Debugging CircuitPython On SAMD w/ Atmel Studio 7

Created by Michael Schroeder

https://learn.adafruit.com/circuitpython-samd-debugging-w-atmel-studio-7

Last updated on 2023-08-29 03:42:38 PM EDT
# Table of Contents

## Overview
- Understanding CircuitPython & Debugging
- Parts Used in this Guide

## Get Ready, To Get Ready
- Building CircuitPython Firmware
- Segger J-Link
- Atmel Studio 7

## Create AS7 Project
- Open Object File to Create An AS7 Project
- Selecting the Target MCU

## Setup J-Link & GDB
- In the Tool section:
- In the Advanced section:

## Can We Debug Yet?
- Ladies and Gentlemen, START YOUR [DEBUG] ENGINES!!!
- Debug Ribbon Icons

## Break. Point.

## Enhance
- Breakpoint Labels
- Breakpoint Settings
- Address:
- Conditions & Actions:

## Using New Firmware
- Running New Firmware With An Existing AS7 Project

## Code + Community
- Share it!

## Troubleshooting

## Oops. I Erased The Entire Chip.

## Device Not Halted

## Extra Credit

## Reading Peripheral Registers
- Patience Is Key
Overview

This guide will help you to start debugging CircuitPython on the SAMD21 or SAMD51, using Windows 10 and Atmel Studio 7. Yes, it can be done. All that is needed is some jumping on one foot, while juggling chainsaws and singing System Of a Down's Chop Suey. Kidding. Mostly.

We'll discuss using the compiled CircuitPython firmware for Atmel Studio project creation and setup. Then we'll delve into setting and maintaining breakpoints, then displaying information from those breakpoints to diagnose problems (or just for "gee-whiz"). Finally we'll cover some tips and troubleshooting steps.

Understanding CircuitPython & Debugging

This process is somewhat technical, but not difficult. It does require some basis of knowledge with regards to how CircuitPython and debugging work. There are a few guides that will help give a more complete understanding of that basis of knowledge, if needed. I decided not to duplicate all of the information here, since it's explained so well already. The guides you may wish to refer to are:

- Welcome to CircuitPython! ()
- CircuitPython Essentials ()
- Debugging the SAMD21 with GDB ()
  (I never got OpenOCD to work with Windows 10, which led to this AS7 solution)
- Proper Debugging of ATSAMD21 Processors ()
  (Arduino-based, but has loads of useful info)

This list is not all-inclusive, and there are a few others placed elsewhere in the guide.
Parts Used in this Guide

You will need one hardware debugger. The Segger J-Link Pro, Basic, EDU, and EDUmini are all compatible, you should choose to get a version according to Segger’s license terms & budget, which is explained on the product page for each device.

1 x SEGGER J-Link EDU Mini - JTAG/SWD Debugger  
Hardware USB debug stick  
https://www.adafruit.com/product/3571

1 x 10-pin 2x5 Socket-Socket 1.27mm IDC (SWD) Cable  
150mm long  
https://www.adafruit.com/product/1675

1 x SWD Cable Breakout Board  
2x5 1.27mm  
https://www.adafruit.com/product/2743

1 x SEGGER J-Link EDU - JTAG/SWD Debugger  
Identical to the J-Link BASE model except for the terms of use.  
https://www.adafruit.com/product/1369

1 x SEGGER J-Link BASE - JTAG/SWD Debugger  
Identical to the J-Link EDU model except for the terms of use  
https://www.adafruit.com/product/2209

1 x JTAG (2x10 2.54mm) to SWD (2x5 1.27mm) Cable Adapter Board  
Large to small debug cable adapter  
https://www.adafruit.com/product/2094

Get Ready, To Get Ready

Building CircuitPython Firmware

First, you must be able to build the firmware on your local machine. If you have not already done this, the Building CircuitPython () Learn Guide will get you up and running.

For any meaningful debugging results, you’ll need to include `DEBUG=1` in your make command. For example: `make clean BOARD=itsybitsy_m0_express; make BOARD=itsybitsy_m0_express DEBUG=1`
Non-Express boards may complain that the resulting compilation will not fit on the board; there are a couple workarounds, so contact one of the CircuitPython helpers on Discord or in the Adafruit CircuitPython forum.

