Pyavrocd

A Platform-Agnostic GDB Server

Bernhard Nebel

Table of contents

1. Installation

1.1 pyavrocd Installation

1.1.1 Arduino IDE 2

If you want to use pyavrocd as part of Arduino IDE 2, you do not need to install it explicitly. It is sufficient to add an "additional boards manager URL" and install the respective core. It will then by installed as a tool for this core. As a Linux user, you may also need to set some permissions and provide udev rules.

If you want to use pyavrocd stand-alone or as part of another IDE, you need to install the pyavrocd package explicitly.

1.1.2 Downloading binaries

Go to the GitHub page (if you are not already there), select the latest release (located on the right-hand side of the page), download the archive containing the binary for your architecture, and then untar the archive. It includes the executable pyavrocd, a folder pyavrocd-util, and additionally avr-gdb, the GDB debugger for AVR chips. Store pyavrocd and pyavrocd-util somewhere in the same folder and include this folder in your `PATH` variable. The avr-gdb debugger has version 16.3, which is relatively recent, and has been compiled for your architecture with only a minimal amount of references to dynamic libraries. It is up to you to decide whether you want to use this version or the one that is already installed on your system.

Since the binaries were generated on very recent versions of the respective operating systems (Windows 11, macOS 15.4, Ubuntu 24.04), it can happen that the binary is not compatible with your operating system. In this case, use one of the methods below.

1.1.3 PyPI

I assume you already installed a recent Python version (>=3.9).

It will be necessary to install pipx first. If you haven't done so already, follow the instructions on the pipx website. Then proceed as follows.

Linux

```
pipx install pyavrocd pipx ensurepath sudo ~/.local/bin/pyavrocd --install-udev-rules
```

The last command will install the necessary udev rules. This can also be done manually by following the instructions in the pyedbglib README.

After unplugging and replugging the debugger and restarting your shell, you can invoke the GDB server by simply typing `pyavrocd` into a shell. The binary is stored under `~/.local/bin/`

Windows and macOS

```
pipx install pyavrocd pipx ensurepath
```

After restarting the shell, you should be able to start the GDB server. The binary is stored under `~/.local/bin/`

1.1.4 GitHub

Alternatively, you can download or clone the GitHub repository. Additionally, you need to install the Python package manager poetry:

```
pipx install poetry
```

With that, you can start executing the script inside the downloaded folder as follows:

```
poetry install poetry run pyavrocd ...
```

Furthermore, you can create a binary standalone package as follows (after having installed the pyinstaller package):

```
poetry run pyinstaller pyavrocd.spec
```

After that, you find an executable `pyavrocd` (or `pyavrocd.exe`) in the directory `dist/pyavrocd/` together with the folder `pyavrocd-util`. You can copy those to a place in your `PATH`. If you want to generate a binary on a Mac that can be shipped to other Macs, you should use `arm64-apple-pyavrocd.spec` or `intel-apple-pyavrocd.spec` in order to include the right `libusb` for the host architecture. Note that fat binaries cannot be generated.

1.2 Debugging software

The GDB server provides an interface to the hardware debuggers on one side and to the GDB debugger on the other side. That means you still need to install the debugger, which in our case means avr-gdb. However, perhaps you even want more than that.

1.2.1 CLI debugging

The most basic option is simply to install avr-gdb, the GDB debugger for AVR chips. You can use the version shipped with the pyavrocd binaries or the version already installed on your system. If avr-gdb is not installed, use your preferred package manager on Linux, Homebrew on macOS, or download a version from Zak's avr-gcc-build repository. This is particularly useful when you want to run debugging software on a 32-bit system.

If you are not a fan of a command-line interface, then an integrated development environment (IDE) or a simple graphical user interface (GUI) for avr-gdb is called for.

1.2.2 Arduino IDE 2

Arduino IDE 2 is probably the most straightforward option. After installing it, you can extend the IDE's capabilities by adding third-party platforms. This is done by adding additional Board Manager URLs in the preferences dialog and selecting a board in the Board Manager. For example, adding the following three Board Manager URLs enables debugging of almost all debugWIRE MCUs.

```
https://felias-fogg.github.io/ATTinyCore/package_drazzy.com_ATTinyCore_index.json https://mcudude.github.io/MicroCore/package_MCU
```

After that, you must install the respective cores. And this is all! Now, you can press the debug button and start debugging. Well, before you can do that, you most probably need to modify the board, and you need to connect the hardware debugger to the target board.

Linux users may need to add a few udev rules. When you first start the Arduino IDE debugger and the hardware debuggers are not recognized, a hint appears in the gdb-server window on how to set the udev rules. You simply need to execute pyavrocd once as root using the command-line option `--install-udev-rules`. Instead, you can create a udev-rules file along the lines described in the README file of pyedbglib.

1.2.3 PlatformIO and Visual Studio Code

PlatformIO is a cross-platform, cross-architecture, multiple framework professional tool for embedded systems engineers. Installed as an extension to the popular Visual Studio Code, it provides a powerful IDE for embedded programming and debugging. Using the `platformio.ini` file, integrating an external debugging framework is very easy. If you want to debug a program on an ATmega328P, the `platformio.ini` file could look as follows (see also the `example` folder)

```
[platformio] default_envs = debug [env:atmega328p] platform = atmelavr framework = arduino board = ATmega328P board_build.f_cpu =
```

Note that the debug environment should be the default one. It should be the first if no default environment has been declared.

I further noticed that the avr-gdb debugger in the PlatformIO toolchain is quite dated and does not start (e.g., under Ubuntu 24.04 and macOS 15.5). Simply replace it with a more recent version from your system or use the version shipped with the pyavrocd binary. The location where PlatformIO stores its copy of avr-gdb is `~/.platformio/packages/toolchain-atmelavr/`, where the tilde symbol signifies the home directory of the user.

1.2.4 Gede

Gede is a lean and clean GUI for GDB. It can be built and run on almost all Linux distros, FreeBSD, and macOS. You need an avr-gdb client with a version of 10.2 or higher. If you have installed Gede somewhere in your PATH, pyavrocd will start Gede in the background if you specify the option `--start gede` when starting pyavrocd.

1.2.5 Other options

There are a few other possible options. The most crucial point is that remote debugging and the specification of alternative debuggers are supported. I believe it should be possible to integrate pyavrocd into **Visual Studio Code**, **CLion**, and **Eclipse**. How to integrate a GDB server into CLion is, for example, described here.

If you have a clear description of how to integrate pyavrocd in an IDE, I'd be happy to add it here.

2. Usage

2.1 Preparing a target board for debugging

When you want to debug a program on a target board, usually some modifications of the MCU fuses, the bootloader, and/or hardware are necessary. For this reason, it is a good idea to record the current state and the changes necessary to enable the board for debugging:

- download the current fuse settings (using avrdude),
- download the currently used bootloader (again using avrdude) or make sure that you are able to reinstall the same bootloader, and
- record necessary physical changes on the target board.

With that, it will be easy to restore the original state after debugging, if desired. If you are working in the Arduino context, restoring fuses and the bootloader is something you can delegate to the Burn Bootloader function. However, you should record any physical changes. You can get some decent development boards from Microchip that contain embedded debuggers, which work well with pyavrocd. In this case, preparations and restoring the original state are not an issue.

2.1.1 General considerations

Depending on the type of debugging interface the MCU provides, different actions must be taken to prepare the target board for debugging. The general rule is that the lines used for debugging should not have any resistive or capacitive loads or active components on them.

Sometimes it may be additionally necessary to change a few fuses before debugging is possible. Some of the fuses will be taken care of by the GDB server, provided pyavrocd is asked to manage these fuses by a command line option, e.g., `--manage dwen`, when invoking the GDB server:

- `Lockbits`: If lockbits are set, then debugging is impossible. For this reason, the GDB server will clear the lockbits by erasing the chip's flash (and perhaps EEPROM) memory, provided pyavrocd has been instructed to manage the lockbits.
- `BOOTRST`: If this fuse is programmed, then instead of starting at address 0x0000, the MCU will start execution at the bootloader address. Since this is usually not intended when debugging, the GDB server unprograms this fuse. For the unlikely case that one wants to debug a bootloader, there is still the option to protect this fuse by not including `bootrst` as a fuse to be managed by the server when starting the GDB server from the command line.
- `DWEN`: This fuse needs to be programmed to use the debugWIRE on-chip debugger. Pyavrocd will program this fuse when asked to do so by the command `monitor debugwire enable`. After the fuse has been programmed, you must power-cycle the target board to enable the debugWIRE interface. Note that afterwards, SPI programming is impossible. With the command `monitor debugwire disable`, the debugWIRE interface will be disabled, and the `DWEN` fuse will be unprogrammed. Of course, DWEN programming by pyavrocd is only performed if pyavrocd is instructed to manage this fuse.
- `OCDEN`: This is the fuse for enabling the JTAG on-chip debugger. It is simpler to deal with than `DWEN`, because one can enable and disable this fuse in every situation. It will be activated before debugging starts and deactivated afterwards. This happens, of course, only if pyavrocd has been instructed to manage this fuse.
- `EESAVE`: If this fuse is programmed, then EEPROM contents will survive chip erase operations. If not, EEPROM content is deleted each time an erase operation is performed, even if this is only organizational. If you want to protect your EEPROM content, allow pyavrocd to manage this fuse. It will then temporarily program this fuse when necessary in order to safeguard the EEPROM content. This is particularly important when loading an executable that contains a code part to be stored in EEPROM.

If you want to leave all the fuse management to pyavorcd, then just specify `--manage all`, which is the default with Arduino IDE2. If you want to play it safe, you can manage these fuses and the lockbits manually using a fuse setting program such as avrdude.

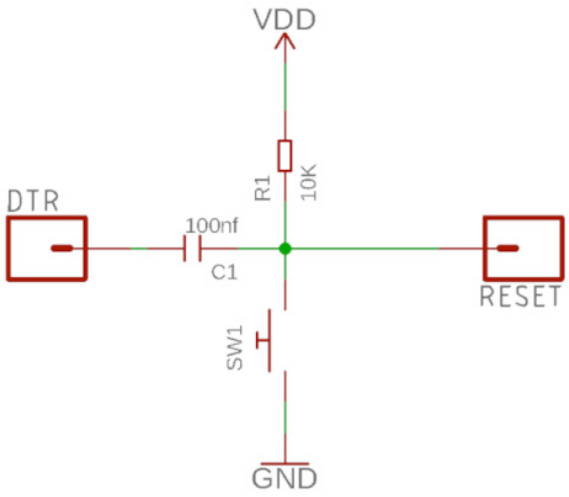
Finally, as already mentioned above, bootloaders will be deleted, so they need to be reinstalled after debugging has finished. Additionally, one cannot use the services some bootloaders offer, e.g., writing to flash memory. If you want to debug such a program, you need to set up a mock object.

