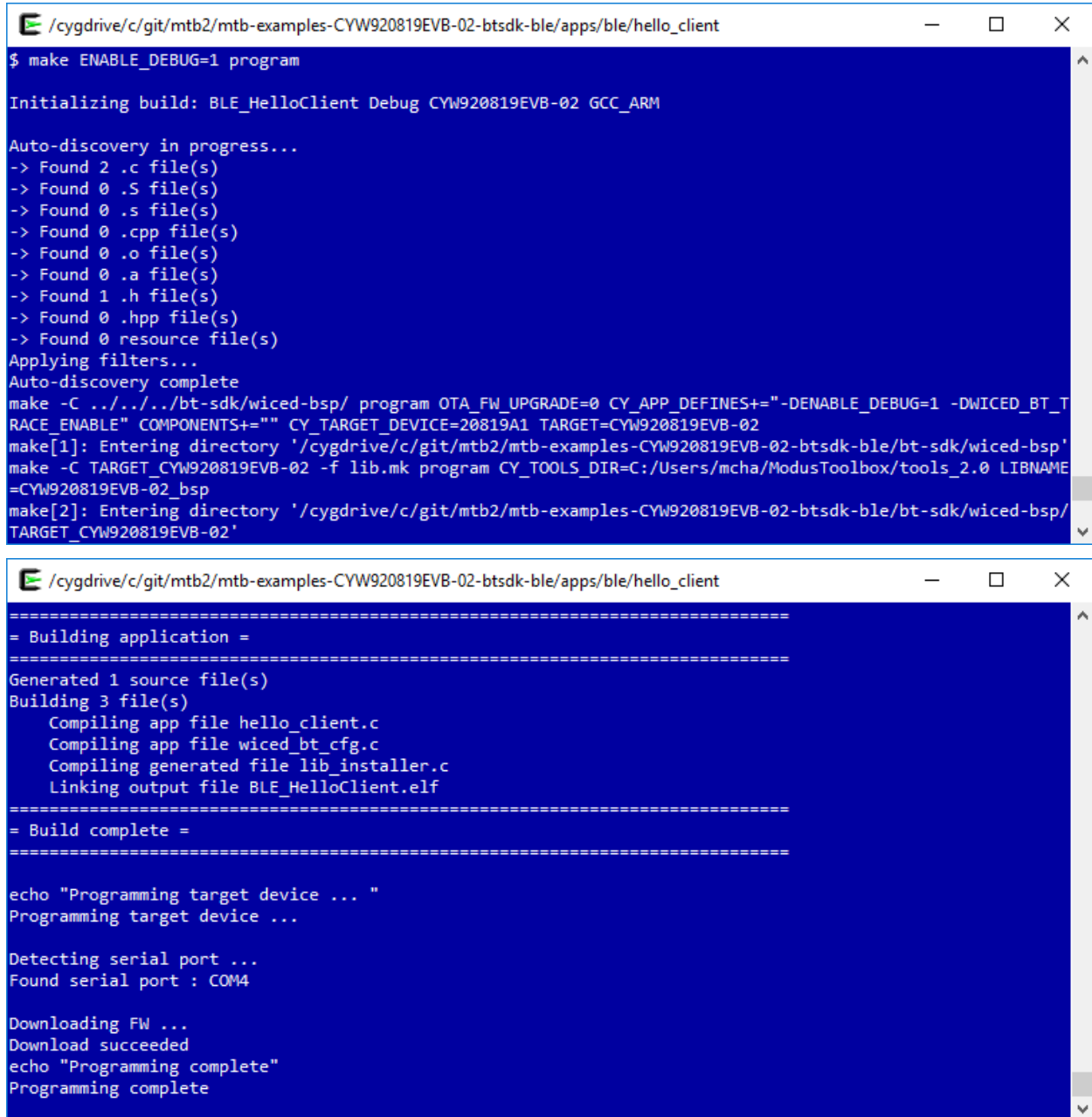
Figure 4-1. Setting the `ENABLE_DEBUG` Feature

Now you can build, program, and download the application.

Similarly, to build from the command line, use the make command line option `ENABLE_DEBUG`:

```
make ENABLE_DEBUG=1 program
```

Additional command line options may be needed depending on the application, `CY_TOOLS_PATHS` path, and platform being used.



```

/cygdrive/c/git/mtb2/mtb-examples-CYW920819EVB-02-btsdk-ble/apps/ble/hello_client
$ make ENABLE_DEBUG=1 program

Initializing build: BLE_HelloClient Debug CYW920819EVB-02 GCC_ARM

Auto-discovery in progress...
-> Found 2 .c file(s)
-> Found 0 .S file(s)
-> Found 0 .s file(s)
-> Found 0 .cpp file(s)
-> Found 0 .o file(s)
-> Found 0 .a file(s)
-> Found 1 .h file(s)
-> Found 0 .hpp file(s)
-> Found 0 resource file(s)
Applying filters...
Auto-discovery complete
make -C ../../../../bt-sdk/wiced-bsp/ program OTA_FW_UPGRADE=0 CY_APP_DEFINES+="-DENABLE_DEBUG=1 -DWICED_BT_T
RACE_ENABLE" COMPONENTS+="" CY_TARGET_DEVICE=20819A1 TARGET=CYW920819EVB-02
make[1]: Entering directory '/cygdrive/c/git/mtb2/mtb-examples-CYW920819EVB-02-btsdk-ble/bt-sdk/wiced-bsp'
make -C TARGET_CYW920819EVB-02 -f lib.mk program CY_TOOLS_DIR=C:/Users/mcha/ModusToolbox/tools_2.0 LIBNAME
=CYW920819EVB-02_bsp
make[2]: Entering directory '/cygdrive/c/git/mtb2/mtb-examples-CYW920819EVB-02-btsdk-ble/bt-sdk/wiced-bsp/
TARGET_CYW920819EVB-02'

=====
= Building application =
=====
Generated 1 source file(s)
Building 3 file(s)
  Compiling app file hello_client.c
  Compiling app file wiced_bt_cfg.c
  Compiling generated file lib_installer.c
  Linking output file BLE_HelloClient.elf
=====
= Build complete =
=====

echo "Programming target device ..."
Programming target device ...

Detecting serial port ...
Found serial port : COM4

Downloading FW ...
Download succeeded
echo "Programming complete"
Programming complete
  
```

Figure 4-2. Command Line Build

5 Using the Hardware Debugger

Before using the debugger, do the following:

- Connect the hardware debugger to the kit.
- Configure the kit for debugging.
- Build, program, and run an application that has debugging enabled on the kit.

Program execution stops progressing at the wait macro `BUSY_WAIT_TILL_MANUAL_CONTINUE_IF_DEBUG_ENABLED()`. See the kit-specific documents for details on the full setup.