Note: "make clean BOARD=\" is a good habit to get into. It ensures that the cross-compiler builds everything from scratch. However, it does add time to the compilation.

Segger J-Link

You'll need a debugging adapter to allow the computer to communicate with the target board. I, and by extension this guide, use a Segger J-Link debugger and their J-Link software and drivers. I'm sure you could use almost any SWD compliant debugger, but that will not be covered in this guide.

The Segger J-Link Pro, Basic, EDU, and EDUmini are all compatible (choose your version according to Segger's license terms & budget). Additionally, you'll need to be able to connect your SAMD board to the J-Link. This is easiest using one (or both) of the following Adafruit breakouts and a 2x5 ribbon cable:

- **JTAG (2x10 2.54mm) to SWD (2x5 1.27mm) Cable Adapter Board**
  (For use with J-Link Pro/Basic/EDU with the 2x10 connector)
- **SWD (2x5 1.27mm) Cable Breakout Board**
  (For use with any 2x5 connection and a breadboard)
- **10-pin 2x5 Socket-Socket 1.27mm IDC (SWD) Cable - 150mm long**
  (None of the debuggers come with a 2x5 cable)

Refer to your target board's pinout to locate the SWDIO & SWCLK pins.

Atmel Studio 7

©Adafruit Industries
Lastly, you'll need Atmel Studio 7 ("AS7"). As luck would have it, Microchip provides it for FREE (at time of writing). Download is available from Microchip's website.

Download Atmel Studio 7

Understand this before moving forward: Atmel Studio can be buggy. You may experience "Not Responding" occurrences and crashes. Nothing new for us Windows users, right? Having said that, I haven't had any experiences of lost or corrupted data. While rarely needed, each board I have debugged has always been recoverable by re-programming when things got screwy. We'll cover that in the guide, as well.

Which brings me to one more thing I will suggest you have before starting: a local copy of the bootloader binary (*.bin) for the board you are debugging. They can be found in the uf2-samd GitHub release page.

https://github.com/adafruit/uf2-samd21/releases

If you've made it this far, you're ready for the joys of debugging! So, lets open Atmel Studio and get ready to debug.

Create AS7 Project

Open Object File to Create An AS7 Project

Unlike using the Arduino approach (referenced in the Overview page), CircuitPython's toolchain is not integrated with AS7. This means that we can't just open up the raw code in Atmel Studio, set breakpoints, then have Atmel Studio compile and program the target board with CircuitPython firmware. We're in luck though.

Atmel Studio has the ability to open "object files" as a project and program/debug a device with it. These are also known as ELF (Executable and Linkable Format) Files, "a common standard file format for executable files, object code, shared libraries, and core dumps" (Wikipedia ()).

With your firmware built, open AS7 and in the File menu, select Open -> Open Object File For Debugging.

The following dialog window will open:
Fill in the dialog window:

- Select The Object File To Debug: type the path, or use the file selector, to "firmware.elf" in your build folder ("circuitpython/ports/atmel-samd/build_board_name/")
- Project Name: type a project name
- Location: location to store the project files (does not move/copy your firmware files)
- Maintain Folder heirarchy for source files & Add File As Link: leave these checked (or don't; your choice)

Here is how one of mine looks:
Once you’re satisfied with everything, click Next >

Selecting the Target MCU

Select the Target MCU you are using. The list can be filtered by clicking the Device Family dropdown.

- For M0 boards, select "SAMD21" in Device Family. Then select the appropriate chip name (ATSAMD21G18 is most common; ATSAMD21E18 for Trinket/Gemma).

- For M4 boards, select "SAMD51" in Device Family. Then select the appropriate chip name (ATSAMD51J19 for Metro M4; other boards unreleased at this time).

- For boards released after this guide, the product page for the board should have the full name of the microcontroller it uses listed. Use that chip name in selecting the Device Family.

Click Finish.

Setup J-Link & GDB

Now, we need to set up our new project with the J-Link, and turn on GDB.

Be sure to connect your J-Link to your computer. AS7 requires it to be connected, so that the backagent can query its information.
If the Project Properties window isn't already open, click on the Project->[Project Name] Properties menu or press Alt+F7.

In the Tool section:

First, we'll setup the J-Link programmer settings.