2.1.2 Preparing a debugWIRE target

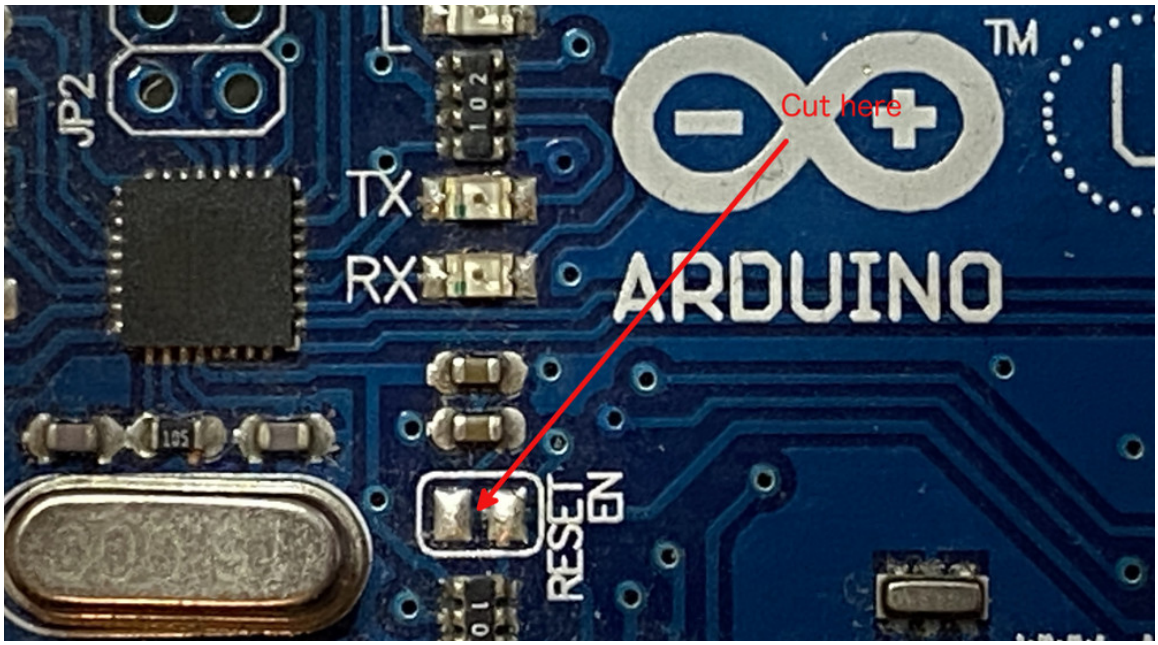
Physical preparations

Since the RESET line is used for communication between the MCU and the hardware debugger, no capacitors should be connected to it. Similarly, pull-up resistors should not be stronger than 10 kΩ. And, there should be no active reset circuit connected to this line. In other words, before debugging starts, disconnect such components from the RESET line.

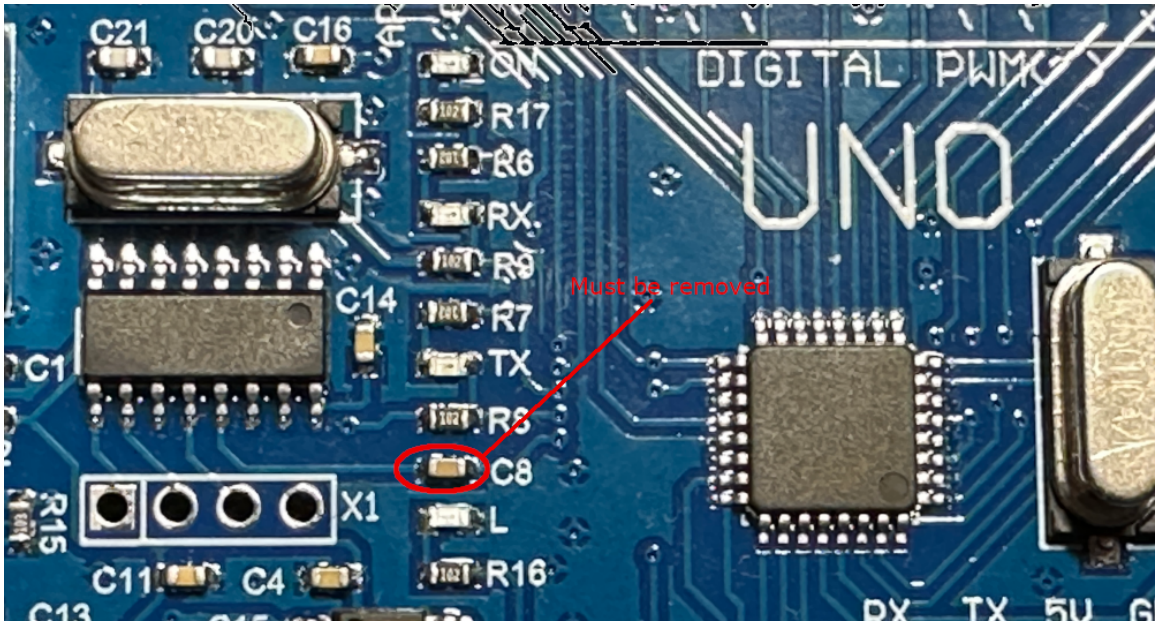
On the **Arduino Uno** and similar boards, an auto-reset capacitor is usually connected to the RESET line, as shown below.



This is responsible for issuing a reset signal when a serial connection is established to the board, which starts the bootloader, which then expects a HEX file sent by the Arduino IDE. On the original Uno board, there is a solder bridge marked 'RESET EN' that needs to be cut to disconnect the capacitor.



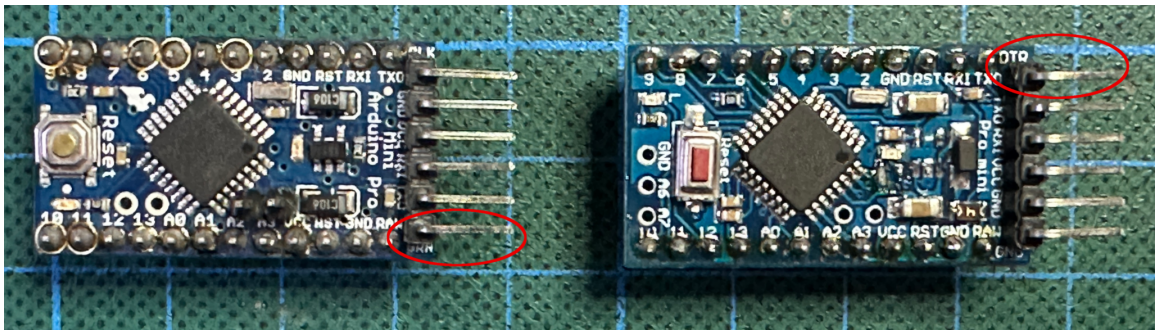
On clone boards with a CH340 serial converter chip, you may have to remove the capacitor marked `C8`.



Things are a bit more complicated with **Arduino Nano** boards. Here, you not only have to remove the auto-reset capacitor but also a strong pull-up resistor of 1kΩ on the RESET line. This is impossible for the original boards because the resistor is part of a resistor array. You may try to cut the trace from Vcc to the resistor, but I doubt this can be done without damaging other parts of the board.

For Arduino Nano clones (those using a CH340 as the serial converter), one can remove the resistor and the capacitor, as described by denMike.

The **Arduino Pro Mini** is a simpler case. The pull-up resistor has a resistance of 10 kΩ, and the auto-reset capacitor is not connected as long as nothing is connected to the DTR pin. This is the header pin, either labeled DTR or GRN. On the original Sparkfun board (left), this is the bottom pin; on some clones (right), it is the top one.



For other boards with ATmega168 and ATmega328 chips, the situation is similar. Find out what is connected to the RESET line and disconnect any capacitors and strong resistors. And the same holds for other debugWIRE MCUs.

Fuse settings

In almost all cases, you do not need to change any fuses on a debugWIRE target before you can start debugging. One exception is when the RESET pin has been disabled (by programming the `RSTDSBL` fuse), allowing it to be used as a GPIO. In this case, you need to unprogram this fuse using high-voltage programming. The same holds when `SPIEN` (enabling SPI programming) is unprogrammed.

The `DWEN` , `BOOTRST` , and `EESAVE` fuses and the `lockbits` will be taken care of by pyavrocd, if this is permitted (see above).

2.1.3 Preparing a JTAG target

Physical preparation

JTAG targets are easier to deal with. Simply do not connect anything to the JTAG lines (`TDI` , `TDO` , `TMS` , `TCK`) or disconnect those components.

Fuse settings

Access to the JTAG pins could be disabled. This is, for example, the case for the Arduino boards. In this case, you need to program the JTAGEN fuse before debugging can start. This has to be done using the SPI programming interface. In the Arduino IDE 2, you can achieve this by setting the `JTAG` attribute in the `Tools` menu to `enabled` and then performing the `Burn Bootloader` action afterward using SPI programming. From then on, you can connect to the board using the JTAG connector.

As in the debugWIRE case, it could be that SPI programming has been disabled. If the JTAG pins are enabled, this does not matter because the JTAG pins are all that is needed. If not, high voltage programming is necessary.

The `OCDEN` , `BOOTRST`, and `EESAVE` fuses and the `lockbits` will be taken care of by pyavrocd (see above).

2.1.4 Preparing a PDI target

Physical preparation

If the PDI interface is used, then the RESET line will be employed as a clock line. Here, we have the same restrictions as in the case of debugWIRE: no resistive or capacitive load, or active reset circuit, on the reset line.

Fuse settings

SPIEN could be disabled. In this case, the above comments apply. Otherwise, there is no need to change any fuses before beginning the debugging process.

2.1.5 Preparing a UPDI target

Physical preparations

Ensure that there is no capacitive or resistive load or active component on the UPDI line and that the UPDI pin is accessible.

On the **Nano Every**, for example, this pin cannot be accessed through the board pins, but there is a pad on the backside of the PCB that can be used to access the UPDI line. And the USB-UART converter is usually disconnected from this pin.

On the **Uno WIFI Rev2**, again the UPDI pin is not exposed. But on this board, a mEDBG debugger is implemented. So you can connect to this debugger.

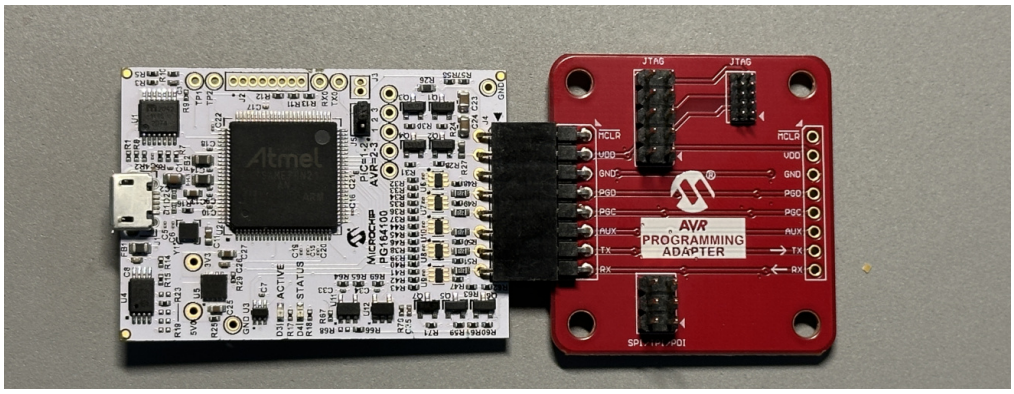
Fuse settings

If the UPDI pin is a dedicated UPDI pin, you do not have to prepare anything. If this is not the case, then the pin might have been programmed to act as a GPIO or the RESET line. To enable debugging and programming over this pin again, you will need to use a high-voltage UPDI programmer. Here, you must ensure that the 12 V pulse does not damage any components on your board.

