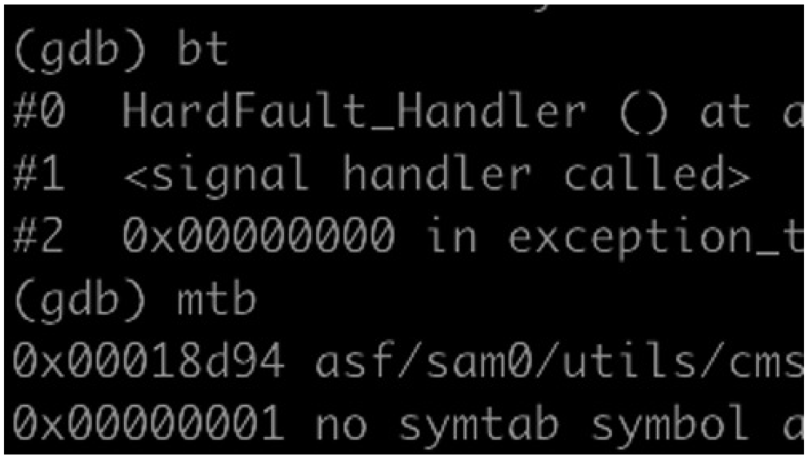
Debugging the SAMD21 with GDB

Created by Scott Shawcroft

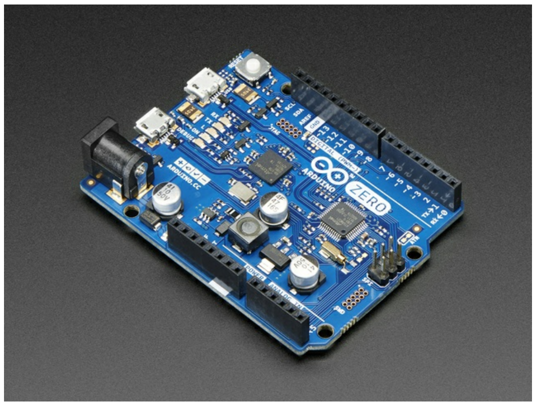


```
(gdb) bt
#0  HardFault_Handler () at a
#1  <signal handler called>
#2  0x00000000 in exception_t
(gdb) mtb
0x00018d94 asf/sam0/utils/cms
0x00000001 no symtab symbol a
```

Last updated on 2019-12-17 04:33:16 AM UTC

Overview

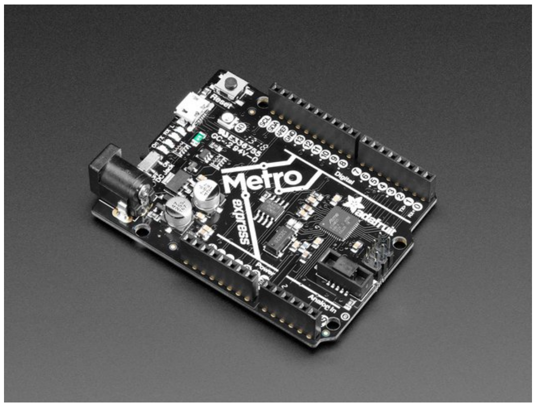
The SAMD21 is a microcontroller developed by Atmel which runs at 48mhz with a Cortex M0+ core. It's used in the Arduino Zero, Metro M0 Express, Feather M0s, Gemma M0 and Trinket M0. Many of our Arduino and CircuitPython libraries use C to interface directly with the hardware and when things go wrong it can be hard to figure out why.



Arduino Zero - 32 bit Cortex M0 Arduino with Debug Interface

OUT OF STOCK

Out Of Stock



Adafruit METRO M0 Express - designed for CircuitPython

$24.95

IN STOCK

Add To Cart

Unlike more common web or server software development the code isn't being run on the same machine as its written on. When code is compiled for a different architecture, such as ARM for the SAMD21, than the architecture the compiler is running on, usually x86, its called cross compiling. There are many different ways to organize how your code compiles we can cover in another guide. What you need to know is that there are a few different toolchains that convert human readable code to machine code. Today we'll focus on the GNU toolchain that is made up of the GNU Compiler Collection (better known as `gcc`), the GNU Debugger (`gdb`) and number of other programs. Since the GNU toolchain is open source, ARM, the designer of the Cortex M0+ core, is able to ensure it produces well working code for their cores. So, we'll be using the ARM version of the GNU toolchain to compile and debug the C code.