5.1 Validate the Hardware Setup

The SWD interface can be tested independently from the ModusToolbox environment. This requires that an embedded application configured with `ENABLE_DEBUG` is first built, downloaded, and running on the kit. Normally, the GDB server application will be run in the background automatically while performing hardware debugging with the Eclipse IDE for ModusToolbox, so you don't need to start it separately. The GUI version of the GDB Server described below can be used to troubleshoot or support command-line debugging, if required.

1. Connect the probe to the kit and both to the computer.
2. Configure the kit and application for hardware debugging.
3. Program the kit with the application.

Steps 4–6 and later depend on the debug probe that you are using:

If you are using a Segger J-Link debug probe:

4. Run the Segger J-Link GDB Server program and set up as shown in [Figure 5-1](#).
5. Once setup is done, click OK to get the window shown in [Figure 5-2](#).
6. Close the GDB server after the test is completed.

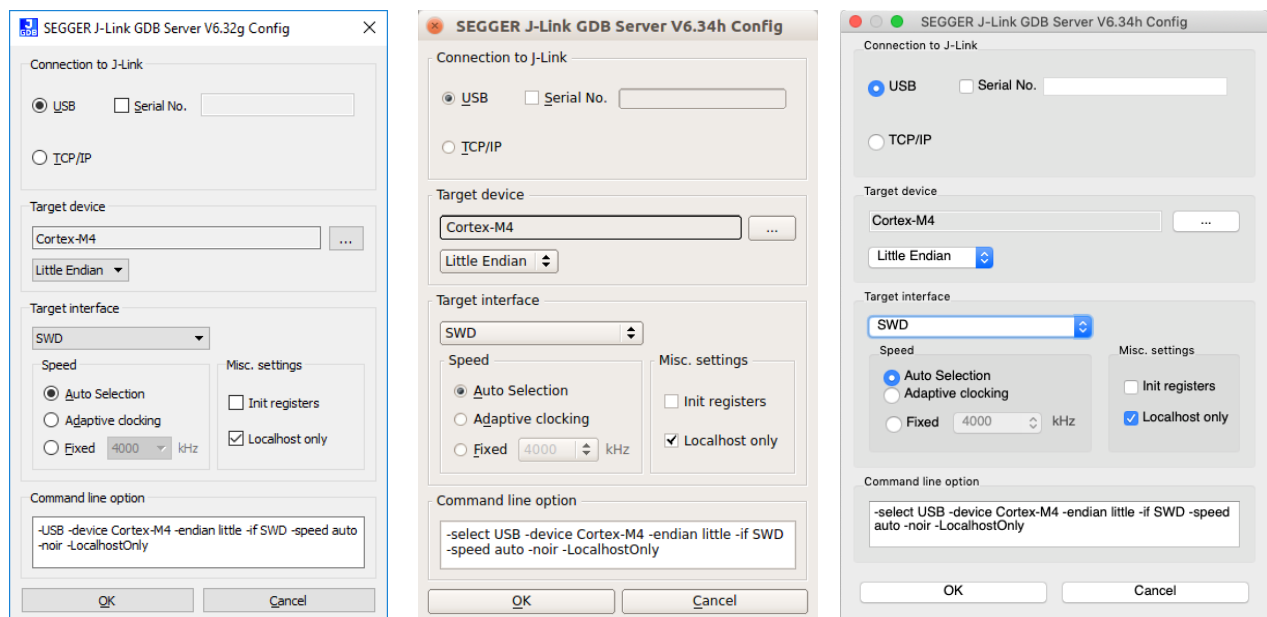


Figure 5-1. SEGGER GUI for Windows, Linux, and macOS

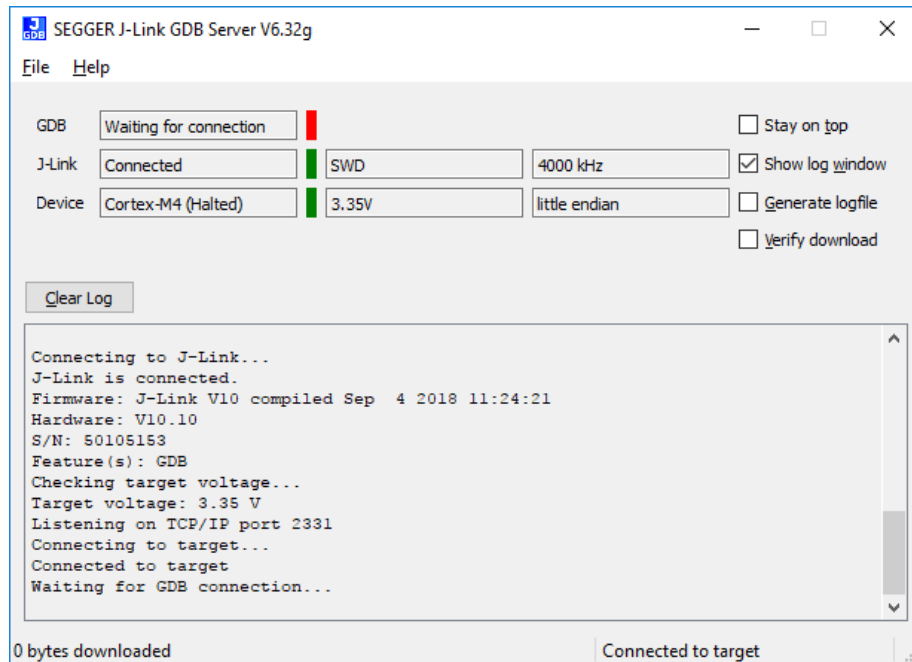


Figure 5-2. Waiting for GDB Connection

If you are using an OpenOCD debug probe such as a MiniProg4:

4. From the application build directory, enter the following command:

```
<ModusToolbox install dir>\tools_2.0\openocd\bin\openocd -s <ModusToolbox install dir>\tools_2.0\openocd\scripts -s ../../../../wiced_btsdk/baselib/<device>/platforms -f <device name>_openocd.cfg
```

Replace <device> with a valid directory name such as 20819A1. Replace <device name> with a valid device name such as CYW20819A1.

