

Arduino Debugger

Author: Jan Dolinay; dolinay [at] utb.cz

Last revised: April 14, 2021

Table of contents

Introduction	3
Release notes	4
Note about this documentation.....	5
Abbreviations and terms used in this document	5
Introduction: How to use this document.....	5
Basic information about the tools.....	6
Steps needed to debug your program	10
Which way to choose	10
Problems with GDB (avr-gdb) included with Arduino	10
Important limitations of the debugger	11
Configuring project for Arduino Uno vs. Mega	12
Using breakpoints in flash memory	13
RAM vs Flash breakpoints – pros and cons	14
RAM breakpoints.....	14
Flash Breakpoints	14
Notes	15
Loading the program via the debugger.....	18
Why load the program through the debugger?.....	18
Why not load the program through the debugger?.....	18
1. Step 1: Preparing Eclipse for Arduino development	19
1.1 Install Arduino	19
1.2 Install Eclipse.....	19
1.3 Install AVR Eclipse plugin	19
1.4 Install Mingw tools	20
1.5 Configure the AVR Eclipse plugin	21
1.6 Install GDB Hardware Debugging launch configuration into Eclipse.....	24
1.7 Notes about the tools	24
Alternative setup with Atmel Studio (Option 2)	25
1.2 Install Atmel Studio	25

Alternative setup without Atmel Studio but with Atmel AVR8 Toolchain (Option 3)	25
Install Atmel AVR Toolchain	26
Configuration of the AVR Eclipse plugin with Atmel Toolchain tools	26
2. Step 2: Create your first project for the Arduino board	28
3. Step 3: Add debugger support to your program	32
Alternative, more portable way for adding the debugger support	34
4. Step 4: Connect to your program with the debugger	36
Important note on the TCP-Serial proxy	36
Step-by-step instructions	36
Working with the debugger	40
Exercise – more complex program	44
5. Step 5: Set up a project in Eclipse with Arduino functions	46
6. Step 6: Enable debugger support in a program with Arduino functions	49
Alternative ways of solving the multiple definitions of vector1 error	52
Option 1 - Comment out the code	53
Option 2 - Use conditional compilation block	53
Opening example projects	54
Set up your development environment	54
Create path variables	54
Import example projects	55
Debug example projects	57
Direct serial connection	58
Connection via TCP-to-Serial proxy	58
Debug session	61
How to enable breakpoints in flash memory	62
Option 1 – using the custom bootloader provided with this package	63
Step 1 - Update the bootloader in your Arduino board and change the fuses	63
Step 2 - Change the option in avr8-stub.h to use flash breakpoints	66
Step 3 - Add files app_api.h and app_api.c to your project	66
Option 2 – using the standard Optiboot bootloader	66
Step 1 - Update the bootloader in your Arduino board and change the fuses	67
Step 2 - Change the option in avr8-stub.h to use flash breakpoints	67
Debugging your program with flash breakpoints	68
Uploading your program via the debugger	69

Step 1 – Configure the debug driver	70
Step 2 - Configure the Debug configuration to load the program	70
Uploading and running the program.....	71
Understanding the load function	72
Troubleshooting problems with debugging	73
Problem: Debug session fails to start.....	73
Problem: The program is not jumping at some line or is jumping somewhere I don't want it to....	74
Problem: The program does not stop on a breakpoint.....	75
Problem: The disassembly cannot be displayed	75
Problem: Loading the program via the debugger ends with error "Load failed"	75
Problem: Loading the program via the debugger ends with some error or hangs.....	75
Arduino library	76
Using this debugger in Visual Studio Code.....	76
Step 1 – Install the avr-debugger library.....	76
Step 2 – Add the library to your program	76
Step 3: Turn off compiler optimizations.....	77
Step 4: Create launch.json file – launch configuration	79
Viewing variables	83
Breakpoint in main	85
Create assembly listing	86
Using this debugger with PlatformIO	86
Step 1 – Install PlatformIO into VSCode.....	86
Step 2 – Create PlatformIO project for your Arduino	86
Step 3 - Install avr-debugger library	87
Step 4 – Update the project configuration for the debugger	87
Step 5 – Modify the code to work with the debugger	88
Step 6 – Debug your program	89
Using TCP-to-Serial proxy (Windows only).....	92
Debugging code with millis and micros.....	94

Introduction

This documentation describes source level debugger for Arduino based on GNU Debugger (GDB). It uses GDB stub mechanism for implementing the debugger. This means a piece of code (stub) is added to your Arduino program. This code then communicates with the GDB debugger. No external

programmer or modification of the Arduino board is required. Eclipse or Visual Studio Code can be used as a graphical frontend for debugging.

Release notes

April 2021

- Added code to use TIMERO as an alternative to using the watchdog timer (AVR8_USE_TIMER0_INSTEAD_OF_WDT = 1).
- Added a new possibility for generating a software interrupt, which does not use any external pin (AVR8_SWINT_SOURCE = -1).
- Shortened the runtime of the debugging ISR from 40 µs to less than 4 µs.

April 2021

- Added code for ATMega1284(P).

March 2021

- Fixed debug_message function; it blocked the debugged program from continuing; now it prints properly even when debugged program is running freely.
- Updated (simplified) instructions for using this debugger with PlatformIO and added info about using millis() and micros() Arduino functions while debugging.

July 2020

- Added information on using standard Optiboot bootloader for flash breakpoints. This allows using flash breakpoints also on Arduino Mega with the Optiboot.

May 2020

- Added information about using the debugger with PlatformIO, thanks msquirogac for suggesting this and providing an example project and modifications for the debugger and bootloader to build with PlatformIO.

June 2019

- Added information about Arduino library and using this debugger in Visual Studio Code IDE.

January 2018

- This document updated based on tests with new versions of the tools – Eclipse Oxygen (64 bit), Arduino IDE 1.8.5, AVR toolchain v3.5.4.

July 2017

- Added support for writing breakpoints into flash memory – Arduino Uno only. This allows debugging program without the significant drop of execution speed which happens when using breakpoints in RAM.
- Added support for loading new program from the debugger – Arduino Uno only. This allows uploading changed program to the MCU and debugging with single click – no need to upload separately using AVRdude.

January 2017

- Added support for Arduino Mega board with ATmega1280 and ATmega2560 MCUs.
- Example programs reorganized and renamed. The name now contains Arduino variant so that example projects for different variants can be imported into single eclipse workspace.
- Fixed bug for ATmega328 (Uno) - the debugger now works for programs larger than 16 kB.
- Documentation updated to describe also direct serial communication with the debugger (without the TCP-to-COM proxy) which seems to work on Windows 10 and with some boards also on Windows 7.

June 2015

- First release. Arduino Uno supported.

Note about this documentation

This document is gradually updated with information related to new versions of the tools used. Some screenshots are updated in this process but not all of them, so your actual screens can be a little different.

Abbreviations and terms used in this document

Arduino MCU Framework – the software for the microcontroller (MCU) shipped with Arduino, that is the core library of Arduino with functions like digitalWrite, etc. Also Arduino software library.

Arduino software library – see Arduino MCU Framework.

Debug driver – (also debug stub); the code which communicates with the debugger. This code is added into your project in Eclipse. It is a program library. This driver is located in avr8-stub.c and avr8-stub.h files.

MCU – microcontroller. The “chip” which is the main part of your Arduino board.

Introduction: How to use this document

This document should help you use the Arduino debugger. This debugger can be used in various IDEs. The original revision of this document assumed that you write and debug your programs in the Eclipse IDE and the step-by-step instructions at the beginning deal with configuration for Eclipse.

In the June 2019 revision chapters were added which describe use of this debugger as an Arduino library in **Visual Studio Code** IDE. These can be found in the chapters [Arduino library](#) and [Using this debugger in Visual Studio Code](#).

In May 2020 revision a chapter was added on using it in **PlatformIO** IDE (which is built on top of Visual Studio Code), see [Using this debugger with PlatformIO](#) chapter.

Setting up Eclipse and other tools is quite complicated, but at the time I started this project I was not aware of any easier way to have an IDE capable of debugging Arduino code. As of now (May 2020) I think using Visual Studio Code (VSCode) is much easier. With VSCode there are two ways documented here: (i) using VSCode with Arduino Extension and (ii) using VSCode with PlatformIO. In my opinion the easiest way is to use Visual Studio Code with Arduino extension, but note that this option uses the Arduino build system (Arduino CLI) which has serious limitations in setting up the build options. If you want better control of the build process and easy per-project configuration try the PlatformIO option. If you prefer the classic IDE, use Eclipse.

The first chapters in this document (“Step n...”) are organized in a step-by-step order, so you can follow the chapters as they are. You can also skip any of them if you feel confident you do not need this step. There are some **generally useful chapters** about the limitations and troubleshooting at the end of this document.

For Eclipse these main steps are covered:

1. Installing and setting up the tools needed to develop and debug your program for Arduino in Eclipse.
2. Building and uploading simple program in C language into the Arduino board, without using the Arduino software library.
3. Enabling the debugger functionality in this program and debugging it.
4. Enabling the Arduino software library functions in your programs in Eclipse.
5. Enabling the debugger functionality in this program and debugging it.

I recommend that you follow these steps including the simple program in C without Arduino software library; this is not a detour but rather a check point. You need to be able to build program in plain C language to be able to build one with the Arduino functions.

Before diving into the step-by-step instructions below, please read the following sections to know what you are doing.

Basic information about the tools

For a typical Arduino user, only one tool is needed – the Arduino IDE.

But you cannot debug your program in the Arduino IDE; it does not provide such functionality. You need an IDE with graphical interface for debugging. The most popular alternatives for Arduino development are probably the Eclipse and Atmel Studio.

I do not think Atmel Studio can be used with the debugger presented here. At least I was not able to set up the GDB debugger in Atmel Studio. It seems there is no access to the settings needed to make

this work. This leaves us with the Eclipse as the only option. Or the command line use of GDB, if you prefer. But I will not describe this case here.

So, we need to write, build, upload and debug our programs for Arduino in Eclipse.

June 2019 Note: Or we can use Visual Studio Code IDE – please see the Introduction: how to use this document chapter above for more information.

Unfortunately, there is no direct support for Arduino in Eclipse. Eclipse is not a “single purpose” IDE; it can be used for all kinds of projects in several languages etc. So you cannot just download Eclipse and start coding for the Arduino (in fact, for Atmel microcontrollers). You need to add some extra tools.

All needed tools are summarized in the following table. Each column represents one tool and a row represents the package (program) in which the tool can be found. Our goal is to have all the tools. You do not need to install all the packages. You are free to combine the packages as you wish to reach this goal.

Package	GUI for debugging	Compiler and linker	Make tool	AVRDude uploader	GDB debugger	Notes
Eclipse (for C/C++ developers)	Yes	-	-	-	-	The IDE we are using.
AVR Eclipse plugin	No	-	-	-	-	Needed to build AVR projects in Eclipse
Arduino IDE (1.8.1)	No	Yes	No	Yes	Yes	Needed for uploading the programs.
Atmel AVR toolchain	No	Yes	No	No	Yes	Is included in Atmel Studio but also available separately.
Atmel Studio	Not usable	Yes	Yes	No	Yes	We do not use the IDE, but contains many tools
Mingw tools	No	?	Yes	?	?	Provides the make tool.

In the following chapters these possible configurations are covered:

- Option 1: Easy Setup based on Arduino IDE and MinGW tools
- Option 2: Easy setup based on Atmel Studio and Eclipse
- Option 3: Setup without Atmel Studio but with Atmel AVR8 Toolchain.

Option 1 was added after Arduino IDE 1.6.5 IDE appeared, because it now again contains the GNU debugger (avr-gdb). Unfortunately, it still does not contain the make tool (make.exe) so this tool must be obtained by installing MinGW tools.

Option 2 with Atmel Studio is also easy, but requires more disk space – you need to install Atmel Studio.

Option 3 requires more steps but less disk space and in principle can be used as a guide for Linux users with some modifications.

The following tables show which packages are needed in each option to obtain the required tools. **We need to have “Yes” in all the columns.**

Option 1 – Using Arduino IDE and MinGW

Package	GUI for debugging	Compiler and linker	Make tool	AVRDude uploader	GDB debugger
Eclipse (for C/C++ developers)	Yes	-	-	-	-
AVR Eclipse plugin	No	-	-	-	-
Arduino IDE	No	Yes	No	Yes	Yes
Atmel AVR toolchain	No	Yes	No	No	Yes
Atmel Studio	Not usable	Yes	Yes	No	Yes
Mingw tools	No	No	Yes	No	No

Option 2 – Using Atmel Studio

Package	GUI for debugging	Compiler and linker	Make tool	AVRDude uploader	GDB debugger
Eclipse (for C/C++ developers)	Yes	-	-	-	-
AVR Eclipse plugin	No	-	-	-	-
Arduino IDE	No	Yes	No	Yes	No
Atmel AVR toolchain	No	Yes	No	No	Yes
Atmel Studio	Not usable	Yes	Yes	No	Yes
Mingw tools	No	No	Yes	No	No

Option 3 – Without Atmel Studio but with Atmel AVR8 Toolchain

Package	GUI for debugging	Compiler and linker	Make tool	AVRDude uploader	GDB debugger
Eclipse (for C/C++ developers)	Yes	-	-	-	-

AVR Eclipse plugin	No	-	-	-	-
Arduino IDE	No	Yes	No	Yes	No
Atmel AVR toolchain	No	Yes	No	No	Yes
Atmel Studio	Not usable	Yes	Yes	No	Yes
Mingw tools	No	No	Yes	No	No

Note: The fact that so many tools are needed is the result of some unfortunate decisions of the Arduino team which removed the make utility from the Arduino package. Ideally there would be some package with most of the tools together, as WinAVR was before it stopped being updated.

Steps needed to debug your program

These are the general steps on the way to debugging your program for Arduino.

Step 1: Prepare the Eclipse development environment

Step 2: Create program for the Arduino board and upload it to the board

Step 3: Add the debugger code (driver) to your program

Step 4: Connect to the program with the debugger (from Eclipse) and debug it

These steps will allow you to debug your program for the Arduino Uno board written in C/C++ language, but without the Arduino software library.

When this works, there are two more steps:

Step 5: Create program in Eclipse with Arduino functions

Step 6: Enable debugger support in a program with Arduino functions

The following sections will provide detailed instructions on these steps.

Which way to choose

There are three ways described, which may be confusing. Here is some help.

- If you have no preference, use the option 1.
- If you have plenty of disk space and do not mind installing big program you will not use, use the option with Atmel Studio. I think it will be more “future-proof” than Arduino IDE based solution.
- If you do not want to rely on tools provided by Arduino, but want to save disk space, use option with Atmel AVR8 toolchain without Atmel Studio. Use this option also if you encounter problems with running the debugger (avr-gdb), see the next section.

Problems with GDB (avr-gdb) included with Arduino

It seems that the avr-gdb included in newer Arduino IDE packages does not work because of some missing DLLs. In my experiments, I was able to use the avr-gdb from Arduino 1.6.8, but not from 1.6.10 and 1.8.1. In general I recommend the following procedure:

- Try to set up everything with build tools from Arduino IDE.
- If the debugging does not work (Eclipse reports errors when starting debug session such as “could not get gdb version...”), try to run the avr.gdb.exe directly by double clicking it in your file explorer. If it does not start, you are experiencing the above mentioned problem.

- In this case install the Atmel AVR toolchain for AVR 8-bit. It is only about 15 MB and the avr-gdb from this toolchain works fine. In this case change the paths in AVR eclipse plugin to point to this toolchain instead of to Arduino, see Configuration of the AVR Eclipse plugin with Atmel Toolchain tools.

Note: It is always good idea to **use matching compiler and debugger**. So if you use the avr-gdb from the Atmel AVR toolchain, use also the build tools from this toolchain – set the paths in AVR Eclipse plugin configuration to the Atmel AVR toolchain instead of Arduino IDE paths as described above.

Important limitations of the debugger

One external interrupt pin must be reserved for the debugger

When debugger is used, one pin must be reserved for its use. It can be any pin with external interrupt function (INT0, INT1, etc..). For Arduino Uno this can be either digital pin 2 or 3 (PD2 or PD3 pin of the MCU). For Mega there are more options. By default, INT0 pin (Uno pin 2, Mega pin 21) is used. To change this, change the value of `AVR8_SWINT_SOURCE` define in `avr8-stub.h` file.

Tip: for Arduino mega you can use INT6 or INT7 (define `AVR8_SWINT_SOURCE` 6 or 7). These pins are not connected on the Arduino Mega board so you will not waste any usable pin.

The debugger needs to handle the INTx interrupt which is in conflict with the `attachInterrupt` function in Arduino software library. Small modification of the Arduino library code is required to build the program. This is described in Step 6 section of this document.

Reason: The debugger needs to generate software interrupts to work. Unfortunately, in the AVR processor there is no dedicated instruction for this and the interrupt must be generated using external interrupt which requires an I/O pin.

Since April 2021, there is the possibility of getting away without using an external interrupt pin. When you set the value of the compile time constant `AVR8_SWINT_SOURCE` to -1, then the `TIMER0_COMPA` interrupt is used as a software interrupt. However, it only works for flash breakpoints and it has its own problems. The main problem is that this interrupt has a lower priority than all the external interrupts mentioned above, the pin change interrupts, and most of the timer interrupts. Because of this, it may happen that when single stepping, the debugger might skip a line (because the interrupts with higher interrupts are served first). If you badly need all pins associated with external interrupts, you are probably happy that there is an alternative. However, you should be aware of the mentioned shortcomings.

Serial communication cannot be used in your program

The Arduino Serial class (or any other library which uses the UART module) cannot be used in your program together with the debugger.

Arduino software library is in conflict with this use of UART by the debugger (to be exact it wants to handle the UART interrupt which the debugger library also needs). It is necessary to exclude the file which implements Serial class from the build in eclipse. Please see Step 5 section for details.

Reason: The debugger needs the UART (serial communication) module to communicate so it cannot be used by the user program. You can output text messages to debugger console using `debug_message` function which is part of the debugger library.

When using flash breakpoints the watchdog cannot be used

The watchdog module is used by the debug driver when it is configured to insert breakpoints into flash memory (as opposed to the default configuration with breakpoints in RAM). So it is not possible to use the watchdog in your application while using the debugger. Arduino library does not use watchdog so this causes no conflicts. If you need to use watchdog in your application, enable the code which works with the watchdog only after the application is debugged and the debugger is removed or disabled. As an alternative, since April 2021 there exists one configuration of the debugger that does not use the watchdog timer (see below under TIMERO interrupts vs WDT interrupts).

When using RAM breakpoints the Arduino functions `millis()` and `micros()` do not work

With the RAM breakpoints the `millis()` and `micros()` Arduino functions will only return meaningful values when the program is running at full speed (that means you clicked the Continue button in the debugger) with no breakpoints set. If you step through the program or let the program run (continue) but there is at least one breakpoint set, the functions will return invalid values.

The solution is to use flash breakpoints. In this mode the timer and other interrupts are executed normally while the program runs between breakpoints. However, timing of the functions will be affected because while the program is stopped in the debugger, the time interrupt is also stopped.

For more info please see the [Debugging code with `millis` and `micros`](#) section.

Configuring project for Arduino Uno vs. Mega

The debugger now supports Arduino Mega board with ATmega1280 and ATmega2560 MCU and also boards with the ATmega1284(P). The procedures in this document describe configuration for Arduino Uno. For Arduino Mega the following changes apply:

- When creating new project in Eclipse select the proper MCU – ATMega1284(P), ATmega1280 or ATmega2560. The clock speed is always 16000000 (16 MHz).
- When configuring the upload command in AVR eclipse plugin (AVRDude configuration, see section 1.5) use the following settings:
 - For ATmega2560 profile “Wiring” and baud rate 115200.
 - For ATmega1280 profile “Arduino” and baud rate 57600.

- For ATmega1284(P) profile “Arduino” and baud rate 115200.

Other than this the settings are the same for Mega and Uno. The code in avr-stub files uses conditional compilation to adjust to the MCU selected in your project, so there is no change required.

The baudrate for communicating with the debugger is 115200 for ATmega2560 and 57600 for ATmega1280.

Note: For ATmega2560 the debugger does not properly display the call stack if the program size is over 128 kB (50% of flash memory).

Using breakpoints in flash memory

This chapter provides basic information about the optional feature of this debug driver which allows debugging program without affecting its execution speed. Instructions on how to actually use this feature can be found in chapter How to enable breakpoints in flash memory.

The new version of this debugger (as of July 2017) supports writing breakpoints into flash memory. In this document these are called *flash breakpoints* while the other option is called *RAM breakpoints*. Even though these names are not quite correct, I use them throughout this documentation and the source code for the lack of better name.

There are two options how to implement breakpoints in an AVR debug driver:

- Option 1 - store the address at which the program should stop in a variable. Then after executing every instruction of the program compare this variable with the current location of the program (the program counter register, PC). If you find a match, stop there and notify the debugger – let the user know that the program stopped on the breakpoint.
- Option 2 – replace the instruction at the address where the program should stop with a special instruction which causes “jump” into the debug driver so that it can notify the debugger.

In earlier versions of this debug driver only the option 1 was available. The disadvantage of this option is that the program must be stopped after executing every instruction to compare the PC with the addresses of desired breakpoint. This slows down the debugged program considerably. But in fact it is hardly noticeable unless you debug code with longer delays implemented by incrementing/decrementing a counter.

The advantage of option 2 is that there is no such slowing down. The program stops itself on the breakpoint. The drawback is that it requires replacing the bootloader in your Arduino.

The advantages and disadvantages are summarized below so that you can make your own decision.

It is recommended to start with the RAM breakpoints in all cases.

The reason is that using flash breakpoints requires replacing the bootloader in your Arduino, which is yet one more thing which may go wrong in the process of setting up your environment for

debugging. So it is better to start the easy way. It is not a detour, it is just taking the road step by step.

Once you are able to debug your program with RAM breakpoints AND if you decide that you can benefit from using the flash breakpoints, take the extra step to enable them in your program.

RAM vs Flash breakpoints – pros and cons

RAM breakpoints

cons

- The program runs slowly when one or more breakpoints are inserted.
It runs perhaps 100 times slower than without breakpoints. In many cases, this makes no difference, you will not notice anything. But if your program contains busy loop delay which, for example, decrements some variable to 0 to obtain desired delay, such a delay will be stretched by a factor of 100 or more.
- You cannot place a breakpoint into an interrupt service routine (ISR).
The RAM breakpoints use external interrupt, e.g. INT0, whose ISR is executed after every instruction in the main code. No other ISR is executed in this situation. So if there is any breakpoint in your program, no ISRs are executed. With flash breakpoints it is possible to stop the program in an ISR but note that the ISR must be declared as `ISR_NOBLOCK`. This means that the interrupts are enabled while the ISR is executed. This makes it possible for the debugger to interrupt the execution of the ISR when it hits a breakpoint.
- The Arduino functions `millis()` and `micros()` do not work unless the program runs with no breakpoints set. The `millis` returns the same value all the time, the `micros` returns value which oscillates around some value within about 1000. The `delay()` function works thanks to “lucky” implementation but the time is stretched about 4 times. For more info please see the [Debugging code with millis and micros](#) section.

pros

- There is no wear of the memory when inserting and removing breakpoints.
See the cons of flash breakpoints for explanation.

Flash Breakpoints

cons

- The flash memory is overwritten often when debugging with flash breakpoints.
The flash memory in the Atmel AVR MCUs used in Arduino should survive about 10 thousand rewrites. When breakpoint is inserted or removed, part of the memory needs to be rewritten. To make things worse breakpoints are inserted and removed not only when you actually insert/remove a breakpoint in the IDE but also automatically when you continue from a breakpoint (the original instruction needs to be restored and executed) and also sometimes when you step through the code the debugger will insert temporary breakpoint, for example to stop the program after stepping over a function. In worst case you, you can

expect one write to flash memory for each step/continue click you do in the IDE. This seems scary but it is still usable and similar to using the debugWIRE hardware on-chip debugger module, please see the notes below. You can check the number of writes to flash when debugging, this is described in the section on enabling flash breakpoints.

pros

- The program runs at full speed.
There is no need to stop the program after each instruction to see if it reached a breakpoint. The program will stop itself when breakpoint is encountered. There is small overhead in periodical testing whether the program stopped on a breakpoint but this has no noticeable effect on the speed of the program.
- It is possible to place breakpoints into interrupt service routines (ISR) if interrupts are enabled in the ISR. If you place a breakpoint into an ISR at a point where interrupts are globally disabled, the program will hang on this breakpoint without ever getting back to the debugger. See the notes below for example code.
- It is possible to use the Arduino millis() and micros() functions even though they will not return absolutely correct values.