While I won't cover compiling the code, I'll be using the CircuitPython in the examples. Its source is here and it uses `make` to run the compiler. I also use an Arduino Zero for all of my debugging because it has a builtin debug chip which converts USB commands to commands the Cortex M0+ core understands.

There are a couple aspects of debugging we'll talk about. First we'll cover a few ways to stop the program at an interesting spot. Second, we'll cover how to inspect the current program state using `backtrace`. Lastly, using the Micro Trace Buffer, we'll inspect the history of the program execution. Lets get setup.

Software Installation

There are two pieces of software we need to install in order to get debugging. First is a GDB server. When using the Arduino Zero, you'll need OpenOCD. OpenOCD is a tool to communicate with debug hardware tools such as the EDBG chip on the Arduino Zero. When using a JLink debugger, you'll use the JLink GDB server. GDB is the GNU Debugger which talks with OpenOCD to control and inspect the raw state of the microcontroller and, using the binary symbols, translate that info back into the source code realm.

GDB Server

Below we'll cover OpenOCD to work with the Arduino Zero. See here for more information on installing the JLink GDB Server.

OpenOCD

You can get OpenOCD from here. Below are quick start instructions.

Windows

Download and install the latest binary from here.

Mac OSX

Installation on Mac OSX is easiest using Homebrew. (If you don't have brew installed see here.)

```
brew install open-ocd
```

Linux

For Ubuntu, install the ARM toolchain using the instructions in the Building CircuitPython Learn Guide. Other Linux distributions may have packages available. Here are some examples.

```
# Ubuntu, Debian, Raspbian, Mint
sudo apt-get install openocd # http://packages.ubuntu.com/search?keywords=openocd&searchon=names
# Fedora
su -c 'yum install openocd' # https://apps.fedoraproject.org/packages/openocd
# Arch
pacman -S openocd # https://www.archlinux.org/packages/community/x86_64/openocd/
```

GDB

GDB is part of the larger ARM toolchain. So don't be surprised to see a number of arm-none-eabi-* binaries installed.

Windows

Download install the win32 executable from here.

Also make sure that you have 32-bit (x86 not x86-64) Python 2.7x installed from here. We'll be using Python to interpret the Micro Trace Buffer later.

Mac OSX

Installation on Mac OSX is easiest using Homebrew. (If you don't have brew installed see here.)

```
brew cask install gcc-arm-embedded
```

Linux

To install the arm gcc toolchain on Ubuntu following the instructions in the Building CircuitPython Learng Guide. Other distributions may have the toolchain available as a package, but check the Building Python Learn Guide for the correct version.

```
# Fedora
su -c 'yum install arm-none-eabi-gdb' # https://apps.fedoraproject.org/packages/arm-none-eabi-gdb
# Arch
pacman -S arm-none-eabi-gdb # https://www.archlinux.org/packages/community/i686/arm-none-eabi-gdb/
```

Setup

Before we can get into the nitty gritty of debugging we need to first get everything running.

GDB Server

First, we'll get the link between our debug hardware and our computer running. Its called the GDB Server.

OpenOCD + Arduino Zero

First, we need to get OpenOCD going to bridge from our computer to the hardware debugger. Its easy with the Arduino Zero.

Connect a USB cable from your computer to the DEBUG USB connector on the Arduino Zero. Now, make sure you have the Arduino Zero config file for OpenOCD available here.

Now run OpenOCD in a terminal. It will stay running while we debug.