5. You should see a result as shown in Figure 5-3.
6. Be sure to close the OpenOCD program when the test is complete.

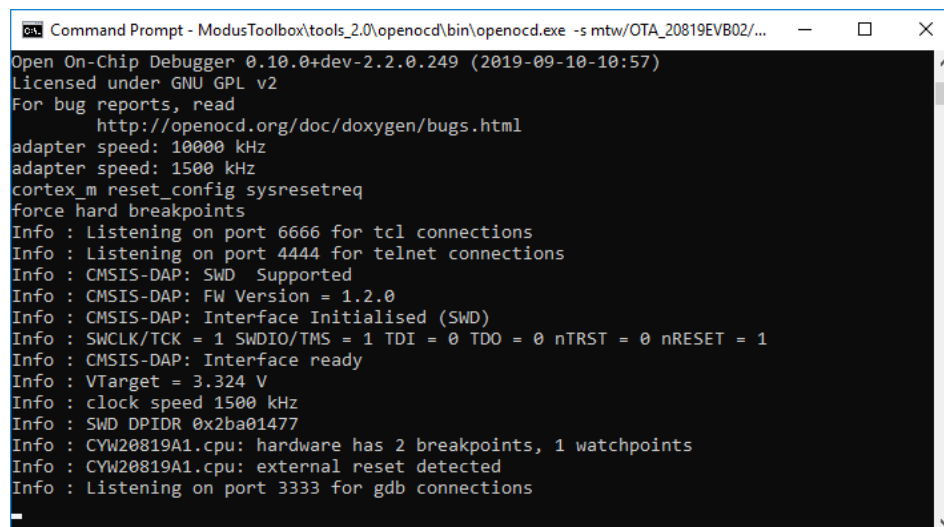


Figure 5-3. Verifying Installation

5.2 Using the Eclipse IDE for ModusToolbox

The Eclipse IDE for ModusToolbox has launch configurations for each platform. These are the Launch items listed in the Quick Panel. Note that this list will change depending on the application project and the platform settings. Although the SWD probe used in the example is MiniProg4, the OpenOCD interface script is more generic and supports any hardware with “KitProg3”. This is why the OpenOCD launch config names include “KitProg3”.

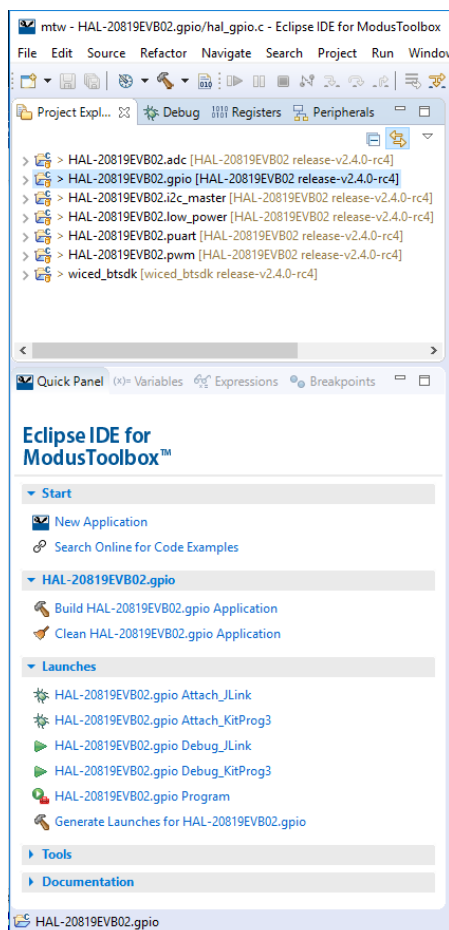


Figure 5-4. Quick Panel

Launch configurations for the Eclipse IDE are generated for each new project using the information from makefiles and .xml files located in the `<ModusToolbox workspace dir>/<project>/wiced_btsdk/baselib/<device>.make/scripts/eclipse` directory. Launch items are configured by default for both J-Link and OpenOCD launch configurations. If a hardware debugger is attached and a debug-enabled application is running on the board, debugging starts when you click one of the **Debug Attach** launch configs. Once launched, clicking the “suspend” debug control button will halt the program execution at the `BUSY_WAIT_TILL_MANUAL_CONTINUE_IF_DEBUG_ENABLED()` loop. This loop can be exited by setting the loop control variable `spar_debug_continue` to non-zero and clicking the debug resume button. At that point, your user application will begin running.