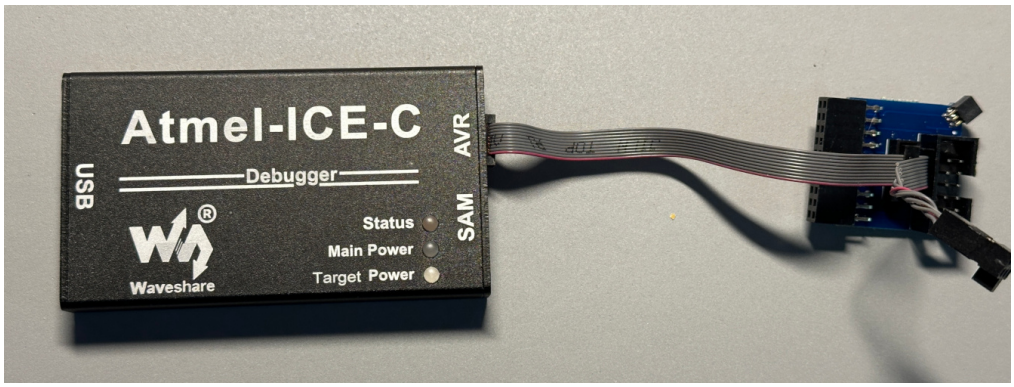
2.2 Connecting the hardware debugger to a target

The hardware debuggers have different connectors. The Microchip debuggers Snap and PICkit4 have an eight-pin SIL connector, where a triangle marks pin 1. This connector is not compatible with any AVR debug connector.

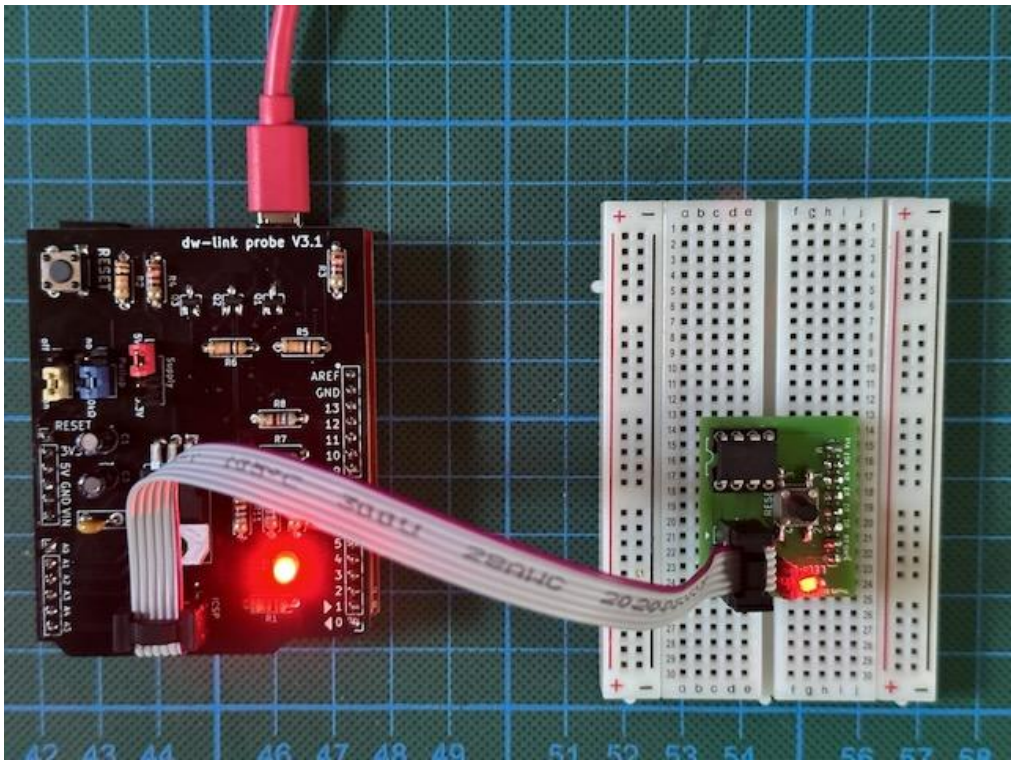
If you want to connect to your target board with a standard SPI or JTAG cable, you can buy an adapter board for AVR connectors from Microchip, as shown in the following picture.



Atmel-ICE, Power Debugger, and JTAGICE3 all feature a keyed 10-pin, 50-mil JTAG header. For these debuggers, adapters are either already included or must be purchased separately. For AVR targets, one should, of course, use the header marked `AVR`.



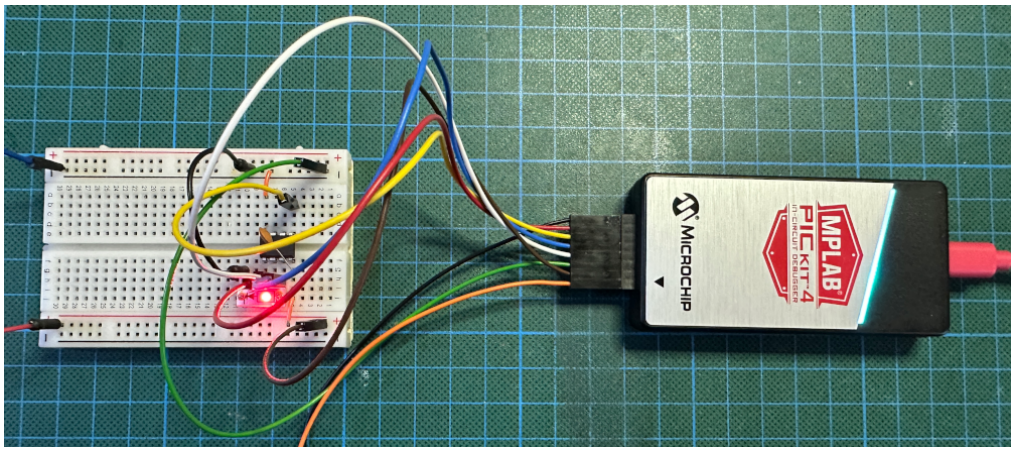
The dw-link debugger uses the header on the Arduino Uno. If a dw-link shield is used, one can use the standard 6-pin SPI header.



Finally, all EDBG debuggers are easy to use. Since they are embedded debuggers, the connection to the target is already on the board.

Depending on which debugging interface the target has, the target board may provide a standard debugging header for this interface. I very much prefer to work with target boards that have the appropriate debugging headers on board. Otherwise, you may easily confuse a connection, and then nothing works.

If you do not have the standard headers on board or you are using a breadboard, then you have to connect each line using a jumper cable or the Atmel squid cable, as shown in the following picture.



In this case, it is essential to consult the user guide of the programmer and the pinout of the MCU in the datasheet to make the correct connections.

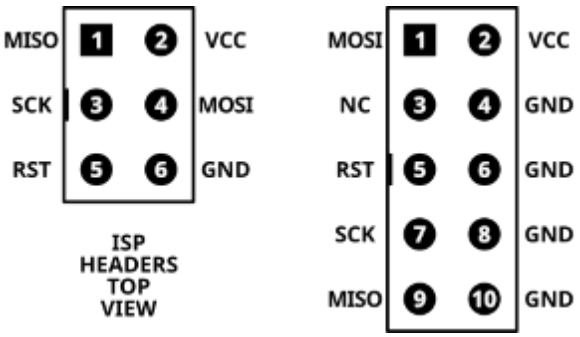
2.2.1 Connecting to a debugWIRE target

In principle, only two wires are necessary to connect your hardware debugger to a target chip or board: the debugWIRE line, which is the target chip's RESET line, and GND. Since the debugger also needs to know which voltage the target board uses, the Vcc line is also necessary. Note that none of the commercial debuggers source the target. They only have voltage-sensing lines to drive the level-shifting hardware.

Since one also wants to change into and out of debugWIRE mode, change fuses, or upload firmware, it is necessary to connect all 6 SPI programming lines to the target: VTG, GND, RESET, MOSI, MISO, and SCK. For this reason, using all SPI programming lines makes a lot of sense. Moreover, most of the time, an SPI connector is already on the target board.

SPI programming header

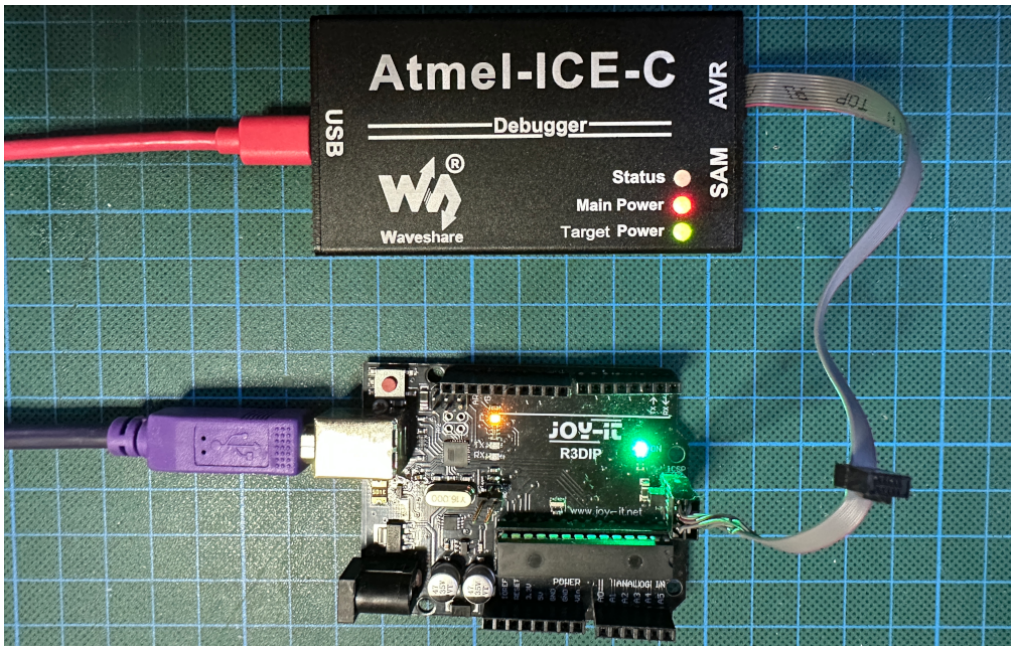
There are two types of SPI programming connectors. The more recent type has six pins, and the older type has 10 pins, as shown in the following diagram (based on a diagram from Wikipedia, which provides a top view of the headers on a PCB.



Note the notches on the left side of the headers. Since almost all SPI programming plugs are keyed, you can only plug them in in the correct orientation. However, the headers sometimes do not have notches. In this case, pin 1 is usually marked in some way, either with a dot, a star, or with the number 1. Similarly, plugs also come unkeyed. In this case, again, pin 1 is marked in some way.

Connecting to targets with an SPI programming header

If the target board has an SPI programming header, it is easy to connect to it. Simply use the SPI programming cable and plug it into the target board's header. Be aware of the correct orientation when the header is not keyed! For all the Arduino boards, pin 1 is always oriented towards the USB connector. However, if you plug it in the wrong way, nothing will be destroyed.



Connecting to targets without an SPI programming header

If the target does not feature an SPI programming header, you need to connect 6 cables. If you are working with a breadboard, you may consider buying an SPI header breadboard adapter. Otherwise, you need to connect each pin individually. **Atmel-ICE**, **Power Debugger**, and **JTAGICE3** have a so-called 10-pin mini-squid cable. The pin mapping for those debuggers is as follows.

Atmel Debugger	Mini-squid pin	Target pin	SPI pin
Pin 1 (TCK)	1	SCK	3
Pin 2 (GND)	2	GND	6
Pin 3 (TDO)	3	MISO	1
Pin 4 (VTG)	4	VTG	2
Pin 5 (TMS)	5		
Pin 6 (nSRST)	6	RESET (debugWIRE)	5
Pin (N.C.)	7		
Pin 8 (nTRST)	8		
Pin 9 (TDI)	9	MOSI	4
Pin 10 (GND)	0		

For **PICkit4** and **SNAP**, such a table looks as follows, with pin 1 marked by a triangle.

