Hardware debugging for AIROC™ Bluetooth® System-on-Chip devices

ModusToolbox™

About this document

Scope and purpose

This document provides a description of hardware debugging support for ModusToolbox™ and Bluetooth® SDK (BTSDK) software and AIROC™ CYWxxxxx devices.

Intended audience

Embedded developers using AIROC™ BTSDK solutions desiring to perform hardware debugging on CYWxxxxx devices.

Table of contents

1 Introduction

ModusToolbox™ provides support for source-code-level debugging of applications running on AIROC™ Bluetooth® System-on-Chip (SoC) devices in the CYW207xx, CYW208xx, CYW43012C0, and CYW307xx chip families, referred to as CYWxxxxx in the rest of this document. 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 CYWxxxxx Arm® CPUs. Third-party JTAG/SWD debug probes may be used to interface between the development PC and the debug interface hardware.

Some kits use built-in debug probes and do not require additional 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 CYWxxxxx 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 CYWxxxxx 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 CYWxxxxx 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 CYWxxxxx devices requires either a SEGGER J-Link probe or an OpenOCD-supported probe.

1.1 Hardware and software requirements

The following items are required to debug an application developed for Infineon AIROC™ SoCs.

Table 1 Hardware reference

Item		Description
Reference design board		A hardware reference design board based on an Infineon CYWxxxxx SoC.
One of these:	Hardware reference design board using KitProg3 USB/serial bridge	No additional hardware dongle is needed when using kits with PSoC™ 5LP-based KitProg3 device, for example CY8C5868LTI-LP039.
	SEGGER J-Link	SEGGER provides several models of JTAG-SWD debug probes that are compatible for hardware debugging. See **www.segger.com/downloads/jlink**.
	Infineon MiniProg4 Programmer/Debugger	See **https://www.infineon.com/cms/en/product/evaluation-boards/cy8ckit-005/**

Item	Description
PC	A computer (Windows, macOS, or Linux) that hosts the following software items: • 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/adapter	An adapter 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.

2 Debug probe software setup

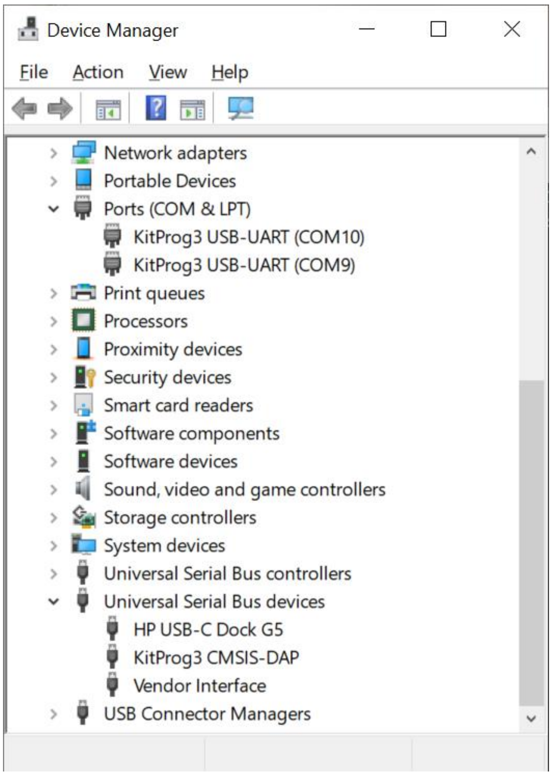
2.1 Debug probe

Purchase debug probes and software separately if needed. SEGGER provides the J-Link probe that has been demonstrated to work well with CYWxxxxx devices. Other debug probes, such as the Infineon MiniProg4, are also compatible. Kits with the KitProg3 USB-serial bridge provide similar built-in functionality as the MiniProg4.

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

2.2 KitProg3

KitProg3 software setup is provided with ModusToolbox™, along with documentation on updates and support.



2.3 Infineon MiniProg4

Follow the instructions provided with MiniProg4 for installation. ModusToolbox™ provides the supporting software.

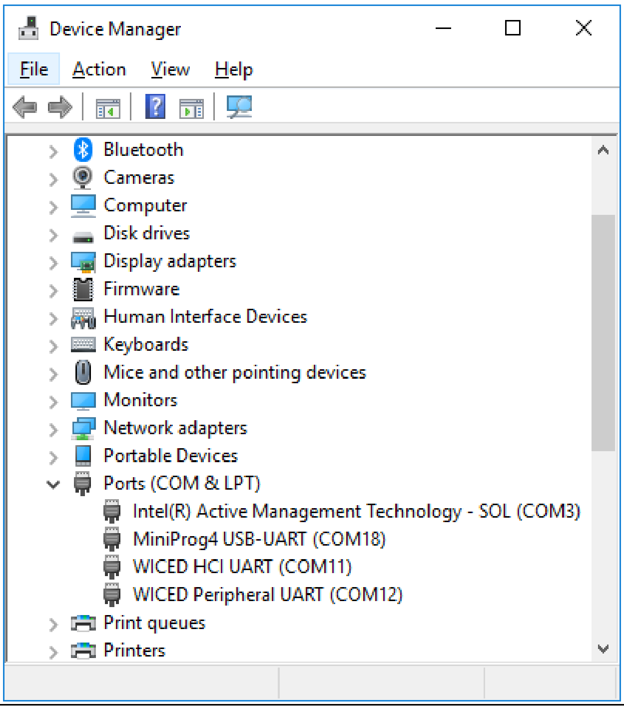


Figure 1 Device Manager with MiniProg4 and AIROC™ kit

2.4 SEGGER J-Link

Do the following to install the SEGGER J-Link GDB Server:

1. Download the software from **www.segger.com/downloads/jlink** and 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** shows.

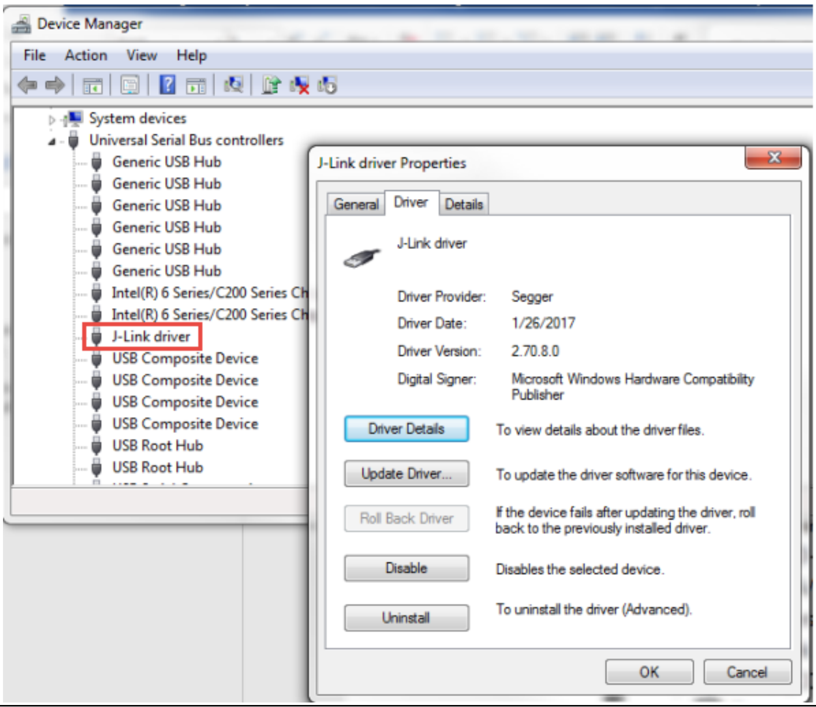


Figure 2 **Device Manager listing J-Link device**

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 macOS operating system, you can use the System Information utility.