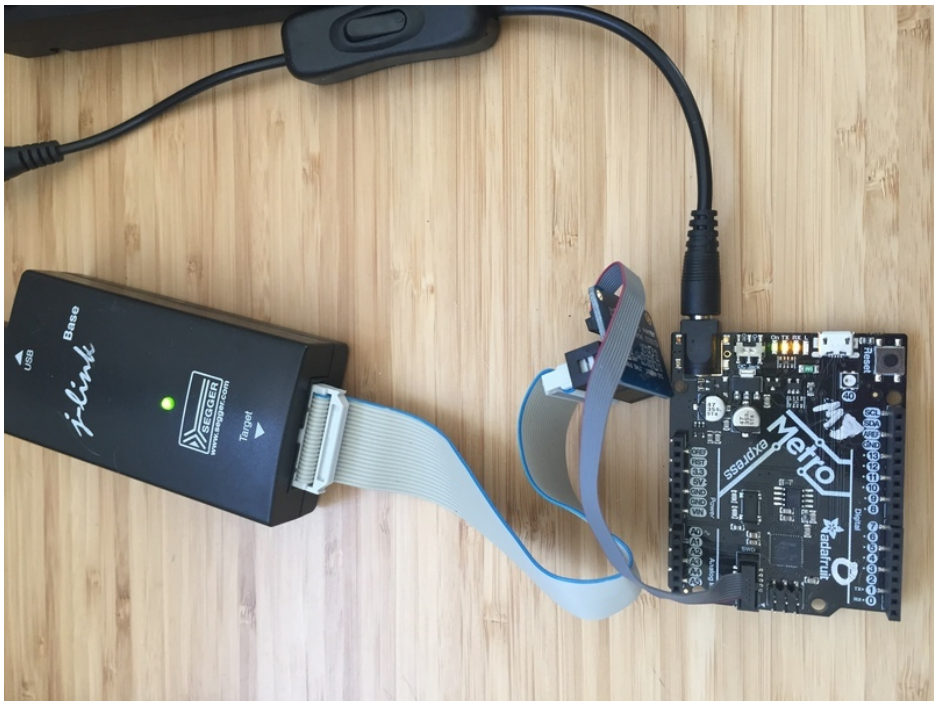
```
openocd -f arduino_zero.cfg
```

You should see that it found the Arduino Zero with output similar to this:

```
Info : CMSIS-DAP: SWD  Supported
Info : CMSIS-DAP: Interface Initialised (SWD)
Info : CMSIS-DAP: FW Version = 02.01.0157
Info : SWCLK/TCK = 1 SWDIO/TMS = 1 TDI = 1 TDO = 1 nTRST = 0 nRESET = 1
Info : CMSIS-DAP: Interface ready
Info : clock speed 500 kHz
Info : SWD IDCODE 0x0bc11477
Info : at91samd21g18.cpu: hardware has 4 breakpoints, 2 watchpoints
```

JLink + Metro M0 Express

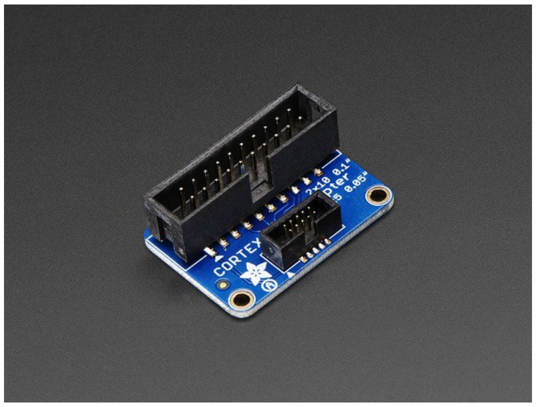
Unlike the Arduino Zero, the Metro M0 Express doesn't have a builtin debug -> USB adapter. To do this conversion you'll need a debugger. Segger's JLinks are the gold standard for debuggers and support many many microcontrollers. (EDU versions are much cheaper for non-commercial use.) To connect the normal sized JLink debugger you'll need an adapter from a JTAG cable to a SWD cable along with a SWD cable.



SEGGER J-Link BASE - JTAG/SWD Debugger

OUT OF STOCK

Out Of Stock



JTAG (2x10 2.54mm) to SWD (2x5 1.27mm) Cable Adapter Board

$4.95

IN STOCK

Add To Cart

10-pin 2x5 Socket-Socket 1.27mm IDC (SWD) Cable - 150mm long

$2.95
IN STOCK

Add To Cart

Now connect the JLink to the Metro express through the adapter board and SWD cable. Once connected, run the JLink GDB server in a terminal.

```
JLinkGDBServer -if SWD -device ATSAMD21G18
```

Most boards will be the ATSAMD21**G**18 except the Trinket M0 and Gemma M0 which are ATSAMD21**E**18 (meaning they are physically smaller).