MBLAP Debugger	Pin #	Target pin	SPI pin
Pin 1 (TVPP)	1		
Pin 2 (TVDD)	2	VTG	2
Pin 3 (GND)	3	GND	6
Pin 4 (PGD)	4	MISO	1
Pin 5 (PGC)	5	SCK	3
Pin 6 (TAUX)	6	RESET (debugWIRE)	5
Pin 7 (TTDI)	7	MOSI	4
Pin 8 (TTMS)	8		

When you want to connect a **dw-link** debugger without a dw-link probe shield to a target, you can use jumper cables using the following pin mapping.

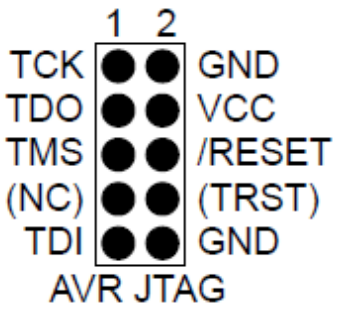
dw-link Arduino Uno pins	Target pin	SPI pin
D8	RESET (debugWIRE)	5
D11	MOSI	4
D12	MISO	1
D13	SCK	3
5V (if powered by debugger)	Vcc	2
GND	GND	6

With a dw-link probe shield, it is best to construct or buy a cable with a 6-pin SPI programming plug on one end and single Dupont pins on the other.

2.2.2 Connecting to a JTAG target

Note that in order to use the JTAG connection, you might first need to enable the JTAG pins by programming the JTAGEN fuse via SPI programming. On chips fresh from the factory, this fuse is programmed. On Arduino boards, JTAGEN is disabled.

For AVR MCUs, there is a standard pinout as follows.



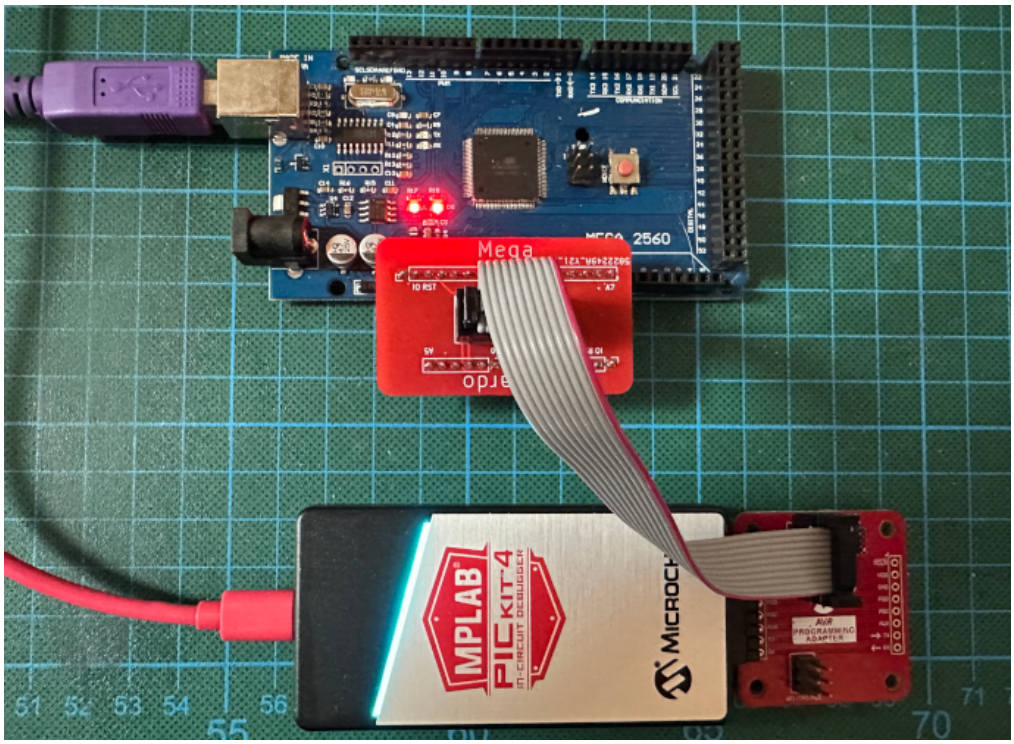
Sometimes, pin 8 is connected to nTREST, which we do not need, though. The crucial pins are TCK (JTAG clock), TDO and TDI (data lines), and TMS (control line). In addition, we have nSRST, the reset line, and VTref and GND.

Connecting to targets with a JTAG header

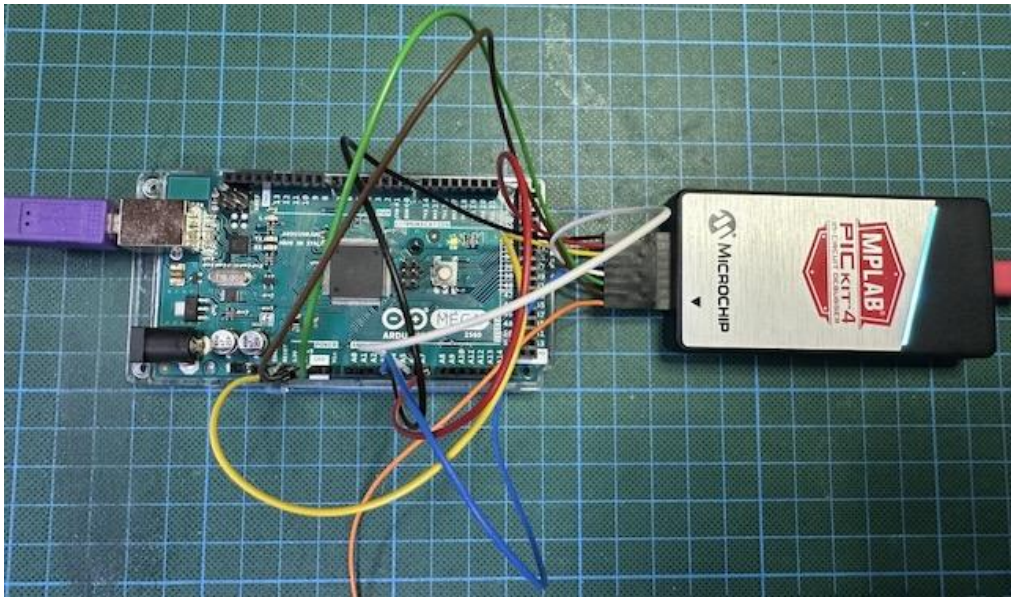
Again, if there is a JTAG header on the board, connecting the board is a breeze. Simply use the right cable.

Connecting to targets without a JTAG header

When debugging a program running on an Arduino Mega or Leonardo, you could use an adapter that plugs into the header, as shown in the following picture.



Otherwise, you must connect the wires individually, referring to the Arduino Mega's pinout and the header on the debugger. Here is an example for connecting PICkit4 (or Snap) to an Arduino Mega 2560.



The pin mapping for the PICkit4 looks as follows, where I have added the Arduino pins in the last column.

MBLAP Debugger	Pin #	Target pin	JTAG pin	Mega pin
Pin 1 (TVPP)	1	NC		
Pin 2 (TVDD)	2	VTG	4	5V
Pin 3 (GND)	3	GND	2, 10	GND
Pin 4 (PGD)	4	TDO	3	A6
Pin 5 (PGC)	5	TSCK	1	A4
Pin 6 (TAUX)	6	RESET	6	RESET
Pin 7 (TTDI)	7	TDI	9	A7
Pin 8 (TTMS)	8	TMS	5	A5

For the Atmel debuggers, the setup appears as follows, where the JTAG pin corresponds to the mini-squid numbering. I have additionally added the corresponding Mega pin.

Atmel Debugger	Mini-squid pin = JTAG pin	Target pin	Mega pin
Pin 1 (TCK)	1	TCK	A4
Pin 2 (GND)	2	GND	GND
Pin 3 (TDO)	3	TDO	A6
Pin 4 (VTG)	4	VTG	5V
Pin 5 (TMS)	5	TMS	A5
Pin 6 (nSRST)	6	RESET	RESET
Pin 7 (N.C.)	7	NC	
Pin 8 (nTRST)	8	NC	
Pin 9 (TDI)	9	TDI	A7
Pin 10 (GND)	0	GND	GND

2.2.3 Connecting to UPDI and PDI

Will be treated later when implemented.

2.3 Invoking pyavrocd

You invoke pyavrocd as follows:

```
> pyavrocd [options]
```

Pyavrocd will then look for a hardware debugger, establish a connection to it, and waits for the GDB debugger to connect to it. You can influence its behavior by the following command-line options.

Option Name	Description
`--command` `-c`	Command to set the gdb port (OpenOCD style), which is used in the Arduino IDE 2. This is an alternative to the `--port` option.
`--device` `-d`	The argument to this option specifies the MCU type of the target chip in lower case. This option is mandatory. If a '?' mark is given, all supported MCUs are listed.
`--debug-clock` `-D`	JTAG clock frequency for debugging (kHz). This value should be less than a quarter of the MCU clock frequency. The default is (a conservative) 200 kHz.
`--help` `-h`	Gives help text and exits.
`--interface` `-i`	Debugging interface to use. Should be one of `debugwire`, `jtag`, `pdi`, or `updi`. Only necessary if an MCU supports more than one interface or if one wants to see only the supported chips with a particular interface.
`--manage` `-m`	Can be given multiple times and specifies which fuses should be managed by pyavrocd. Possible arguments are `all`, `none`, `bootrst`, `nobootrst`, `dwen`, `nodwen`, `ocden`, `noocden`, `eesave`, `noeesave`, `lockbits`, and `nolockbits`. Later values in the command line override earlier ones. Any fuses not managed by pyavrocd need to be changed 'manually' before and/or after the GDB server is activated. Note that dw-link ignores this option!
`--port` `-p`	IP port on the local host to which GDB can connect. The default is 2000.
`--prog-clock` `-P`	JTAG programming clock frequency in kHz. This is limited only by the target MCU silicon, not by the actual MCU clock frequency used. The default is (a conservative) 1000 kHz.
`--start` `-s`	Program to start or the string `noop`, when no program should be started
`--tool` `-t`	Specifying the debug tool. Possible values are `atmelice`, `edbg`, `jtagice3`, `medbg`, `nedbg`, `pickit4`, `powerdebugger`, `snap`, `dwlink`. Use of this option is necessary only if more than one debugging tool is connected to the computer.
`--usbsn` `-u`	USB serial number of the tool. This is only necessary if one has multiple debugging tools connected to the computer.
`--verbose` `-v`	Specify verbosity level. Possible values are `all`, `debug`, `info`, `warning`, `error`, or `critical`. The default is `info`. The option value `all` means that, in addition to the debug output, all communication with GDB is logged.
`--version` `-V`	Print pyavrocd version number and exit.
`--install-udev-rules`	Install the udev rules necessary for Microchip's EDBG debuggers. Needs to be run with `sudo` and is only present under Linux.

You can also use the monitor command options as command-line options in order to set debugger values already at startup. For example, you may specify `--timers freeze,` which has the same effect as issuing the command `monitor timers freeze` in the debugger at startup. You can also use one-character abbreviations for such option values and the usual abbreviation rules for options, shortening it to `--ti f` .

In addition to options, one can specify file names prefixed with a '@'-sign. Such files can contain additional arguments. Arguments read from such a file must be one per line and are treated as if they were in the same place as the original file referencing argument on the command line. If the file does not exist, no error is raised.

The argument `@pyavrocd.options` is always added to the command line. In other words, if there is such a file in the folder where the GDB server is invoked, then the arguments in this file will override the command line. This is the way to override options on a per-project basis in an IDE, where the IDE invokes the GDB server.