TIMER0 interrupts vs WDT interrupts – pros and cons

When using flash breakpoints, the default is to employ the watchdog timer to check whether a breakpoint has been reached. Since April 2021, there exists an alternative to this method, namely, to use TIMER0, which is the timer used in Arduino cores to count milliseconds. You can enable this method by setting the compile time constant `AVR8_USE_TIMER0_INSTEAD_OF_WDT = 1`. Let us have a brief look at the pros and cons of this new method:

cons

- It requires that an Arduino core is used and that the timer is running.
- It uses the output compare register OCR0A and the corresponding interrupt TIMER0_COMPA. These resources are not used by Arduino cores. However, it means that OCR0A cannot be used by the user program.
- The TIMER0 interrupt is raised every millisecond while the WDT interrupt is raised only every 500 ms. In other words, 500 times more CPU time is “wasted.” However, since the interrupt routine has been optimized, only 3µs are spent in this routine for every call. This is the same order of magnitude as the Arduino’s millis interrupt. So the additional 0.3% of runtime are probably hardly noticeable.

pros

- One can use and debug watchdog interrupts (see the discussion about breakpoints in ISRs below).

Notes

Breakpoints in ISRs

It is only possible to place breakpoints into an ISR if you use flash breakpoints and if interrupts are globally enabled in the interrupt routine, e.g., by declaring the ISR as *non-blocking*. This is because the debugger uses interrupts internally to detect the situation when the program stops on a breakpoint. If interrupts are disabled (which they are by default while any ISR is executed) the debugger cannot stop the program.

Example of ISR which can be debugged:

```
ISR(TIMER0_COMPB_vect, ISR_NOBLOCK )
{
    // toggle the LED
    PORTB ^= _BV(LED_PIN);
    result += 5;
    if ( result == 10 )
        result = 0;
}
```

One should note, however, that debugging ISRs is a very delicate matter. If you declare your ISR as non-blocking, you might run into the problem that the same ISR is called again, which is something ISRs are usually not prepared for. For instance, the example ISR above is called every millisecond when it is used in the Arduino framework. So, when you continue from a breakpoint in the above ISR, the first thing that will happen is that the ISR is called again (inside the current invocation). So, you might end up at the same breakpoint (but with one more invocation on the stack).

If you want to debug an ISR, then you should take some precautions such as disabling the ISR-specific interrupt, clearing the interrupt flag and only then enabling interrupts globally. Before leaving the ISR, one should disable interrupts again and then enable the ISR specific interrupt, so that it can be served again after leaving the ISR. So, a safer way to debug the above ISR would be to include some conditional code as follows:

```
ISR(TIMER1_COMPB_vect)
{
#ifdef DEBUG
    TIMSK0 &= ~_BV(OCIE0B); // disable OCR0B interrupt
    TIFR0 |= _BV(OCF0B);    // clear potentially set interrupt flag
    sei();                  // enable interrupts globally
#endif
    // toggle the LED
    PORTB ^= _BV(LED_PIN);
    result += 5;
    if ( result == 10 )
        result = 0;
#ifdef DEBUG
    cli();                  // disable interrupts globally
    TIMSK0 |= _BV(OCIE0B); // enable OCR0B interrupt
#endif
}
```


If you want to debug the watchdog interrupt (in case you have enabled `AVR8_USE_TIMER0_INSTEAD_OF_WDT`), these precautions are not necessary, though, because the debugger calls internally `wdt_reset()` as often as possible in order to reset the WDT timer.

Flash memory wear

To minimize the wear of the flash memory when using flash breakpoints I recommend using the following guidelines (the first two are taken from Atmel documentation for debugWIRE hardware debugging):

1. Try to minimize on the number of breakpoint additions and removals, as each one require a FLASH page to be replaced on the target
2. Try to add or remove a small number of breakpoints at a time, to minimize the number of FLASH page write operations
3. Use only as many breakpoints as really needed. Often you will only need one breakpoint at a time. Consider that the debugger must always update all the breakpoints when the program is run, not just the one which will be hit next – the debugger does not know which one will be hit next.
4. When you no longer need a breakpoint, remove it. This is not to say that you should remove and re-insert a breakpoint each time the program stops on it, but when you know that you will be dealing with other part of the program and the breakpoint will not be needed for some time, remove it. This may seem in conflict with guideline 1, but most of the time breakpoints are inserted and removed automatically when you step through the code. It only takes one write to remove a breakpoint upon your command while this same breakpoint would be unnecessarily removed and inserted automatically every time you let the program run (see guideline 3).

Disabled interrupts and breakpoints

In the flash breakpoint mode you should never set a breakpoint into code which runs with interrupts disabled – that is the code after `cli()` instruction and before the interrupts are re-enabled.

If you insert a breakpoint the program will hang/freeze and stop communicating with the debugger. This is because the debugger will replace the original instruction at the point of a breakpoint with an infinite loop and then check if the program stopped at the breakpoint in a watchdog interrupt service routine. If the interrupts are disabled, the program will just execute the infinite loop infinitely, no watchdog ISR will be executed. The infinite loop will remain in the program memory even after restart/power off. You will need to upload the program again to make it work again.

When using RAM breakpoints there is no harm in setting a breakpoint like this. The program will just not stop at the breakpoint but the communication with the debugger will be restored as soon as breakpoints are enabled.

Loading the program via the debugger

Together with the option to write breakpoints to flash there is also new option to load the program into the MCU using the debug driver instead of AVRdude.

This option is not functionally connected with the flash breakpoints, you can use it with RAM breakpoints as well. What this option and flash breakpoints have in common is that they both require replacing the bootloader in your Arduino.

If you replace the standard Arduino bootloader with the one provided with this debug driver, you can use the flash breakpoints and program load together or any of these features alone.

Why load the program through the debugger?

Because it is easier and faster to debug your program this way. Basically, the workflow can be just like this:

- Edit the code
- Click Debug button (or rather expand the Debug button menu and select debug configuration you want to run from the list). The code is built automatically before load.
- Debug the program

If you load the program via AVRdude the workflow is as follows:

- Edit the code
- Build the code
- Upload the code
- Click Debug button (or rather expand the Debug button menu and select debug configuration you want to run from the list).
- Debug the program

May 2020 update: When using this debugger with PlatformIO you do not need to use this feature and you are still able to load and debug the program with single click. This is because the PlatformIO debugger can automatically load the program using AVRdude before debugging.

Why not load the program through the debugger?

There are two obstacles.

- You need to replace the bootloader in your Arduino. This can be challenging for some people and you need additional hardware to do this – an ICSP programmer or another Arduino to work as such a programmer.
- When loading through the debugger the flash memory wears faster than when loading through AVRdude. If you do not know about flash memory wear please see the “cons” section for RAM vs Flash breakpoints – pros and cons section. I assume that when uploading via AVRdude the memory is written once. When uploading via the debugger the memory is written twice. So in theory the MCU should survive 10 thousand uploads via AVRdude and only 5 thousand via the debugger. For most people both these numbers are high enough, but you should be aware of this.

1. Step 1: Preparing Eclipse for Arduino development

This section describes how to set up Eclipse to be able to develop programs for Arduino boards with Atmel AVR MCUs. As mentioned above, there is more than one option how to do this. This section covers the easiest one which requires installing Arduino IDE 1.6.5 or later, Eclipse with AVR Eclipse plugin and small subset of MinGW tools.

There are two other options described in the next sections.

1.1 Install Arduino

From <http://www.arduino.cc/en/Main/Software> download the Arduino software. Current version (January 2018) is 1.8.5. You can either download installer or a zip file. This text assumes you use the zip file.

Extract the downloaded zip file into any folder on your computer. The following text assumes you extracted it to c:\programs\arduino-1.8.5

To start the Arduino IDE, run arduino.exe from the folder. You do not need to run it now.

1.2 Install Eclipse

From <https://eclipse.org/downloads/> download the “Eclipse IDE for C/C++ Developers”. Current version (January 2018) is Oxygen.

There is no installation needed. You download a zip file which you can extract into any folder on your computer, for example, to c:\programs\eclipse.

Go to the eclipse folder and run eclipse.exe to start the IDE. You may want to create link on your desktop for quick access.

When Eclipse starts, you need to select location for your workspace. This is a folder on your computer where all your projects will be located. Later you can use several workspaces and change the default location but for now I recommend just accepting the default offered by Eclipse and moving on.

1.3 Install AVR Eclipse plugin

Start the Eclipse IDE and in the main menu select Help > Install New Software...

In the Work with box enter this update site address: <http://avr-eclipse.sourceforge.net/updatesite> and press Enter on the keyboard.

After a while the list below will display AVR Eclipse Plugin item. Select it by checking the box next to the AVR Eclipse Plugin name and click the Next button below.

Follow the wizard to install the plugin. It takes quite a while. You will need to agree with license agreement and confirm installing of unsigned plugin. After the installation, accept the offer to restart Eclipse.

TIP: You can find tutorial with pictures on the AVR Eclipse plugin website at:

http://avr-eclipse.sourceforge.net/wiki/index.php/Plugin_Download

1.4 Install Mingw tools

The MinGW tools are needed to obtain the GNU make utility. If you have Atmel Studio 6.x installed on your computer, you can skip this step; you will find make.exe in the [Atmel Studio location]\shellUtils. We will need this path in AVR Eclipse plugin configuration as described later.

To install MinGW:

From <http://www.mingw.org/> follow the Downloads link on the left. This will take you to <http://sourceforge.net/projects/mingw/files/>

Near the top of the page you will find "Looking for the latest version?" line and a link to download.

Download the mingw-get-setup.exe (direct link:

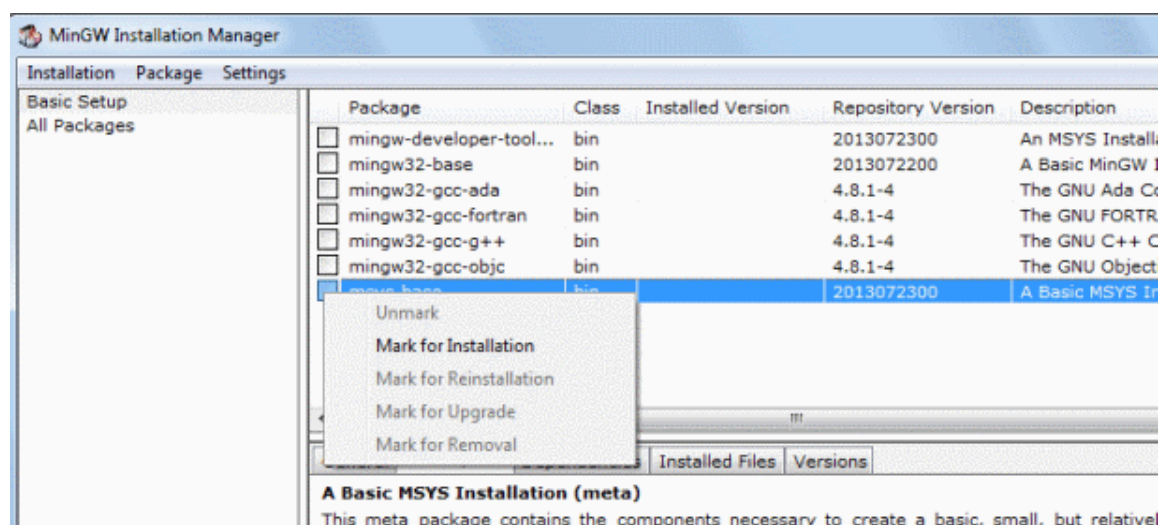
<http://sourceforge.net/projects/mingw/files/latest/download?source=files>).

Run the setup program. In the first screen leave all settings at defaults, change the Installation Directory if you prefer and click Continue.

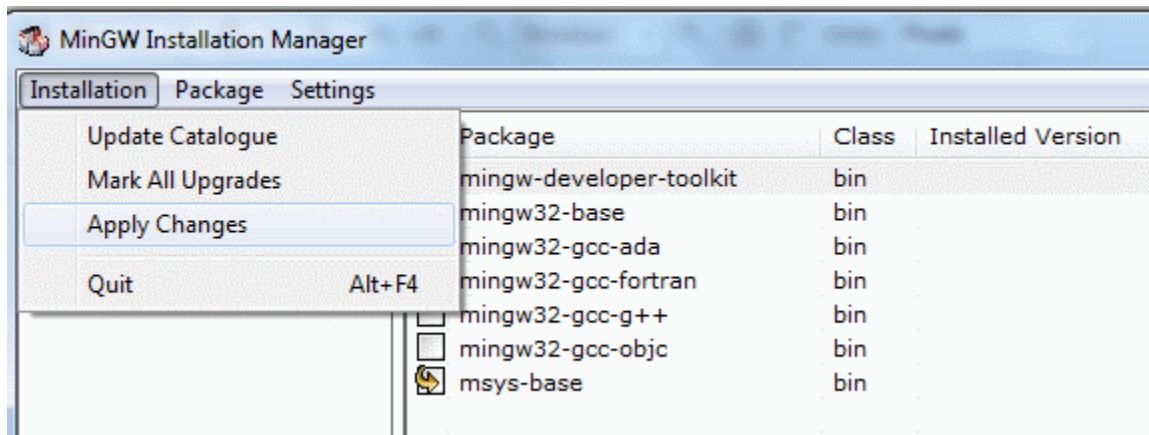
In the next step wait for the download of some files and click Continue.

You should see a MinGW Installation Manager window. In the list of available packages click on the box next to "msys base" (the last item in the list).

From the context menu select Mark for Installation.



From the menu at the top of the window select Installation > Apply Changes.



In the next step click Apply button.

After a while the MinGW tools should be installed. Close the window.

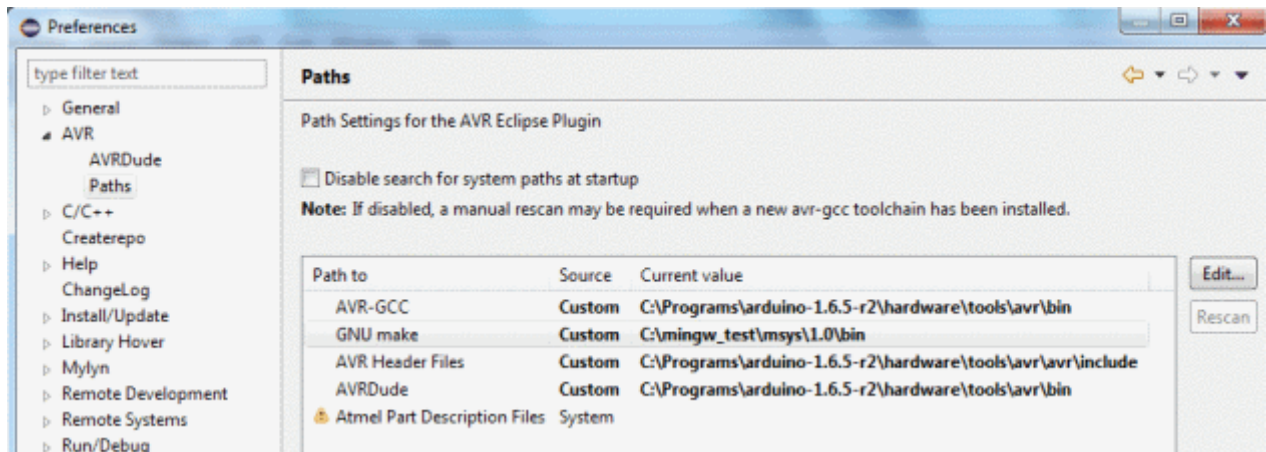
We are now ready to configure the AVR Eclipse plugin.

1.5 Configure the AVR Eclipse plugin

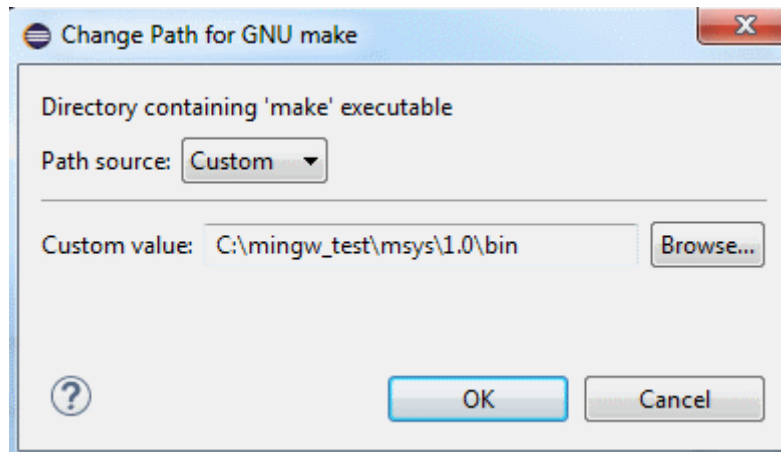
Once the AVR Eclipse plugin is installed, in the Eclipse main menu select Window > Preferences.

In the Preferences window expand AVR > Paths.

Use the Edit button on the right to set the paths as shown in the following picture.



The paths may be different on your system, depending on where you installed the tools. Use the Browse button in the window for path selection and "Custom" Path source; see the picture below.



Here are the paths on my computer:

AVR-GCC: C:\Programs\arduino-1.6.5-r2\hardware\tools\avr\bin

GNU make: C:\mingw_test\msys\1.0\bin

AVR Header Files: C:\Programs\arduino-1.6.5-r2\hardware\tools\avr\avr\include

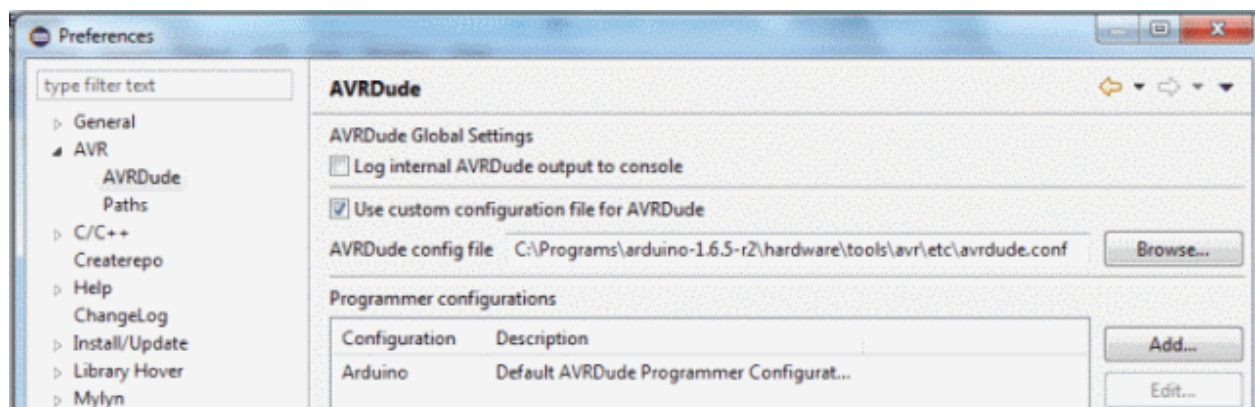
AVRDude: C:\Programs\arduino-1.6.5-r2\hardware\tools\avr\bin

Note that instead of the tools from Arduino installation shown here, you can also configure the plugin to use the tools from Atmel Toolchain, see the alternative setups below for more information.

Now select the AVRdude category on the left.

Check the box “Use custom configuration file for AVRdude”.

In “AVRdude config file” box browse to the path of this file in your Arduino installation. Here is how it looks:



The path is, for example, C:\Programs\arduino-1.6.5-r2\hardware\tools\avr\etc\avrdude.conf.

Close the preferences window by clicking the OK button.

Note: It seems closing the Preferences window is required to apply the changes. Without this, the next step, adding the programmer, fails with error that AVRDUDE cannot find its configuration file "".

Open again the Preferences from Window menu.

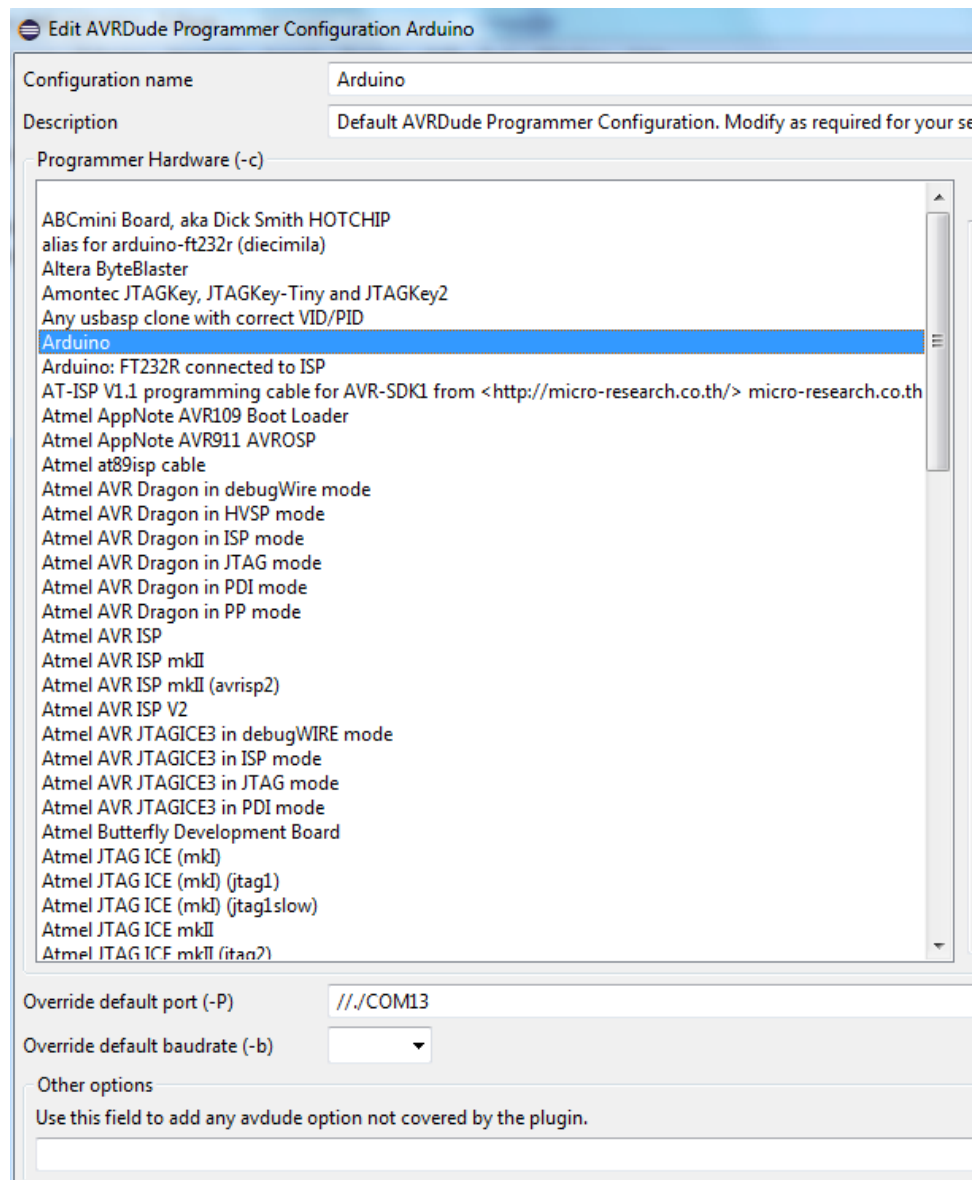
Select AVR > AVRDUDE.

Click the Add button on the right.

In the window which opens, select Arduino from the "Programmer Hardware" list.

Enter some name in the Configuration name field above the list, for example, Arduino.

Enter the serial port to which your Arduino is connected into the "Override default port" box below the list. Note that the port name must be written like this on Windows: "\\.\\COM13". See the picture below.



Close the window with OK and then close the Preferences window also.