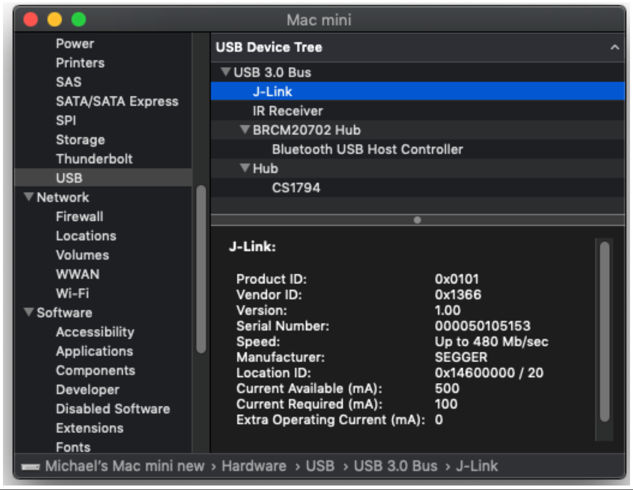


Figure 3 **Using System Information utility**

3 Kit setup

Each Infineon IoT development kit has particular setup requirements for the debug interface. Many kits use the KitProg3 USB-serial bridge chip, for example the CYW920835M2IPA1. These kits do not require a dongle for hardware debugging. 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.

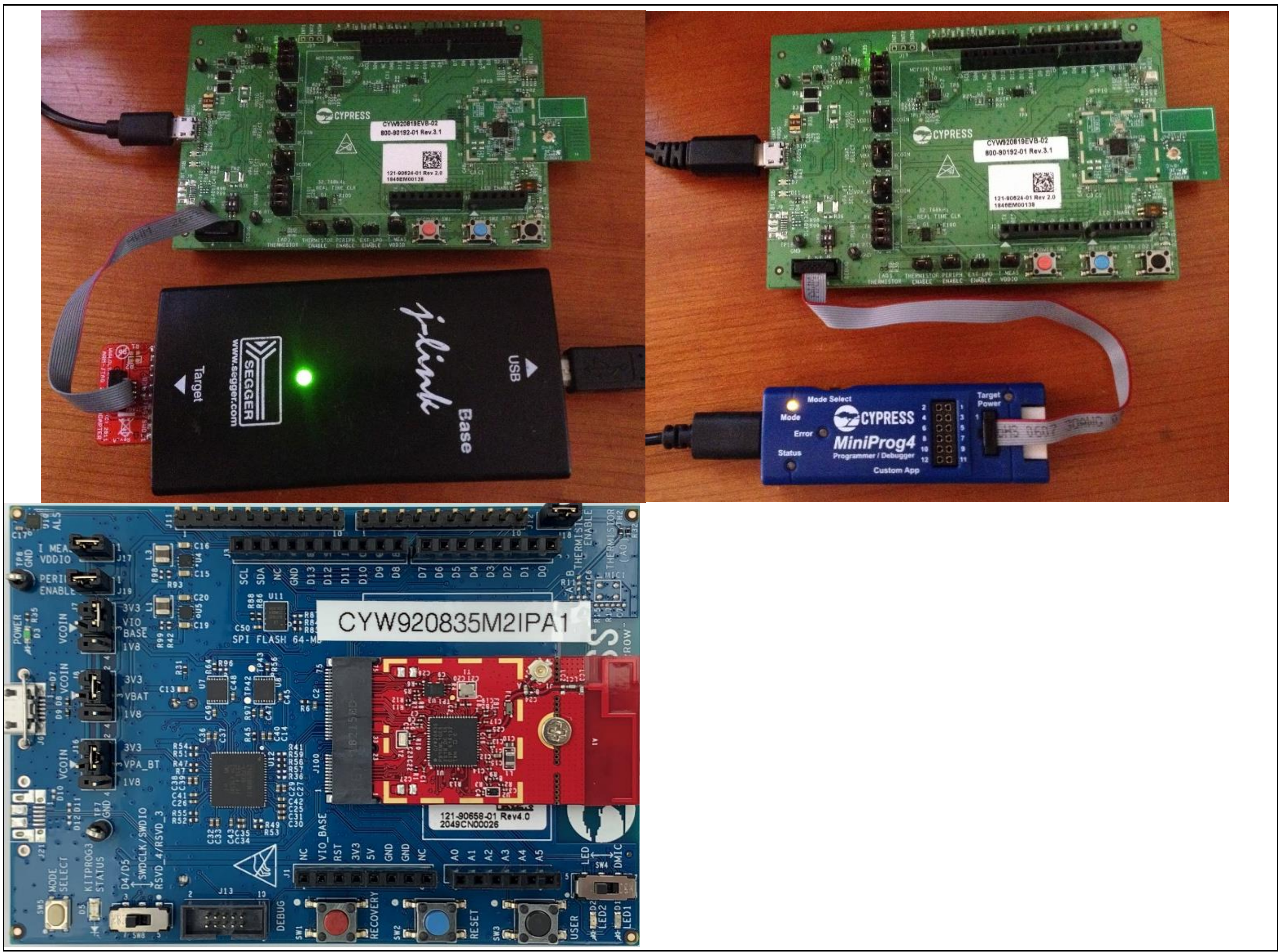


Figure 4 Typical kit with with and without hardware debug probes

4 Preparing the embedded application for debug

By default, CYWxxxxx 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 *<workspace dir>/mtb_shared/wiced_btsdk/baselib/<device>/<branch>/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, 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: *The 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.*