2.4 Debugging the target

2.4.1 Debugging with a command-line interface

After compiling your program, e.g., varblink0.ino, you can start the GDB server and the GDB debugger. When calling the compiler, you should provide the following two options: `-Og` and `-ggdb3`. The first one optimizes for debugging (instead of size or speed), the second requires including as many symbols from the source program as possible.

When starting the GDB server from the command line, you need to specify the MCU you want to connect to. In addition, you should specify the option `-m all`, so that the GDB server manages the debug-related fuses (see Preparing a target board):

```
> pyavrocd -d atmega328p -m all [INFO] Connecting to anything possible [INFO] Connected to Atmel-ICE CMSIS-DAP [INFO] Starting py
```

In another terminal window, you can now start a GDB session:

```
> avr-gdb varblink0.ino.elf GNU gdb (GDB) 15.2 Copyright (C) 2024 Free Software Foundation, Inc. ... (gdb) target remote :2000 Re
```

If you have reached this point, I trust that you are familiar with GDB and know how to proceed.

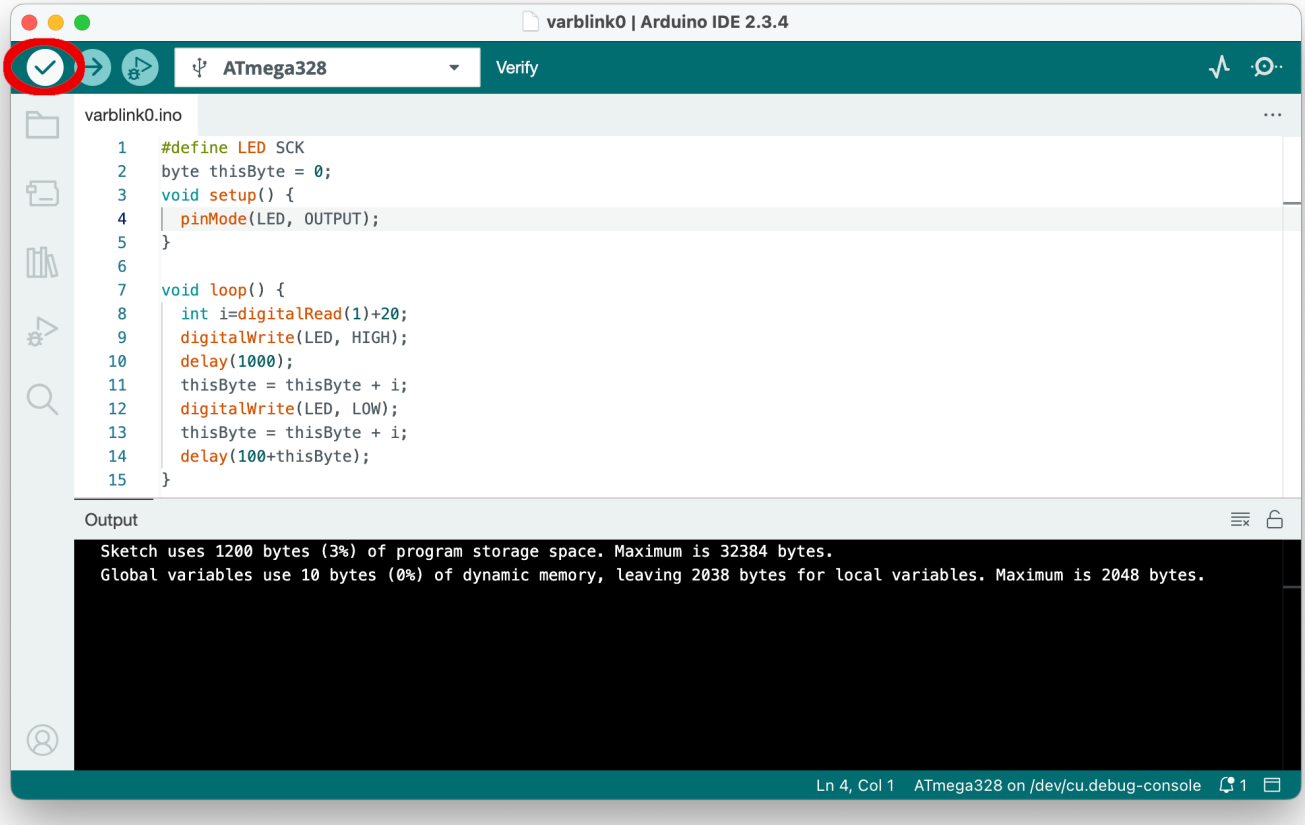
Note the request to power-cycle the target system, which will only appear when dealing with debugWIRE targets. You then need to disconnect and reconnect the power to the target. Afterward, debugWIRE mode is enabled, and you can debug. The debugWIRE mode will not be disabled when you leave the debugger! It will only be disabled when you issue the command `monitor debugwire disable`. This means that until then, the RESET button will not be of any use; you cannot upload anything using SPI programming, nor can you change fuses. Since pyavrocd needs to delete the bootloader as well, you also cannot upload anything over the serial line.

2.4.2 Debugging using the Arduino IDE 2

If you prefer to use an IDE instead of a CLI, the Arduino IDE 2 is the most straightforward option.

Compiling the sketch

You must load the sketch into the editor and select a board as usual. If you want to debug an Arduino Uno R3 board, choose ATmega328 from the `MiniCore` in the `Tools` menu. Before you can debug your code, you need to compile it, which will be done when you click on the Verify button in the upper left corner of the Arduino IDE window (see below).

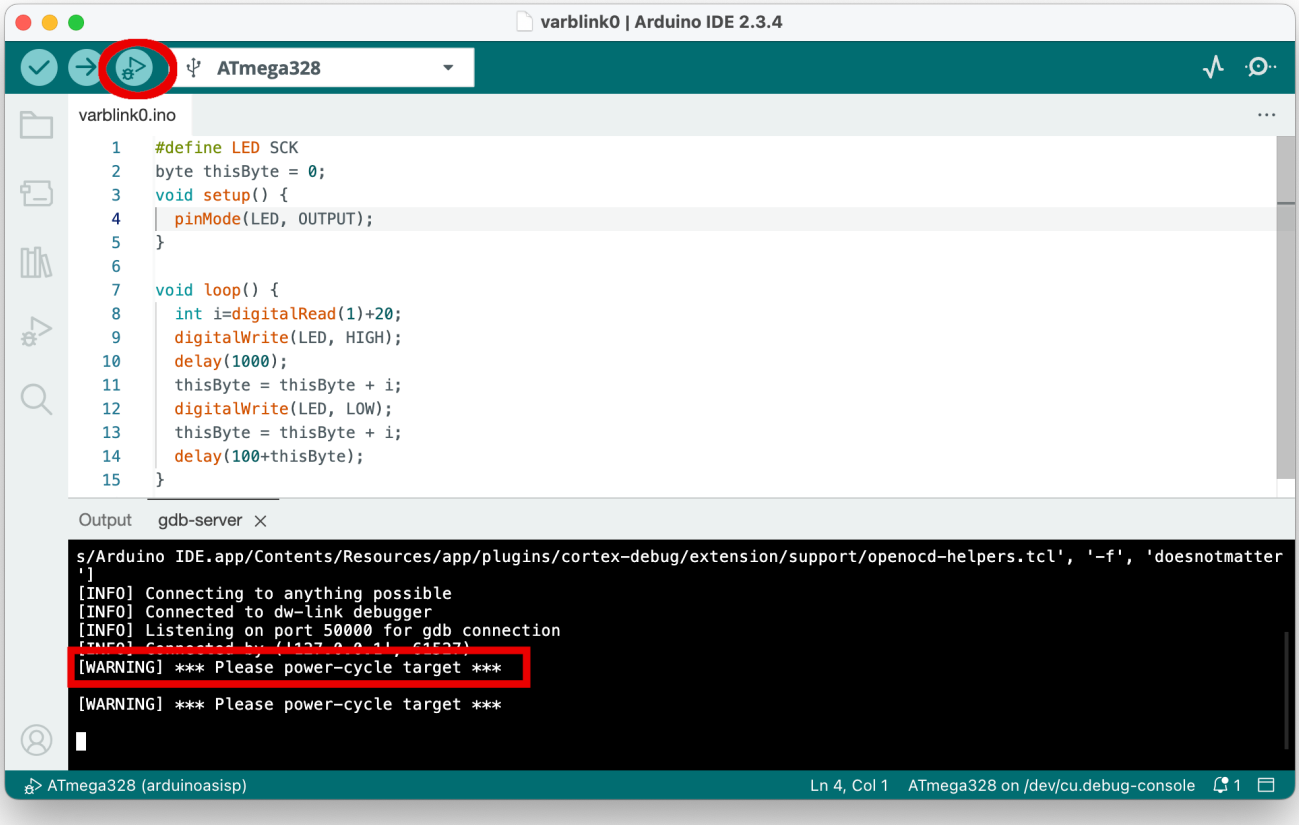


Before clicking the `Verify` button in the upper left corner, choose `Optimize for Debugging` in the `Sketch` menu. This is necessary so that the compiler optimizes the code in a way that makes debugging straightforward. Otherwise, the compiler may rearrange source code lines, which can be confusing when single-stepping through the code.

Starting the debugger

After compiling the sketch, it is time to start debugging by clicking the debug button in the top row. This will start the debug server.

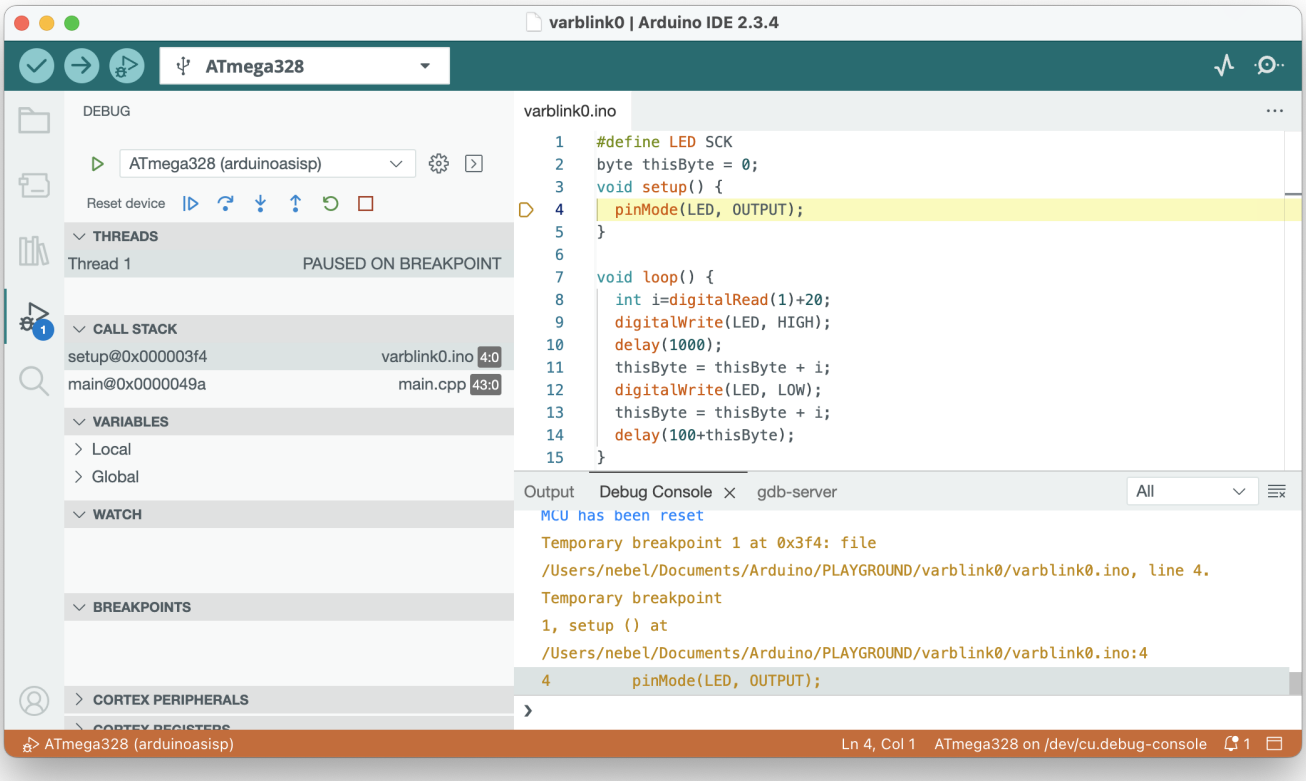
Instead of the message shown in the following screenshot, a warning "No hardware debugger discovered" may be displayed. The reason may be that the Arduino IDE 2 reserved the debugger's serial line for the `Serial Monitor`. Simply close the `Serial Monitor` console and try again. On Linux, another reason could be that the udev rules have not yet been installed (see installation instructions). Or maybe you forgot to connect a hardware debugger altogether.