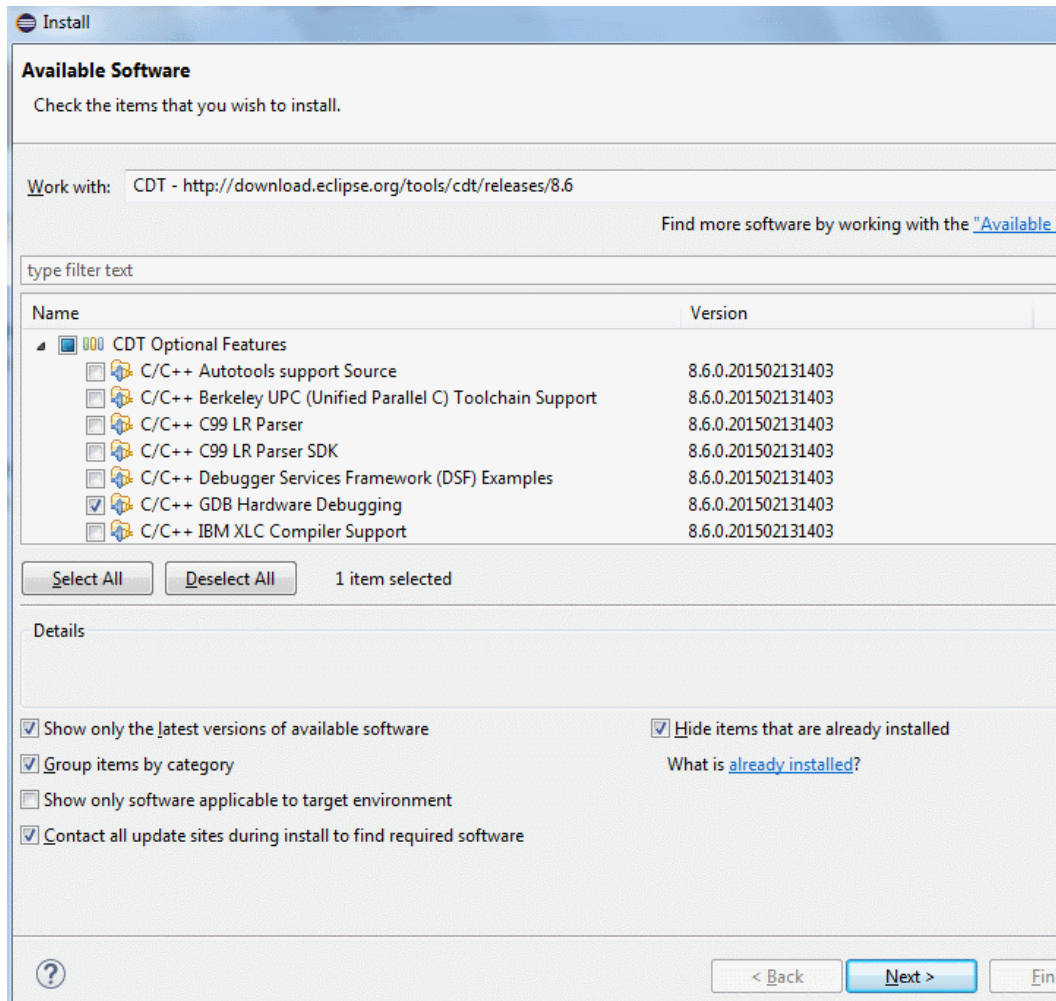
1.6 Install GDB Hardware Debugging launch configuration into Eclipse

In Eclipse go to menu Help > Install New Software...

In the “Work with” box enter this update site address and press the Enter key:

<http://download.eclipse.org/tools/cdt/releases/8.6>

After a while the list below will display some items. Expand the CDT Optional Features and select the C/C++ GDB Hardware Debugging.



Follow the wizard to install this feature and restart Eclipse when prompted to finish the installation.

Your development environment is now ready. You can continue with creating your first program as described in section “Step 2: Create your first project for the Arduino board” below.

1.7 Notes about the tools

If the above steps seem too complicated to you, you are not alone. Unfortunately, this is the simplest setup I could figure out. If you find simpler setup, let me know. There are several unfavorable factors, which caused this situation:

- The WinAVR toolchain seems no longer updated, so it cannot be used as the base for building AVR projects in Eclipse.
- Atmel AVR toolchain does not contain make utility, so make must be obtained separately. If you install Atmel Studio, the make is present there in the shellUtils directory. If you do not want to install Atmel Studio, you can use MinGW package or Cygwin.
- Arduino in recent versions does not include the make. From 1.6.5 it again contains avr-gdb, which can be used instead of the one from Atmel Toolchain.

Alternative setup with Atmel Studio (Option 2)

This section describes alternative way of setting up your development environment. The difference compared to the first way is that we do install Atmel Studio to obtain the build and debug tools rather than relying on the tools provided with Arduino IDE. This option requires more disk space (Atmel Studio is large) but it may be more stable and more up-to-date as the Atmel Studio and Atmel Toolchain are professional tools, unlike the Arduino IDE. Only the differences are described here. Please refer to the previous chapter for the other steps.

Difference(s) compared to option 1:

- Do not install MinGW tools (skip that step) and instead install Atmel Studio. All the other steps remain the same.
- For configuration of the AVR Eclipse plugin look at the section below rather than the one included with option 1.

1.2 Install Atmel Studio

From <http://www.atmel.com/tools/atmelstudio.aspx> download Atmel Studio. Current version (May 2015) is 6.2.

Run the installer and install the program.

When configuring the AVR Eclipse plugin (described in next sections), use the paths to build tools pointing to Atmel toolchain instead of into Arduino folder.

Alternative setup without Atmel Studio but with Atmel AVR8 Toolchain (Option 3)

This section describes alternative way of setting up your development environment without using the Arduino build tools. But note that you still need to have Arduino software! The difference is just in the setup of the build tools in AVR Eclipse plugin (described in next sections) – we do not point the plugin to tools in Arduino folder but rather into the Atmel Toolchain folder.

Difference(s) compared to option 1:

- Install all the tools as in option 1 plus install the Atmel AVR Toolchain.
- For configuration of the AVR Eclipse plugin look at the section below rather than the one included with option 1.

Install Atmel AVR Toolchain

This is needed to have the build tools and also the GDB debugger (avr-gdb.exe).

Note 1: If you have Atmel Studio installed on your computer, you will already have the AVR Toolchain installed. You can find the toolchain in: c:\Program Files\Atmel\Atmel Toolchain\AVR8 GCC. You also do not need to install MinGW tools, because you can use the Make utility found in [Atmel Studio location]\shellUtils instead of the one from MinGW tools.

Note 2: You can also use other AVR toolchain (AVR-GCC) instead of the one provided by Atmel, but make sure it contains current version of the GDB debugger (avr-gdb). Popular WinAVR will probably not work because it seems it is not updated for long time.

To download the Atmel AVR Toolchain:

Go to [http://www.microchip.com/avr-support/avr-and-arm-toolchains-\(c-compilers\)](http://www.microchip.com/avr-support/avr-and-arm-toolchains-(c-compilers)).

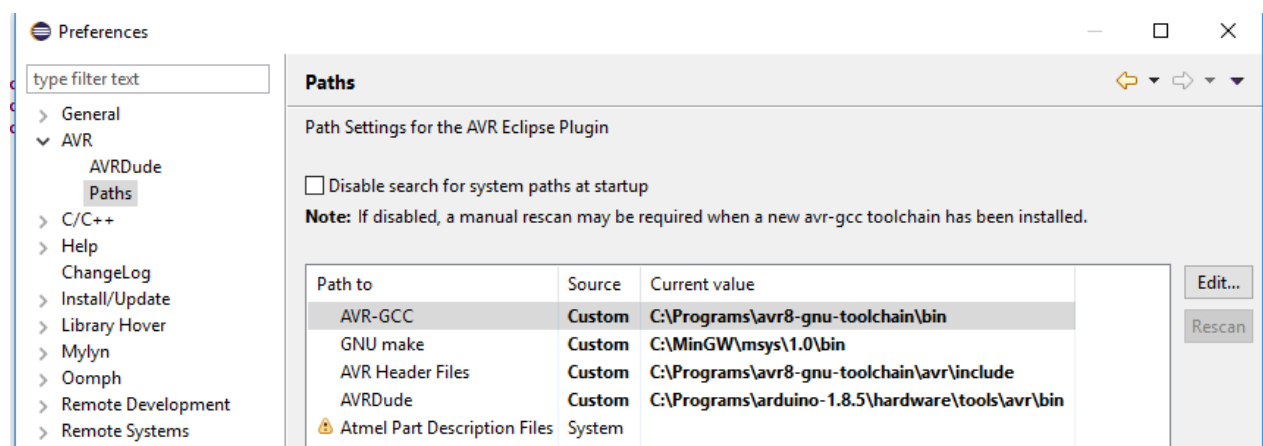
In the Downloads table select **AVR 8-bit Toolchain v3.5.4 – Windows**.

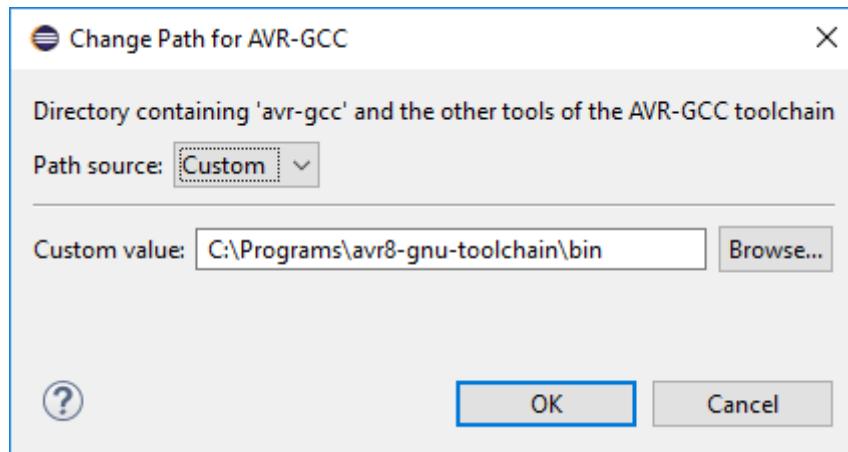
This is self extracting archive. Extract the toolchain to some folder on your computer, for example to c:\programs\avr8-gnu-toolchain.

Configuration of the AVR Eclipse plugin with Atmel Toolchain tools

In the pictures below you can see the AVR Eclipse plugin configured with the GNU make tool from Atmel Studio and the other paths pointing to Atmel AVR8 Toolchain instead of the Arduino folders.

Use this as a reference for setups based on Atmel Studio or Atmel Toolchain.





Here are examples of the paths:

AVR-GCC: C:\Programs\avr8-gnu-toolchain\bin

GNU make: C:\MinGW\msys\1.0\bin

AVR Header Files: C:\Programs\avr8-gnu-toolchain\avr\include

AVRDude: C:\Programs\arduino-1.8.5\hardware\tools\avr\bin

2. Step 2: Create your first project for the Arduino board

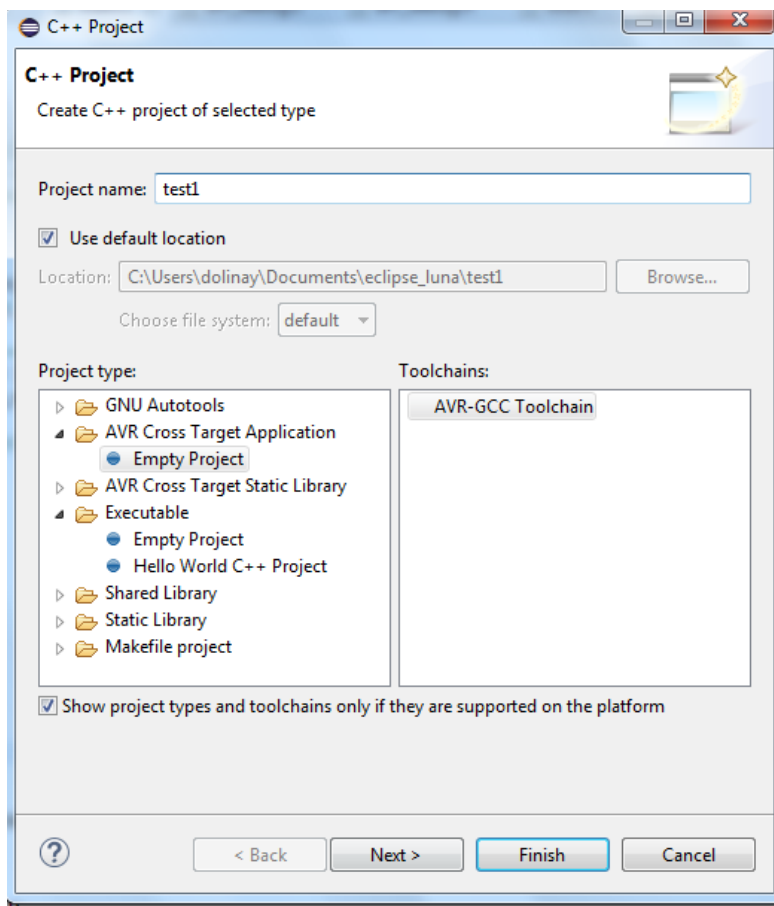
This section describes how to create project in Eclipse which will produce a program that you can upload to your Arduino Uno board. The program is written in “plain” C++ language; it does not use the Arduino software libraries.

Note: This project will work for Arduino Uno board and possibly also boards with the same microcontroller (ATmega328). For Arduino Mega you will need to select different MCU Type and modify the program code slightly so that it controls pin 7 instead of 5. Or you can import an example project from examples/mega2560/c_mega. See Opening example projects section for instructions.

In Eclipse select New > C++ Project.

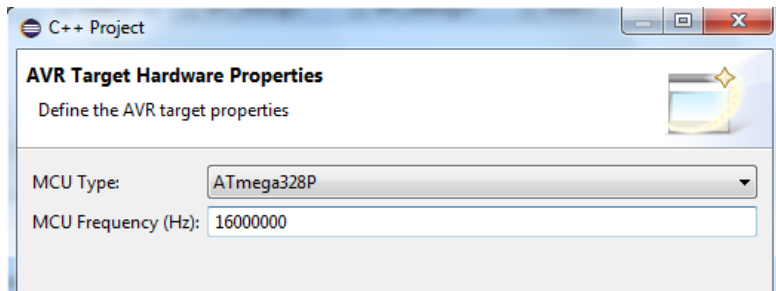
Enter a name for the project, for example, “test1”.

Select AVR Cross Target application > Empty project.



In the next step leave everything as offered (this will create Debug and Release configuration) for the project.

In the next step select the MCU Type: ATmega328P and MCU Frequency (Hz): 16000000.

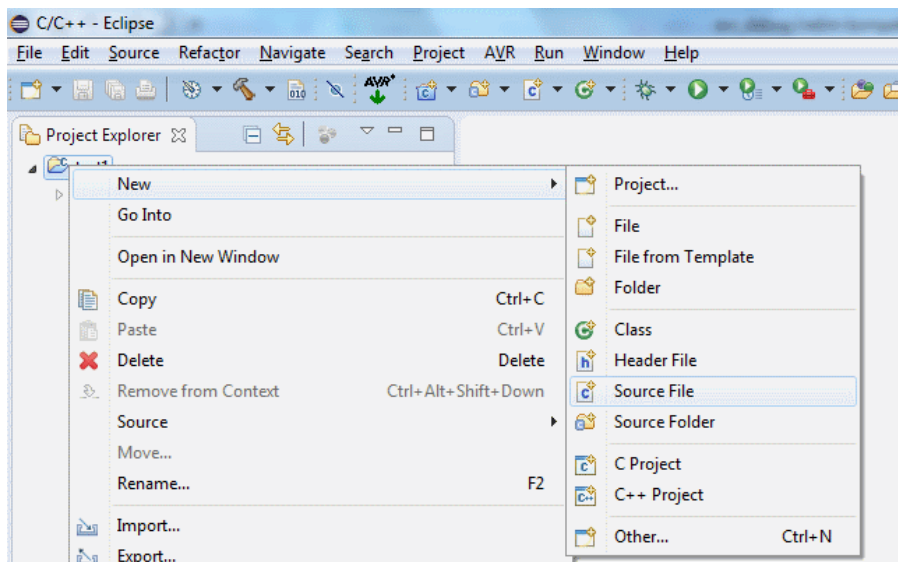


Click Finish to create the project.

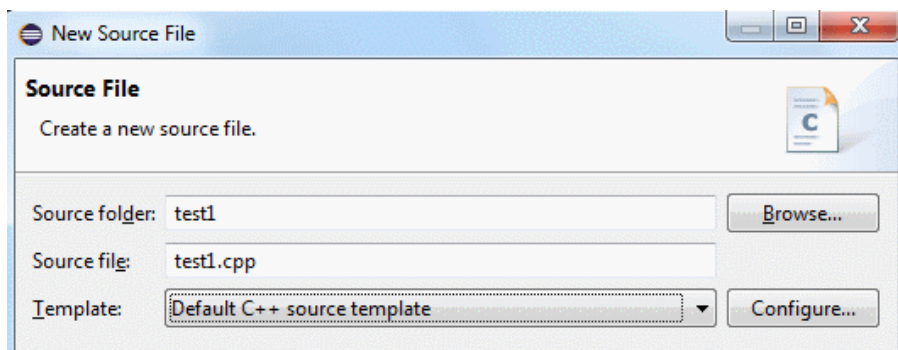
Now create main source file (.cpp) in the project:

Close the Welcome screen if you haven't closed it yet. Now you should see Project Explorer window on the left and the new project "test1" in it.

Right-click the project in Project Explorer and select New > Source File from the context menu.



In the New Source File window enter the name of the file, for example, use the same name as your project, "test1" with a .cpp extension.



Click Finish to create the file. It should appear under the project item in the Project Explorer on the left and also open in the editor window on the right.

Copy and paste the following code into the file you've just created.

```

#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>

int main(void)
{
    DDRB |= _BV(5); // pin PB5 to output (LED)
    sei();           // enable interrupts
    while(1)
    {
        PORTB |= _BV(5); // LED on
        _delay_ms(100);
        PORTB &= ~_BV(5); // LED off
        _delay_ms(100);
    }
    return 0;
}

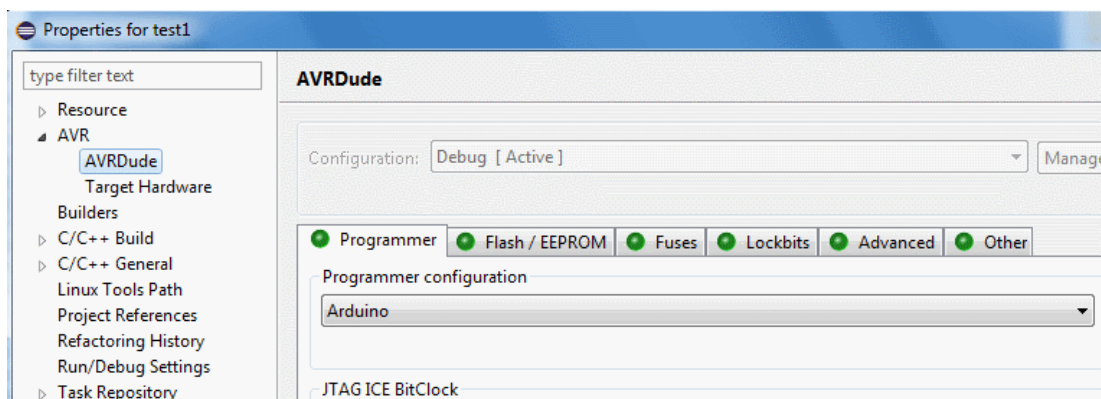
```

Save the file (Ctrl + S).

TIP: Enable automatic save before build in Window > Preferences > General > Workspace: "Save automatically before build". Without this you will often forget to save the changes after modifying the code and will be actually building the old version of your program, without the changes.

Right-click the project in the Project Explorer and select Properties from the context menu. You can also press Alt + Enter to open the Properties window.

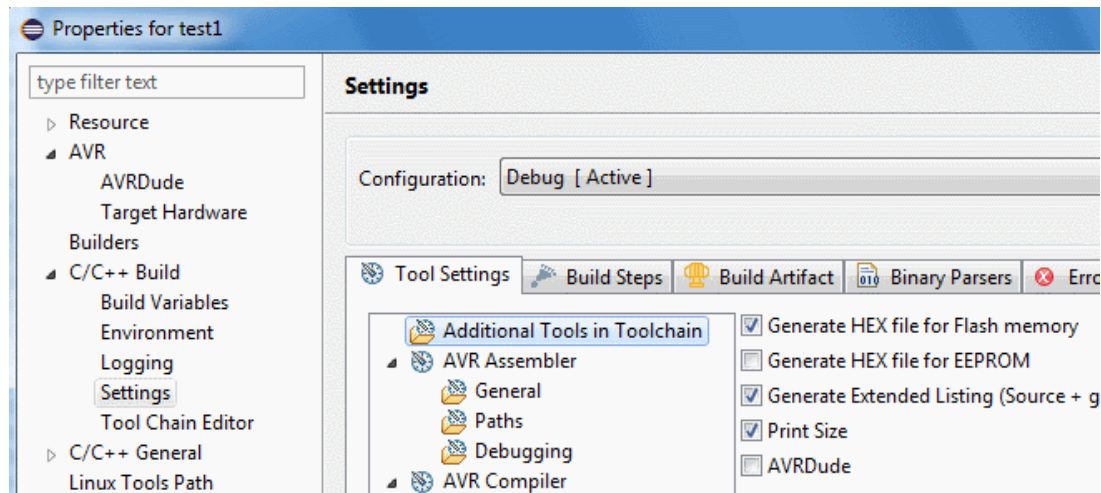
In the left part of the Properties window select AVR > AVRdude. On the Programmer tab in Programmer configuration select Arduino. This is the programmer configuration we created earlier, when configuring the AVR Eclipse plugin.



Still in the Properties window expand the C/C++ Build > Settings.

In the right-side window with sub-categories select "Additional Tools in Toolchain" category.

Check the "Generate HEX file for Flash memory" box.



Close the Preferences window with OK (Apply and Close) button.

To build the project:

Right-click the project and select Build Project from the context menu. You can also click the hammer icon in the toolbar.

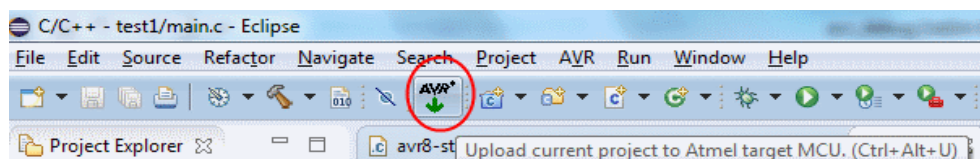
The project should build without errors. Check the Console tab in the bottom part of the Eclipse IDE for build messages. A warning about compiler optimizations from delay.h is OK for now.

To upload the program to Arduino board:

Connect your Arduino board to the computer if not already connected.

Click the AVR icon in the toolbar.

This will upload your program to the board. Check the result in the Console window. There will be some messages from AVRDUDE. It should finish with “avrdude done. Thank you.”



You should now see that the LED on your Arduino is blinking fast (100 ms on, 100 ms off). This means our program is running.

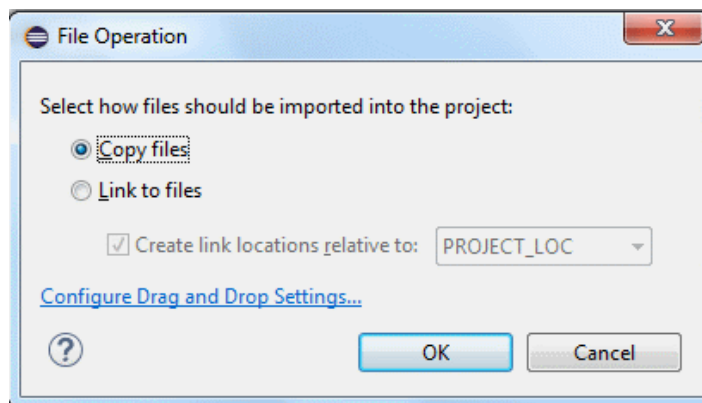
If you get error messages such as “port is blocked” you probably have wrong COM port number in the AVR Eclipse plugin configuration. Go to Window > Preferences > AVR > AVRDUDE. In the “Programmer configurations” list select your Arduino item and click Edit. Make sure the number of COM port (for example //./COM15) is correct. You can find this number either in Arduino IDE or in Hardware manager of your computer.