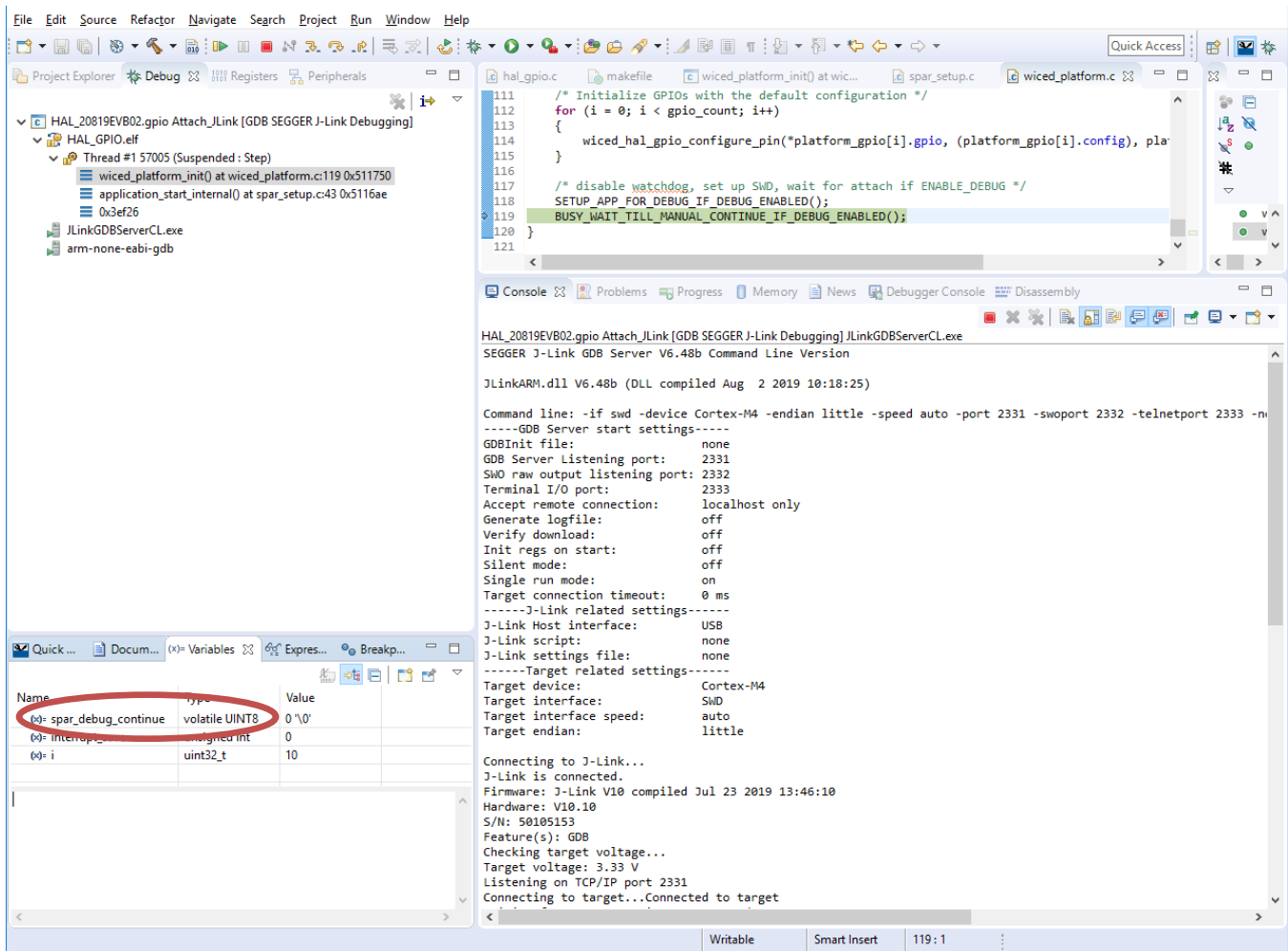


Figure 5-5. Debug View after Pause Using Segger J-Link

An alternative way to access the debug launch configurations is by accessing the menu item **Run > Debug Configurations....** This method also launches a dialog that can be used to modify or define new debug launch configurations.

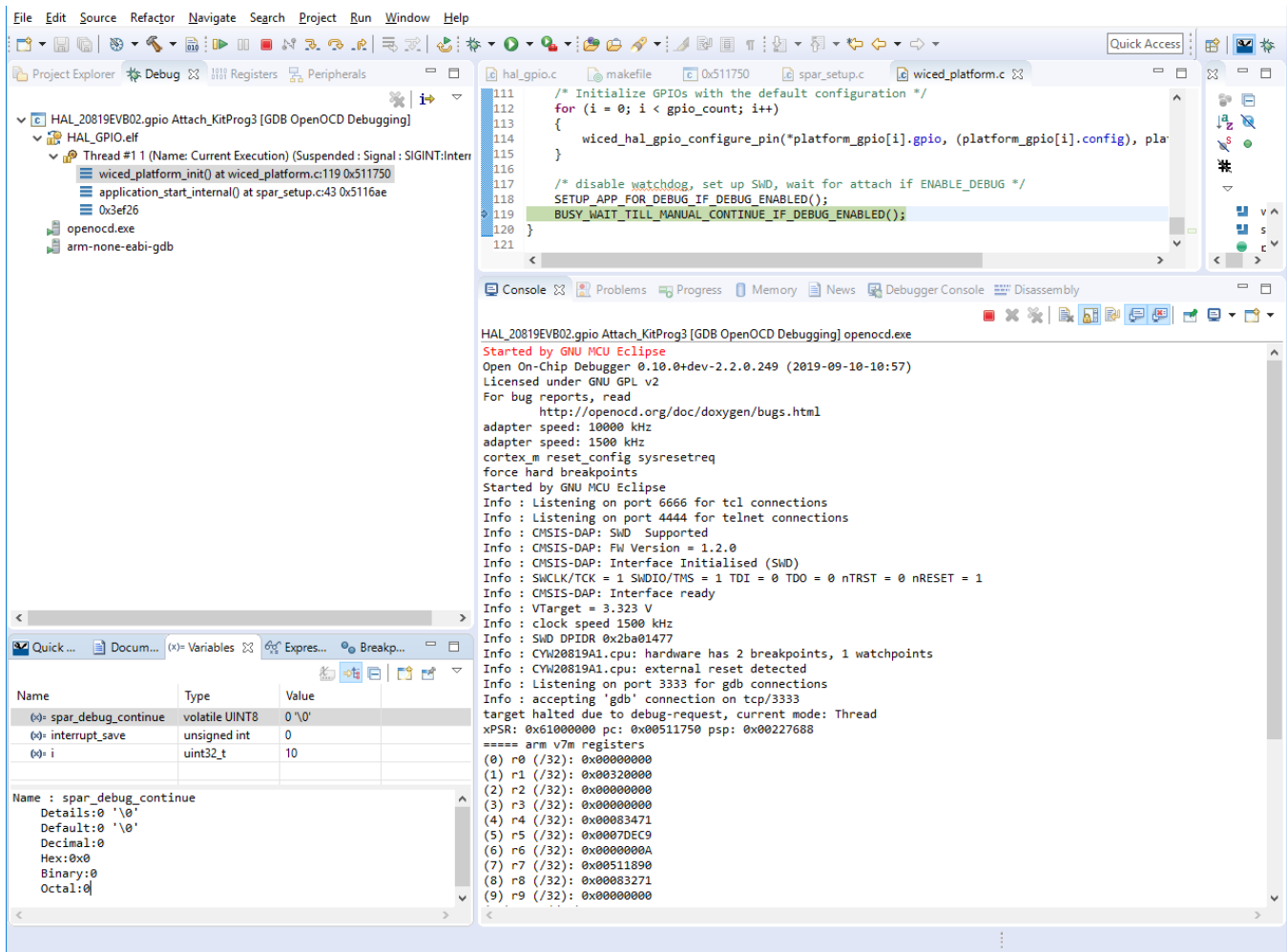


Figure 5-6. Debug View after Pause Using OpenOCD

5.3 Using the Visual Studio Code IDE

ModusToolbox provides a customization method for Visual Studio Code (VS Code). Run the `make ... vscode` command in an application directory to generate a `.vscode` subdirectory. This subdirectory contains `*.json` files used by VS Code for configuration of Intellisense, build tasks, and debugger launches.

1. Install ModusToolbox and VS Code.
2. Install “C/C++” and “Cortex-Debug” extensions in VS Code.
3. Create the `wiced_btsdk` and app(s) either using the Project Creator GUI or from the command line.
4. From the command line in the application directory, run one of the following commands to create the `.vscode` subdirectory.

```
make vscode
```

or

```
make TARGET=<target> vscode
```

This generates the *following* files.

File	Function
<i>tasks.json</i>	Contains the build information
<i>c_cpp_properties.json</i>	Provides build paths and defines to Intellisense
<i>settings.json</i>	Provides paths to gdb and the gdb server
<i>launch.json</i>	Provides debugger launch information

- If using SEGGER J-Link, edit the global *settings.json* file to provide the path and file name of the gdb server.