- Select the J-Link you wish to use in the "Selected debugger/programmer" dropdown list. The "Interface" dropdown will automatically populate with "SWD", since it's the only available option for J-Link. A button to open the J-Link Control Panel will also appear.

- The SWD Clock setting will be set to the default. I have found no need to change this.

- In the "Programming Settings" section lies a very, very important setting. You will notice a dropdown list that most likely says "Erase entire chip". If you were programming/debugging firmware that contained its own bootloader, you could leave this. However, CircuitPython's firmware and bootloader are separate, and this setting can cause you much headache. So, change this to "Erase only program area".

When configured, your properties window should look like this:

If you leave the Programming Settings set to "Erase entire chip", you will be left in an unusable state. You can recover from it, but it is an extra step.
In the Advanced section:

Now we'll set up GDB.

All you need to do is make sure the "Use GDB" checkbox is checked. I didn't change any of the GDB Settings, but feel free to explore.

Now save the project. This is another good time to repeat this adage: save often!

The above steps only need to be done once if you continue to use the same project file. I will suggest that you use a different project file for each board you debug, even though you could probably re-map the object files. This project file will still be usable if you make a change to the firmware and re-compile, as explained in the Using New Firmware section.

This guide does not cover laying out your AS7 environment, that's up to personal preference. I like using an environment configured with these windows: Disassembly,
Can We Debug Yet?

So, let's debug! I'm going to explain the different tools and approaches that work for me, using the latest issue I've been debugging as an example: **High Frequency Pulse In**. I mention "work for me" because you may work differently; feel free to explore different approaches.

To start off, I wrote two CircuitPython scripts: one for PWM output from a Feather M0 Express, and the other for PulseIn detection on an ItsyBitsy M0 Express (board being debugged). This is what my screen will look like through most of the process:

Ladies and Gentlemen, START YOUR [DEBUG] ENGINES!!!

To get the target board running, click the "Start Debugging" icon or press the F5 key.

You will only be able to view information while the board is "paused" (exception: any information output from a Breakpoint will be displayed in "real-time" in the Output window).

If you know the information you're interested in, or want to set a breakpoint based on a memory location, you can use the "Start Debugging and Break" icon or press Alt+F5, set your breakpoint(s), and then "Start Debugging".
Debug Ribbon Icons

1. Start Debugging (F5): programs the board, and starts running the board.

2. Start Debugging and Break (Alt+F5): programs the board, starts running the board, but immediately pauses execution.

3. Attach to Target: establishes connection to the target board, does not program the board, and does not start running the board.

4. Stop Debugging (Ctrl+Shift+F5): stops any paused or running board.

5. Break All (Ctrl+F5): pauses a running board at the current memory address.

When you click either Start option, a window will pop up and show progress of each of the 4 steps AS7 takes: Compare, Erase, Program, and Verify. Then the board will start running and/or pause depending on your selection.

You will notice that the board's USB drive ejects and re-connects to Windows. Any connected terminal for REPL will need to be re-connected. If you chose "Start and Break", this will not happen since the board immediately stops and doesn't run the bootloader.

Also, while the board is paused, REPL activities will obviously not work. However, once the board is running again, the terminal will flush and send any commands that were entered while paused (at least PuTTY does).

Pro Tip: Avoid adding/changing files to/on the board during debugging. When you stop the board (or the board hangs), it is NOT properly ejected from Windows and may corrupt the board's filesystem if changes have occurred.

Getting back to the issue I was debugging, I ran the scripts until the board lockup that was described in the GitHub issue occurred. The first place I started to investigate,
was a look at the Call Stack. So, I clicked the "Break All" icon (or Ctrl+F5). Then, opened up the Call Stack window:

As you can see, the callstack (aka backtrace, stacktrace) shows that we're currently in the `pulsein_interrupt_handler()` function.

Since the High Frequency PulseIn issue was resulting in a "board lockup", it seemed logical that we were getting stuck in a loop somewhere. I decided to set a breakpoint on this function and watch the output a few times.

Now that we have a clue, it's time to get into the weeds...

Break. Point.

[insert your favorite tennis or Keanu Reeves pun here]
Breakpoints can be considered the sorcerers of debugging. Want to know if a function is giving the correct result of 42? Set a breakpoint, and all will be revealed.