3. Step 3: Add debugger support to your program

This section describes how to add support for the debugger to the project and how to work with the debugger. It assumes that you have set up your Eclipse with the AVR Eclipse plugin to be able to build programs for Arduino and that you have created a project in Eclipse which you can build and upload to your Arduino board.

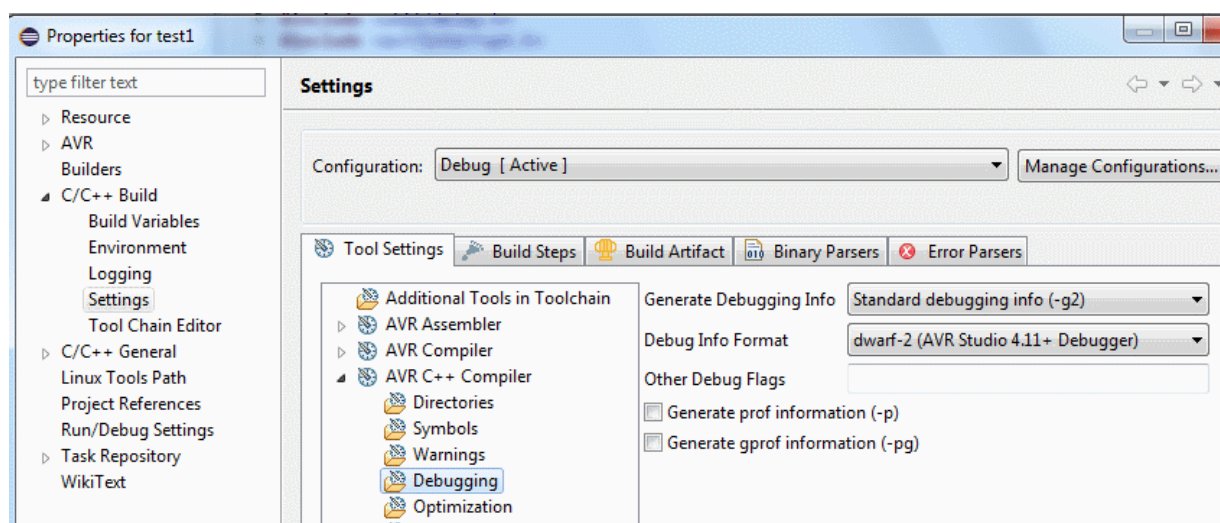
Add **avr8-stub.c** and **avr8-stub.h** files into your project. These files are located in the [Arduino debugger]\avr8-stub folder. You can drag the files from your file manager and drop them on the project “test1” in the Project Explorer view in Eclipse. Select Copy files option in the File Operation window which appears after dropping the files.

If you want to use flash breakpoints, add also the files **app_api.h** and **app_api.c** from the same location.



Open Properties of your project (Alt + Enter) and go to C/C++ Build > Settings.

Select AVR C++ Compiler > Debugging. In the “**Debug Info format**” select dwarf-2.



Do the same for the AVR C Compiler. It is not necessary now but can come in handy later when we also have some .c files in the project.

Close the Preferences window with Apply and close (OK) button.

Replace the program with the following one:

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include "avr8-stub.h"

int cnt = 0;
int main(void)
{
    debug_init();
    DDRB |= _BV(5); // pin PB5 to output (LED)
    sei();          // enable interrupts
    while(1)
    {
        PORTB |= _BV(5); // LED on
        cnt++;
        PORTB &= ~_BV(5); // LED off
        cnt++;
    }
    return 0;
}
```

Note these changes:

- Included the header file for the debug driver (#include "avr8-stub.h")
- Added call to function debug_init(); to initialize the debug driver.
- Added call to sei() to enable interrupts.
- Added global variable cnt, so that we have something to inspect in the debugger.
- Removed delays (_delay_ms()).

Build your project.

Connect your Arduino board.

Click the AVR icon in the toolbar. In the Eclipse console window you should see how your program is uploaded, ending with "avrdude done. Thank you."

Now the program is uploaded in the microcontroller and running. The LED seems to be constantly ON. We cannot see that it is blinking because the speed is too fast. But we will be able to see it turning on and off when debugging.

We now need to create **Debug configuration** in Eclipse to be able to connect to the program and debug it. This is covered in the next section – 4. Step 4: Connect to your program with the debugger .

Summary of the important steps to add the debugger support to your program

- Add avr8-stub.c and avr8-stub.h files into your project.

- Add #include "avr8-stub.h" into your source file.
- Call debug_init() somewhere at the beginning of your main function.
- Enable interrupts by calling sei().
- Optionally, call function breakpoint() at the point(s) where the program should stop. You can also break the program when you connect to it from the debugger.

Alternative, more portable way for adding the debugger support

The method for adding the debugger files into your project described above is simple, but not the best one for advanced users. For example, if the debug driver is updated and you download new version, you would have to replace all the avr8-stub.c and avr8-stub.h files in all projects which use it to update the driver in these projects.

Recommended method for more advanced users is as follows:

In the Eclipse main menu select Window > Preferences.

In the Preferences window go to General > Workspace > Linked Resources.

In the right-side part of the window create new Path variable using the New button. Name it, for example, AVR8_STUB_PATH. Use the Folder... button in the New Variable window to point the variable to the location of the debug driver files on your system. For example, c:\avr_debug\avr8-stub.

Close the Preferences window.

Right-click your project in Project Explorer and select New > Folder from the context menu. New Folder window will open.

In the Folder name box enter avr8-stub.

Expand the Advanced options.

Select "Link to alternate location (Linked Folder)" option.

Click the Variables button and select the AVR8_STUB_PATH variable which we've created earlier.

Click Finish. A folder avr8-stub should now appear under your project.

Go to preferences for your project (Alt + Enter or right-click the project and select Properties).

Expand C/C++ Build > Settings category.

At the top of the window, in "Configuration" select [All configurations].

In the AVR C++ Compiler > Directories add the following path:

"\${workspace_loc}/\${ProjName}/avr8-stub".

You can either copy-paste it from here or use the Workspace button in the Add directory path window and select your project > avr8-stub folder. The result should be the same. This will add the

avr8-stub folder into the search path of the compiler so that the compiler can find the header file(s) located in this folder.

Do the same procedure also for the directories of the C compiler in AVR C Compiler > Directories.

Your project should now build without error. Do not forget to enable the HEX file output and select the programmer for the project as described in the chapter about creating your first project for the Arduino board above (if you haven't done it yet).

This method has several advantages:

- If you update the debug driver, you just need to update it in the original location and all the projects will automatically use the new version.
- If you move the location of the debug driver, you just need to update the value of the AVR8_STUB_PATH variable and all projects will still build correctly.
- If you move the project itself, for example, to another computer, you will just need to create or update the AVR8_STUB_PATH variable so that it points to the correct location of the debug driver.

4. Step 4: Connect to your program with the debugger

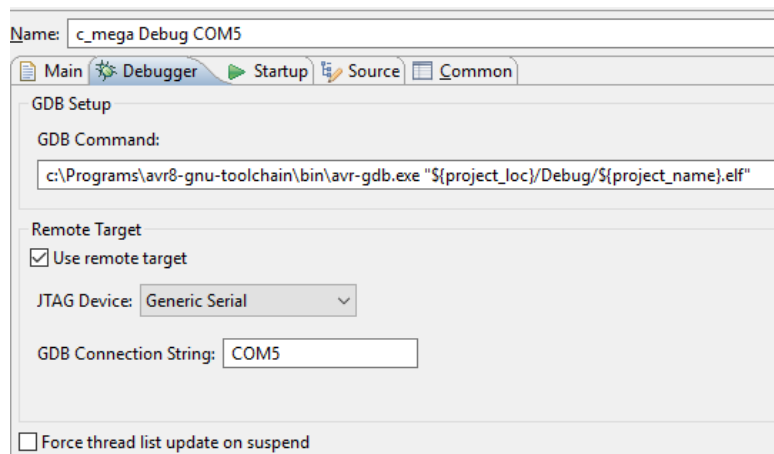
This section describes how to create the debug configuration in Eclipse to be able to debug the program in Arduino. It assumes that the program running in Arduino was built with the debug driver as described in the previous section.

Important note on the TCP-Serial proxy

The procedure below assumes use of a program which converts TCP/IP communication from the IDE to serial communication used by the debugger. This was the only option for Windows at the time of first release of this debugger. It seems with some boards (USB drivers) and especially on Window 10 this convertor is not necessary and the debugging will work directly over serial line. I recommend the following procedure:

- If you are using Linux, just use direct connection to serial line (e.g. to /dev/ttyACM0)
- On Windows, first set up the debug configuration with direct serial connection
- If you are able to debug your program, use this way and ignore the TCP to Serial proxy instructions.
- If direct serial connection does not work, use the TCP to Serial proxy as described below.

To use direct serial connection, select “Generic Serial” in the JTAG Device field in debug configuration as shown in the picture below. For details follow the Step-by-step instructions below.

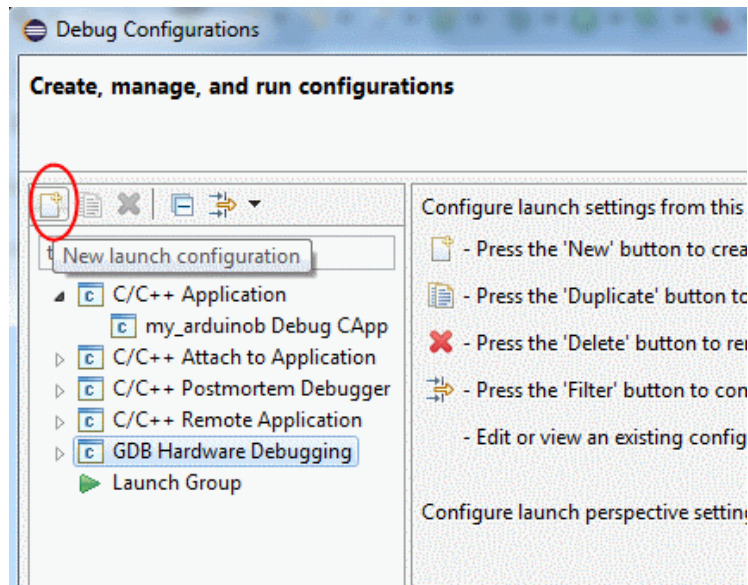


Note: if you get errors “COM x. No such file or directory” try to enter the COM port name in this format: `\\.\COM10` (example for COM 10). It seems that for higher port numbers the simple name without the backslashes does not work.

Step-by-step instructions

Right-click your project in the Project Explorer in Eclipse. From the context menu select Debug As > Debug configurations...

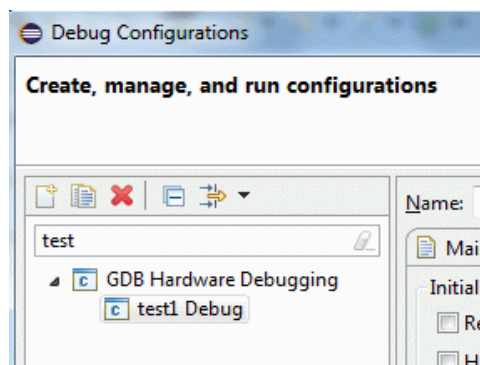
In the Debug Configurations window select GDB Hardware debugging item and click the New launch configuration button in upper left corner of the window.



This will create new launch configuration under the GDB Hardware debugging item.

Note: If you do not see the GDB Hardware Debugging item in the list, you probably haven't installed this type of configuration. Please refer to the chapter about setting up your development environment.

Select the new configuration under GDB Hardware debugging to configure its properties:



In the Debugger tab in the "GDB command" field enter (or browse to) the path to the GDB executable `avr-gdb.exe`, followed by the path to your "executable" file (.elf).

The path to your file can use eclipse variable to refer to the project executable. Here is my example for this field:

```
c:\Programs\arduino-1.6.5-r2\hardware\tools\avr\bin\avr-gdb.exe  
"${project_loc}/Debug/${project_name}.elf"
```

Or if you use the **Atmel Toolchain build tools** instead of those from Arduino package.

```
C:\Programs\avr8-gnu-toolchain\bin\avr-gdb.exe "${project_loc}/Debug/${project_name}.elf".
```

TIP: Use the Browse button to select the avr-gdb.exe. Then enter space after the path into the edit box and then paste the following line: "\${project_loc}/Debug/\${project_name}.elf".

Check the "Use remote target" box.

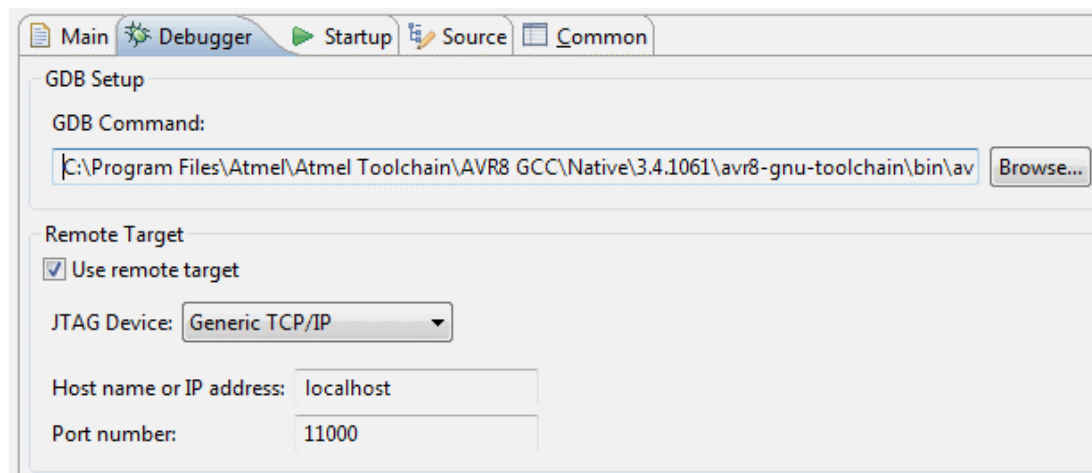
Note: The following steps describe the connection via TCP-to-Serial converter. It may be easier to use direct serial connection. To use it select **Generic Serial** in the JTAG Device field and in the GDB Connection String enter you serial port, e.g. \\.\COM5. For details see "Important note on the TCP-Serial proxy" section above.

To set up the connection via TCP to Serial proxy, do this:

In "JTAG device" select "Generic TCP/IP" and enter:

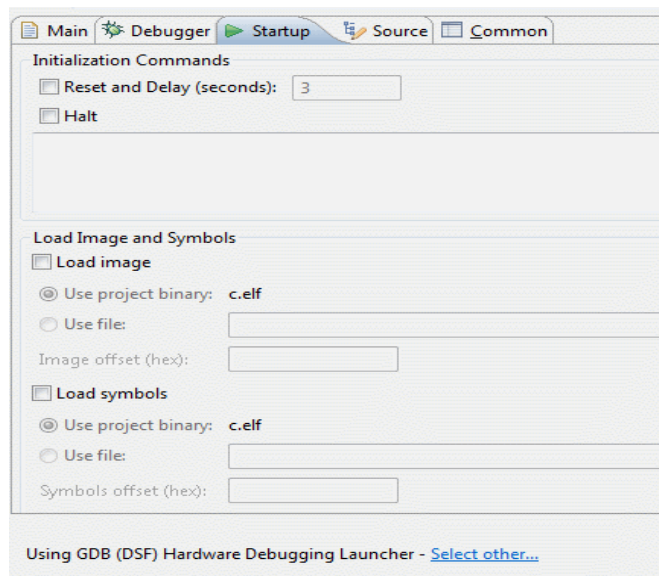
Host name or IP address: localhost

Port number: 11000.



This next step is used both for direct serial and TCP to serial proxy option.

Switch to Startup tab. Uncheck (clear) all the boxes (Reset and Delay, Halt, Load image and Load symbols).



Click **Apply** button to save the changes, but do not close the Debug configurations window yet.

If you use direct serial communication, you can now click the **Debug** button at the bottom of the window to start debugging. Skip to the section Working with the debugger below.

If you use the TCP to Serial connection continue with these steps:

Now use your file manager to open the folder where the Arduino debugger package is located. For example, c:\avr_debug. You should see a start_proxy.bat file in this folder.

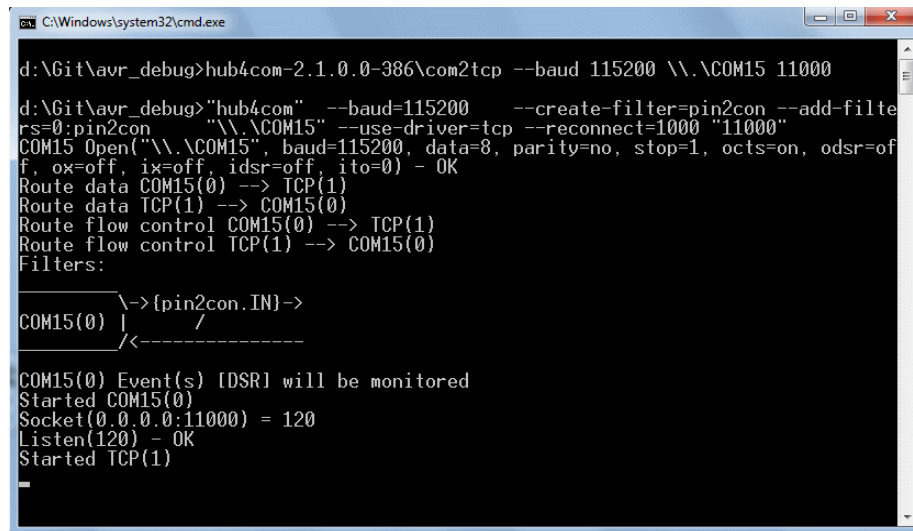
Open the start_proxy.bat file in Notepad or other text editor. Change the number of the COM port in this file. There is this line:

```
hub4com-2.1.0.0-386\com2tcp --baud 115200 \\.COM15 11000
```

Just change the number after COM from 15 to the number of your COM port to which the Arduino board is connected. If you prefer, you can also use the com2tcp program directly; use the command in this .bat file as an example.

Save and close the start_proxy.bat file.

Run the bat file. This will start convertor between TCP/IP port used by the GDB (as configured above) and the serial port to which your Arduino is connected. You should see a console window with some information. This window will be opened all the time during the debugging.