This file is in OS-dependent locations as follows:

- Windows: %APPDATA%/Code/User/settings.json
- macOS: \$HOME/Library/Application Support/Code/User/settings.json
- Linux: \$HOME/.config/Code/User/settings.json

For example, add:

```
"cortex-debug.JLinkGDBServerPath": "C:/Program Files (x86)/SEGGER/JLink_V648b/JLinkGDBServerCL".
```

- In VS Code, select **File > Open Folder** (or, on the Welcome page, select **Start > Open folder**) to open the application directory.
- To build the application, select **Terminal > Run Task** or **Terminal > Run Build Task**.
- To update the configuration select "Utility: refresh" from **Terminal > Build Task**.

```

/cygdrive/c/git/btsdk240/BT-SDK
$ make vscode

make -C ../../../../wiced_btsdk/dev-kit/bsp/TARGET_CYW920819EVB-02 vscode OTA_FW_UPGRADE=1 CY_APP_DEFINES+=-DWICED
B-02
make[1]: Entering directory '/cygdrive/c/git/btsdk240/BT-SDK/wiced_btsdk/dev-kit/bsp/TARGET_CYW920819EVB-02'

Prebuild operations complete
Commencing build operations...

Initializing build: CYW920819EVB-02_bsp Debug CYW920819EVB-02 GCC_ARM

Auto-discovery in progress...
-> Found 12 .c file(s)
-> Found 0 .S file(s)
-> Found 0 .s file(s)
-> Found 0 .cpp file(s)
-> Found 0 .o file(s)
-> Found 0 .a file(s)
-> Found 15 .h file(s)
-> Found 0 .hpp file(s)
-> Found 0 resource file(s)
Applying filters...
Auto-discovery complete

=====
= Generating IDE files =
=====

To use VSCode with CYW920819EVB-02:
1. Open VSCode
2. Install "C/C++" and "Cortex-Debug" extensions
3. File->Open Folder (Welcome page->Start->Open folder)
4. Select the app root directory and open
5. Builds: Terminal->Run Task
6. Debugging: Edit makefile (or override with command line) to build for hardware debug
   by setting ENABLE_DEBUG=1, rebuild application, program, attach debugger, then
   launch debugger with Run->Start debugging, or F5.
   See document "WICED Hardware Debugging for Bluetooth Kits" for details
  
```

Figure 5-7. make vscode

5.4 Use VS Code for Hardware Debugging

1. Edit the application makefile to set `ENABLE_DEBUG=1`.
2. Clean and rebuild and program the application.
3. Ensure that the debug probe is attached.

Note that BT devices boot from ROM and do not set up SWD support until early in the application initialization (see [Preparing the Embedded Application for Debug](#)). Because of this, hardware debugging BT devices always involves attaching the debugger to a running process.