If there is a connection to the debugger and the target, the GDB server will start up. When you deal with a debugWIRE target, you may be asked to power-cycle the target, i.e., to disconnect and reconnect power to the target. As mentioned above, power cycling is only necessary once. The next time you start a debugging session, the MCU will already be in debugWIRE mode, and the debugger will not stop at this point.

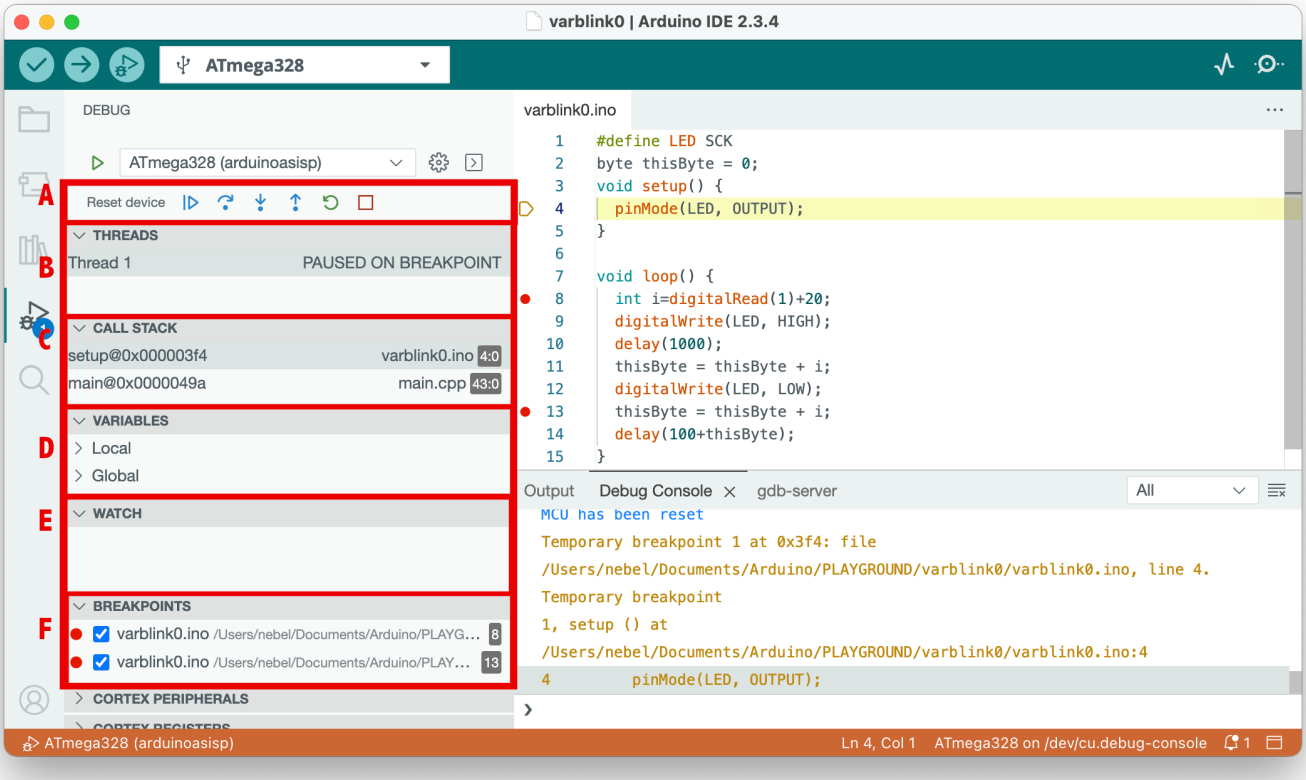
After power-cycling the target, the debugger starts. Eventually, execution is stopped in line 4 at an initial internal breakpoint, indicated by the yellow triangle left of line 4 in the following screenshot. It may take some time before we reach that point, as the debugger must also load the program.

After stopping, the IDE rearranges the layout, showing the debugging panes on the left and the sketch on the right. It will also switch from displaying the `gdb-server` console to the `Debug Console`, which displays the output of the GDB debugger. In the last line of this console, a prompt symbol `>` is shown, where you can enter any GDB command, in particular the `monitor` commands to control the GDB server. Here, the command `monitor debugwire disable` is crucial because it will disable the debugWIRE mode.



Now is a good time to familiarize yourself with the window's layout. The source code is on the right side. Below that is a console window, and to the left are the debug panes. To set a breakpoint, click to the left of the line numbers. Such breakpoints are displayed as red dots, such as those located to the left of lines 8 and 13.

Debugging



The debugging panes are organized as follows. Pane A contains the debug controls. From left to right:

- *Reset*ting the device
- *Continue* execution or *pause*
- *Step over*: execute one source line
- *Step into*: execute stepping into the function, if in this line one is called
- *Step out*: finish the current function and stop after the line where it was called
- *Restart*: Same as Reset
- *Stop*: Terminate debugging

Pane B shows the active threads, but there is just one in our case. Pane C displays the call stack starting from the bottom, i.e., the current frame is the topmost. Pane D displays variable values. Unfortunately, global variables are not shown if *link-time optimizations* are enabled, which is the default. Pane E can be populated with watch expressions, for example, with the names of global variables. Finally, in pane F, the active breakpoints are listed.

The panes below pane F are interesting if you are deep into the MCU hardware. The `CORTEX PERIPHERALS` pane displays all I/O registers of the MCU, decodes their meanings, and allows you to change the contents of these registers. The `CORTEX REGISTERS` pane displays the general registers. For more information on debugging, refer to the Arduino debugging tutorial.

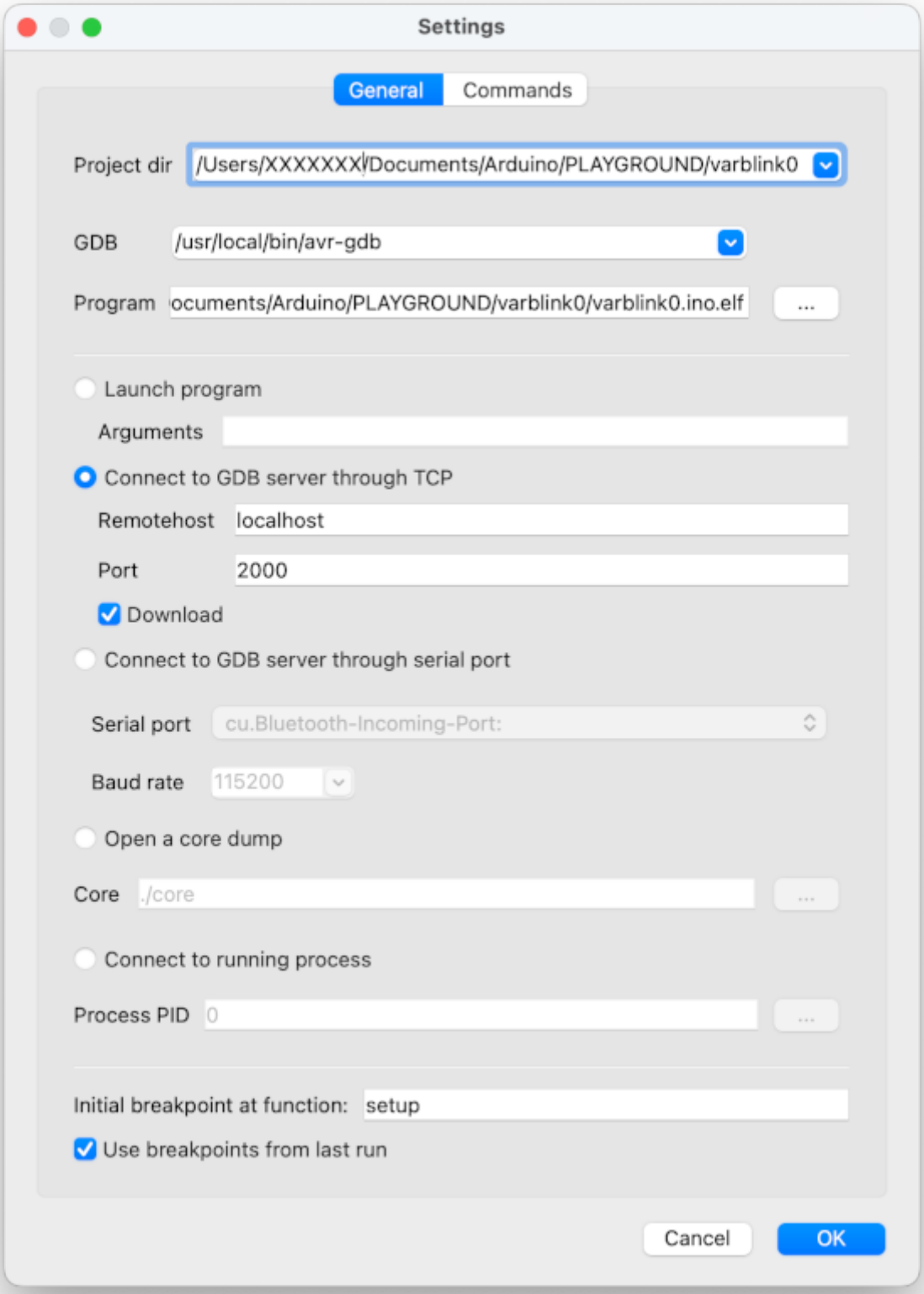
When you have decided to change the source code, remember to terminate the debugger (red square), then recompile the sketch using the upper left `Verify` button, and finally start another debugging session.

2.4.3 Debugging using PlatformIO/VSC

Debugging a program/sketch in PlatformIO/VSC is very similar to doing the same thing in the Arduino IDE 2. The reason is that both IDEs are based on VS Code. Compared to the Arduino IDE 2, PlatformIO/VSC offers several features that work better, such as easy adaptability through the `platformio.ini` configuration file and support for disassembled code. However, it may not be the proper IDE for beginners. In any case, if you are opting for PlatformIO/VSC, you are probably familiar with the tool, and I do not need to preach to the converted. A `platform.ini` that can be used to start the debugger is provided here.