```
C:\Windows\system32\cmd.exe
d:\Git\avr_debug>hub4com-2.1.0.0-386\com2tcp --baud 115200 \\.\COM15 11000
d:\Git\avr_debug>"hub4com" --baud=115200 --create-filter=pin2con --add-filter=0:pin2con --use-driver=tcp --reconnect=1000 "11000"
COM15 Open("\\.\COM15", baud=115200, data=8, parity=no, stop=1, octs=on, odsr=off, ox=off, ix=off, idsr=off, ito=0) - OK
Route data COM15(0) --> TCP(1)
Route data TCP(1) --> COM15(0)
Route flow control COM15(0) --> TCP(1)
Route flow control TCP(1) --> COM15(0)
Filters:
COM15(0) \->{pin2con.IN}->
          /
          /<-----
COM15(0) Event(s) [DSR] will be monitored
Started COM15(0)
Socket(0.0.0.0:11000) = 120
Listen(120) - OK
Started TCP(1)
```

Note: You may want to configure your firewall to block access to the port 11000 from other computers. You can also change the port number both in the .bat file and in the Eclipse debug configuration.

Now return to Eclipse. We still have the debug configurations window opened.

Click the **Debug** button at the bottom of this window.

Working with the debugger

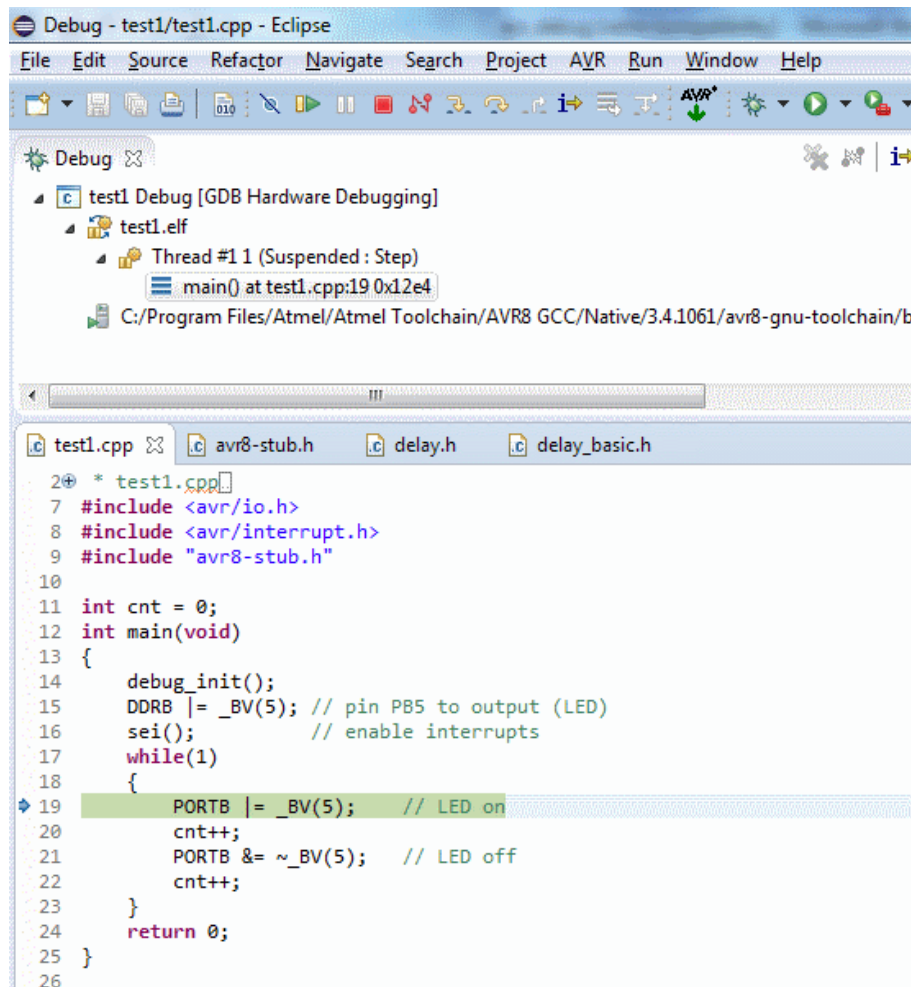
After some time, Eclipse should ask you if you want to switch to debug view (Perspective). Answer Yes.

TIP: Switch between the Debug and C/C++ perspective using the buttons in upper right corner of the window. When you finish debugging, click the C/C++ button to reorganize the windows for coding. When you start debugging, Eclipse will automatically switch to window arrangement suitable for debugging.

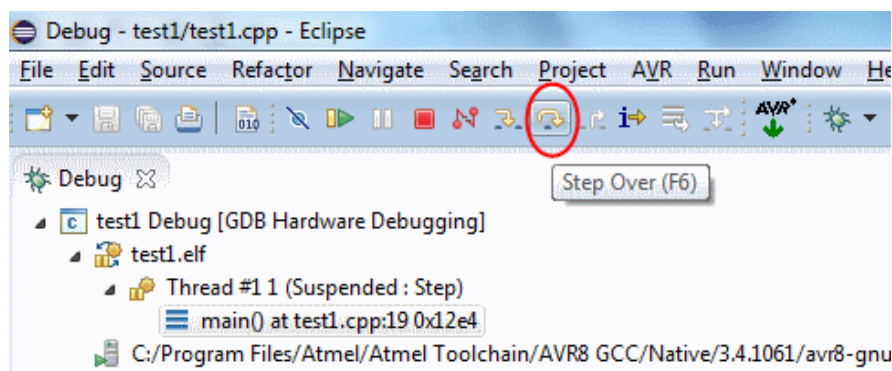
You should see the program stopped in debugger, as in the following picture.

The point at which the program stops is random; the program is stopped run when the debugger connects.

Note: If you encounter an error during this, please see the section Problem: Debug session fails to start. There seems to be problem with the avr-gdb.exe included in newer versions of Arduino IDE – it fails to start because of some missing DLLs. Solution is described in the Reason 4 section of the above mentioned topic – error message **could not determine GDB version**.



You can now single step the program using the **Step over** button in the toolbar. This will advance the program by one line. Note that when you step over the `PORTB |= _BV(5);` line, the LED on Arduino board will turn on.



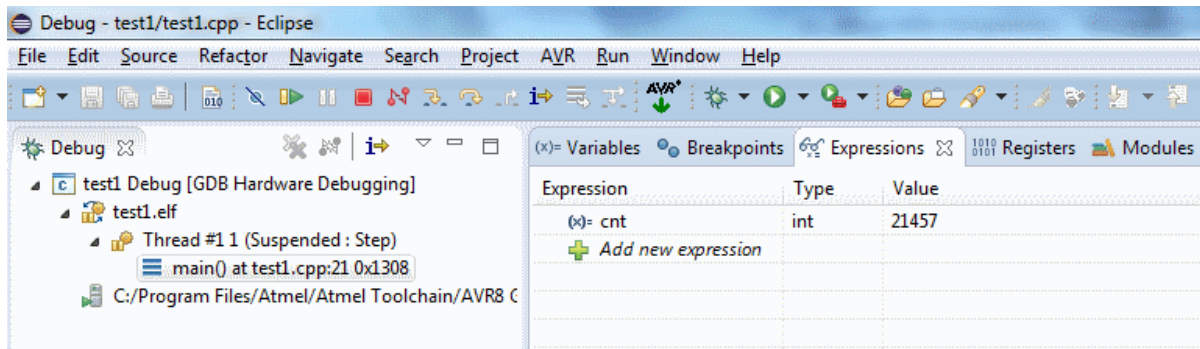
We can also look at the value of the “cnt” variable. Just hover the cursor at the `cnt++;` code and the value will be displayed. Don’t be surprised the value is high, remember, we interrupted the program at random moment.

We can also change the value of the cnt variable:

In eclipse menu select Window > Show View > Expressions. In the upper right corner an Expressions tab will open.

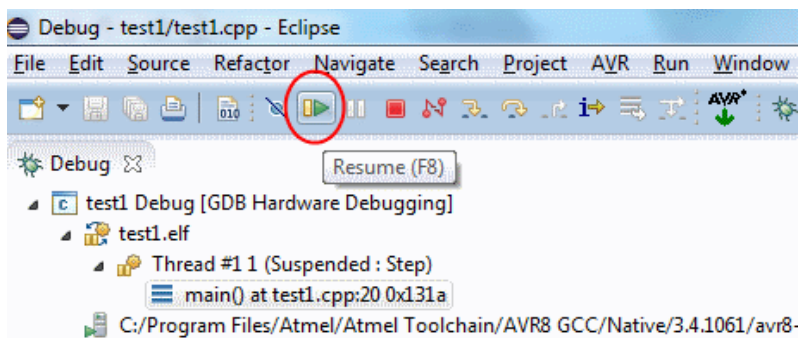
Click at the Add new expression in the Expression tab. Enter “cnt” and press Enter.

You should now see the value of the cnt variable there, as in the following picture.



Click at the value and enter 0. Then press Enter. The cnt variable should now have value 0. If you step through the code, you can see how the value increases with every cnt++; command.

You can also let the program run at full speed. Click Resume button in the toolbar to do so.



Interrupt the program at any time using the **Suspend** button (pause) which is next to the **Resume** button.

Breakpoints

You can also stop the program at any line. To do so, right click into the gray left margin in the editor at the line where you want the program to stop.

From the context menu select Toggle breakpoint. A blue point will appear. To remove a breakpoint, use the same procedure.

Now resume the program (click **Resume** button). It will stop at the breakpoint. You can then inspect the variable(s), step or resume again.

To end the debug session, click the **Terminate** button (red square) in the toolbar.

If you modify your program, make sure you rebuild and upload it to the board before debugging again. If you use the TCP to Serial proxy, it must be stopped so that the serial port is free for the upload. Here is the procedure for modifying the program:

- Edit and build your program
- Close the console window with COM to TCP proxy server, if still opened.
- Upload the program using the AVR button in Eclipse
- Start the TCP to Serial proxy server (use the start_proxy.bat file)
- Start the debug in Eclipse (Expand the Debug button in the toolbar and click your debug configuration, for example “test1 Debug”).

If you use direct serial connection the procedure is simpler:

- Edit and build the program
- Upload the program using the AVR button in Eclipse
- Start the debug in Eclipse (Expand the Debug button in the toolbar and click your debug configuration, for example “test1 Debug”).

Note about the delay function

Earlier, we removed the `_delay_ms` function from the program, because it would make the debugging time-consuming. This kind of delay with busy loops is affected very much by the breakpoints in RAM used by the debugger; the program runs much slower when there is any breakpoint set or when stepping over a line of code. The reason is explained in the description of the principles of the debug driver. In the next section, when we enable the Arduino functions, we will see that the Arduino delay which is based on timer will perform much better and it is quite comfortable to step through the code even with delay.

Troubleshooting

If you encounter an error right after clicking the debug button with “gdb –version”, select your project in the Project Explorer and try again. Alternatively, instead of using the Debug button in toolbar, right-click the project and select Debug As > Debug configurations. Select the debug configuration and click Debug.

If you encounter an error later during the startup of the debug session, make sure that your program calls the `debug_init()` function at the beginning. If you are using the TCP to Serial proxy server, make also sure that it is running.

Please see also the Troubleshooting problems with debugging later in this document for more tips.

Exercise – more complex program

Here is another program which contains also a function and local variables. Try to build, upload and debug this program as an exercise.

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include "avr8-stub.h"

int cnt = 0;
int function(int a);

int main(void)
{
    debug_init();
    DDRB |= _BV(5); // pin PB5 to output (LED)
    sei();           // enable interrupts
    breakpoint();
    while(1)
    {
        PORTB |= _BV(5); // LED on
        cnt++;
        cnt = function(cnt);
        PORTB &= ~_BV(5); // LED off
        cnt++;
    }
    return 0;
}

int function(int a)
{
    int n;
    n = 2*a;
    return n;
}
```

Things to note

We have added call to `breakpoint()` function before the `while(1)` cycle. This will stop the program; it will wait for us to connect with the debugger instead of running freely until we connect. Then we can step or resume.

If you step through the code in the `while(1)` cycle with Step Over, you will never get into the `function()`. If you use the step into function you should eventually get there after several steps. Easier way to get into the function is to set a breakpoint inside it (on the `n = 2*a;` line). Then Resume the program and it will stop inside the function.

The behavior of the program when stepping through is sometimes different than expected. This is caused by compiler optimizations which reorganize the code or completely skip some commands. By default the project is built with “No optimizations(O0)” option, but that does not mean that the compiler will exactly follow the order of the code as you write it. Try to experiment with “-Og” option. This option is not available in the selection list, but you can enter it into the Other Optimization Flags box. Optimizations are set in project Properties > C/C++ Build > AVR C++ Compiler > Optimization. It can also be useful if you are running out of program memory (your program is too

large to fit into the microcontroller). The `-Og` option should save some memory without affecting the debug experience much.

Local variables (n in the function) can be seen on the Variables tab (in the upper right corner view). These are displayed automatically; you do not need to add them.

5. Step 5: Set up a project in Eclipse with Arduino functions

This section describes how to create project with Arduino software library (MCU Framework), so that you can use the Arduino functions in your program.

Create new project in Eclipse as described earlier in Step 2. Here is summary of the main points to select:

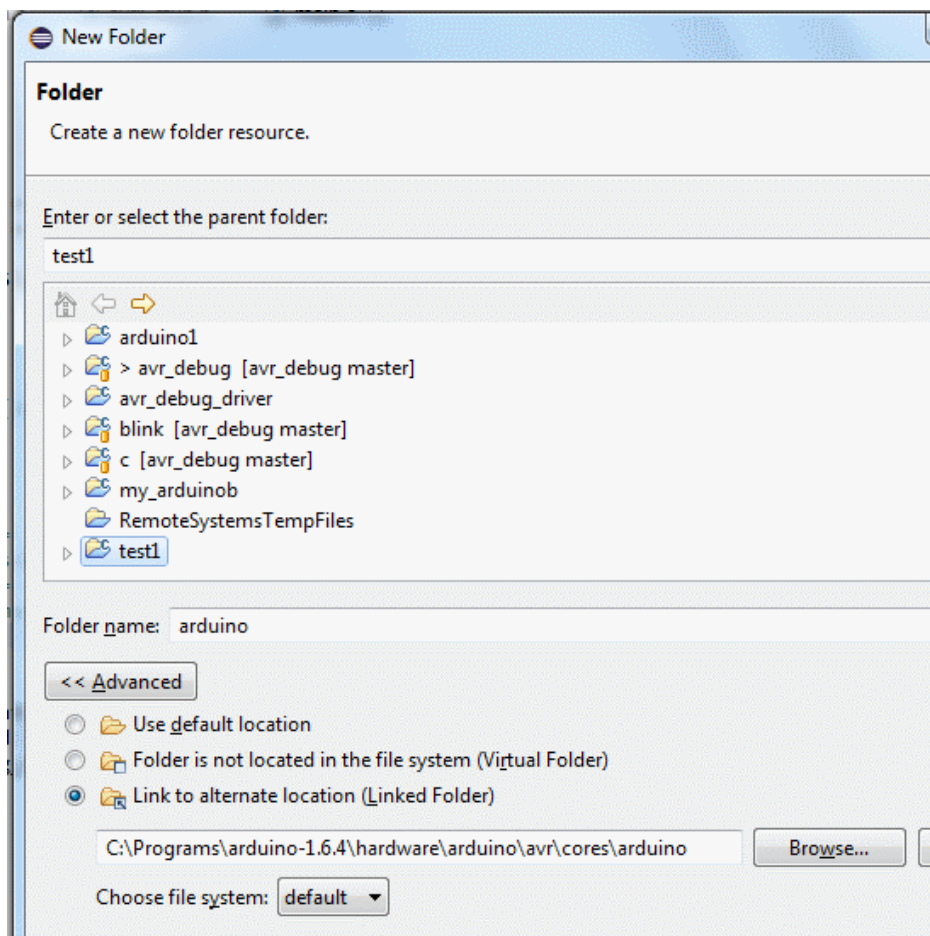
- File > New > C++ Project
- AVR C++ Cross Target Application > Empty project,
- MCU Type Atmega328P, MCU Frequency 16000000.

After the project is created, right-click the project in Project Explorer in Eclipse and select New > Folder. New Folder window will appear.

In the Folder name box enter arduino.

Click the Advanced button at the bottom of the window. Folder options will appear. Select the type "Link to Alternate location (Linked folder)".

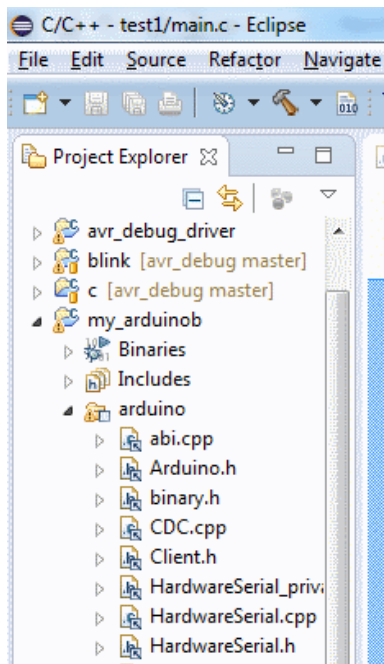
Click the Browse button and select the location of folder hardware\arduino\avr\cores\arduino in your Arduino installation. See the picture below.



Click Finish to create the folder.

Follow the same steps to create another folder. Name it “standard” and point it to hardware\arduino\avr\variants\standard in your Arduino installation.

You should now see under your project in Eclipse the subfolders “Arduino” and “standard” and if you expand them, you will see the files from these folders. In the “Arduino” folder, there are many files. In the “standard” folder there is only one file – pins_arduino.h.



Note that these folders are linked to the location in your Arduino installation; they are not copied into the project. If you modify or delete the files under these folders; you could damage your Arduino installation.

Right-click the project and select New > Source File.

Enter some name for the file with .cpp extension and click Finish. The name could be the same as the project’s name or **sketch.cpp**.

Paste the following code into the new file and save it:

```
#include "arduino.h"

void setup(void)
{
    pinMode(13, OUTPUT);
}

void loop(void)
{
    digitalWrite(13, HIGH);
    delay(200);
    digitalWrite(13, LOW);
    delay(500);
}
```

Go to preferences for your project (Alt + Enter or right-click the project and select Properties).

Expand C/C++ Build > Settings category.

At the top of the window, in "Configuration" select [All configurations].

In the AVR C++ Compiler > Directories add the two Arduino folders we now have in the project (Arduino and standard). The easiest way is to copy-paste the following two paths, including the quotation marks (click the small Add button and then paste one path; repeat for the other):

```
"${workspace_loc}/${ProjName}/arduino"
```

```
"${workspace_loc}/${ProjName}/standard"
```

Alternatively, click the Add button, then Workspace button in the Add directory path window. In the Folder selection window select your project > Arduino and click OK. Click OK again to close the Add directory path window. Repeat the same for the "Standard" folder.

This will add the two folders with Arduino files into the search path of the compiler so that the compiler can find the header file(s) located in these folders.

Do the same also for the directories of the C compiler in AVR C Compiler > Directories.

Your project should now build without errors. There are just some warnings from delay.h which can be ignored for now.

It is a good idea to upload the program into the board now to see if it works. The LED should blink.

Important: To try the program at this point:

- Enable the **Generate HEX file for Flash memory** option in Properties > C/C++ Build > Additional Tools in Toolchain and
- **Select the programmer** for the project in Properties > AVR > AVRDUDE. This is described in the chapter about creating your first project for the Arduino board above

In the next section we will add the debug driver so that we can finally debug the program.

TIP: When adding the Arduino folders to your project, you can also use a variable instead of the absolute path. Create a variable, for example, ARDUINO_LOCATION in Eclipse menu Window > Preferences > General > Workspace > Linked Resources. Then in the New Folder window use the Variables button and Extend... to point to the Arduino folders using this variable instead of full absolute path. This was described in more details earlier, in the chapter about adding the debug driver into your project in a more portable way.

6. Step 6: Enable debugger support in a program with Arduino functions

This section explains how to add debugger support to the program with Arduino software library created in previous step. It does not provide detailed instructions for the steps covered in previous chapters, so please refer to these chapters as needed.

Important note

With the debugger support included your program cannot use the Serial functions (the hardware serial) and it cannot use one of the pins with external interrupt function (INT0 by default, but this can be changed in avr8-stub.h). These are both used by the debug driver.

We will start with the project created in Step 5. That is we now have a project which can use Arduino functions, we are able to build this project and upload it to the board.

Add the debug driver into you project as described in Step 3, that is:

Drag and drop the avr8-stub.c and .h files from [avr_debug]/avr8-stub folder onto your project in Eclipse Project explorer (with Copy files option).

Set the Debug Info Format in your project's Preferences > Settings > AVR Compiler > Debugging and in AVR C++ Compiler > Debugging to **dwarf-2**.

In the code, add the #include "avr8-stub.h" and call to debug_init() and breakpoint(). Here is the code:

```
#include "arduino.h"
#include "avr8-stub.h"

void setup(void)
{
    debug_init();
    pinMode(13, OUTPUT);
}

void loop(void)
{
    breakpoint();
    digitalWrite(13, HIGH);
    delay(200);
    digitalWrite(13, LOW);
    delay(500);
}
```

Note: This example code assumes that interrupts are enabled by the Arduino core – which happens in the main() function contained in the Arduino core code. However, if you use your own main code and call the setup() and loop() yourself, please enable interrupts by calling sei() after the call to debug_init(), see the example code in Step 3 above.

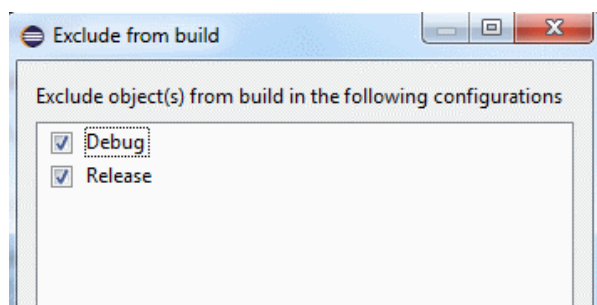
The program will not build now because of some errors. The linker is complaining about “multiple definition of `__vector_1`” and vector 18. These are interrupt vectors for the INT0 external interrupt (on pin 2) and interrupt from UART module which signals that a character was received through the serial line. Both these interrupts are needed for the debug driver to work but are currently handled also by the Arduino software library.

To fix these errors:

Expand the arduino folder in your project in Project Explorer and locate file HardwareSerial0.cpp.

Right-click this file and from the context menu select Resource Configurations > Exclude from Build...

In the window which opens select both Debug and Release configurations and click OK.



Now this file will not be built with your program. This will solve the multiple definition for vector 18 (UART) but it also means the Arduino Serial functions will not work. Note that this applies only to this project. Other programs you create either in eclipse or in the Arduino IDE are not affected. You are not modifying anything in your Arduino installation.

Repeat the same procedure for the file WInterrupts.c, that is exclude this file for build as well.

This solves the multiple definitions for vector 1, but by excluding WInterrupts.c from build, your program cannot use the attachInterrupt Arduino function. If you need to use attachInterrupt in your program, please see the subsection about Alternative ways of solving multiple definitions below.

Build the project. It should now build without error.

Create Debug configuration for your project as described in Step 4 section. Do not start the proxy or start debugging yet, just set up the options.

TIP: You will see the configuration for our previous project (test1) in the Debug Configurations window. You can copy the settings from this configuration by duplicating it (use the button next to the New launch configuration button). Then you just need to change the Project and C/C++ Application fields on the Main tab. Use the Browse and Search Project... buttons to select the proper project and the corresponding .elf file.

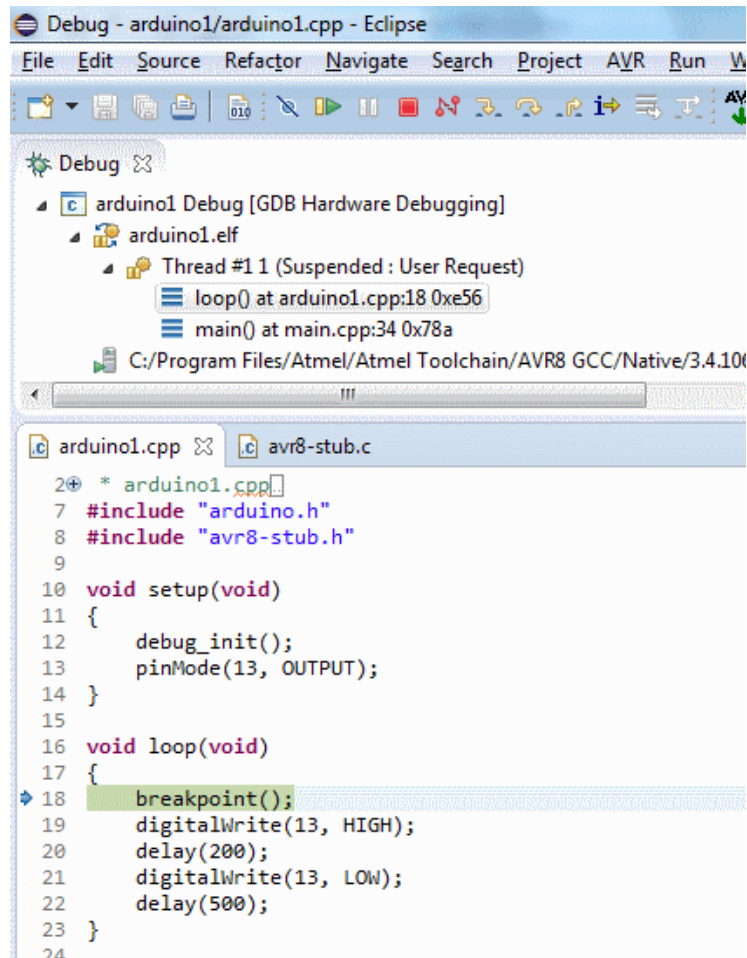
Close the Debug Configurations window.

Upload the program into your board if you haven't done it yet.

Start the TCP to Serial proxy server if you are not using direct serial connection (see Step 4 for details).

In Eclipse expand the Debug button, select Debug Configurations, and then your configuration and click Debug.

After a while, the Eclipse will switch to debug perspective and you should see something like this:



The program is stopped at the breakpoint line.

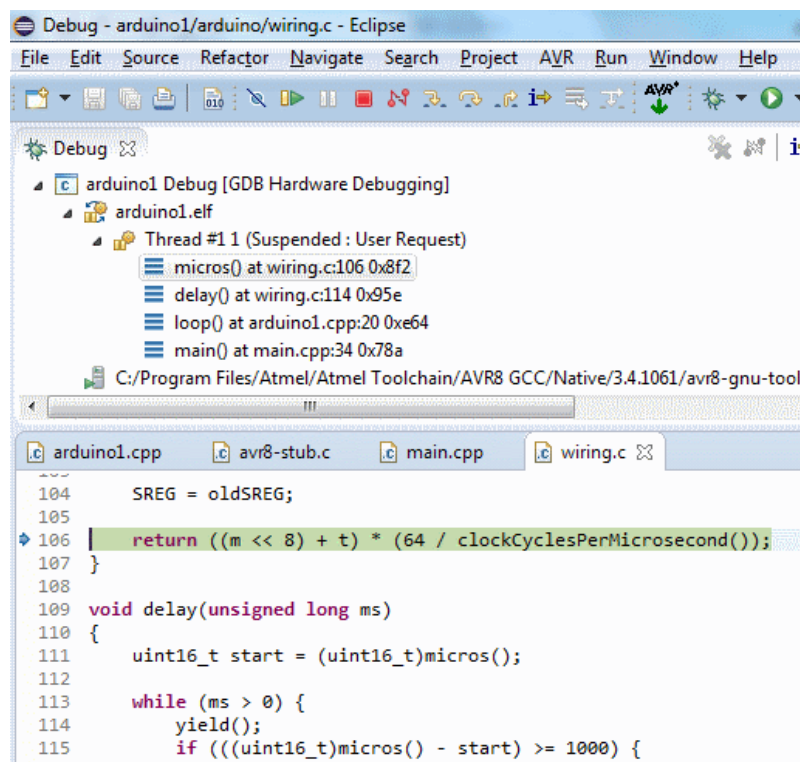
You can now step through the code (Step over button in toolbar) to see the LED go on, etc. Note that after stepping from the end of the loop, you will find yourself in the Arduino library's main.cpp file. If you continue stepping, you will get into your loop again. Also it seems like the setup is called again, but this is just discrepancy between the code you see in the C language and the real code generated by the compiler; the setup is not really executed again. You can use the Resume button to let the program run until it hits the breakpoint we have "hard-coded" at the beginning of loop.

You can also place breakpoints by right-clicking the left margin and selecting Toggle Breakpoint from context menu.

You can also remove the call to breakpoint() from the loop if you want to see the program running at full speed. Note that you need to rebuild and re-upload the program into the board. See Step 4 for details.

Note about debugging the program without the breakpoint() function in the code

If you do not place the call to `breakpoint()` function into your program, it will run (LED blinking) right after upload. When you connect with the debugger, it will stop at any random place; most likely somewhere in the `delay()` code. You may see something like this:



In the upper window (Debug) there is so called **call stack**, the “chain” of calls which led into current place. The program is stopped inside the `micros()` function, which was called from the `delay()` function, which was itself called from `loop()` function and so on.

To quickly get to your own code, click `loop()` in the Debug window (select the `loop` function). This will display code of the `loop` function in the lower window. Now you can place a breakpoint, for example, on the `digitalWrite(13, HIGH);` line and resume the program. It will stop at the breakpoint.