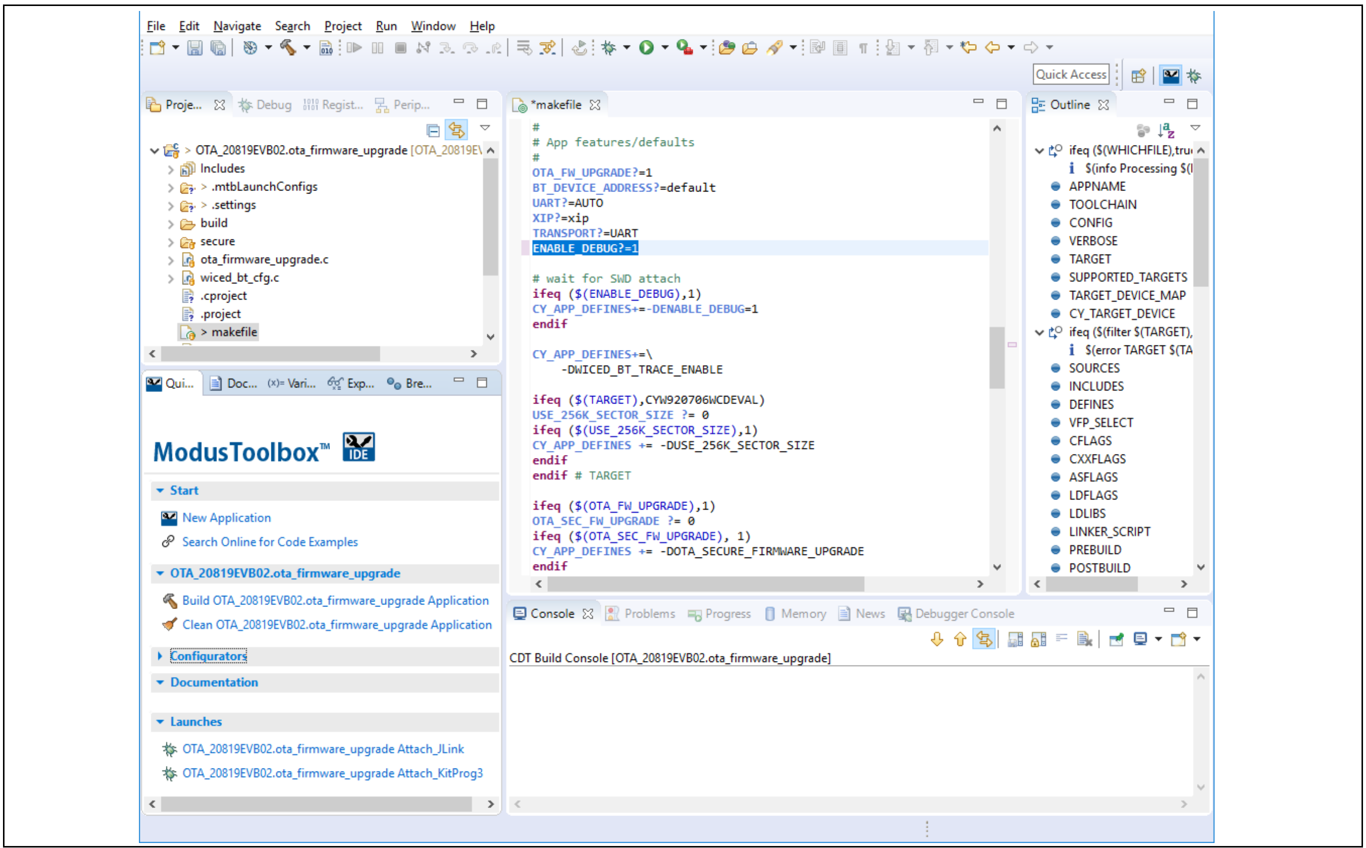


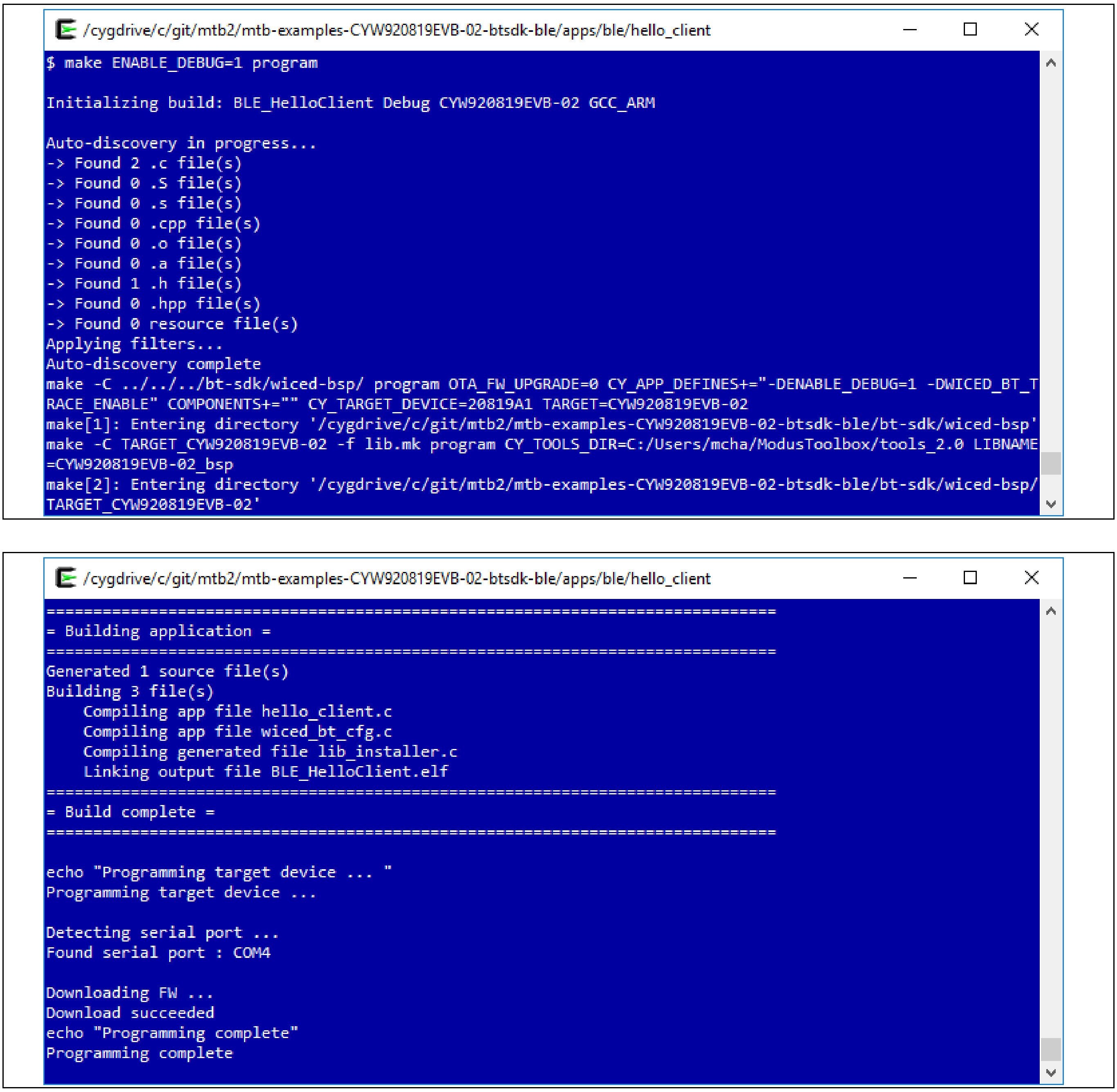
Figure 5 Setting the ENABLE_DEBUG feature

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'
```

```
/cygdrive/c/git/mtb2/mtb-examples-CYW920819EVB-02-btsdk-ble/apps/ble/hello_client
==============================================================================
= 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 6 **Command line build**

5 Using the hardware debugger

Before using the debugger, do the following:

1. Connect the hardware debugger to the kit.
2. Configure the kit for debugging.
3. 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 7**.
5. Once setup is done, click **OK** to get the window shown in **Figure 8**.
6. Close the GDB server after the test is completed.