There are a few different ways to set a breakpoint. In the previously mentioned situation, where the board was paused and the function we want to set a breakpoint for is in the callstack, it's as simple as right-clicking on the function in the Call Stack window and adding the breakpoint in the Breakpoint sub-menu:

![Breakpoint Setting](image)

However, if you need to find a function not in the callstack there are a few options.

The easiest option is to try using the New->Function Breakpoint in the Breakpoints window. Simply type the name of the function you want to set a breakpoint for in the "Function Name" textbox, and set any conditions and actions (explained below). This method can even be used while the board is running. There is a minor downside to this method, which is detailed in the Using New Firmware section.

©Adafruit Industries
Another option is the Disassembly window. In the Disassembly window, you can type a function name in the "Address" combo-box and hit the Enter key.

You'll usually get lucky and it will highlight the memory address of the function. Then you can do a simple right-click and add the breakpoint. This method can only be used when the board is paused.

The last option, is the GDB Console window. I have found a few function addresses here, that weren't found in the Disassembly window.

In the GDB Command textbox, type "print" and the function name. Example: \texttt{print SysTick_Handler}

If GDB finds the function, it will print some info including the memory address. Then you can use either option above using the address. This method can only be used when the board is paused.

If none of these options result in a breakpoint, you might be forced to Step Over and/or Step In to find the function in the Call Stack window (have only had to this once on a deeply optimized function, so it's a low-rate occurrence).

Once you've set the breakpoint (as you may have seen, I set two for this PulseIn issue), you can customize it a little. You can do this while the target board is running; no need to pause. You can even turn them on and off while the board is running with the checkbox.
Enhance

Now that we have breakpoints set, we should enhance them a little.

To get to the enhancing options for a breakpoint, simply right-click the breakpoint:

![Breakpoint Menu](image_url)

**Breakpoint Labels**

One thing I have gotten into the habit of doing when setting breakpoints is adding labels. When using any of the breakpoint creation options, other than New->Function Breakpoint, the function names aren't added to the breakpoint.

In the breakpoint menu, select Edit Labels..., put your desired name in the "Type a new label" textbox, and click Add or press Enter.

This helps to keep your breakpoints organized, without having to remember memory addresses. Also, you can use the labels in a different manner, and apply multiple labels to the same breakpoint or one label to multiple breakpoints.
Breakpoint Settings

The functional options are in the Settings… menu.

Address:

The address of the breakpoint. This is useful when a variable output needs to be tweaked (explained below in the Actions section), or if you've re-compiled changes to the firmware and the function's address has changed. If you change the address of an existing breakpoint, AS7 will force you to select a "Language". I have used "C#" with no problems. Also, this is not available when using New->Function Breakpoint since it is replaced with the function name instead of the address.

Conditions & Actions:

Conditions

• You can set conditional expressions for when the breakpoint is triggered. Syntax and gotchas are explained in Actions.

Actions

• This is my personal favorite. You can set the breakpoint to print out information you want to evaluate. You can let the breakpoint pause the board, or let the board keep running by checking "Continue execution". The information is printed to the Output window. This is really helpful in situations where code
execution needs to continue to see a larger set of data or when there are dependent functions/interrupts that need to continue. Now, not all information will be available. You will come across variables that are either optimized out by the compiler, or just not at a scope level that AS7 can evaluate.

To explain the action syntax a little, here is what I know to be available: plain text, "$" macros, and "{???}" variable calls.

For the macros, just type "$" and a list of options will appear. $CALLSTACK and $CALLER are nice for a snapshot w/o stopping the board. $ADDRESS is useful if you have multiple breakpoints on the same function.

For the variable calls, there are a couple things to mention:

- The value (or lack thereof) output is based on a variable's state at the breakpoint's address. If you're getting a lot of could not evaluate results, try moving the breakpoint's address "down" a little. You can also set multiple breakpoints on a single function, assuming it takes up multiple addresses. This will let you watch a variable through the function's cycle.

- Pointers work…most of the time. What I mean is this will work: {last_us[$elf->channel]}. In the case of indexed values, you can also just put the index in if you know it: {last_us[5]}

One last note on Actions. While using "Continue Execution" will keep the board running, it does cause a delay in execution. If you're working with things that require precise timing, you may have to factor that into your approach.