Alternative ways of solving the multiple definitions of vector1 error

As mentioned above, by excluding `WInterrupts.c` from build, your program cannot use the `attachInterrupt` Arduino function.

If you need to use `attachInterrupt` in your program, do not exclude the `WInterrupts` file from build, but just disable (comment out) the code related to the `INT0` interrupt in this file.

There are three ways how to do it:

- Comment the code out
- Put the code into conditional block `#ifndef`
- Use modified `WInterrupts.c` file provided with this debugger with the conditional blocks already added and define the `AVR_DEBUG` symbol in compiler symbols for your project. The modified file is located in `avr_debug/arduino` folder. You can copy it to your Arduino

package replacing the original file. Please **do make a backup** and use the appropriate file for your version of Arduino. The file can be changed in new versions of Arduino and if you replace the original file with one from older Arduino version, it may not work.

Here the options are described in detail.

Option 1 - Comment out the code

Open the WInterrupts.c file from eclipse (it is located in the Arduino virtual folder in your project).

Locate this code:

#else

```
ISR(INT0_vect) {  
    if(intFunc[EXTERNAL_INT_0])  
        intFunc[EXTERNAL_INT_0]();  
}
```

Note the exact look, including the #else at the beginning!

The same code is located at 3 points in the file, this is the last one; at line about 305. You can use the red dot in the left margin indicating the place of the error to quickly locate the code.

Comment the block out by placing /* above the ISR() and */ below the “}”; it should look as follows:

#else

```
/*  
ISR(INT0_vect) {  
    if(intFunc[EXTERNAL_INT_0])  
        intFunc[EXTERNAL_INT_0]();  
}*/
```

By this you just disable the code which handles INT0 (the interrupt used by the debug driver), but still be able to use the other interrupts.

You should now be able to build the project.

Important note

The advantage of this method compared to exclude from build is that you can still use the attachInterrupt function.

The **disadvantage** is that attachInterrupt will not work for the interrupt INT0 in any program. The change in the WInterrupts.c file affects all your Arduino projects.

Option 2 - Use conditional compilation block

You may want to do this instead of commenting the block out

```
#ifndef AVR_DEBUG  
ISR(INT0_vect) {  
    if(intFunc[EXTERNAL_INT_0])  
        intFunc[EXTERNAL_INT_0]();  
}  
#endif
```

Then define the AVR_DEBUG symbol in project properties > C/C++ Build > Settings > AVR Compiler > Symbols. This way only projects with AVR_DEBUG symbol defined will exclude this code. Other project will not be affected.

Opening example projects

The debugger for Arduino comes with example programs. You can import these programs into your workspace in Eclipse to quickly try the debugger. This is easier than creating your own program from scratch as described in the other parts of this document.

Set up your development environment

First, please set up your development environment as described in Step 1 chapter in this document.

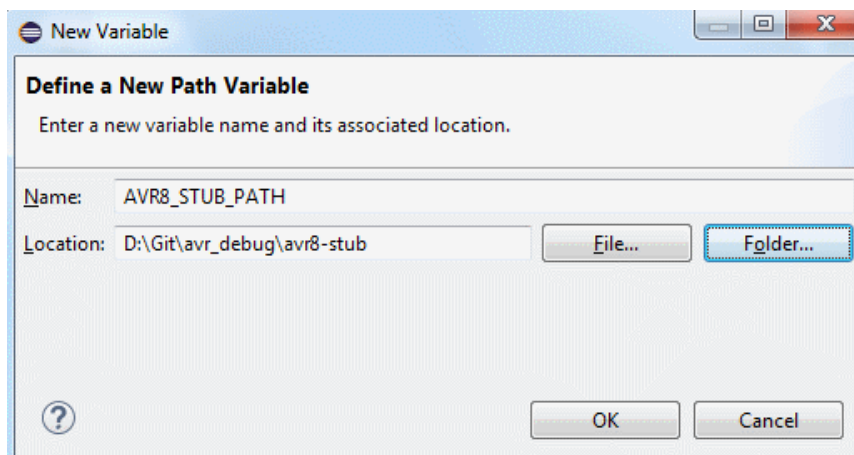
Create path variables

Next, we need to create path variables which are used in the example projects to refer to the location of the Arduino software library and the debug driver. This is only done once in a new workspace. If you've already created these variables in your workspace, skip this step.

Go to Eclipse menu **Window > Preferences** and expand **General > Workspace > Linked resources**.

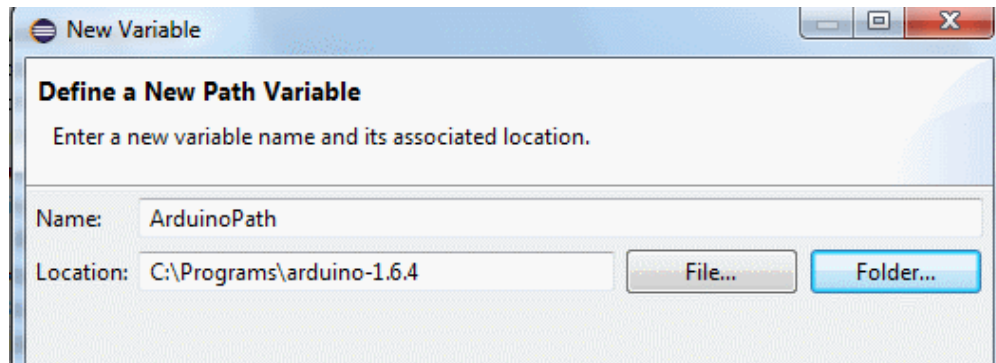
In the right-side part of the window create new Path variable using the New button:

The name must be: AVR8_STUB_PATH. Use the Folder... button in the New Variable window to point the variable to the location of the debug driver files on your system. For example, c:\avr_debug\avr8-stub.

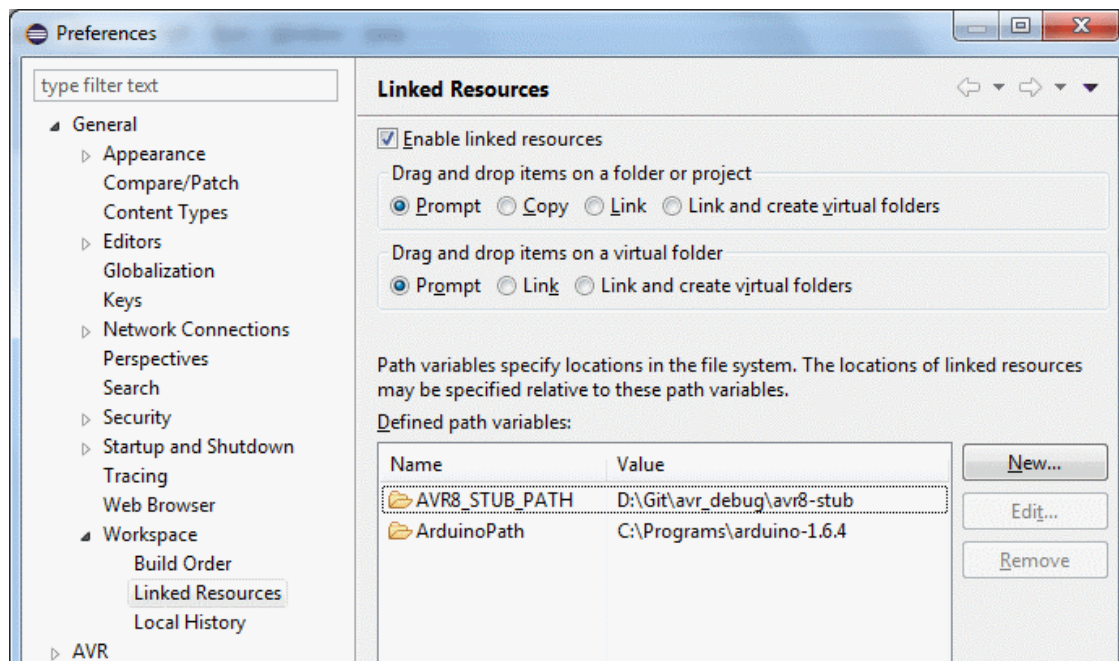


In the same way create another variable named ArduinoPath.

Set the location to the root of your Arduino installation, for example C:\Programs\arduino-1.8.5.



There should now be these two variables in the list:

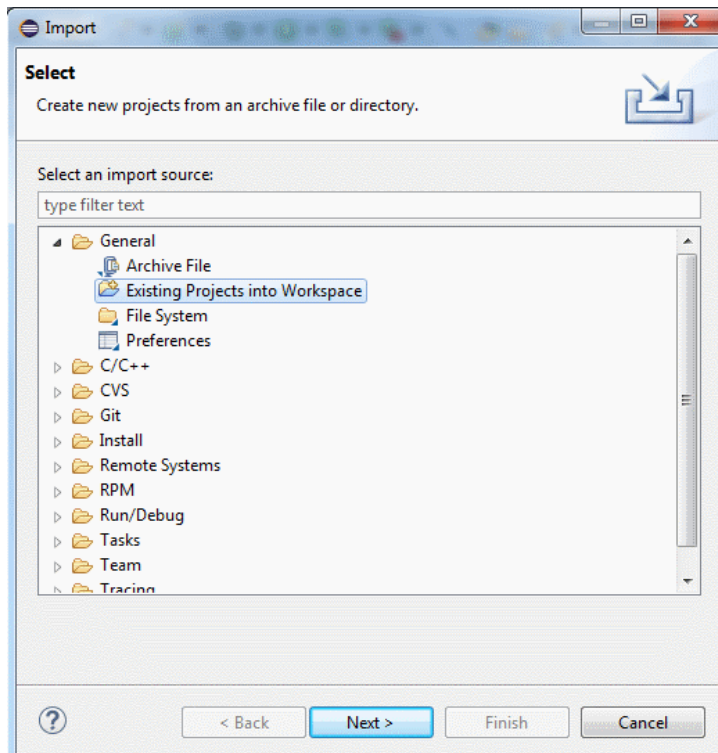


Import example projects

To import an example program into your workspace:

Start Eclipse and select **File > Import** from the main menu.

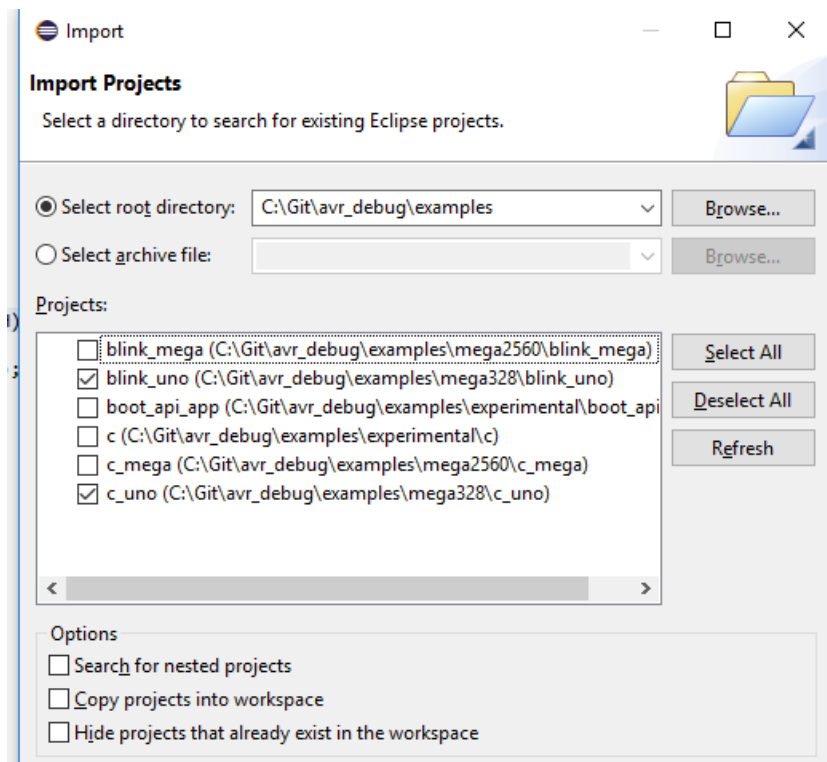
In the Import window select **General > Existing projects into workspace**. Click **Next**.



In the “Select root directory” use the Browse button to locate the examples folder in the AVR debug directory, for example `c:\avr_debug\examples`. You should see the example projects in this folder.

For Arduino Uno there are projects **blink_uno** or **c_uno**.

Select one or more projects to import. Make sure the “Copy projects to workspace” option is NOT enabled.



Build the project(s). There should be no errors. But note that if you are in a new workspace, you need to configure the paths in AVR eclipse plugin – see 1.5 Configure the AVR Eclipse plugin.

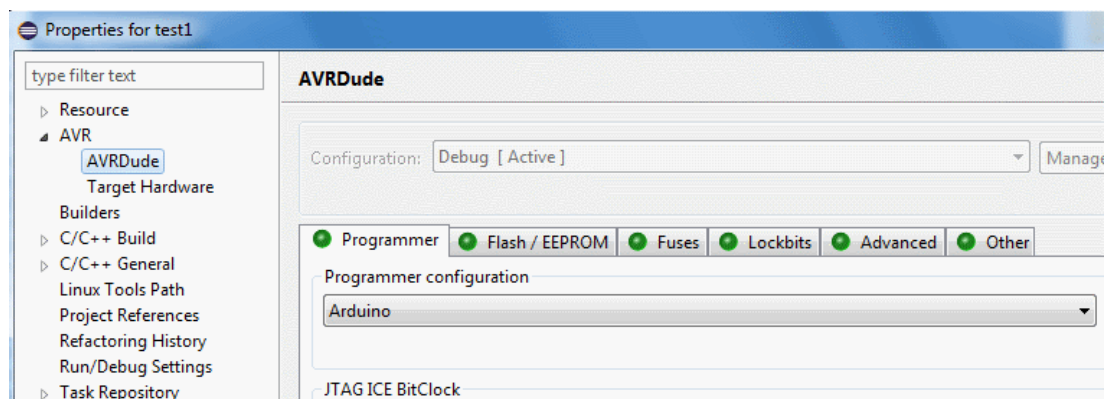
Note that the project with Arduino libraries (blink_uno) assumes that you have modified version of the file WInterrupts.c in your Arduino location – the project uses the AVR_DEBUG symbol for conditional compilation as described in the Alternative ways of solving the multiple definitions of vector1 error section. You may need to use the Clean command before building after you change the file.

Before uploading to the board we need to select the programmer. To do so:

Go to preferences for your project (Alt + Enter or right-click the project and select Properties).

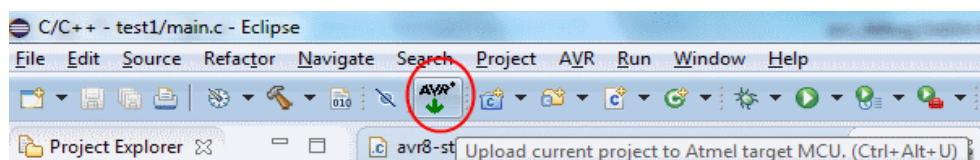
Expand AVR > AVRdude category.

On the Programmer tab select your Programmer configuration from the list. There should be the “Arduino” configuration we have created when setting up the development environment (see the Step 1 chapter, Configure the AVR Eclipse plugin subsection).



Close the Properties window with the OK button.

Click the AVR icon in the toolbar (Upload current project) to upload the program to Arduino.



In the Eclipse console window you should see how your program is uploaded, ending with "avrdude done. Thank you."

Debug example projects

Now we will connect to the program with debugger. There are two options:

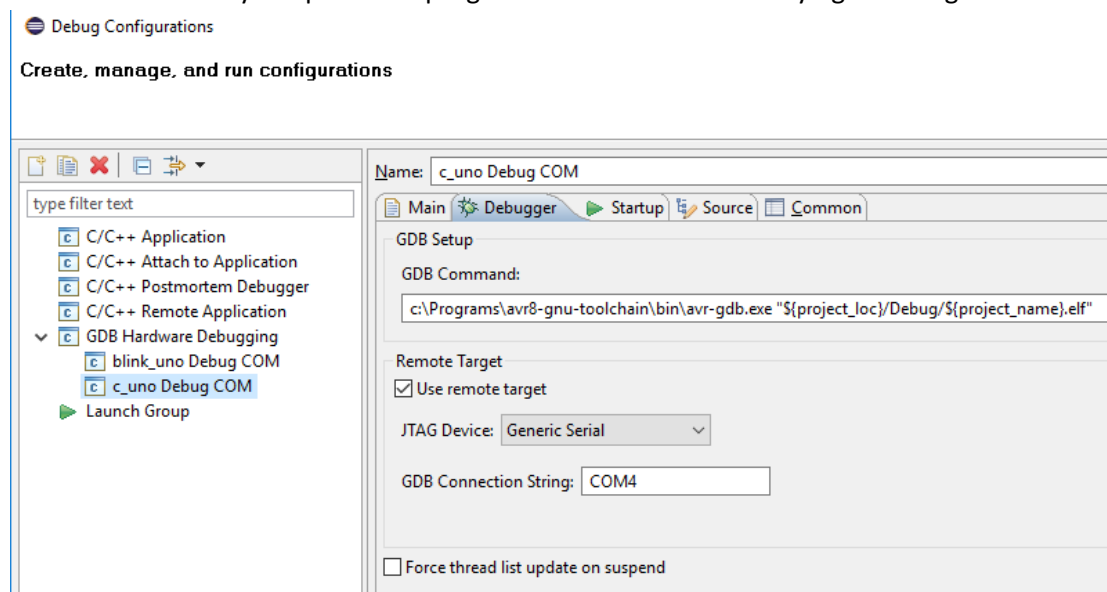
- Direct serial connection
- Connection via TCP to serial port converter (proxy server)

The direct connection is easier to use, so I recommend trying it first. If it does not work, use the connection via TCP-to-Serial proxy.

Direct serial connection

There is debug configuration included in the project. You just need to change the serial port number – COMx. To do so:

- Right-click the project in eclipse and select **Debug As... > Debug Configurations** from the context menu.
- In the Debug configurations window expand **GDB Hardware debugging** category and select the configuration for the project you wish to debug, e.g. c_uno Debug COM.
- On the right side select Debugger tab.
- Change the number of the COM port to the port where you Arduino is connected. See the picture below.
- In the GDB Command box make sure the path to the GDB debugger (avr-gdb.exe) is valid. The example projects use the default path but if you installed the Atmel AVR8 toolchain into another location or if you have another version, the path will not be valid.
If you need to change the path, use the Browse button to select the avr-gdb.exe. Then enter a space after the path into the edit box and then paste the following line:
"`${project_loc}/Debug/${project_name}.elf`".
- You can now start debugging by clicking the Debug button in the bottom part of this window. Make sure you upload the program to the board before trying to debug it.



Connection via TCP-to-Serial proxy

If the direct serial connection does not work, use this option.

Before starting debug session in eclipse, we need to start a special program which converts the commands sent by the debugger to a TCP port to serial port where the Arduino board is connected (COM to TCP proxy server).

Use your file manager to open the folder where the Arduino debugger package is located. For example, c:\avr_debug. You should see a start_proxy.bat file in this folder.

Open the start_proxy.bat file in Notepad or other text editor (right-click the file and select Edit or drag and drop it into Notepad window).

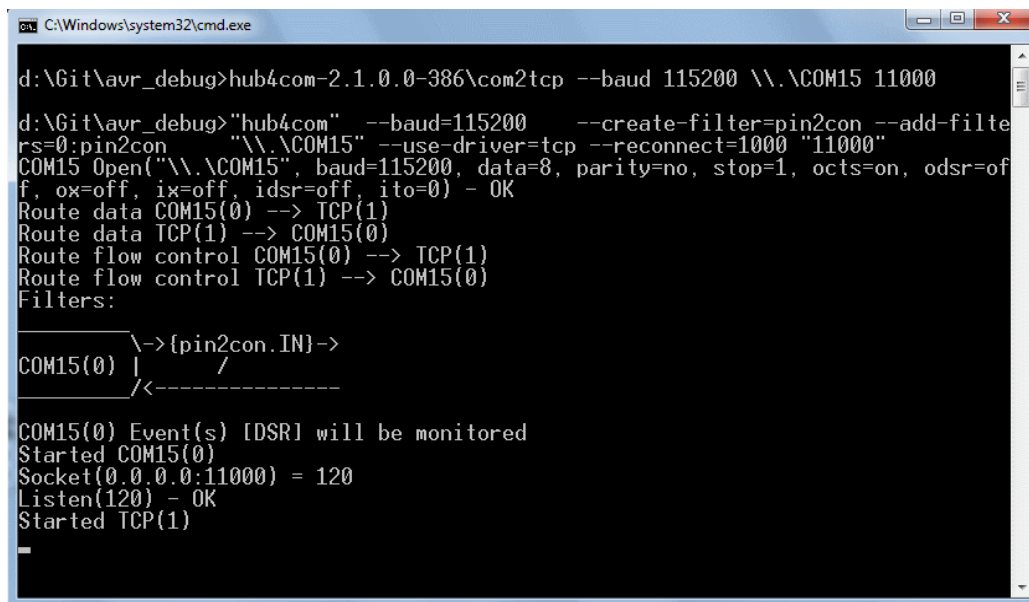
Change the number of the COM port in this file. There is this line:

```
hub4com-2.1.0.0-386\com2tcp --baud 115200 \\.\COM15 11000
```

Just change the number after COM from 15 to the number of your COM port to which the Arduino board is connected. If you prefer, you can also use the com2tcp program directly; use the command in this .bat file as an example.

Save and close the start_proxy.bat file.

Run the start_proxy.bat file. This will start the proxy server. You should see a console window with some information. This window will be opened all the time during the debugging.



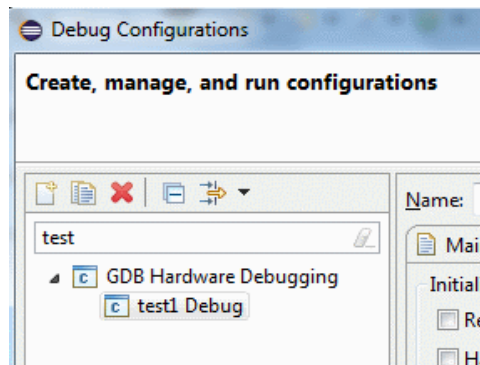
```
C:\Windows\system32\cmd.exe
d:\Git\avr_debug>hub4com-2.1.0.0-386\com2tcp --baud 115200 \\.\COM15 11000
d:\Git\avr_debug>"hub4com" --baud=115200 --create-filter=pin2con --add-filter=0:pin2con "\\.\COM15" --use-driver=tcp --reconnect=1000 "11000"
COM15 Open("\\.\COM15", baud=115200, data=8, parity=no, stop=1, octs=on, odsr=off, ox=off, ix=off, idsr=off, ito=0) - OK
Route data COM15(0) --> TCP(1)
Route data TCP(1) --> COM15(0)
Route flow control COM15(0) --> TCP(1)
Route flow control TCP(1) --> COM15(0)
Filters:
  \->{pin2con.IN}->
COM15(0) | /
  /<-----
COM15(0) Event(s) [DSR] will be monitored
Started COM15(0)
Socket(0.0.0.0:11000) = 120
Listen(120) - OK
Started TCP(1)
```

Note: You may want to configure your firewall to block access to the port 11000 from other computers. You can also change the port number both in the .bat file and in the Eclipse debug configuration.

Now return to Eclipse.

Right-click the project and select Debug as > Debug Configurations from the context menu.

In the Debug Configurations window you should see a configuration with name in this format "[project_name] Debug" under the GDB Hardware Debugging category. For example, if you are using the "blink" example project, there will be "blink Debug" configuration.



Select the configuration and in the right-side window select the Debugger tab.

In the GDB Command box make sure the path to the GDB debugger (avr-gdb.exe) is valid. The example projects use the default path but if you installed the Atmel AVR8 toolchain into another location or if you have another version, the path will not be valid.

If you need to change the path, use the Browse button to select the avr-gdb.exe. Then enter a space after the path into the edit box and then paste the following line:

"\${project_loc}/Debug/\${project_name}.elf".

Here is example of the value for this field using tools from Arduino package:

c:\Programs\arduino-1.6.5-r2\hardware\tools\avr\bin\ avr-gdb.exe

"\${project_loc}/Debug/\${project_name}.elf".