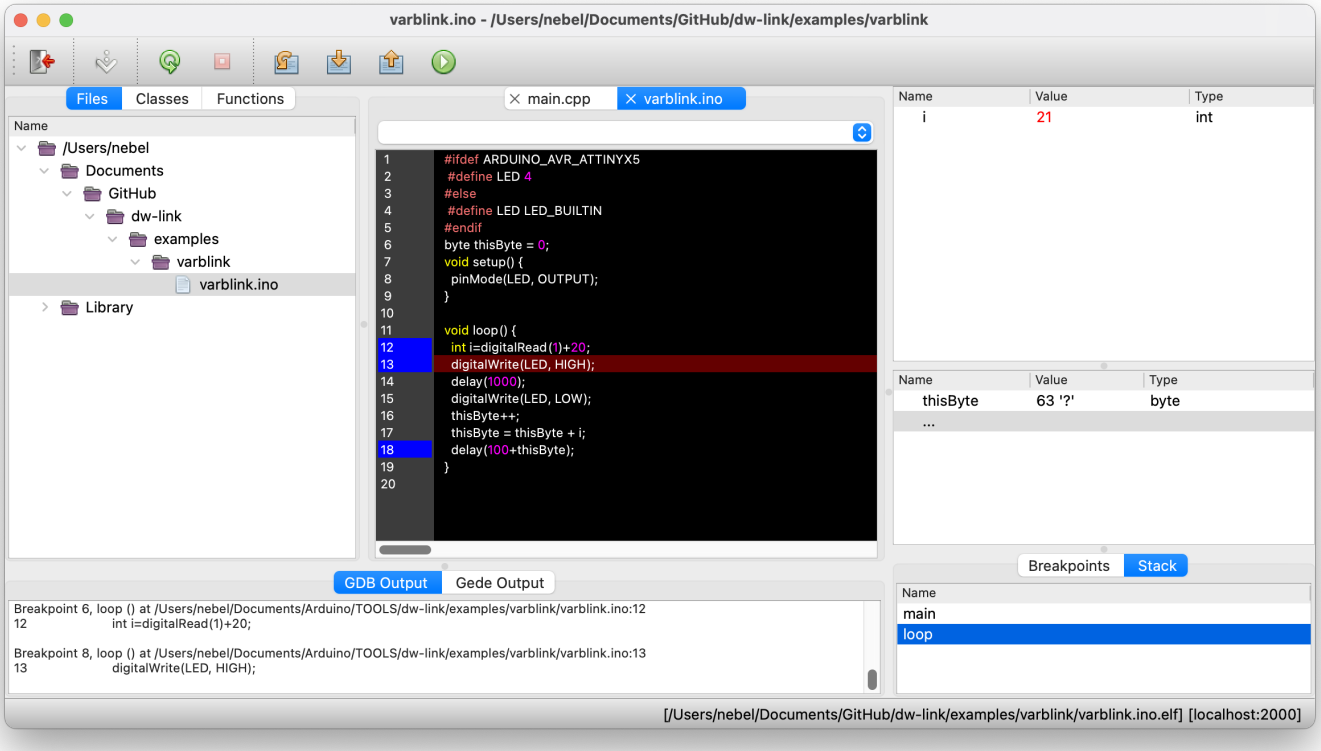
2.4.4 Debugging using Gede

Gede is a lean and clean GUI for GDB. It can be built and run on almost all Linux distros, FreeBSD, and macOS. You need an avr-gdb client with a version greater than or equal to 10.2. If you have installed Gede somewhere in your PATH, you can start Gede by specifying the option `--gede` or `-g` when starting pyavrocd.



`Project dir` and `Program` are specific to your debugging session. The rest should be copied as it is shown. Before you click `OK`, you should switch to the `Commands` section, where you need to enter the command `monitor debugwire enable` if you are working with a debugWIRE target (otherwise it does not hurt).

Clicking on OK, you start a debugging session. The startup may take a while because the debugger always loads the object file into memory. After a while, you will see a window similar to what is shown below.



2.5 Monitor commands

Pyavrocd implements several `monitor` commands. These can be used to control important aspects of the GDB server. One important command is the `monitor debugwire enable` command, which enables debugWIRE mode on MCUs supporting this interface.

Command	Action
`monitor atexit` [`stayindebugwire` | `leavedebugwire`]	When specifying `leavedubgwire`, then debugWIRE mode will be left when exiting the debugger. This is useful when dealing with embedded debuggers. The default is `stayindebugwire`, i.e., debugWIREmode will not be left when exiting the debugger. **(+)**
`monitor breakpoints` [`all` | `software` | `hardware`]	Restricts the kind of breakpoints the hardware debugger can use. Either `all` types are permitted, only `software` breakpoints are allowed, or only `hardware` breakpoints can be used. Using `all` kinds is the default.
`monitor caching` [`enable` | `disable`]	The loaded executable is cached in the gdbserver when `enabled`, which is the default setting. **(+)**
`monitor debugwire` [`enable` | `disable`]	DebugWIRE mode will be `enable`d or `disable`d. When enabling it, the MCU will be reset, and you may be asked to power-cycle the target. After disabling debugWIRE mode, one has to exit the debugger. Afterward, the MCU can be programmed again using SPI programming.
`monitor erasebeforeload` [`enable` | `disable`]	This monitor option controls whether the flash is erased before an executable is loaded, which is the default for all targets, except for debugWIRE targets, which do not have a chip erase command in debug mode. **(+)**
`monitor help`	Display help text.
`monitor info`	Display information about the target and the state of the debugger.
`monitor load` [`readbeforewrite` | `writeonly`]	When loading an executable, either each flash page is compared with the content to be loaded, and flashing is skipped if the content is already there, or each flash page is written without reading the current contents beforehand. The first option is the default option for debugWIRE targets. For JTAG targets, the overhead of checking whether the page content is identical is so high that the `writeonly` option is the default.
`monitor onlywhenloaded` [`enable` | `disable`]	Execution is only possible when a `load` command was previously executed, which is the default. If you want to start execution without loading an executable first, you need to `disable` this mode.
`monitor rangestepping` [`enable` | `disable`]	The GDB range-stepping command is supported or disabled. The default is that it is `enable`d. **(+)**
`monitor reset`	Resets the MCU.
`monitor singlestep` [`safe` | `interruptible`]	Single-stepping can be performed in a `safe` way, where single steps are shielded against interrupts. Otherwise, a single step can lead to a jump into the interrupt dispatch table. The `safe` option is the default.
`monitor speed` [`low` | `high`]	Set the communication speed limit to the target to `low` (=150kbps) (default) or to `high` (=300kbps); without an argument, the current communication speed and speed limit are printed.**(*)**
`monitor timer` [`run` | `freeze`]	Timers can either be `frozen` when execution is stopped, or they can `run` freely. The latter option is helpful when PWM output is crucial and is the default.
`monitor verify` [`enable` | `disable`]	Verify flash after loading each flash page. The default setting is for this option to be `enable`d.
`monitor version`	Show version of the gdbserver.

All commands can, as usual, be abbreviated. For example, `mo d e` is equivalent to `monitor debugwire enable`. If you use a command without an argument, the current setting is printed.

Commands marked with **(+)** are not implemented in dw-link; those marked with **(*)** are specific to dw-link.

2.6 Restoring a target to its original state after debugging

If, after debugging, you want to restore a target board to its original state, a few things have to be done:

1. If your target is a debugWIRE MCU, then you need to disable debugWIRE mode. This can be done by entering the debugger and then issuing the `monitor debugwire disable` command.
2. You need to undo the physical changes you have made to the board. This may be to restore a solder bridge (or solder a header in its place). Alternatively, you may need to solder a capacitor again or reconnect a reset circuit. My advice is not to do that, but mark the board instead for **debug use** only. In this case, you can also ignore step 3.
3. You may need to reflash the bootloader and likely need to set the correct fuses.
4. In the Arduino IDE, you can do this by using the `Burn Bootloader` command in the `Tools` menu. This will set the correct fuses and reinstall the bootloader.

To restore the board to its exact original state before debugging, you must download the bootloader and record the fuses before initiating the debugging process. Then you can easily restore the original state using an SPI programmer.

2.7 Configuration

You do not have to set up any configuration file before you can use pyavrocd. However, sometimes it may be convenient to store some options in a file so that you do not have to type them every time you invoke pyavrocd. Or, you may want to override options that are set in an IDE. For this purpose, the `@` notation is very helpful. If you place the string `@file.ext` on the command line, then arguments are read from `file.ext` and spliced into the command line. These arguments are read line by line.

Let us assume, `file.ext` contains the following lines:

```
--manage eesave --prog=3000 --to atmelice --veri=e
```

When you now invoke pyavrocd with `pyavrocd -t dwlink @file.ext`, then this is expanded into

```
pyavrocd -t dwlink --manage eesave --prog=3000 --to atmelice --veri=e
```

With the usual abbreviation rules, the fact that the equal sign can simply be substituted by space, and the rule that later arguments override earlier ones, this is equivalent to

```
pyavrocd --manage eesave --prog-clock 3000 --tool atmelice --verify enable
```

Note that implicitly `@pyavrocd.options` is added to the end of the command line. This means that even if you cannot change the command line that invokes pyavrocd, because, e.g., pyavrocd is invoked by an IDE, you still can specify arguments that have precedence by using the configuration file `pyavrocd.options`.

3. Supported devices

3.1 Supported MCUs and Boards

This is the list of all AVR MCUs, which should be compatible with pyavrocd. MCUs tested with pyavrocd are marked bold. MCUs known not to work with pyavrocd are struck out.

3.1.1 MCUs with debugWIRE interface

ATtiny supported by *MicroCore*

- **ATtiny13**

ATtinys supported by the *ATTinyCore*

- **ATtiny43U**
- **ATtiny2313, ATtiny2313A, ATtiny4313**
- **ATtiny24(A), ATtiny44(A), ATtiny84(A)**
- **ATtiny441, ATtiny841**
- **ATtiny25, ATtiny45**, **ATtiny85**
- **ATtiny261(A), ATtiny461(A), ATtiny861(A)**
- **ATtiny87, ATtiny167**
- **ATtiny828**
- **ATtiny48, ATtiny88**
- **ATtiny1634**

ATmegas supported by *MiniCore*

- ~~**ATmega48**~~, **ATmega48A**, **ATmega48PA**, ATmega48PB,
- ~~**ATmega88**~~, **ATmega88A**, **ATmega88PA**, Atmega88PB,
- **ATmega168**, **ATmega168A**, **ATmega168PA**, **ATmega168PB**,
- **ATmega328**, **ATmega328P**, **ATmega328PB**

The ATmega48 and ATmega88 (without the A-suffix) sitting on my desk suffer from stuck-at-one bits in the program counter and are, therefore, not debuggable by GDB. They also act strangely when trying to switch to debugWIRE mode or back (you can easily brick them this way).

I suspect that this applies to all chips labeled this way. Even chips recently purchased through an official distributor had these issues. For this reason, pyavrocd will refuse to handle them.

Other ATmegas

- ATmega8U2, ATmega16U2, ATmega32U2
- ATmega32C1, ATmega64C1, ATmega16M1, ATmega32M1, ATmega64M1
- AT90USB82, AT90USB162
- AT90PWM1, AT90PWM2B, AT90PWM3B
- AT90PWM81, AT90PWM161
- AT90PWM216, AT90PWM316
- ATmega8HVA, ATmega16HVA, ATmega16HVB, ATmega32HVB, ATmega32HVBrevB, ATmega64HVE2

Supported Arduino boards

All Arduino boards equipped with one of the chips mentioned above can be debugged. This includes the **Arduino Uno R3**, **Arduino Nano**, and **Arduino Pro Mini** (as well as clones). Note that in all these cases, one must ensure that the RESET line is not connected to a capacitor and that the pull-up resistor on the RESET line is not stronger than 10 kΩ. This means that in most cases, the board must be physically changed before debugging is possible.