That's about all there is for Breakpoints. Simple, but effective. AS7 does have a "Data Breakpoint" available, but in my early use I couldn't get it to work. I may re-visit it, and update this guide accordingly.

To show all of this in action, here is a screenshot of a PulseIn run:
Using New Firmware

Running New Firmware With An Existing AS7 Project

So the whole point to debugging is to fix things, right? And we would never make a change without testing it for bugs, right?

To continue using the PulseIn issue I've been working, let's assume I've made some code changes and built new firmware to test. Since our firmware builds in the same location everytime for CircuitPython, and our project file is already mapped for that location, why create a new project? Especially when Atmel Studio will recognize that the firmware files have been updated, and give us the option to use the update. However, it isn't a perfect, seamless experience. So here are a few tips:

- Close AS7: I have gotten into the habit of closing Atmel Studio when ever I'm building new firmware with changes. The not-as-smooth-as-it-could-be process goes a little smoother. So, I suggest closing AS7 (or at least the project), building the new firmware, then re-opening AS7 and/or project. Using the available "Recent" open commands cuts down on a couple clicks.

- Wait: AS7 is a LARGE program (Visual Studio framework, after all). It takes a while for it to scan and load all of the project files (YMMV if your PC has worse/better specs than mine). Don't immediately click "Start Debugging...". You can have the Solution Explorer window open to watch the status: it will change from "Loading" to "1 project" when it is complete.
- **Reload**: Once AS7 finishes loading the updated firmware files, you will be greeted by the following message. Click "Reload".

![Reload Window](image)

- **Remap**: The following window will eventually pop up. Click "Finish" to confirm that all the files have been reloaded. (Note: the "Missing" message is misleading; scroll through the actual list to find any errors)

![Remap Window](image)

- **Choose The Right Option**: Once AS7 finishes remapping the updated firmware files, you will be greeted by one of those scary message boxes with too much text and too many buttons. It seems counterintuitive at first, but the option you'll want to select to use the updated firmware is "Discard". After reading the descriptions in the message box a few times, it makes sense. Here it is in all its glory:
Reset The Programming Settings: Remember that "very, very important setting" discussed earlier? Well, AS7 likes to forget all about the fact that you changed it. So, before you send any programming commands to the J-Link, make sure you go back to the Project Properties window and change "Erase entire chip" back to "Erase only program area". Then, save. If you miss or forget about this step, fear not; it is recoverable.

UPDATE: Due to bug fixes in an AS7 update, released 6/22/18, the issue above seems to be corrected. However, keep an eye on it.

Verify Breakpoint Addresses: It is most likely that at least some of the functions you have breakpoints set for, are at new address locations. The quickest way I've found to verify them, is to "Start and Break..." the board into a paused state, right-click each breakpoint in the Breakpoints window, and select "Goto Disassembly...". That will highlight the breakpoint address in the Disassembly window. The function addresses are usually blocked together by function, with spaces or filepaths separating any jumps. If the highlighted address appears to be in a different location than it should, just search for the function in the Disassembly or GDB window as described earlier. Then change the breakpoint's address. Another tell-tale sign is if any output actions are giving totally different results (changed to could not evaluate or optimized out) than before.

- Breakpoints set with New->Function Breakpoint: Any of these breakpoints that have moved memory addresses with updated firmware, will have the breakpoint address set to $0x0000$. Simplest way to fix it: change the breakpoint function name (e.g. drop the last character), save, then change it back. Automatic address update. You will need to have the board paused for this.

DON'T ERASE THE ENTIRE CHIP!!!
Now it is once again safe to start debugging. The process is a little tedious, but after a while it just becomes natural. Like hitting the snooze button six times in the morning . . .

Code + Community

We've debugged. We've written awesome code to fix a bug (or bugs). Only one thing left to do before we're done...

Share it!

Although not a requirement, CircuitPython gets better when everyone contributes. That includes you, dear reader.

This guide will help you reach out to the community if you need help getting your changes incorporated: Welcome To The Community ()

Troubleshooting

Things happen. System setups are different. This page is here to help. Its kind of like a superhero...