After connecting you should see something like:

```
SEGGER J-Link GDB Server V6.12e Command Line Version

JLinkARM.dll V6.12e (DLL compiled Jan  6 2017 17:21:41)

-----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:      yes
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:                 ATSAMD21G18
Target interface:              SWD
Target interface speed:        1000kHz
Target endian:                 little

Connecting to J-Link...
J-Link is connected.
Firmware: J-Link V10 compiled Dec 23 2016 12:00:00
Hardware: V10.10
S/N: 50103114
Feature(s): GDB
Checking target voltage...
Target voltage: 3.29 V
Listening on TCP/IP port 2331
Connecting to target...Connected to target
```

GDB

GDB is similarly straightforward. The most important thing is that your current directory is near your binary. With Adafruit's CircuitPython I like to be in the `atmel-samd` directory where our binary is `build-arduino_zero/firmware.elf`. (If you are following along with CircuitPython you can compile it with `make BOARD=arduino_zero DEBUG=1`.)

```
arm-none-eabi-gdb-py build-arduino_zero/firmware.elf
```

Now you should see some version information and a prompt that start with `(gdb)`. All examples that start with (gdb) should be run in gdb and you do not need to type (gdb) in.

OpenOCD

Now we need to tell GDB to debug through OpenOCD rather than on this computer.

```
(gdb) target extended-remote :3333
```

JLink

Now to need to tell GDB to debug through JLink rather than on this computer.

```
(gdb) target extended-remote :2331
```

Loading, Resetting and Running

Loading, resetting and running the currently running program on the microcontroller is critical to the debugging process. To load a new version of the program after you've compiled outside of `gdb` do:

```
(gdb) load
Loading section .text, size 0x2bb84 lma 0x0
Loading section .data, size 0x5a4 lma 0x2bb84
Start address 0x0, load size 180520
Transfer rate: 5 KB/sec, 13886 bytes/write.
```

OpenOCD

To reset the microcontroller to the start of the new program you need to ask OpenOCD via `monitor` to reset to the initialization state.

```
(gdb) monitor reset init
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x00018dd0 msp: 0x20008000
```

JLink

To reset the microcontroller to the start of the new program you need to ask JLink via `monitor` to reset to the initialization state.

```
(gdb) monitor reset
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x00018dd0 msp: 0x20008000
```

Running

Finally, to make the program run type `continue` or `c` and hit enter. The prompt won't return until your program finishes, hits a breakpoint or you type ctrl-c.

Breakpoints

Its great if your program runs as expected but what if it doesn't? It may crash or return an error when you least expect. Breakpoints are a great way to stop the execution of your code so you can inspect how you got a certain place in your code and with what state. There are two primary ways I use breakpoints:

1. to stop when a function is called.
2. to stop at a particular line of code.

Once stopped, we can use `backtrace` and the Micro Trace Buffer to see our current state and how we got there.

If the steps below don't work, make sure you compiled your binary with symbols. (-ggdb for gcc)

Stopping

To add a breakpoint where you want it to stop use `break`. For example, to break at the function `mp_hal_stdin_rx_chr` in CircuitPython you would type:

```
(gdb) break mp_hal_stdin_rx_chr
Breakpoint 1 at 0x1730c: file mphalport.c, line 141.
```

If you want to break close to a particular source code line you can give the filename and line number instead. For example, with file `mphalport.c` and line `150` it'd be:

```
(gdb) break mphalport.c:150
Breakpoint 2 at 0x1739a: file mphalport.c, line 150.
```

Breaking by line can get fuzzy if you enabled optimizations while compiling because the compiler may rework the underlying machine code for efficiency and inadvertantly mix code for different source lines. With GCC, you can make sure and compile with `-O0` (letter O and zero) to disable optimizations.