And from Atmel Toolchain:

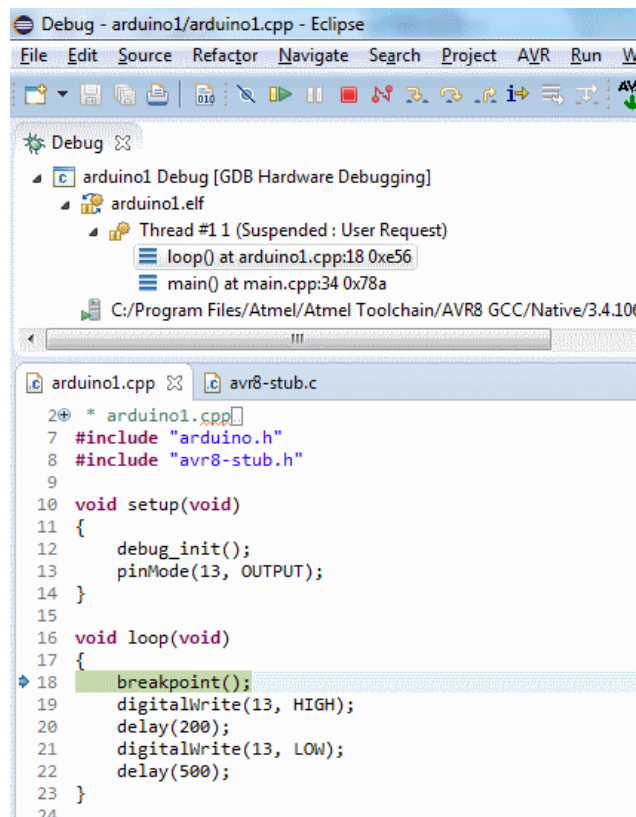
C:\Program Files\Atmel\Atmel Toolchain\AVR8 GCC\Native\3.4.1061\avr8-gnu-toolchain\bin\avr-gdb.exe "\${project_loc}/Debug/\${project_name}.elf".

Click Apply button to save the changes, if needed.

Click the Debug button in lower right corner of the window.

Debug session

After a while, the Eclipse will switch to debug perspective and you should see something like this:



The program is stopped at a line where breakpoint() function is placed in the code.

You can now step through the code (Step over button in toolbar) to see the LED go on, etc. Note that after stepping from the end of the loop, you will find yourself in the Arduino library's main.cpp file. If you continue stepping, you will get into your loop again. Also it seems like the setup is called again, but this is just discrepancy between the code you see in the C language and the real code generated by the compiler; the setup is not really executed again. You can use the Resume button to let the program run until it hits the breakpoint we have "hard-coded" at the beginning of loop.

Of course, you can also place breakpoints by right-clicking the left margin and selecting Toggle Breakpoint from context menu.

Please see Step 4 chapter for more information on debugging.

How to enable breakpoints in flash memory

This section provides instructions on using the flash breakpoints for debugging. For explanation of flash breakpoints and their comparison to RAM breakpoints please see the chapter RAM vs Flash breakpoints – pros and cons.

Here is overview of the steps needed to use flash breakpoints:

1. Update the bootloader in your Arduino Uno board
2. Change the option in `avr8-stub.h` to enable flash breakpoints
3. If Needed: Add files `app_api.h` and `app_api.c` to your project – ONLY if you are using the bootloader provided with this package, see the June 2020 update below.

June 2020 Update

There are now 2 options for enabling the flash breakpoints:

- Use the bootloader provided with this package
- Use Optiboot bootloader available here (<https://github.com/Optiboot/optiboot>).

Optiboot is the bootloader used by default in Arduino Uno. It supports many other Arduino boards (Atmel AVR MUCs).

The Optiboot bootloader since version 8 supports writing to flash memory so it can be used by this debugger stub to implement flash breakpoints. Optiboot is the standard bootloader used for Arduino boards based on Atmega328 (Uno, Nano, Micro) but it supports many other Atmel AVR MUCs and thus also Arduinos, like the Arduino Mega.

In an ideal world you would not need to update the bootloader in your board and could use the flash breakpoints. In the current world, however, you will need to update the bootloader because the Arduino uses old Optiboot version 4.4 and writing to flash is supported only since version 8.

Optiboot pros and cons

- (+) the bootloader is smaller, you can use 1.5 kB more flash for your program compared to the bootloader provided with this package
- (-) loading via debugger is not available
- (-) the debugger uses more RAM
- (+) Optiboot is available for many AVR chips; the special bootloader included in this package is only available for Atmega328. If there is support for new MCUs added to this debugger, it will use the Optiboot for flash breakpoints. I don't plant to update the bootloader to support other chips – it is easier to use the Optiboot.

Table of supported configurations/features

AVR chip	Arduino board	RAM breakpoint	Flash breakpoints	Load via debugger
Atmega328	Uno, micro, nano	Yes	With custom bootloader and Optiboot	Yes with custom bootloader
Atmega1280	Mega, older version	Yes	Optiboot	No
Atmega2560	Mega, newer	Yes	Optiboot	No
Atmega1284(P)	Sleeping Beauty, MightyCore development board	Yes	Optiboot	No

Option 1 – using the custom bootloader provided with this package

These instructions apply to the custom bootloader provided with this package. For instructions for using standard Optiboot bootloader please [see Option2 below](#).

Step 1 - Update the bootloader in your Arduino board and change the fuses

You need to do this step only once per Arduino board. Once you have the bootloader in your Arduino, you can use the flash breakpoints and also upload the programs via the debugger. You can also upload the programs as usual from Arduino IDE or Eclipse via AVRDUDE.

The bootloader binary can be found in avr_debug/bootloader.

For Arduino Uno use the file **optiboot.hex** in avr_debug\bootloader\optiboot\Debug.

For other Arduinos based on Atmega328 it could be possible to use the same hex file or build the bootloader from sources – see the **Notes about bootloader** section below for more information or see the readme file in the bootloader folder.

Arduino Mega bootloader is not supported. Please use the Optiboot bootloader.

To “burn” the bootloader into your Arduino you need an ICSP programmer or another Arduino board to serve as such programmer. The software used for burning the bootloader can be the Arduino IDE, Atmel Studio or some custom program made by the manufacturer of the programmer. Please refer to the documentation for your programmer or search the internet.

The settings presented here are general information which you should be able to apply in any tool. Below is also screenshot with the appropriate settings in Atmel Studio Device programming window used with ICSP programmer compatible with STK500 protocol.

Please note that you also need to **change the fuse settings**.

Raw fuses values

EXTENDED: 0xFD
HIGH: 0xD2

LOW: 0xFF

Fuse settings for humans

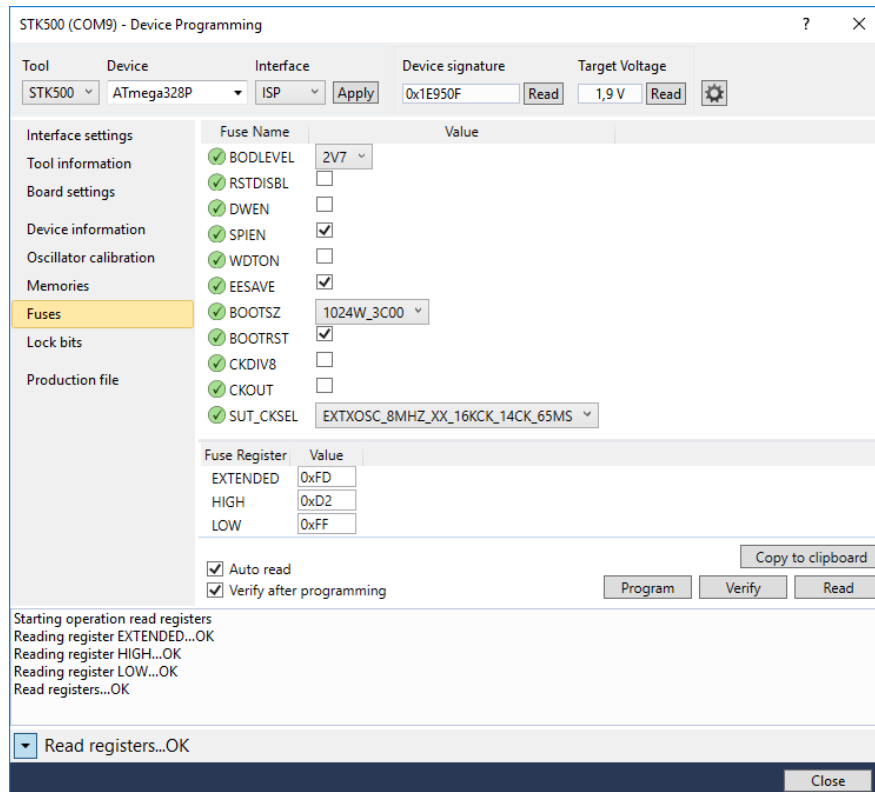
Enable SPIEN

Enable BOOTRST

Enable EESAVE

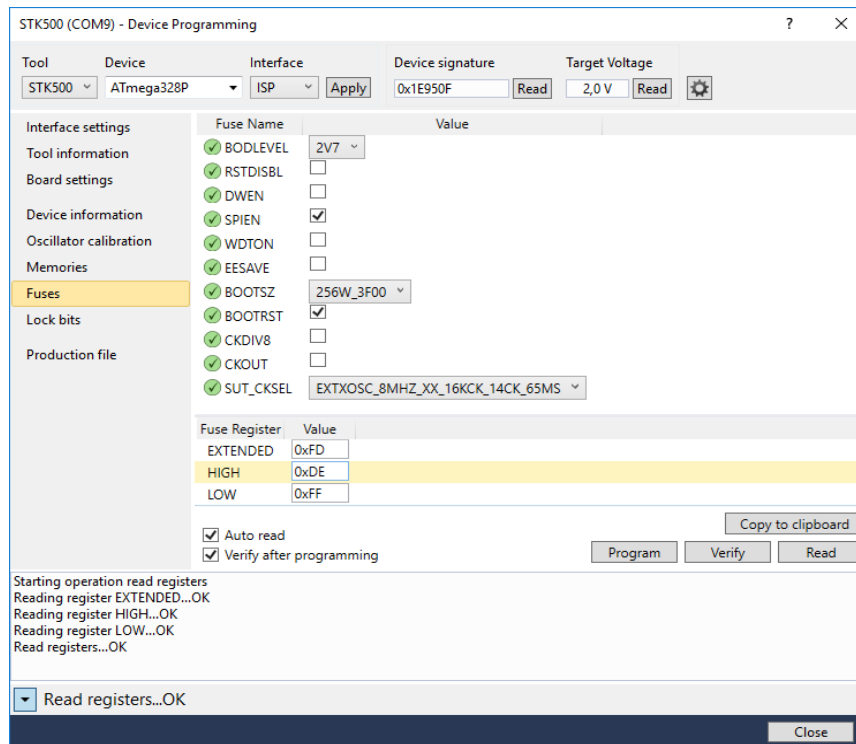
Set BOOTSZ to 1024 words (bootloader start address 0x3c00)

Set clock to EXT OSC 8 or 16 MHz



Fuse settings for the modified bootloader.

For your reference (and if you need to change the fuses back), here are the fuse settings for Arduino Uno as shipped from factory. The original bootloader is located in the Arduino installation in [Arduino folder]/ arduino-1.8.5\hardware\arduino\avr\bootloaders\optiboot\optiboot_atmega328.hex



Fuse settings for Arduino Uno for the standard bootloader (factory shipped).

Updating the bootloader from Arduino IDE

You can use the Burn bootloader command in Arduino IDE to update the bootloader. To do so:

- Copy the hardware folder into your Arduino sketchbook folder, for example, Documents/Arduino.
- Restart Arduino IDE if it is running.
- In the Tools menu in Board you should see avr-debugger (in Sketchbook) category. Under it there are 2 options: the bootloader for Atmega328 for Arduino Uno and Optiboot bootloader for Arduino Mega 2560.
- Select the board to burn the bootloader to your Arduino board you also need to set the Programmer in the tools menu depending on the hardware ISP programmer you have and maybe also the port of this programmer.

Arduino Mega 2560 note

The Arduino Mega bootloader provided in this package is the binary (hex) of the original Optiboot bootloader available here: <https://github.com/Optiboot/optiboot>

The binary is included here for convenience because the Optiboot release archive does not contain binary for Atmega2560 and building it from sources can be complicated for many users (the original instructions seem outdated).

If you want to use the Optiboot bootloader for Arduino Uno or any other board, please see the instructions below in [Option 2 – using the standard Optiboot bootloader](#).

Notes about bootloader

The modified bootloader works also for normal uploading from Arduino IDE or Eclipse using AVRdude. It is not limited to uploading via the debugger. So you can use your Arduino with this bootloader as usual and program it, for example, with Arduino IDE.

The bootloader is bigger than the standard bootloader shipped with Arduino Uno. It uses 2 KB of program memory. So with this bootloader your programs are limited to 30 KB of program memory (flash).

If you want to build the bootloader yourself, there is an **eclipse project** in `avr_debug/bootloader/optiboot`. You can import this project into Eclipse and build it in the same way as the example projects provided with this debug driver. You can also use the **PlatformIO** to build the bootloader, please see the `readme_platformio.txt` file in the `avr_debug/bootloader/optiboot`.

Step 2 - Change the option in `avr8-stub.h` to use flash breakpoints

Open the file `avr8-stub.h`. You will have this file in your eclipse project when you add this debugger to your program.

Locate the following line in the file:

```
#define AVR8_BREAKPOINT_MODE (0)
```

Set the value of `AVR8_BREAKPOINT_MODE` to 0.

Value 0 means that the breakpoints are written to flash memory. Detailed explanation of the possible values can be found in the comment above the definition.

Step 3 - Add files `app_api.h` and `app_api.c` to your project

The files `app_api.h` and `app_api.c` are located in the `Arduino debugger\avr8-stub` folder – in the same place as the main debug driver files `avr8-stub.h` and `avr8-stub.c`.

Drag and drop these files into your project in Eclipse.

These files contain the interface between the debug driver and the bootloader. This is needed to write to flash memory.

Note: if you will not use flash breakpoints or load through the debugger, remove these files from the project or exclude them from build. This will make the program smaller.

Option 2 – using the standard Optiboot bootloader

These instructions apply to the standard Optiboot bootloader available here <https://github.com/Optiboot/optiboot>.

Arduino Mega 2560 note

As on June 2020 the Optiboot release package with pre-built binaries does not contain the .hex file for Atmega2560 or ATmega1284. **If you are using Arduino Mega 2560, the instructions below will**

not work. You can use the .hex file included in this package – see [Updating the bootloader from Arduino IDE](#). Or if you are brave enough you can build the bootloader from sources.

Step 1 - Update the bootloader in your Arduino board and change the fuses

The instructions for burning the bootloader can be found in the Readme file in the Optiboot package here <https://github.com/Optiboot/optiboot>. See the section **To install into the Arduino software** in the Readme file.

The author recommends using supported "Arduino Core" which is probably easier but I'd prefer to use the Board manager packages as mentioned later in the Readme. Here is a quote of the instructions for your convenience:

The following instructions are based on using the Arduino "Board Manager", present in IDE versions 1.6.5 and later.

- Find the desired Optiboot release on the [Optiboot Release page](#).
- Use the "Copy link address" feature of your browser to copy the URL of the associated .json file.
- Paste this url into the "Additional Boards Manager URLs" field in the Arduino IDE "Preferences" pane. (Separate it from other URLs that might be present with a comma or click the icon to the right of the field to insert it on a new line.)
- After closing the Preferences window, the Tools/Boards/Boards Manager menu should include an entry for that version of Optiboot. Select that entry and click the Install button.

Basically by following the above steps you will be able to burn the ready-to-use Optiboot bootloader into your board. In the Arduino IDE Tools menu you will find Optiboot 8.0 item with sub-menu of the available MCUS. You then select the "CPU", the Processor and Clock speed in the Tools menu and can burn the bootloader.

Here is example for the Arduino Uno (Atmega328):

- For Board select Optiboot on 28-pin cpus
- For Processor select Atmega328P
- For CPU speed select 16 MHz.
- Now you can select your programmer and burn the bootloader.

Note: To "burn" the bootloader into your Arduino you need an ICSP programmer or another Arduino board to serve as such programmer.

Step 2 - Change the option in avr8-stub.h to use flash breakpoints

Open the file avr8-stub.h and locate the following line:

```
#define AVR8_BREAKPOINT_MODE (2)
```

Set the value of AVR8_BREAKPOINT_MODE to 2.

Value 2 means that the breakpoints are written to flash memory using Optiboot `do_spm` function. Detailed explanation of the possible values can be found in the comment above the definition.

Debugging your program with flash breakpoints

There is virtually no difference in usage when you switch to flash breakpoints. One example of a program where you can see the difference is a program with delay implemented as busy loop.

If you have code like this in the main loop:

```
digitalWrite(13, HIGH);  
my_delay();  
digitalWrite(13, LOW);  
my_delay();
```

And the delay function looks like this:

```
void my_delay(void)  
{  
    unsigned long i = 50000;  
    while ( i > 0 )  
        i--;  
}
```

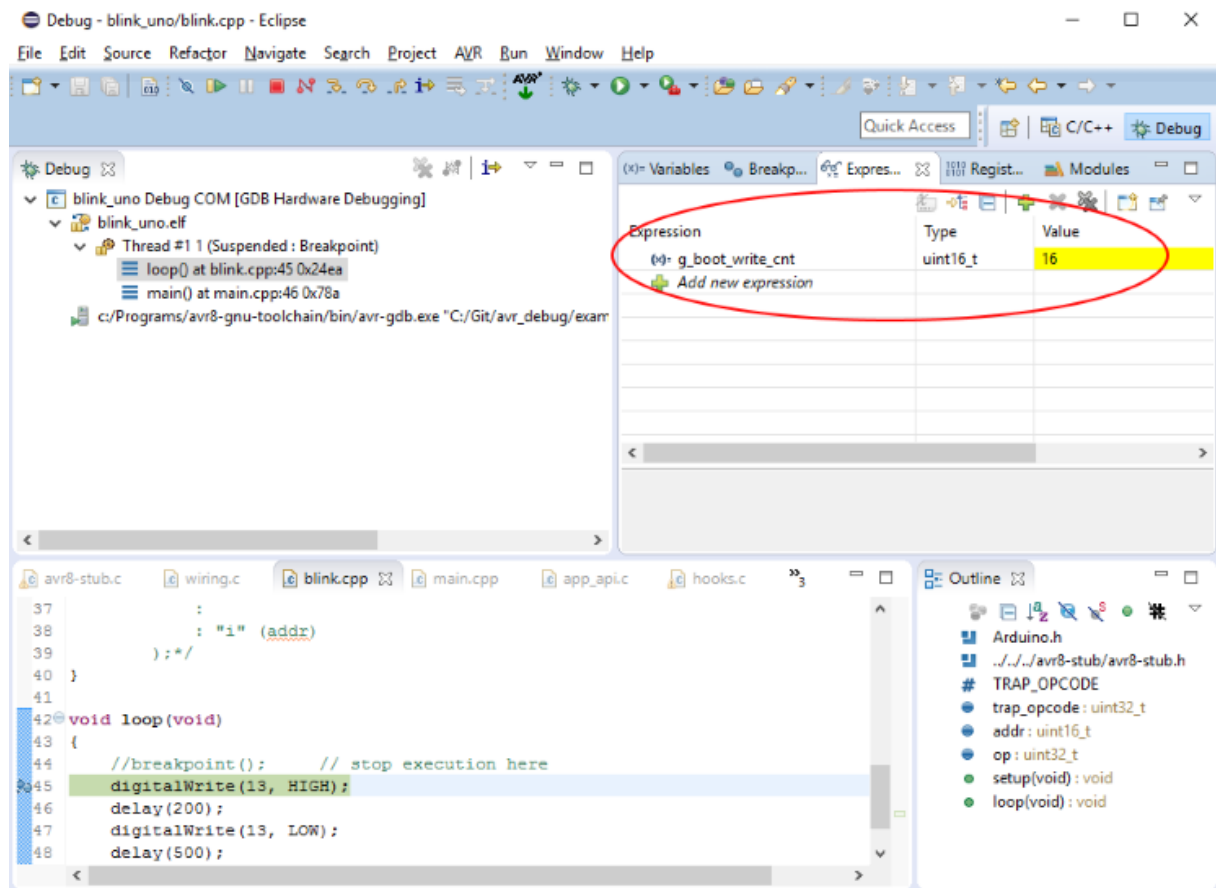
Then using RAM breakpoints it will take several seconds to step over the `my_delay` functions. If the program runs without breakpoints, the delay is some fraction of a second, but if there is a breakpoint, it takes several seconds. With flash breakpoints there is no such difference; the delay should still take only fraction of a second to execute.

Please note that the flash memory is rewritten several times when you debug with flash breakpoints and that the flash memory can survive only limited number of writes. This is discussed in detail in chapter Using breakpoints in flash memory.

You can view how many times the flash memory has been written to during debug session if you enable certain option in the debug driver. To do so:

- In your project open the `app_api.c` file.
- Uncomment the line **`#define AVR8_API_DEBUG`**
- In Eclipse when debugging go to menu Window > Show View > Expressions.
- In the Expressions window click new line with “Add new expression” and copy-paste or type this name of a variable: **`g_boot_write_cnt`**.
- This variable shows how many times flash memory was written. Note that this counts the total writes in any location of the memory, not the writes into particular page of the memory.

The picture shows the flash write counter in the Expressions window in eclipse.



There is one breakpoint at the digitalWrite(13, HIGH) command at line 45. When you step through the program, you will notice that the write counter variable g_boot_write_cnt increases by 2 for each step. This is because to step over a function the debugger inserts a temporary breakpoints after the function call and continues the program. So there are actually 2 breakpoints written/erased each time – the one set by the user at line 45 and the temporary breakpoint.

Uploading your program via the debugger

This option is available if you update the bootloader in your Arduino (which is also required to use flash breakpoints). It allows you to upload and debug your program by just starting the debug session – no need to first upload via AVRdude and then connect with the debugger. For information on updating the bootloader please see chapter Option 1 – using the custom bootloader provided with this package.

To upload the program to Arduino via debugger you need to:

- Configure the debug driver
- Set the options in the Debug configuration to load the program.

Step 1 – Configure the debug driver

In your project in Eclipse open the file **avr8-stub.h**.

Locate the line with this definition: **#define AVR8_LOAD_SUPPORT (1)**

Set the value of the AVR8_LOAD_SUPPORT to 1.

Value 1 means that the debugger will support loading the program into target MCU.

Step 2 - Configure the Debug configuration to load the program

In your project in eclipse open the Debug configurations (right-click the project and select Debug As – Debug configuration).

In the list on the left hand side select your project under the GDB Hardware Debugging category.

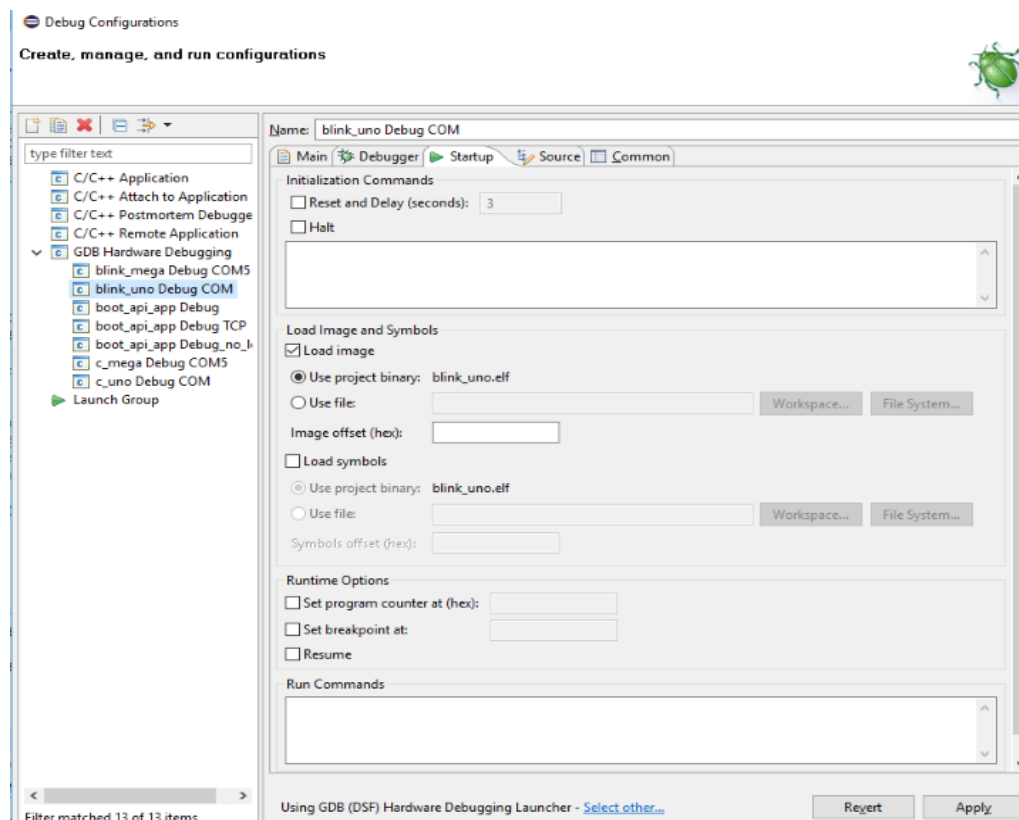
In the Debug configurations window go to the **Startup** tab.