- Oops. I Erased The Entire Chip ()
- Device Not Halted ()

This will be a work in progress page for a while. If you come across an issue, use the feedback link on the left to send in your issue and we'll see if we can get it added.
Oops. I Erased The Entire Chip.

It happens. And that's ok. It's a pretty easy fix, actually. If you didn't already do so, download a copy of the bootloader binary file (*.bin) for your board:

https://github.com/adafruit/uf2-samd21/releases

Now open the Device Programming window.

With the J-Link tool selected and the Device set to your SAMD chip (should be populated based on your project file), click the Apply button. If everything checks out, the buttons next to "Device signature" and "Target voltage" should enable.

Click the Read button next to "Device signature"
If the device ID reading returns OK, click Memories on the left and the window will look like this:

In the "Flash (256 KB)" section, type/select the path to the bootloader binary file you downloaded earlier.

Ensure the "Erase Flash before programming" checkbox is checked.

Click the Program button.
If everything was successful, close the Device Programming window. Since the flash was erased, anything you had on the board's flash will be gone (firmware, any libraries, etc).

The board should have reset, and connected to Windows in bootloader mode (i.e. FEATHERBOOT, TRINKETBOOT, etc). To keep the variables to a minimum with the file system, I usually drag and drop a firmware .uf2 onto the board before running it in AS7.

Again, resist the urge to manipulate the file system while running the board from AS7. But, now that you know this process, you can recover from it if you do.

---

**Device Not Halted**

Occasionally, you'll run into a problem where AS7 won't connect/run the target device.

Common error messages look like this:

I have yet to solidify what causes this. But, the fix is usually as simple as doing one (or more) of the following:

- Disconnect and reconnect the SWD cable from the target device and the J-Link. Wait a few seconds for the J-Link and the backagent to sync up.

- Close AS7, then re-open your project.

- If those don't work, starting over completely (rarely requires Windows restart) is the ultimate fixer.

- And if nothing is still working, you may need to check the "Fuse" settings in the Device Programming window.
Extra Credit

In case you need some extra points for that test I didn't give you, here are some things that I'll make up as I go along. That's to say, as I...no, as WE discover more fun stuff, it will be dropped here unless it applies elsewhere. Seriously, if you come across some cool black magic that isn't documented here, write it up and/or send it in. Would definitely love to share (and attribute accordingly)!

Each of the following are worth 5 additional non-points for the non-test:

- Reading Peripheral Registers ()

Reading Peripheral Registers

If you're diving deep enough into CircuitPython's core code, you might be dealing with the chip's peripheral registers. Now, while you can get A LOT done with breakpoints and GDB `print` commands, the peripheral registers can largely remain elusive. Which can be a bit of a headache at times.

Using the same PulseIn issue that has been used throughout this guide, I have been working on a replacement function to capture frequencies better. Aptly, it is named `FrequencyIn`. The SAMD chips can use their Timer/Counter ("TC") or Timer/Counter for Control Applications ("TCC") to capture frequency periods (and PWM...but that's for a later date). If you're interested, read the datasheet(s).

Moving on.... In order to use the TC peripheral to capture frequency periods, we have to set up the peripheral which is accomplished through the registers/helper functions. These registers are also used to check if a signal has been captured, and to retrieve the captured result. As mentioned, the elusive part is finding out if what we think is happening is actually happening at the register level.

Enter the IO Window!
Its elegantly simple. You can open the IO Window from the Debug->Windows menu. In order to read the peripheral register values, the board must be paused. The values will be updated automatically when pausing the board.

You can also SET the register values here. I may update this section later to illustrate how to use this, but since this guide is about debugging, reading the registers is beneficial on its own.

Patience Is Key

As mentioned earlier in the guide, Atmel Studio is a large program and can be buggy. One thing I constantly encounter when using the IO Window, is the time it takes to update the peripheral register values. There are a lot of them. So, exercise patience when using the IO Window. You will most likely see intermittent "Not Responding" or "Atmel Studio Is Busy" periods, and this dialog window will open:

If you’re still looking for register values, click Wait 1 more minute. The dialog window may open more than once before you can get the information you seek.
If you're done looking, click Stop waiting, and AS7 will abandon the process it is trying to accomplish.

As best I can tell, this is all related to communication with the debugger and the backagent program.