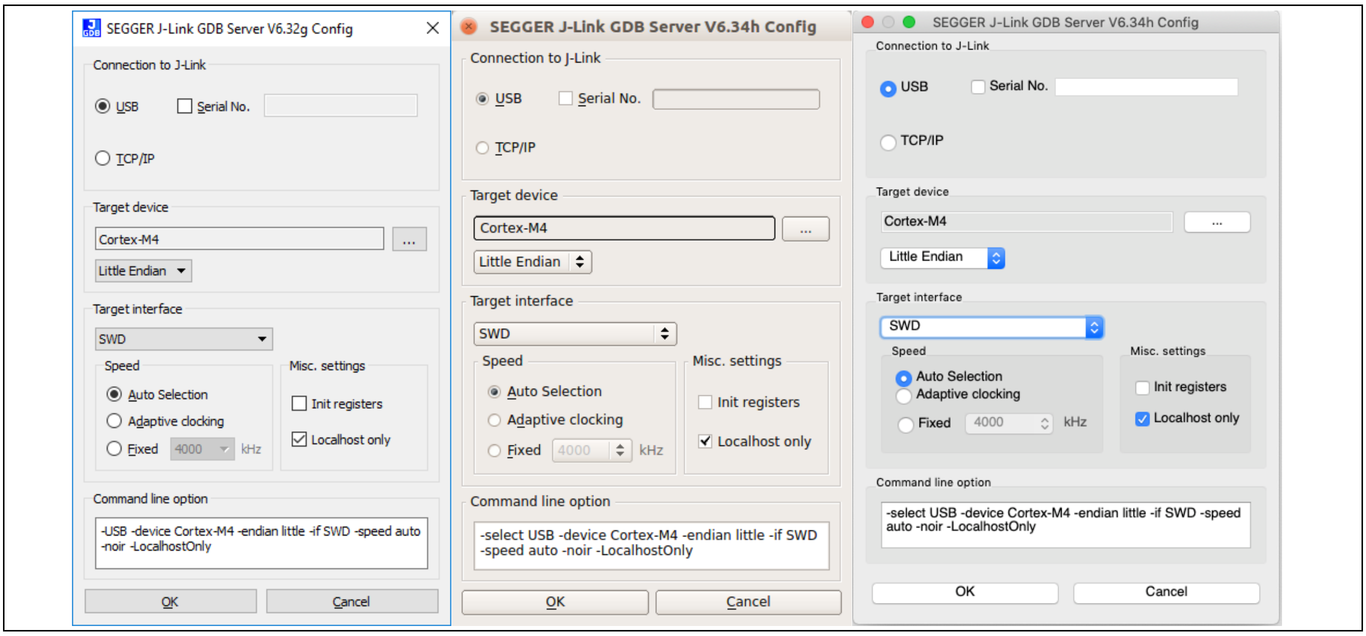


Figure 7 SEGGER GUI for Windows, Linux, and macOS

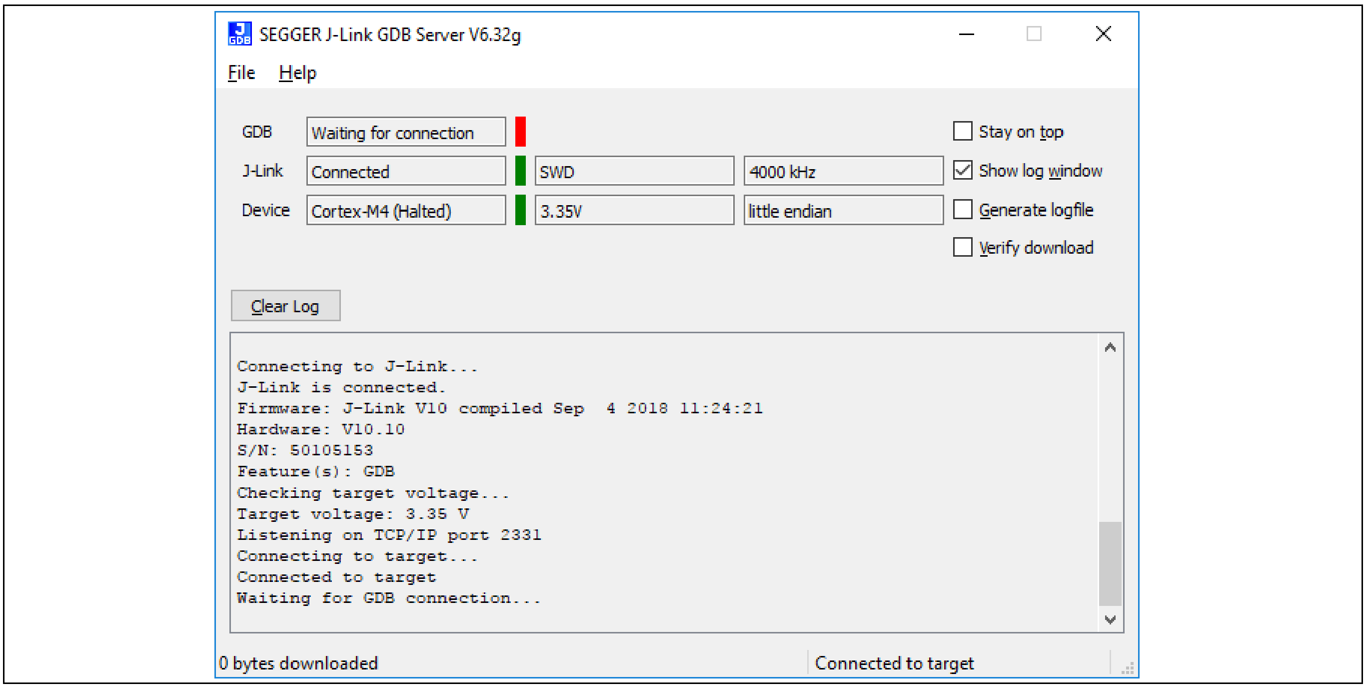


Figure 8 **Waiting for GDB connection**

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

7. From the application build directory, navigate to the ModusToolbox™ install folder and enter the following command:
```
<install dir>\tools_<X.Y>\openocd\bin\openocd -s <install-dir>\tools_<X.Y>\openocd\scripts -s <workspace-dir>/mtb_shared/wiced_btsdk/baselib/<device>/<branch>/platforms -f <device name>_openocd.cfg
```
   - Replace <install-dir> with the ModusToolbox™ install directory on your system.
   - Replace tools_<X.Y> with the version of the tools folder found in your install directory.
   - Replace <workspace-dir> with the workspace path where your application has been created, usually in the user home directory in the 'mtw' folder.
   - Replace <device>/<branch> with the directory name for your device such as 20819A1 and the git branch subfolder created during application project creation.
   - Replace <device name> with a valid device name such as CYW20819A1.

   You should see a result as shown in **Figure 9**.

8. Be sure to close the OpenOCD program when the test is complete.

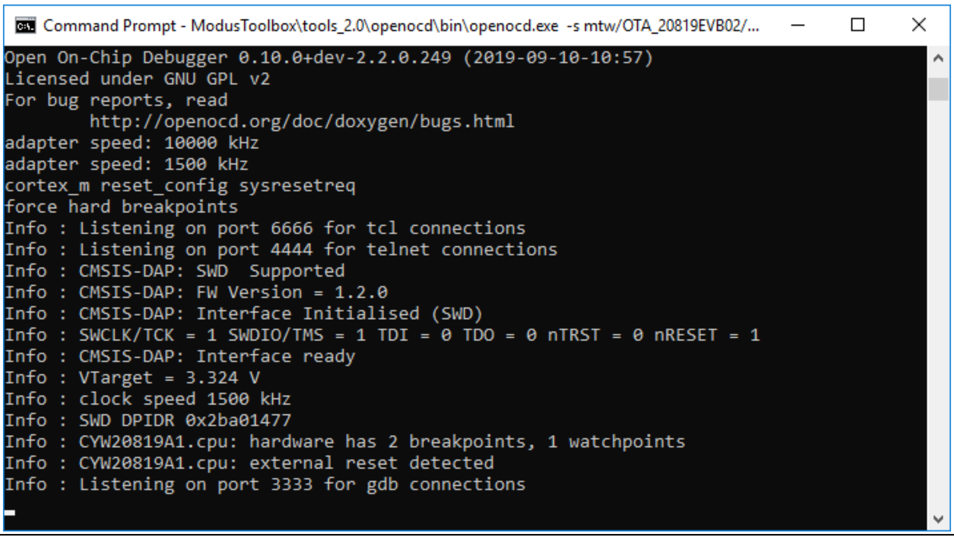


Figure 9 Verifying Installation

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”.

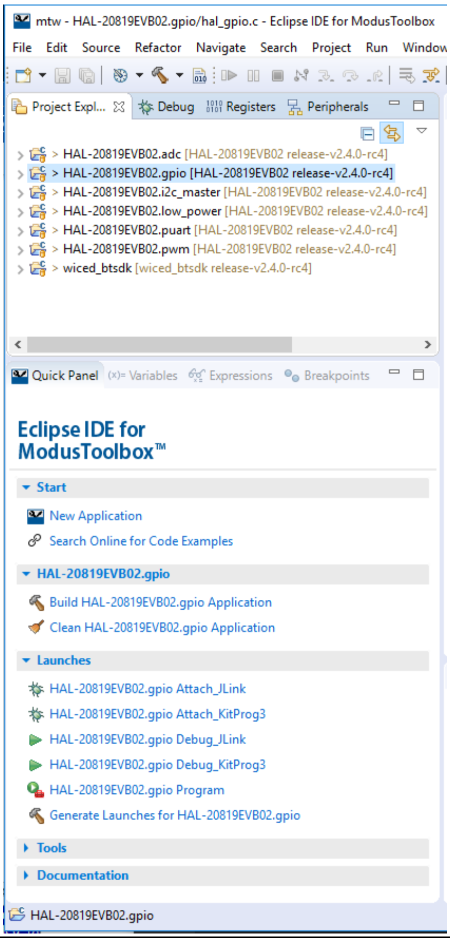


Figure 10 Quick Panel

Launch configurations for the Eclipse IDE are generated for each new project using the information from Makefiles and *.xml* files located in the *< workspace-dir>/mtb_shared/wiced_btsdk/baselib/<device>/<branch>/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.