Once you add a breakpoint the program will be stopped there everytime its reached. To view the existing breakpoints do:

```
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x0001730c in mp_hal_stdin_rx_chr at mphalport.c:141
2       breakpoint     keep y   0x0001739a in mp_hal_stdin_rx_chr at mphalport.c:150
```

To delete a single breakpoint, such as `1`:

```
(gdb) delete 1
```

To delete all breakpoints:

```
(gdb) delete
```

Continuing

We'll cover how to inspect state in just a bit but lets just reiterate how to get back going. As before, if you want to load a new version (hopefully fixed) you can do:

```
(gdb) load
(gdb) monitor reset init
(gdb) continue
```

(No `init` with JLink.)

If you just want to keep going you can simply do:

```
(gdb) continue
```

Additionally if you just want to go another little bit you can do:

- `step` executes one more source code line and goes into functions
- `next` executes one more source code line but skips going into functions
- `finish` executes until the function finishes

More info about control is here.

Backtrace

Backtrace is the most common way to understand where a program currently is stopped. Also known as a stack trace, backtrace crawls up the stack in memory to output the current function heirarchy.

Here is an example backtrace. The #0 is the function that contains the currently executing code and #1 is the function that contains the #0 function and so forth up to #3 which is the top-level main function.

```
(gdb) backtrace
#0  mp_hal_stdin_rx_chr () at mphalport.c:144
#1  0x0001bf6a in readline (line=line@entry=0x20007fc8, prompt=prompt@entry=0x2aeca "&gt;&gt;&gt; ")
at ../lib/mp-readline/readline.c:425
#2  0x0001b8d0 in pyexec_friendly_repl () at ../lib/utils/pyexec.c:412
#3  0x0001641a in main (argc=, argv=) at main.c:206
```

Not only does the backtrace include the names of the functions but it also includes the value of the arguments. For example, the backtrace above shows the prompt argument of readline was ">>> ", just like a Python prompt.

This is really cool but you can do even more. You can also see the state of global and local variables using `print`.

Printing

Globals

Printing globals is easy because they are accessible from anywhere. So, simply do:

```
(gdb) print usb_rx_count
$1 = 0 '\000'
```

This means the value of `usb_rx_count` is `0`.

Locals

Printing locals is a little bit tricky because they are limited by scope. Lets see how we can print`prompt` from within the function. This takes two steps, first we must switch frames to the function that has `prompt` in the backtrace its #1 and then we can print `prompt`.

```
(gdb) frame 1
#1  0x0001c216 in readline (line=line@entry=0x20007fc8, prompt=prompt@entry=0x2b20a "&gt;&gt;&gt; ")
at ../lib/mp-readline/readline.c:425
425             int c = mp_hal_stdin_rx_chr();
(gdb) print prompt
$2 = 0x2b20a "&gt;&gt;&gt; "
```

Awesome! So now we can understand what function(s) we're in and the current local and global state.

Micro Trace Buffer

Backtraces are a great tool for understanding where you are stopped in the program and the current state. But, what happens when our backtrace looks like this?

```
(gdb) backtrace
#0  HardFault_Handler () at asf/sam0/utils/cmsis/samd21/source/gcc/startup_samd21.c:284
#1
#2  0x00000000 in exception_table ()
```

We obviously got where we are somehow but its not clear from the backtrace. Well, the Micro Trace Buffer is a way to sort out the history of the program rather than just the current state.

Unlike a backtrace, which shows the active function and its parents, the Micro Trace Buffer (MTB) records a history of what code was executed. The log it keeps is a circular buffer which always keeps a fixed amount of history. Once it is full, the oldest entry is overwritten with the newest entry and so on.

The MTB keeps the program counter history. The program counter keeps track of which assembly instruction is currently being executed. In basic procedural code flow such as the assembly for `x = 0` code flows sequentially and the program counter is simply incremented. However, the program counter may change non-sequentially when a jump is made because of control flow statements such as if statements and loops. The MTB on the Cortex M0+ only records a packet when a jump is made because between jumps the code is sequential. When a jump is made it records both the program counter of where it is and the program counter of where its jumping to.