4. Select **Run > Start Debugging** to launch the hardware debug process.

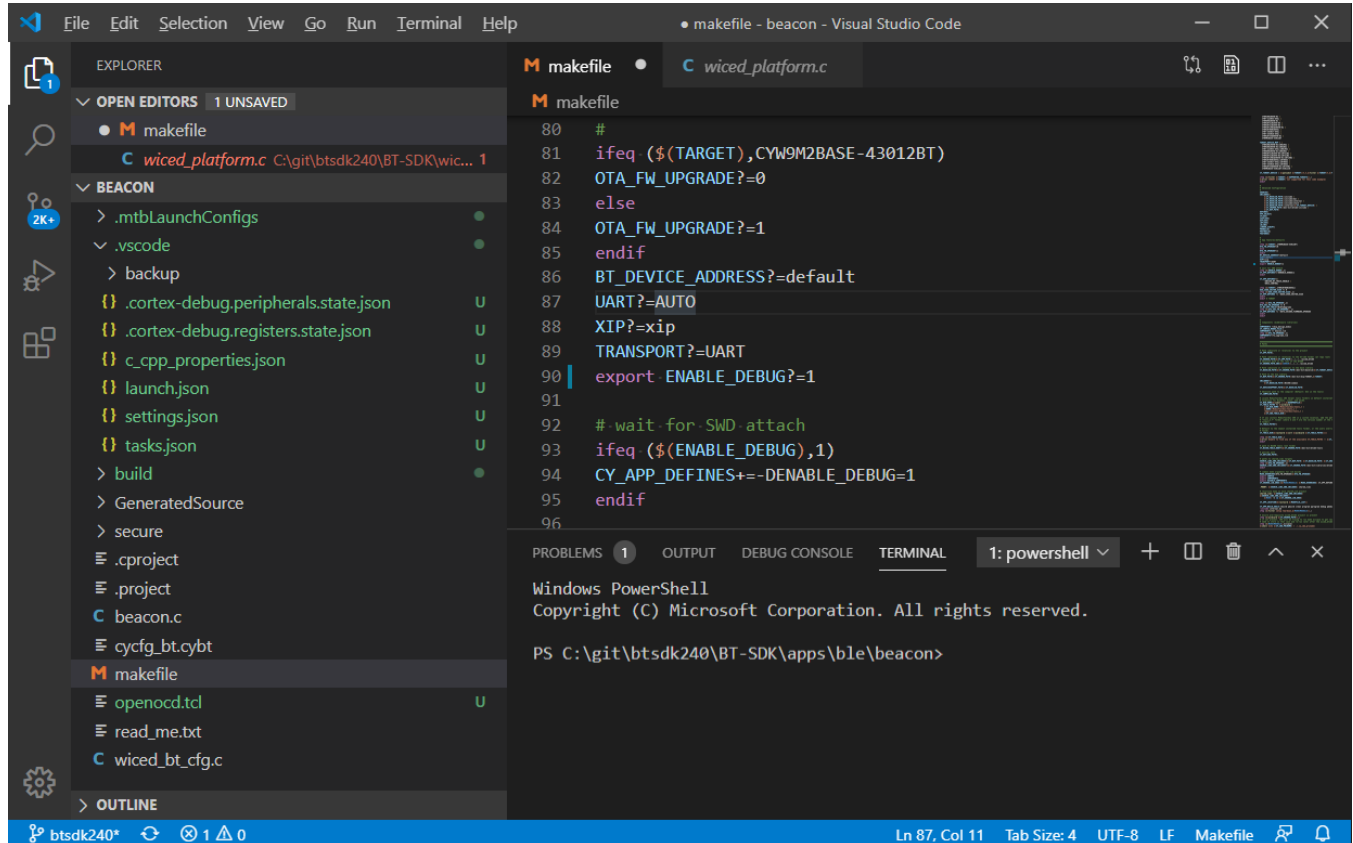


Figure 5-8. VS Code, Application Opened

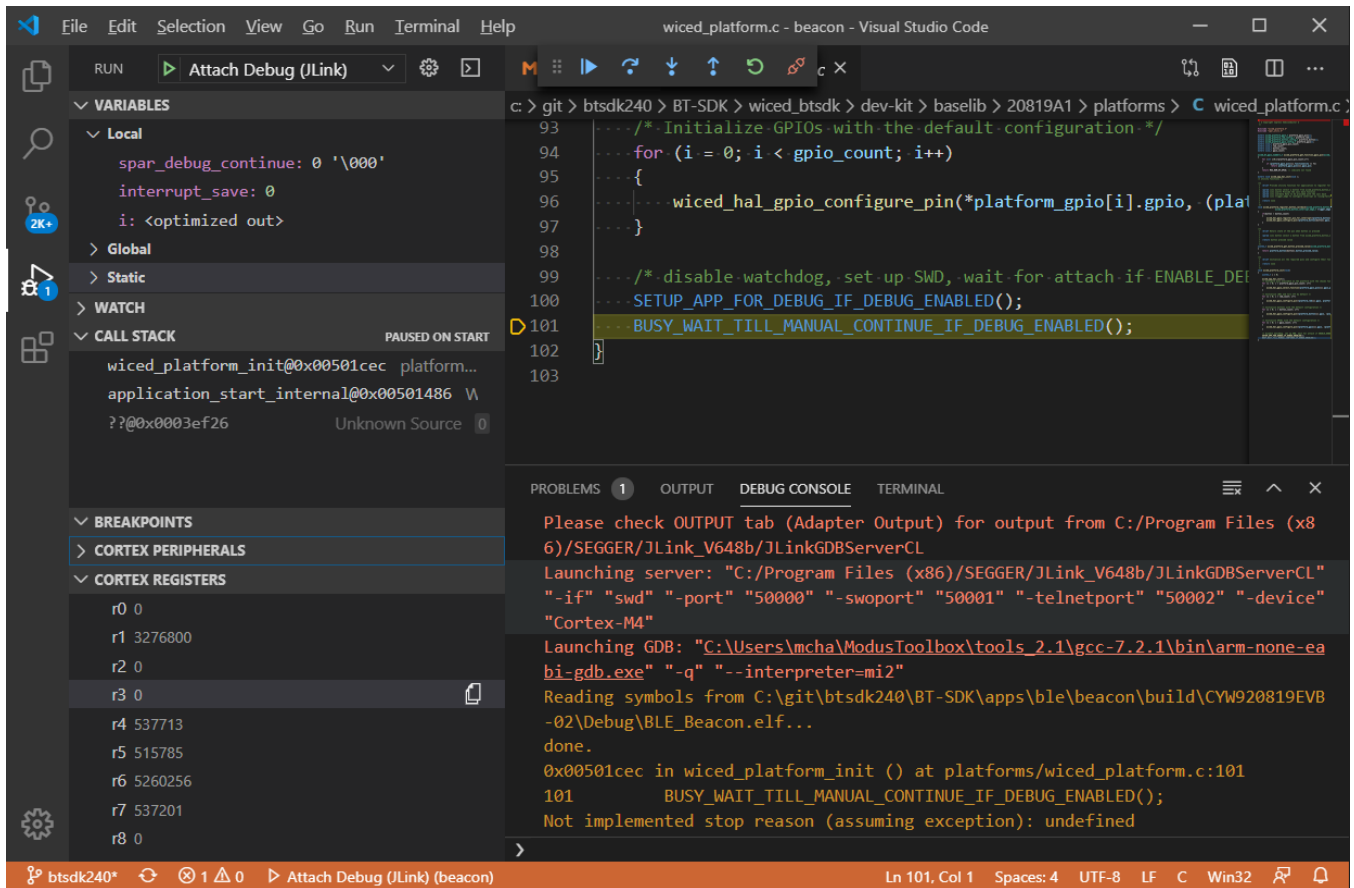


Figure 5-9. VS Code J-Link Debug Session

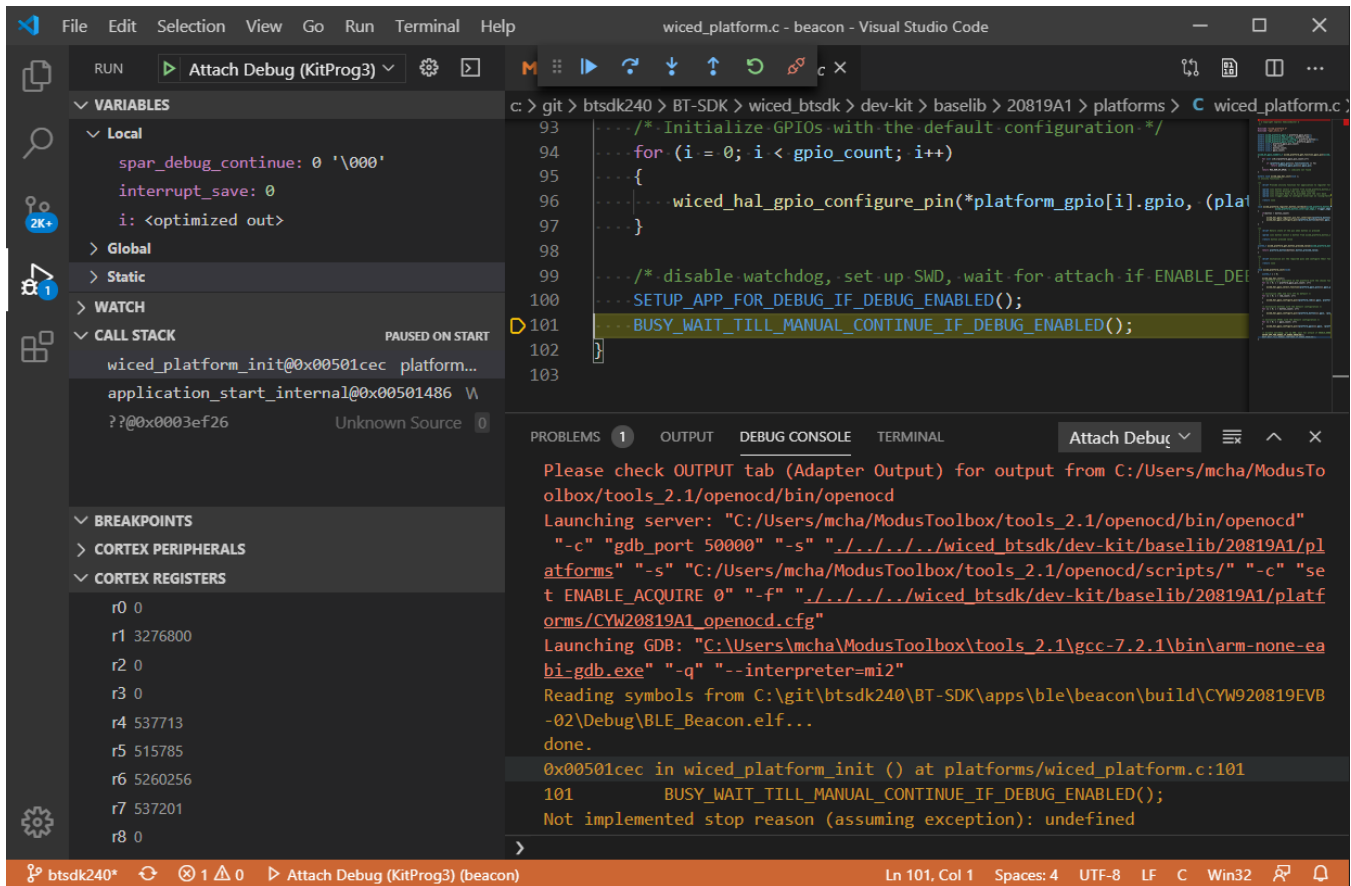


Figure 5-10. VS Code OpenOCD Debug Session

5.5 Hardware Debugging from the Command Line

1. The command line interface can also be used to build applications, download them, launch GDB servers, and run GDB with symbols to perform hardware debugging. Some `make` targets, `make debug`, `make qdebug`, and `make attach`, have been provided to assist with command line hardware debugging. These `make` targets must be run from the application directory. The following steps assume that a probe has connected the host computer to the kit, the kit and application have been configured for hardware debugging, and the kit has been programmed with the application.
2. Launch the GDB Server from the command line.

To launch the OpenOCD GDB Server from the command line, use `make debug` to include a rebuild if needed or `make qdebug` to launch directly. The `makefile` recipe will perform the following command line:

```
<ModusToolbox install dir>\tools_2.0\openocd\bin\openocd.exe -s
<workspace>/wiced_btsdk/baselib/<device>/platforms -s <ModusToolbox install
dir>/tools_2.0/openocd/scripts -f CYW<device>_openocd.cfg
```

To use the SEGGER GDB Server from the command line, define `GDB_SERVER` to be `jlink`: `make debug GDB_SERVER=jlink`.

In this case, the `makefile` recipe will perform the following command line:

```
C:\Program Files (x86)\SEGGER\JLink_V632g\JLinkGDBServerCL.exe" -USB -device Cortex-M4 -
endian little -if SWD -speed auto -noir -LocalhostOnly
```

```

C:\Program Files (x86)\SEGGER\JLink_V632g\JLinkGDBServerCL.exe -USB...
SEGGER J-Link GDB Server V6.32g Command Line Version

JLinkARM.dll V6.32g (DLL compiled Jun 15 2018 17:26:15)

WARNING: Unknown command line parameter -USB found.
Command line: -USB -device Cortex-M4 -endian little -if SWD -speed auto -noir -LocalhostOnly
-----GDB Server start settings-----
GDBInit file:          none
GDB Server Listening port: 2331
SWO raw output listening port: 2332
Terminal I/O port:     2333
Accept remote connection: localhost only
Generate logfile:      off
Verify download:       off
Init regs on start:    off
Silent mode:           off
Single run mode:       off
Target connection timeout: 0 ms
-----J-Link related settings-----
J-Link Host interface:  USB
J-Link script:          none
J-Link settings file:   none