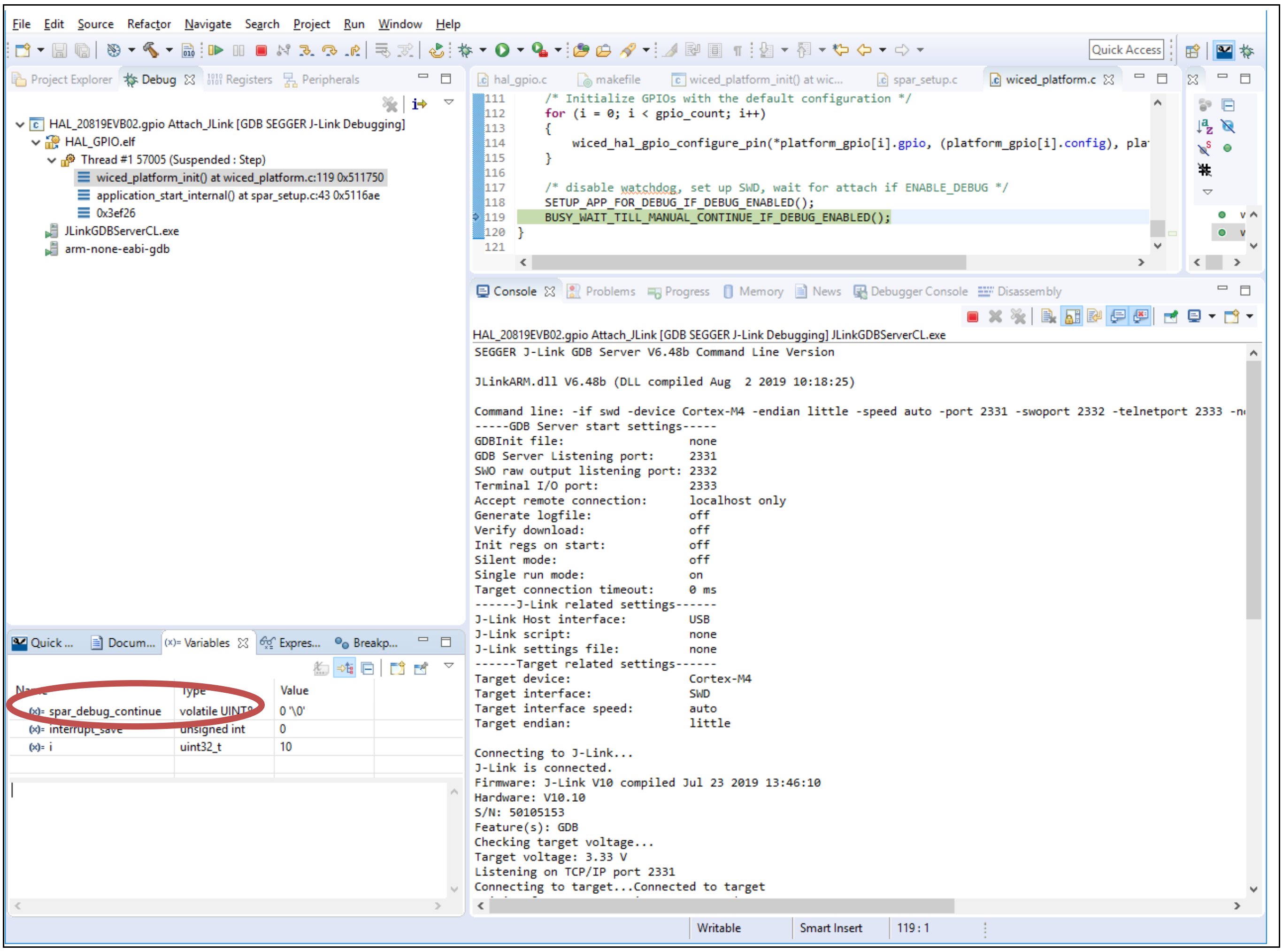


Figure 11 **Debug view after pause using SEGGER J-Link**

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.

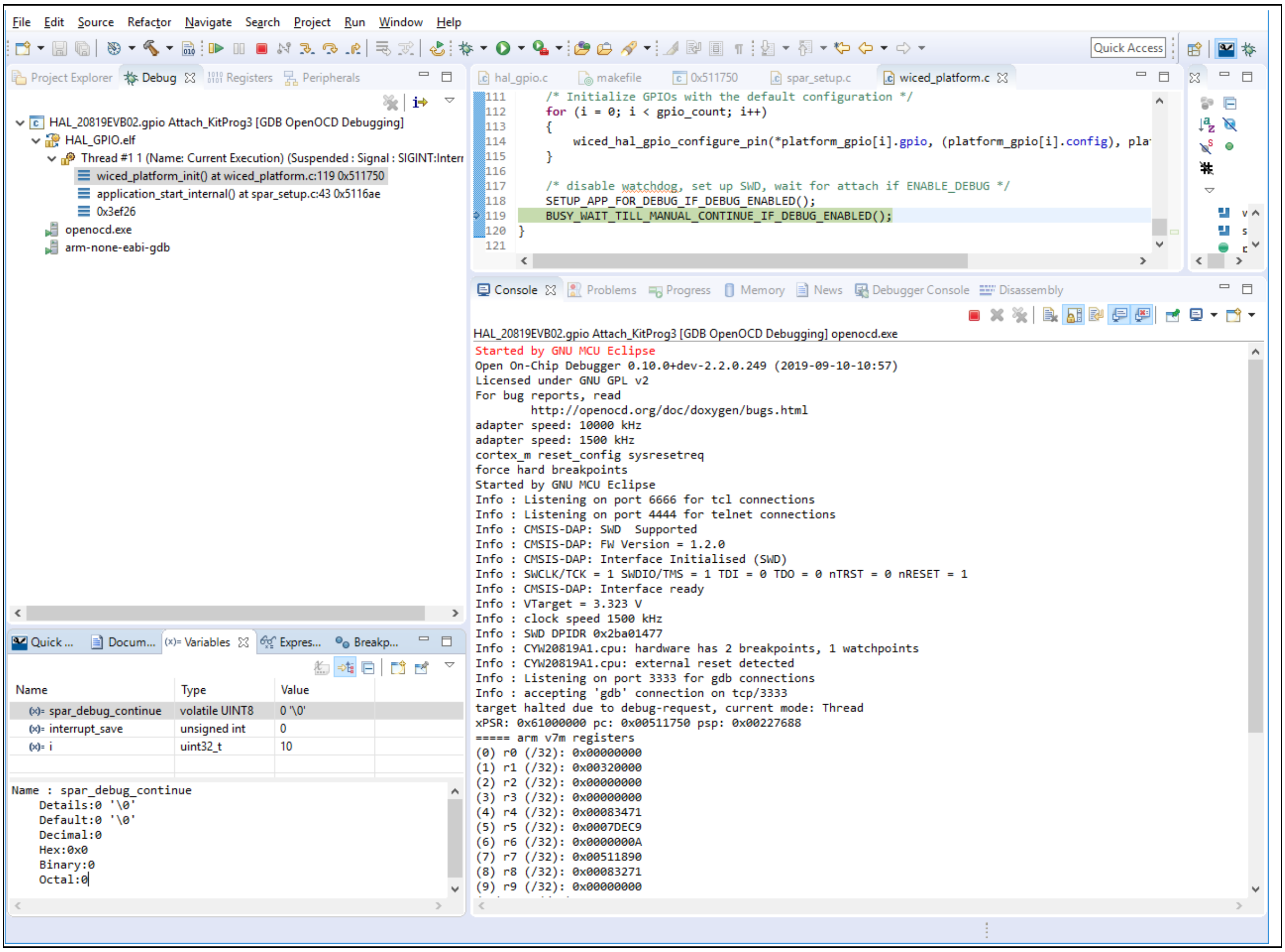


Figure 12 Debug view after pause using OpenOCD

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 application and dependencies (wiced_btsdk etc) 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.