This raw program counter history is difficult to understand as a flat array so I've created a GDB plugin to reinterpret the program counter into the line of code that was run instead. This isn't a perfect approach though because a line of code may generate more than one assembly instruction and the compiler may also reorganize lines during optimization. But, its still better than assembly.

Code changes

Before we talk about the GDB side we need to initialize the MTB in our code itself. (I've already added it to CircuitPython for debug builds.)

First, create a global variable to hold the trace buffer in RAM. It must be called `mtb` to work with the GDB command. Notice it must also be aligned to the size of the MTB because the buffer wraps based on the lower bits of the address.

```
#define TRACE_BUFFER_SIZE 256
__attribute__((__aligned__(TRACE_BUFFER_SIZE * sizeof(uint32_t)))) uint32_t mtb[TRACE_BUFFER_SIZE];
```

Next, to turn on the MTB we must configure the starting position and flow policy to act as a circular buffer over `mtb`. The last configuration step configures the size of the buffer in bytes and enables the tracing. I recommend placing this code at the start of your `main` method so that you can read the MTB at any point after that.

```
REG_MTB_POSITION = ((uint32_t) (mtb - REG_MTB_BASE)) &amp; 0xFFFFFFF8;
REG_MTB_FLOW = ((uint32_t) mtb + TRACE_BUFFER_SIZE * sizeof(uint32_t)) &amp; 0xFFFFFFF8;
REG_MTB_MASTER = 0x80000000 + 6;
```

One thing to be aware of is that any loop will add many entries to the MTB. Empty infinite loops are even worse because they will fill up the entire buffer with nearly identical PCs! Infinite loops like this are popular in fault handlers when execution has effectively come to a hault. So, to preserve the MTB in fault handlers you should turn the tracing off before the empty infinite loop. Here is our basic HardFault_Handler:

```
void HardFault_Handler(void)
{
    // Turn off the micro trace buffer so we don't fill it up in the infinite
    // loop below.
    REG_MTB_MASTER = 0x00000000 + 6;
    while(true) {}
}
```

Installing GDB Helper

Now, once the trace is being written to RAM you can read it back with GDB. You can do this manually by printing the `mtb` variable but its more helpful to see the trace history by line of code.

First, download and source micro-trace-buffer.py into your gdb. Make sure you are using gdb with python enabled. In the arm toolchain use `arm-none-eabi-gdb-py` instead of `arm-none-eabi-gdb`.

```
(gdb) source micro-trace-buffer.py
```

Use

Now, during a breakpoint you can run `micro-trace-buffer` or `mtb` for short to get the history from newest to oldest. (If it paginates you can type `q` then enter to quit early.) Here is an example:

```
(gdb) mtb
0x00018d94 asf/sam0/utils/cmsis/samd21/source/gcc/startup_samd21.c 282 1 times
0x00000001 no symtab symbol and line for , line 0
0x00000000 no symtab symbol and line for , line 0
0x00017ac4 asf/common/services/storage/ctrl_access/ctrl_access.c 447 1 times
0x00017aba asf/common/services/storage/ctrl_access/ctrl_access.c 437 1 times
0x00018574 asf/common/services/usb/class/msc/device/udi_msc.c 806 1 times
0x00018562 asf/common/services/usb/class/msc/device/udi_msc.c 794 1 times
0x00018394 asf/common/services/usb/class/msc/device/udi_msc.c 609 1 times
0x00018386 asf/common/services/usb/class/msc/device/udi_msc.c 594 2 times
```

The first number is the last pc to occur at that line in the log. The second part is the source file name. The third part is the source line number. The last part is the number of times the line occurs in the log. This is useful to condense tight loops.

Also, beware that this log only includes the pcs around jumps. If you see neighboring entries with the same file then the code between lines may have been run rather than jumped over.

So, going back to our example from the backtrace section, we can see that we jumped to a null function (pc 0x000000) from `asf/common/services/storage/ctrl_access/ctrl_access.c 447`. Bingo! Going there we can see a function call that must be null! Fix that and it works!

Wrap Up

GDB is a powerful tool for debugging SAMD21 boards. You can get really far with simple breakpoints, backtraces and the micro trace buffer. Good luck!