-----Target related settings-----
Target device:          Cortex-M4
Target interface:       SWD
Target interface speed: auto
Target endian:          little

Connecting to J-Link...
J-Link is connected.
Firmware: J-Link V10 compiled Sep  4 2018 11:24:21
Hardware: V10.10
S/N: 50105153
Feature(s): GDB
Checking target voltage...
Target voltage: 3.35 V
Listening on TCP/IP port 2331
Connecting to target...Connected to target
Waiting for GDB connection...
  
```

3. Run the GNU GDB command-line application in a separate console. You can perform this from the application directory using `make attach`. The recipe for this makefile target will launch `gdb` using a command line like: `<ModusToolbox install>/tools_2.0/gcc-7.2.1-1.0/bin/arm-none-eabi-gdb.exe`.

GNU GDB is an interactive command-line application. The following steps are demonstrated in the figure below.

4. Connect GDB with the proper GDB server port with the command `target remote localhost:<port>`.
Replace `<port>` with 3333 when using OpenOCD defaults, or 2331 for J-Link.
5. Load the symbol file for the application with `symbol-file <application *.elf>`. Use `monitor halt` to halt the embedded application and `1` (as in `list`) to list the source code at that location.
6. To break out of the `BUSY_WAIT_TILL_MANUAL_CONTINUE_IF_DEBUG_ENABLED()` loop, set the variable `spar_debug_continue` to non-zero with `set var spar_debug_continue=1`.

```
Command Prompt - tools\gcc-7.2.1-1.0\bin\arm-none-eabi-gdb.exe
GNU gdb (GNU Tools for Arm Embedded Processors 7-2018-q2-update) 8.0.50.20171128-git
Copyright (C) 2017 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software; you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=i686-w64-mingw32 --target=arm-none-eabi".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb) target remote localhost:2331
Remote debugging using localhost:2331
warning: No executable has been specified and target does not support
determining executable automatically. Try using the "file" command.
0x00215b6c in ?? ()
(gdb) symbol-file ../mtw/BLEHelloSensor_mainapp/Debug/BLEHelloSensor_mainapp.elf
Reading symbols from ../mtw/BLEHelloSensor_mainapp/Debug/BLEHelloSensor_mainapp.elf...done.
(gdb) monitor halt
(gdb) set var spar_debug_continue=1
(gdb) s
123     }
(gdb) l
118         }
119
120     /* disable watchdog, set up SWD, wait for attach if ENABLE_JTAG */
121     SETUP_APP_FOR_DEBUG_IF_DEBUG_ENABLED();
122     BUSY_WAIT_TILL_MANUAL_CONTINUE_IF_DEBUG_ENABLED();
123 }
(gdb) _
```

6 Troubleshooting

Here are some issues faced during hardware debugging sessions:

- When breakpoints have been set, and you perform a clean/build followed by the Debug launch (program device and attach hardware debugger), the breakpoints are no longer operative. As a workaround, set breakpoints after the rebuild, program, and attach sequence.
- The ending of debug sessions, when stopped at a breakpoint using the J-Link and OpenOCD probes show different behaviors. When using the J-Link probe, execution continues after the session is ended. When using the OpenOCD probe execution stops at the breakpoint when the session is ended.
- When using the OpenOCD probe and a device such as CYW20819A1 that supports only two hardware breakpoints, the message "can't add breakpoint: resource not available" is displayed if two hardware breakpoints are already in use and an attempt is made to single step.
- During a hardware debug session, an attempt to "restart the process" without first terminating and restarting manually will fail. As explained in Section 4, the CYW20xxx device reboots to ROM code that initializes GPIO to a default state, disabling the SWD configuration. Restart cannot be supported.
- When using breakpoints or stepping in XIP (execute in place) code, typical for the CYW20819A1, use hardware breakpoints. The code resides in flash memory and cannot be directly overwritten with soft breakpoints. Be aware of the limited number of hardware breakpoints and watchpoints supported by the devices.
- Immediately after Attach is launched, the debugger may stop at a location that does not show source code and instead has a message like "Break at address "0x5121d0" with no debug information available, or outside of program code". In this case, use the **Resume** button (or select **Run > Resume**), and then halt the execution with the **Suspend** button (or select **Run > Suspend**). Source code including the line with the `BUSY_WAIT_TILL_MANUAL_CONTINUE_IF_DEBUG_ENABLED()` macro should then be displayed.
- When single-stepping through sources, progress is stopped, and a message is displayed like "Break at address "0x7ecba" with no debug information available, or outside of program code". This is typical when stepping into a library or ROM code where no debug symbols are available. The way out is to select the function further up the call stack and work with a subsequent breakpoint at that source code level.
- If the GDB server fails to launch, look for and close any other instances of the GDB client application that may have been left running (*arm-none-eabi-gdb.exe*).
- Remember to "make clean" before rebuilding sources after making any edits to the makefile.

Document History

Document Title: Hardware Debugging For CYW207xx And CYW208xx

Document Number: 002-20504

Revision	ECN	Submission Date	Description of Change
**	5861331	08/23/2017	Initial release
*A	6373682	11/02/2018	Updated for ModusToolbox
*B	6486075	02/15/2019	Updated title Changed ENABLE_JTAG setting to ENABLE_DEBUG Added references to Linux and Mac Added description for CYW208xx Added troubleshooting section
*C	6554890	04/23/2019	Removed Associated Part Family
*D	6592701	06/11/2019	Updates throughout the document
*E	6674063	09/26/2019	Updated for ModusToolbox 2.0 and MiniProg4.
*F	6700905	10/15/2019	Added troubleshooting note.
*G	6735026	11/20/2019	Added additional description for command line debugging and a troubleshooting note.
*H	6848929	04/08/2020	Added support for VS Code
*I	6943882	08/05/2020	Added troubleshooting for Eclipse Debug launch.

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at [Cypress Locations](#).

Products

Arm® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

[PSoC 1](#) | [PSoC 3](#) | [PSoC 4](#) | [PSoC 5LP](#) | [PSoC 6 MCU](#)

Cypress Developer Community

[Community](#) | [Code Examples](#) | [Projects](#) | [Videos](#) | [Blogs](#) | [Training](#) | [Components](#)

Technical Support

cypress.com/support

All other trademarks or registered trademarks referenced herein are the property of their respective owners.



Cypress Semiconductor
An Infineon Technologies Company
198 Champion Court
San Jose, CA 95134-1709

© Cypress Semiconductor Corporation, 2017-2020. This document is the property of Cypress Semiconductor Corporation and its subsidiaries ("Cypress"). This document, including any software or firmware included or referenced in this document ("Software"), is owned by Cypress under the intellectual property laws and treaties of the United States and other countries worldwide. Cypress reserves all rights under such laws and treaties and does not, except as specifically stated in this paragraph, grant any license under its patents, copyrights, trademarks, or other intellectual property rights. If the Software is not accompanied by a license agreement and you do not otherwise have a written agreement with Cypress governing the use of the Software, then Cypress hereby grants you a personal, non-exclusive, nontransferable license (without the right to sublicense) (1) under its copyright rights in the Software (a) for Software provided in source code form, to modify and reproduce the Software solely for use with Cypress hardware products, only internally within your organization, and (b) to distribute the Software in binary code form externally to end users (either directly or indirectly through resellers and distributors), solely for use on Cypress hardware product units, and (2) under those claims of Cypress's patents that are infringed by the Software (as provided by Cypress, unmodified) to make, use, distribute, and import the Software solely for use with Cypress hardware products. Any other use, reproduction, modification, translation, or compilation of the Software is prohibited.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS DOCUMENT OR ANY SOFTWARE OR ACCOMPANYING HARDWARE, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. No computing device can be absolutely secure. Therefore, despite security measures implemented in Cypress hardware or software products, Cypress shall have no liability arising out of any security breach, such as unauthorized access to or use of a Cypress product. CYPRESS DOES NOT REPRESENT, WARRANT, OR GUARANTEE THAT CYPRESS PRODUCTS, OR SYSTEMS CREATED USING CYPRESS PRODUCTS, WILL BE FREE FROM CORRUPTION, ATTACK, VIRUSES, INTERFERENCE, HACKING, DATA LOSS OR THEFT, OR OTHER SECURITY INTRUSION (collectively, "Security Breach"). Cypress disclaims any liability relating to any Security Breach, and you shall and hereby do release Cypress from any claim, damage, or other liability arising from any Security Breach. In addition, the products described in these materials may contain design defects or errors known as errata which may cause the product to deviate from published specifications. To the extent permitted by applicable law, Cypress reserves the right to make changes to this document without further notice. Cypress does not assume any liability arising out of the application or use of any product or circuit described in this document. Any information provided in this document, including any sample design information or programming code, is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. "High-Risk Device" means any device or system whose failure could cause personal injury, death, or property damage. Examples of High-Risk Devices are weapons, nuclear installations, surgical implants, and other medical devices. "Critical Component" means any component of a High-Risk Device whose failure to perform can be reasonably expected to cause, directly or indirectly, the failure of the High-Risk Device, or to affect its safety or effectiveness. Cypress is not liable, in whole or in part, and you shall and hereby do release Cypress from any claim, damage, or other liability arising from any use of a Cypress product as a Critical Component in a High-Risk Device. You shall indemnify and hold Cypress, its directors, officers, employees, agents, affiliates, distributors, and assigns harmless from and against all claims, costs, damages, and expenses, arising out of any claim, including claims for product liability, personal injury or death, or property damage arising from any use of a Cypress product as a Critical Component in a High-Risk Device. Cypress products are not intended or authorized for use as a Critical Component in any High-Risk Device except to the limited extent that (i) Cypress's published data sheet for the product explicitly states Cypress has qualified the product for use in a specific High-Risk Device, or (ii) Cypress has given you advance written authorization to use the product as a Critical Component in the specific High-Risk Device and you have signed a separate indemnification agreement.

Cypress, the Cypress logo, Spansion, the Spansion logo, and combinations thereof, WICED, PSoC, CapSense, EZ-USB, F-RAM, and Traveo are trademarks or registered trademarks of Cypress in the United States and other countries. For a more complete list of Cypress trademarks, visit cypress.com. Other names and brands may be claimed as property of their respective owners.