Table 2

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

5. 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".

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

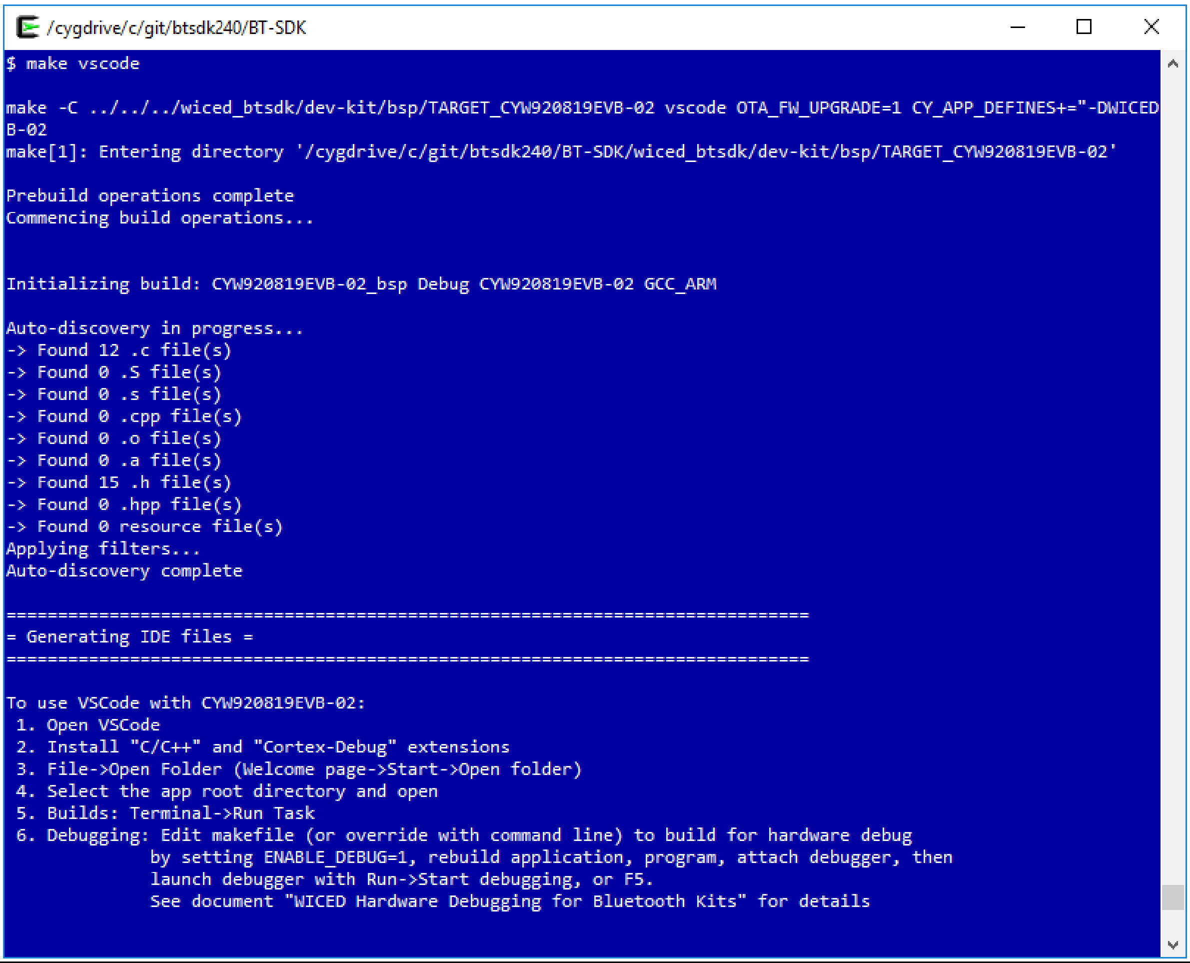


Figure 13 make vscode

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: AIROC™ Bluetooth® SoC devices boot from the 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 such devices always requires attaching the debugger to a running process.*