3.1.2 ATmegas with JTAG interface

Note that the MCUs are supported by pyavrocd. However, the cores have not been extended yet to allow for debugging with the Arduino IDE 2.

ATmegas supported by *MightyCore*

- ~~**ATmega16**~~, ATmega16A, **ATmega32**, ATmega32A
- ATmega164A, ATmega164P, **ATmega164PA**, ATmega324, ATmega324A, ATmega324PA, **ATmega324PB**, **ATmega644**, ATmega644A, ATmega644PA, ATmega1284, **ATmega1284P**

The ATmega16 MCUs (without an A-suffix) have a stuck-at-1-bit in the program counter, which does not show when reading the program counter in the debugger. But when retrieving return addresses from the stack, it is apparent. Since this confuses GDB, this MCU cannot be debugged.

ATmegas supported by *MegaCore*

- ATmega64, ATmega64A, ~~**ATmega128**~~, ~~**ATmega128A**~~
- ATmega640, **ATmega1280**, **ATmega2560**
- ATmega1281, ATmega2561
- ATmega165, ATmega165A, ATmega165P, ATmega165PA, ATmega325, ATmega325A, ATmega325P, ATmega325PA, ATmega645, ATmega645A, ATmega645P, ATmega645PA

- ATmega169, ATmega169A, **ATmega169P**, ATmega169PA, ATmega329, ATmega329A, ATmega329P, ATmega329PA, ATmega649, ATmega649A, ATmega649P, ATmega649PA
- ATmega3250, ATmega3250A, ATmega3250P, ATmega3250PA, ATmega6450, ATmega6450A, ATmega6450P, ATmega6450PA
- ATmega3290, ATmega3290A, ATmega3290P, ATmega3290PA, ATmega6490, ATmega6490A, ATmega6490P, ATmega6490PA
- AT90CAN32, AT90CAN64, AT90CAN128

The ATmega128 MCUs do not allow software breakpoints. For this reason, debugging is currently impossible because only 1 hardware breakpoint is permitted. This will change in the near future.

ATmega supported by *MajorCore*

- **ATmega162**

Other ATmegas

- AT90USB646, AT90USB647, AT90USB1286, AT90USB1287
- ATmega644rfr2, ATmega1284rfr2, ATmega2564rfr2
- ATmega64rfr2, ATmega128rfr2, ATmega256rfr2
- ATmega128rfa1
- ATmega16U4, **ATmega32U4**
- ATmega406

Supported Arduino boards

All boards with the chips listed above can be debugged. This is, in particular, the **Arduino Mega (2560)**, **Arduino Leonardo**, and **Arduino Micro**. Note that you should not connect any load to the JTAG lines. Furthermore, you must first enable the JTAG pins by SPI programming because on the Arduino boards, JTAG is disabled by default.

3.2 Supported hardware debuggers

Except for dw-link, this list is copied from the README file of pyedbglib. Boldface means that I have tested the debuggers and they work with pyavrocd.

- **MPLAB PICkit 4 In-Circuit Debugger** (when in 'AVR mode')
- **MPLAB Snap In-Circuit Debugger** (when in 'AVR mode')
- **Atmel-ICE**
- **Atmel Power Debugger**
- **JTAGICE3 (firmware version 3.0 or newer)**
- **EDBG - on-board debuggers on Xplained Pro/Ultra**
- **mEDBG - on-board debuggers on Xplained Mini/Nano**
- nEDBG - on-board debuggers on Curiosity Nano
- **dw-link - DIY debugger running on an Arduino UNO R3** (only debugWIRE)

My JTAGICE3, being the oldest one of the set of supported debuggers, is sometimes a bit shaky. In particular, with lower voltages and when the MCU has a clock less than 8 MHz, sometimes it emits errors when other debuggers work without a hitch. It is not clear whether these issues are with my sample or a general problem for these debuggers.

3.2.1 Switching to AVR mode

Note that Snap and PICkit4 need to be switched to 'AVR mode'. This can usually be accomplished as follows by using avrdude (>= Version 7.3):

```
avrdude -c snap_isp -Pusb -xmode=avr
```

With PICkit4, it is similar:

```
avrdude -c pickit4_isp -Pusb -xmode=avr
```

In both cases, you can check whether you were successful by typing the same command again. If you get the message that the debugger is still in 'PIC' mode, you need to flash new firmware first using MPLAB X.

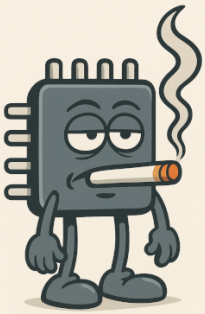
4. Caveats

4.1 Disclaimer

Note that, as is usual in life, there are some risks involved when using a tool. In particular, the debugWIRE interface can be a serious health risk to your MCU. Even worse, all other aspects of the debugging package have the potential of doing harm to the tested MCU, to the target board, or to the attached devices. Flash wear, for instance, is an issue when you use software breakpoints. Further, when using a debugger on an MCU that controls a system, stopping the execution of the controlling program might lead to erroneous behavior of the controlled system.

All in all, bear always in mind that the software is provided "as is", without warranty of any kind.

4.2 ~~Smoking~~ debugWIRE can be Dangerous to the Health of your MCU



While debugWIRE is an excellent concept, as it requires no GPIO sacrifice for debugging, it can be harmful to the MCU. Once the MCU has been brought into debugWIRE mode (using, for example, the `monitor debugwire enable` command), the RESET line can no longer be used to reset the chip, and it is impossible to use SPI programming to change fuses, particularly the debugWIRE enable (DWEN) fuse. If something goes wrong while entering debugWIRE mode, this could mean that you "bricked" your chip, since communication with the MCU is no longer possible. So, what can go wrong, and how can you resurrect the chip?

There are essentially five different scenarios:

1. The classical problem is a capacitor on the RESET line, either for noise suppression or as a means to implement auto-reset on an Arduino board such as the Uno. Similarly, a resistor that is too strong or a dedicated reset circuit could pose a problem. In these cases, one can change the DWEN fuse using SPI programming, but communication over the debugWIRE line (the RESET line) is impossible. The cure is apparent: Remove the resistor, capacitor, or reset circuit (or cut the trace to it). Afterward, it should be possible to connect to the MCU using the debugger (via pyavrocd).
2. All **ATmega48** and **ATmega88** variants without a P or A suffix exhibit unusual behavior. Note that this applies to MCUs purchased from official distributors. They have stuck-at-1 bits in the program counter, they refuse to let their DWEN fuse be set, or, if one is successful, it is impossible to leave debugWIRE mode again. By now, they are identified by pyavrocd before the DWEN bit is set, so that the only annoying thing about them is that they are not debuggable.
3. Another cause for trouble could be that the MCU is operated in an unstable electrical environment. This could mean that the supply voltage is fluctuating, an unstable external clock is used, blocking capacitors between (A)Vcc and (A)GND are missing, or, another classic, AVcc and/or AGND are not connected to the power rail. In these cases, unpredictable things can happen, and the MCU might not be responsive after having been switched into debugWIRE mode. In this case, repairing the fault, e.g., soldering a blocking capacitor between Vcc and GND, may or may not resolve the issue.
4. The MCU could be a non-genuine product. Since such products do not satisfy all the specifications of genuine MCUs, these MCUs might be able to enter debugWIRE, but then one is stuck. Or debugWIRE mode is not supported at all.
5. It could be that you can enter debugWIRE mode and debug your chip, but getting back to normal mode is impossible. This may be caused by setting some fuses when switching to debugWIRE mode that prevent the return to normal mode. If you unprogrammed `SPIEN` (Serial program downloading) and/or programmed `RSTDSBL`, the fuse to disable the reset line, then it is possible to leave debugWIRE mode, but you cannot use SPI programming afterward. When you let pyavrocd handle the fuses, this cannot happen.
6. There are apparently unknown reasons that can make a chip unresponsive when switching to debugWIRE mode. I have no idea why this happens. And usually, there is no easy recovery method (but see below).

If none of the above-mentioned recovery methods work, the last resort is *high-voltage programming*. This means that 12 volts are applied to the RESET line and then signals are sent to the MCU over different lines. If you have an MCU with a DIP footprint, you can use a breadboard high-voltage programmer or a specially designed "HV fuse programmer". For MCUs with an SMD footprint, you would need to buy a breadboard adapter.

Having said all that, my experience is that if you take care of the potential problems mentioned in points 1-5, it is unlikely that your MCU will get bricked. But it doesn't mean that it is impossible either. JTAG and UPDI are definitely the more robust debugging interfaces.

4.3 Flash Wear

When setting a breakpoint in a program, one usually does not think about the underlying mechanism that stops the program at the particular point where the breakpoint has been set. Technically, this can be done by *hardware breakpoints* or *software breakpoints*. A hardware breakpoint is implemented as a register that is compared to the actual program counter. If the PC is equal to the register value, execution is stopped. Usually, only a few such hardware breakpoints are available. On a debugWIRE device, there is just one. On AVR JTAG ATmegas, we have four; on UPDI MCUs, there are two. Software breakpoints are implemented by placing a particular trap instruction into the machine code. On AVRs, this is the `BREAK` instruction.

There are pros and cons to each type of breakpoint. Hardware breakpoints are faster to set and to clear because they do not involve reprogramming flash memory. Further, they do not lead to *flash wear* as software breakpoints do. However, as mentioned, there are usually only very few hardware breakpoints.

4.3.1 The flash wear problem

So, how severe is the flash wear problem? The data sheets state that for classic AVR MCUs, the guaranteed flash endurance is 10,000 write/erase cycles. For the more recent MCU with UPDI interface, it is only 1000 cycles! These are probably quite conservative numbers guaranteeing endurance even when the chips are operated close to the limits of their specification (e.g., at 50° C).

Let’s assume an eager developer who reprograms the MCU every 10 minutes with an updated version of the program and debugs using five software breakpoints that she sets and clears during each episode. That will probably result on average in 3 additional reprogramming operations on an individual page, leading to 4 such operations in 10 minutes or 192 such operations on one workday. So, she could hit the limit for the modern AVR MCUs after one working week already. The classic AVRs can be used for 10 weeks. This holds only if she does not set and clear breakpoints all the time, but is instead rather careful about doing so. Further, the software debugger needs to make sure not to require superfluous breakpoint set/clear operations.

Different GDB servers vary in their ability to minimize flash wear, with pyavrocd being very competitive. However, all in all, as Microchip states, you should not ship MCUs to customers that have been used heavily in testing.

4.3.2 Using only hardware breakpoints

Can it be a solution to use only hardware breakpoints? It will definitely reduce flash wear to zero (well, except for reprogramming the target). However, it can also be very inconvenient because there are only a few of them. You can force the use of only hardware breakpoints by the `monitor` command:

```
monitor breakpoint hardware
```

One must be aware, though, that there is a slight problem here when single-stepping. If GDB continues from a location at which a breakpoint is set, it will assert all user-requested breakpoints except the one at this particular location. Then GDB requests a single step, and afterwards asserts the breakpoint at the original location. Since a stepping-over operation uses a temporary breakpoint, this can lead to the situation where, after starting the stepping-over operation with a single step, it is discovered that too many breakpoints are necessary to complete the step-over operation. This is not a disaster, but it is very confusing. Unfortunately, there is no way to detect this problem early enough at the level of the GDB server. And there does not seem to be an easy way to solve the issue on the GDB level.

So, when using only hardware breakpoints, do not use the maximum number of breakpoints or refrain from the stepping-over operation (called `next` in GDB).

5. About

5.1 Release Notes

No release yet.