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

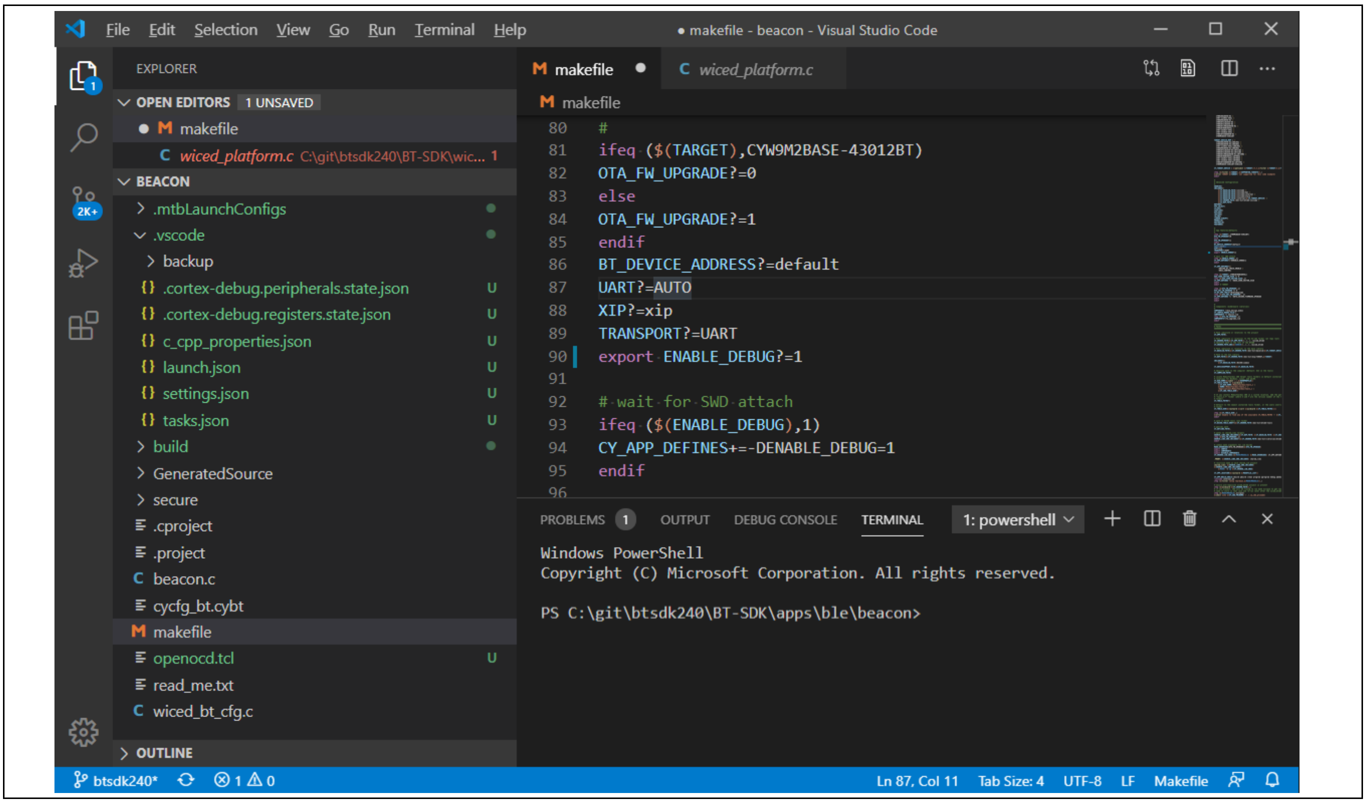


Figure 14 VS Code, application opened

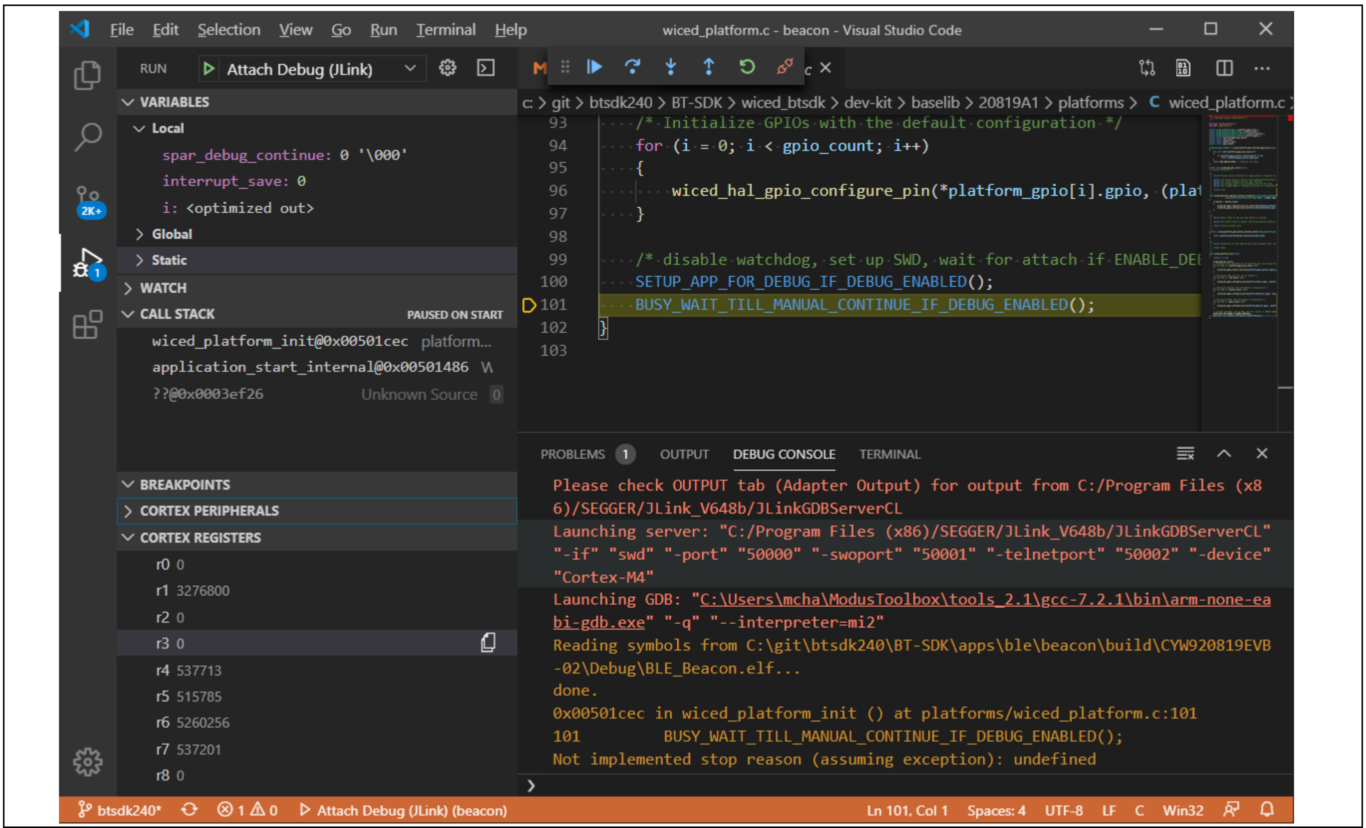


Figure 15 VS Code J-Link debug session

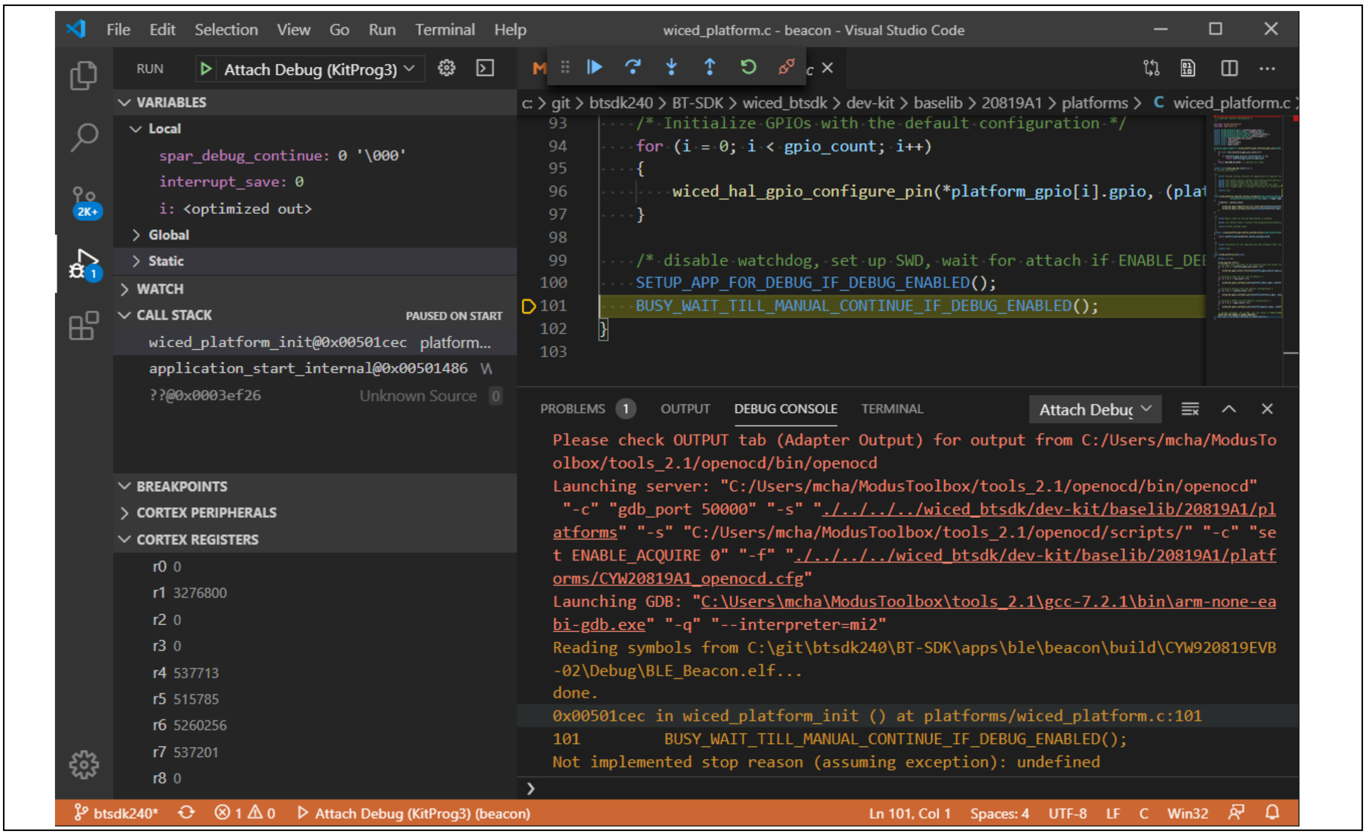


Figure 16 **VS Code OpenOCD debug session**

5.5 Hardware debugging from the command line

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.

1. Launch the GDB Server from the command line.

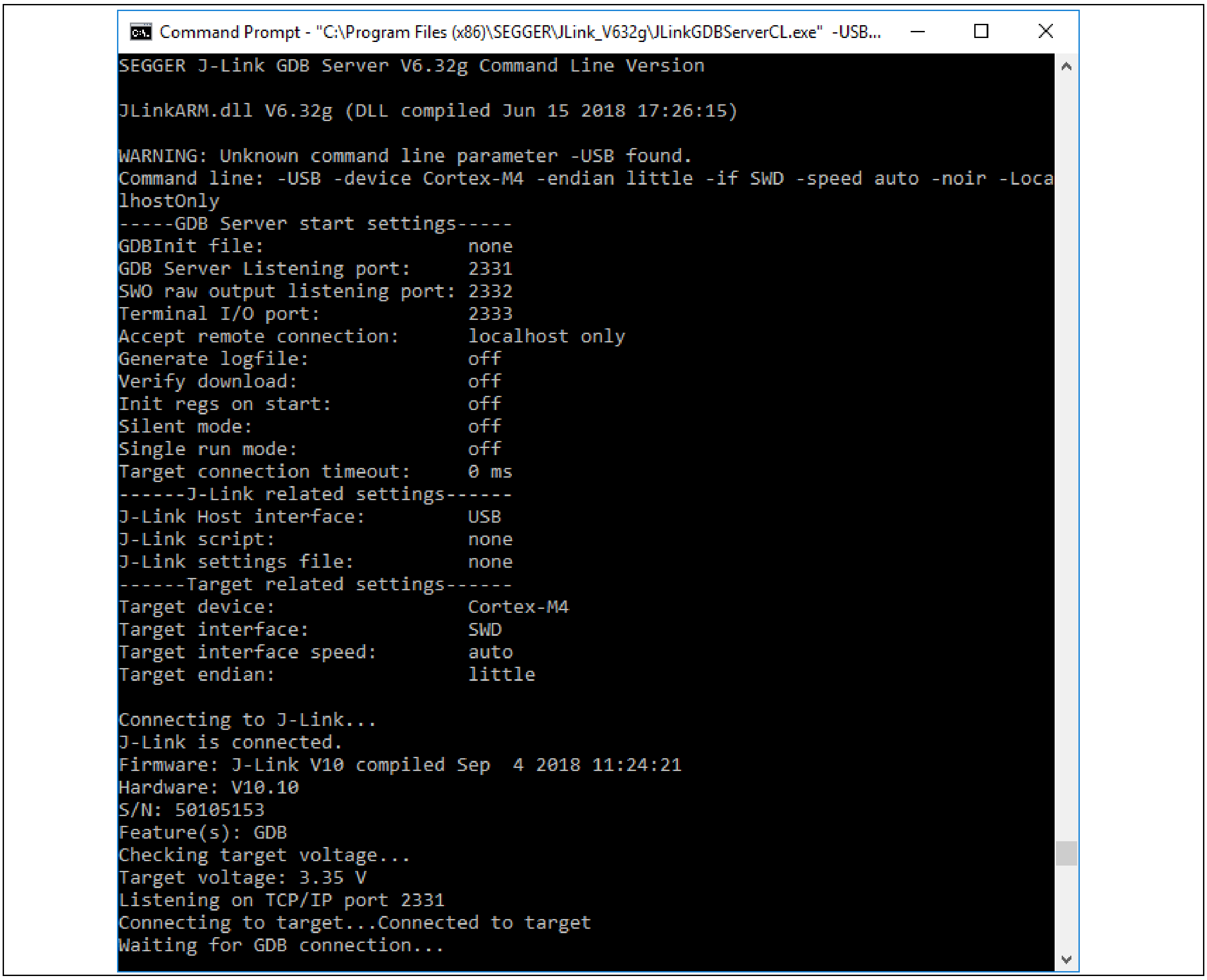
   To launch the OpenOCD GDB Server from the command line, use `make debug` to perform a rebuild of the application image if needed, or use `make qdebug` to launch directly. The Makefile recipe will perform the command documented in Section 5.1 of this document.

   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
   ```



2. Run the GNU GDB command-line application in a separate console. You can perform this from the application directory by using `make attach`. The recipe for this Makefile target will launch GDB using a command line like: *<install-dir>/tools_<X.Y>/ gcc-<version>/bin/arm-none-eabi-gdb.exe*.

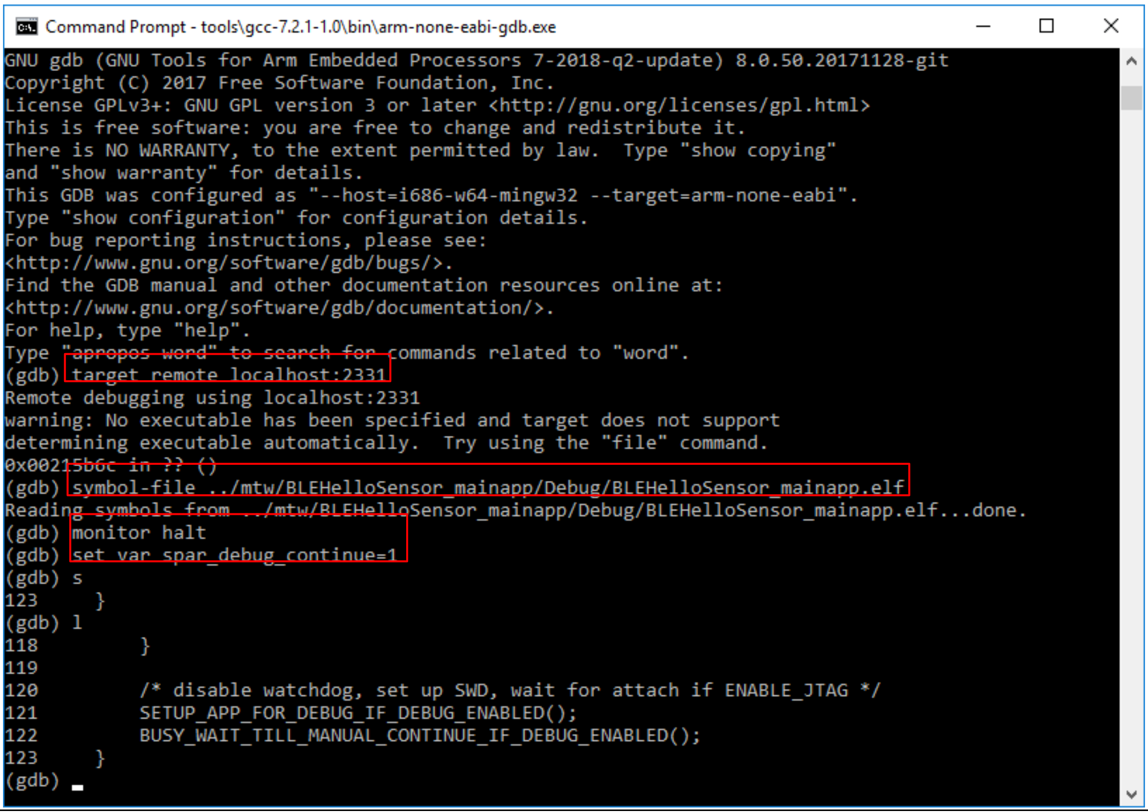
   GNU GDB is an interactive command-line application. The following steps are demonstrated in the figure below.

3. 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.

4. Load the symbol file for the application with *symbol-file <application *.elf>*. Use `monitor halt` to halt the embedded application and `l` (as in *list*) to list the source code at that location.

5. 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 CYWxxxxx 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.

Revision history

Document version	Date of release	Description of changes
**	2017-08-23	Initial release.
*A	2018-11-02	Updated for ModusToolbox™.
*B	2019-02-15	Updated title. Changed ENABLE_JTAG setting to ENABLE_DEBUG. Added references to Linux and macOS. Added description for CYW208xx. Added troubleshooting section.
*C	2019-04-23	Removed Associated Part Family.
*D	2019-06-11	Updates throughout the document.
*E	2019-09-26	Updated for ModusToolbox™ 2.0 and MiniProg4.
*F	2019-10-15	Added troubleshooting note.
*G	2019-11-20	Added additional description for command line debugging and a troubleshooting note.
*H	2020-04-08	Added support for VS Code.
*I	2020-05-08	Added troubleshooting for Eclipse Debug launch.
*J	2021-11-18	Branding updates, added CYW307xx family.
*K	2022-01-13	Added CYW43012C0 to support list and description of hardware debugging for kits with KitProg3 bridge.
*L	2022-03-02	Updated hyperlinks across the document.

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

Edition 2022-03-02

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about this document?

Go to www.infineon.com/support

Document reference

002-20504 Rev. *L