Check the box **Load image**.

If you want the program to run after load you can also check the box **Resume** in the lower part of the window. Otherwise the program will be stopped at random location after load.

Do not enable any other options as they are not supported by the debug driver anyway.

The following picture shows the configuration for blink_uno project.



Tip: You can create two debug configurations for your project. One with load disabled and one with load enabled. If you need to connect to a program which is already running in the MCU or a program

uploaded via AVRdude, you can use the no-load configuration. If you need to upload the program, use the load configuration. This saves the flash memory from unnecessary rewrites because every load means overwriting the flash memory.

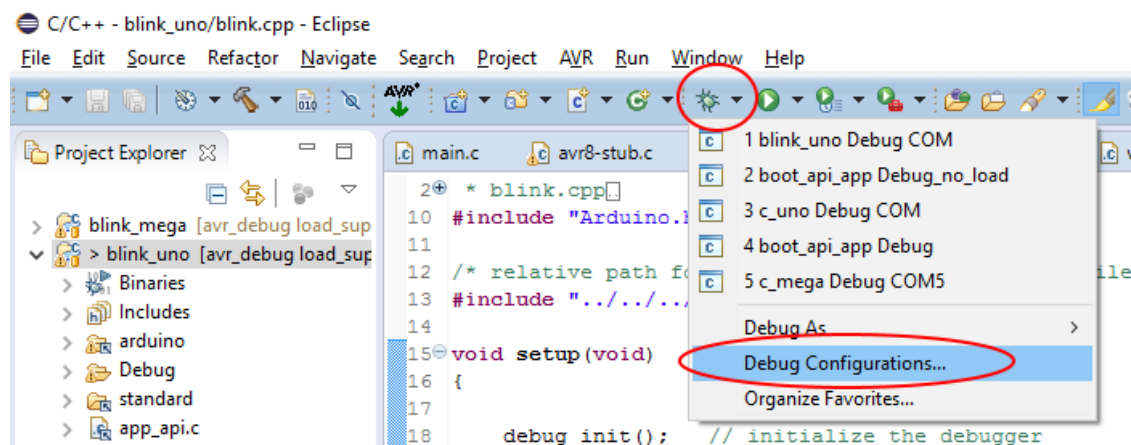
Important note: After enabling the load support in your program you need to upload the program using AVR dude (that is click the AVR button in the toolbar). Only after the program with load-via-debugger enabled is in the MCU, you will be able to load using the debug button. For explanation please see the Understanding the load function section.

Uploading and running the program

To upload and run the program after a change in code, just click the arrow of the Debug button and select Debug Configurations from the menu. Then select the configuration you want to run.

Eclipse will build the program, upload it and start debugging.

Note: To debug the program next time you can just select the configuration from the menu – there will be list of recently run configurations. So you don't need to go to the Debug Configurations window to run the program every time.



The program will be loaded and you will find it stopped in the debugger at random location (unless you've enabled the Resume option in the debug configuration).

Note that at this point the debugger does not really know where it stopped and it will show no call stack. Click the **Step over** button to advance the program one step and now the call stack will be shown correctly.

If you enable the Resume option the program will be running and you can stop it using the **Suspend** button. In this case it will show correct call stack.

You can use the Step Over (F6) command to continue stepping from the current location. Or you can insert a breakpoint in the program and use the Resume (F8) command to let the program run until it hits the breakpoint.

You can also insert call to **breakpoint()** function into your code to stop the program at desired point. For example, insert it into the setup function right after the initialization of the debugger as in the following picture.

```
15 void setup(void)
16 {
17
18     debug_init(); // initialize the debugger
19     breakpoint(); // stop execution here
20     pinMode(13, OUTPUT);
21     digitalWrite(13, HIGH);
22 }
23
```

Note that after the program is loaded it does not show the correct location of the breakpoint call. But after you execute Step Over command, it will point to the correct location and you can continue stepping from the breakpoint line onwards.

Understanding the load function

This section provides some background information about loading the program into the target MCU via the debugger.

When you instruct the debugger to load the program and start debugging, it tries to communicate with the target MCU to upload the new program. There must be a program in the target MCU already which is able to handle this communication.

For uploading your programs via AVRdude (from Arduino IDE), such a program is always present in the MCU – it is the bootloader. But for uploading via the debugger the debug driver is used (the code in the avr8-stub.c file). In order to load the program there must already be a program with the debug driver in the MCU. So **before you can upload via the debugger, you need to load your program into the MCU by AVRdude**. After this initial load, you can do the subsequent loads via the debugger.

In principle it would be possible to put the code for loading via debugger into the bootloader so it would be always present in the MCU, but there are several problems. First, it would increase the size of the bootloader. Second, normally the user program runs all the time; not the bootloader. When you load your program via AVRdude, it resets the MCU to interrupt the user program and activate the bootloader. But the debugger does not know how to reset the program before loading. The user would have to reset the MCU manually before uploading the program. So in general it would be possible to further improve loading via the debugger but I believe that the current version is usable – once you have some program with the debug driver in your MCU, you can upload via the debugger over and over again...until the program gets somehow corrupted, which should not happen often. And if it happens, you can just upload via AVRdude once and then again upload via the debugger.

Troubleshooting problems with debugging

Problem: Debug session fails to start.

Reason 1: Project is not selected in Eclipse Project Explorer.

Solution: If you encounter an error right after clicking the debug button; message tells you about “gdb –version”, select your project in the Project Explorer and try again. Alternatively, instead of using the Debug button in toolbar, right-click the project and select Debug As > Debug configurations. Select the debug configuration and click Debug.

Reason 2: COM port is not set correctly (error message about No such file or directory).

Solution: Make sure the Arduino board is connected and the COM port number is correct – verify this in Windows Device Manager. If it is correct, try to enter the COM port name in this format:

`\\.\COM10` (example for COM 10). It seems that for higher port numbers the simple name such as COM10 without the backslashes does not work.

Reason 3: Communication with the board cannot be established.

Solution: If you encounter an error later during the startup of the debug session, make sure the COM to TCP proxy server is running and also that your program calls the `debug_init()` function somewhere at the beginning (in `setup()` function if you use Arduino functions, or at the beginning of `main` for plain C/C++ programs).

Reason 4: The debugger itself (`avr-gdb.exe`) cannot be started – **could not determine GDB version**.

Solution: If you encounter error later during startup of debug session saying “could not determine GDB version with command...” or so, try to start `avr-gdb.exe` directly from your file manager – go to the folder when `avr-gdb.exe` is located and double click it. If it cannot be started because of some missing `.dlls`, install the Atmel AVR toolchain and use the `avr-gdb` from this toolchain instead of the one included in Arduino IDE package. For more information please see section

Problems with GDB (`avr-gdb`) included with Arduino in the Introduction of this document.

Reason 5: Trying to use flash breakpoints or load via debugger on a board without special bootloader.

Solution: If starting the debug session seems to hang at late stage (progress about 90%), it is possible that you have enabled the flash breakpoints or load support in your project (in `avr8-stub.h` file) but you have not updated the bootloader in your board. You need to use bootloader provided with this debug driver instead of the standard bootloader provided in Arduino by default. Please see Option 1 – using the custom bootloader provided with this package chapter.

The debug driver will stop the program in the call to `debug_init` in case it cannot find the special bootloader. Normally, the debug session should start and you should see this situation in the debugger. If it does not start, try changing the program so that it blinks the onboard LED in the main loop. If it does not blink it is likely that the program is stopped in `debug_init`.

Problem: The program is not jumping at some line or is jumping somewhere I don't want it to...

Reason 1: There is another program in the microcontroller than in the debugger.

Solution: If you made any changes in the program make sure you rebuild and re-upload the program to the board. If you just opened some older project, make sure you upload it to the board before you start debugging.

Reason 2: Compiler optimizations and code reordering by the compiler.

Often, when you debug your program, it does not behave as you expect. In most cases, this is not caused by an error in the debugger but by the difference between the program as you have written it (and see it in the debugger) and the actual code generated by the compiler. For example, when you write:

```
while ( i-- > 0 ) ;
```

The compiler can translate it into:

1: jump to line 3.

2: subtract 1 from i.

3: if (i > 0) jump to line 2.

This can happen even with no optimizations (-O0 option). Things can be even more confusing if you enable optimizations.

Solution: By default, use -O0 option (no optimizations) – this is enabled by default when you create new project in Eclipse. You can also try -Og which results in more optimized program but still with reasonable debugging experience – useful if your program grows too big to fit into the MCU with O0.

Reason 3: Interrupts are not enabled in your program.

Solution: Call sei() in your code. See example below.

Note: In Arduino programs interrupts are enabled automatically by the Arduino core code, but if you create a “plain” C/C++ program, you need to enable it yourself by calling sei().

```
int main(void)
{
    debug_init();
    DDRB |= _BV(5);    // pin PB5 to output (LED)
    sei();             // enable interrupts
    breakpoint();
    while(1)
    {
        ...
    }
}
```

Problem: The program does not stop on a breakpoint

Reason: There are more breakpoints set in the program than the debug driver supports.

Solution: Check how many breakpoints are there in the Breakpoints window in Debug perspective. If you cannot see this window, open it from the main menu Window > Show View > Breakpoints.

Delete the breakpoints you do not need.

To see the supported number of breakpoints search for AVR8_MAX_BREAKS symbol in avr8-stub.h file. Default is 8 breakpoints for RAM breakpoints and 4 breakpoints for Flash breakpoints. Note that you should not set more than this number – 1, e.g. more than 7 breakpoints in case of the breakpoints-in-RAM configuration. There should be one extra breakpoint available for use by the debugger – it sets temporary breakpoints when stepping over a function and so on.

Problem: The Console view in Eclipse shows error “No source file named ...”.

Reason: This is because Eclipse remembers all the breakpoints you set, even from other projects you debugged earlier. If you debug project “Test1” and place a breakpoint in test1.cpp file and then start debugging “test2” project, this error can appear for test1.cpp file.

Solution: Remove the unneeded breakpoints in the Breakpoints view – switch to the view in upper right corner of Eclipse and select Remove / Remove all breakpoints command.

Problem: The disassembly cannot be displayed

Reason: This is long time known bug in avr-gdb. See here for more information:

https://sourceware.org/bugzilla/show_bug.cgi?id=13519.

Solution: It should be possible to apply the patch mentioned on the webpage above to avr-gdb sources and build it on your own. I have not tried it yet because I can live without the disassembly.

Problem: Loading the program via the debugger ends with error “Load failed”.

Reason: The load via the debugger is not enabled in the debug driver or the program with this option enabled is not loaded in the MCU yet.

Solution: Please see the AVR8_LOAD_SUPPORT define in avr8-stub.h and make sure the load is enabled. Then rebuild your program and **upload it to the MCU using AVR Dude**. Once the program with this option enabled is in the MCU, you can upload via the debugger. Note that there is no need to update the bootloader after this change but you do need to have the modified bootloader in your MCU for this option to work.

Problem: Loading the program via the debugger ends with some error or hangs.

Solution: Try to increase the stack size in the debug driver – in file avr8-stub.c locate the definition of GDB_STACKSIZE and increase the number. Currently the size is set to 144. Increase it to, e.g. 176 and

try if it helps – be sure to first upload the program via AVR dude after this change and only then try to load it via the debugger again.

Arduino library

This chapter describes an alternative way of using this debugger as an Arduino Library. The library should make things easier for those who don't want to play with configuring the eclipse IDE.

The Arduino library is named **avr-debugger** and can be found in the `arduino/library` sub-folder. The sources and headers are exactly the same as those in the `avr8-stub` folder (unless I forget to update them after some change/bug-fix). There is no difference between the sources used in the standard way and the Arduino library sources. Ideally there would be only one copy of the sources but I wanted to provide the Arduino library as a self-contained package, so I copied the sources to the library. For future revisions of the debugger I plan to make any changes in the `avr8-stub` folder and then replace the files in Arduino library with these changed files.

To use the Arduino library just copy the `avr-debugger` folder into your folder with Arduino libraries – which on Windows is in `Documents/Arduino/libraries`.

Then you should see the `avr-debugger` library in the Contributed libraries categories in your IDE.

In your program, in the setup function **call the `debug_init()` function**.

Note that you cannot debug your programs in the Arduino IDE; there is no debugging support in this IDE. You can debug your programs in Visual Studio Code and possibly also in other IDEs which allow you to build Arduino programs and contain interface for the GNU debugger - GDB.

Using this debugger in Visual Studio Code

This chapter describes how to use this debugger in Visual Studio Code. I assume you are able to build your Arduino programs in VS Code, that is, you configured the Arduino extension.

Note: the tutorial assumes your output binary files is located in build sub-folder of the folder with your Arduino program. To achieve this, add the following line to the `arduino.json` file which you can find in `.vscode` sub-folder of your program folder:

```
"output": ".\\build",
```

Step 1 – Install the avr-debugger library

Just **copy the `avr-debugger` folder** into your `Documents/Arduino/libraries` folder. You should now have `Documents/Arduino/libraries/avr-debugger` folder.

Restart VS Code if it is now running so that the new library is loaded.

Step 2 – Add the library to your program

In VS Code open the Command Palette (F1) and type Arduino, then select Arduino: Library manager.

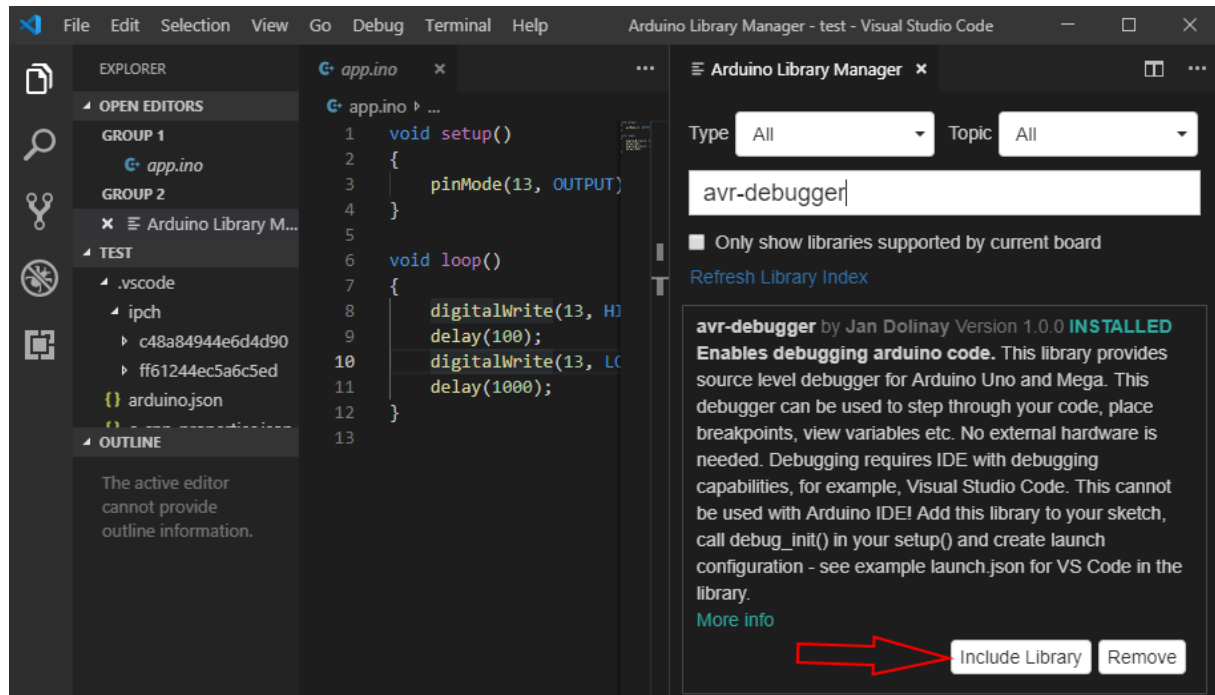
Type `avr-debugger` into the Filter your search... box.

The avr-debugger library should appear in the list.

Click the **Include library** button.

This will add 2 include files into your program:

```
#include <app_api.h>
#include <avr8-stub.h>
```



Add call to function `debug_init` to your setup function. See the picture below step 3.

Verify (build) the program and note the size. In my case it is 4852 bytes.

Step 3: Turn off compiler optimizations

The program is now built with optimizations, which means the compiler which translates your code into the native language of the microcontroller can make changes in the organization of the code to make it smaller. Of course, the compiler makes sure the program works the way you wrote it; it just makes it more efficient in some cases. So optimizations are good in general; they let you put more code into small memory of the MCU. But they are not so good for debugging because we want to be able to step through the code the way we wrote it and see it in the editor. So it's a good idea to turn off the optimizations when you debug your program.

To turn off the optimizations

In your favorite file manager go to the avr-debugger folder and locate file **platform.local.txt**.

Copy this file to the folder where your Arduino IDE is installed into sub-folder `hardware/arduino/avr`.

For example, on my system the platform.local.txt file should be placed in c:\Program Files (x86)\Arduino\hardware\arduino\avr\. You may need admin privileges to copy the file into Program Files.

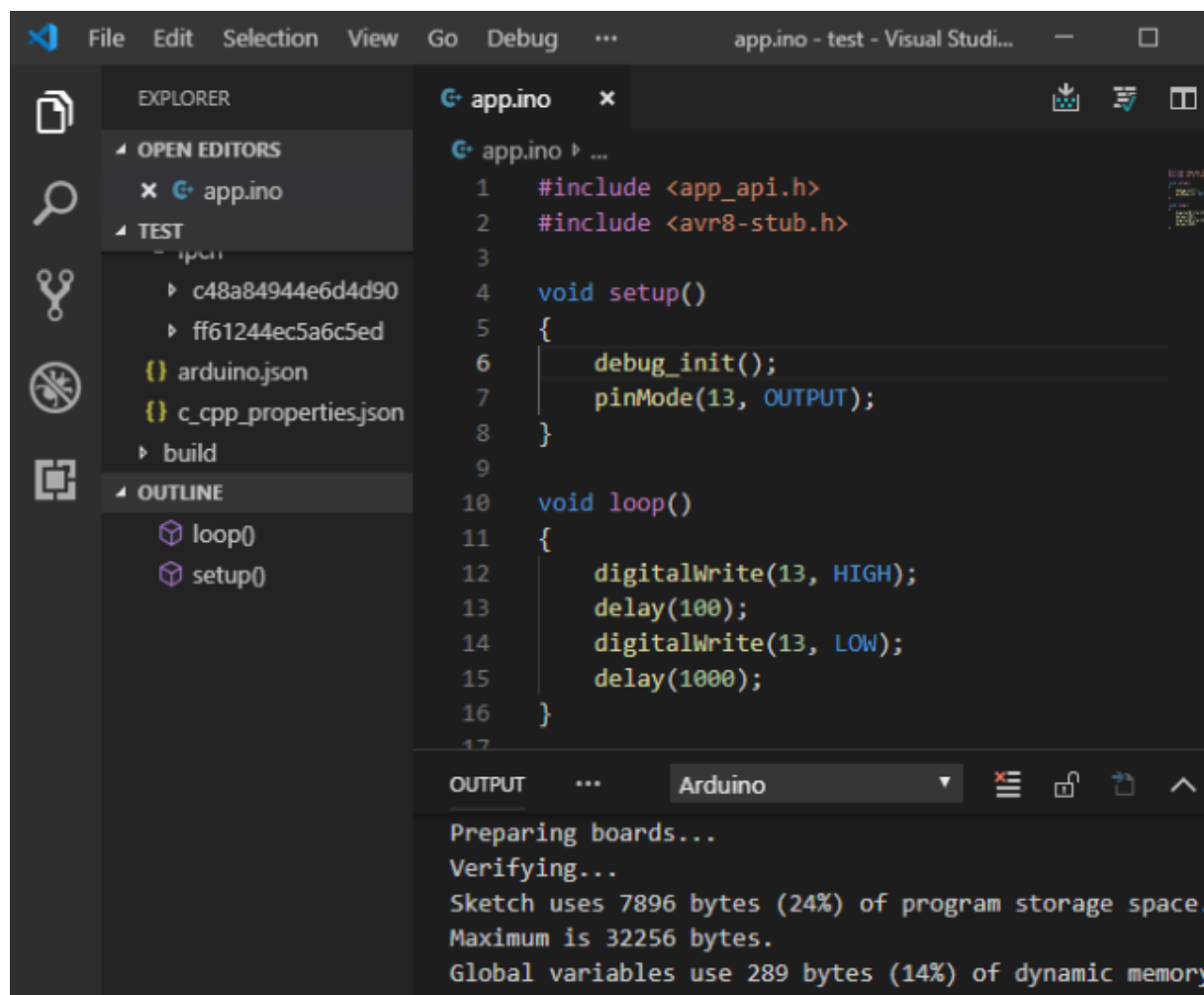
Restart VS Code.

In File manager also delete the contents of the build subfolder in your program folder – Documents/Arduino/vscode/test/build.

Verify the program again and note the size. It should be much larger than before.

In my case 7896 bytes for simple blink program with optimizations off.

If the program is not about this size, there is something wrong. Try deleting the build folder again and restarting VS Code again and double-check that the location of the platform.local.txt file is correct.



Two notes about turning off the optimizations

1. This step is optional - if you don't use the platform.local.txt file, you will still be able to debug your programs; it just sometimes may do strange things like jumping to another line than expected.

2. The platform.local.txt file is the only way I found to define compiler options for Arduino. It affects all your Arduino programs including those built in Arduino IDE. When you are done debugging, you can delete or rename this file to get back to normal, optimized programs. I wish it was possible to set the build options per project but it is not; the platform.local.txt file must be in above mentioned folder to work.

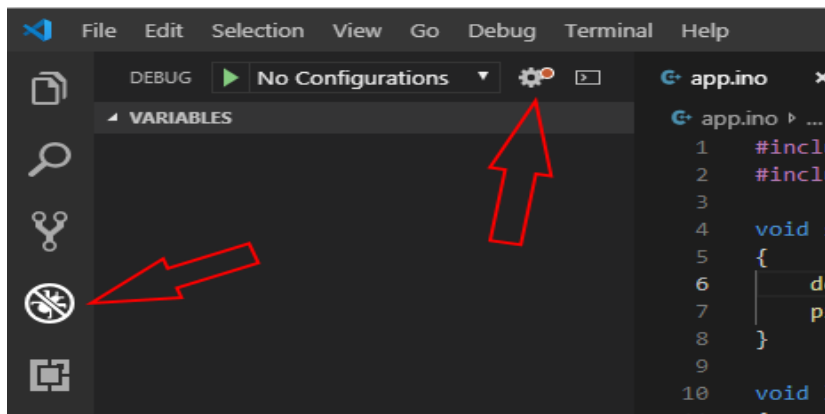
Step 4: Create launch.json file – launch configuration

To be able to debug the program in VS Code, you need to create launch configuration which tells VS Code how to start your program.

Click the debug button in the Action bar on the left-hand side – it's the icon with the bug.

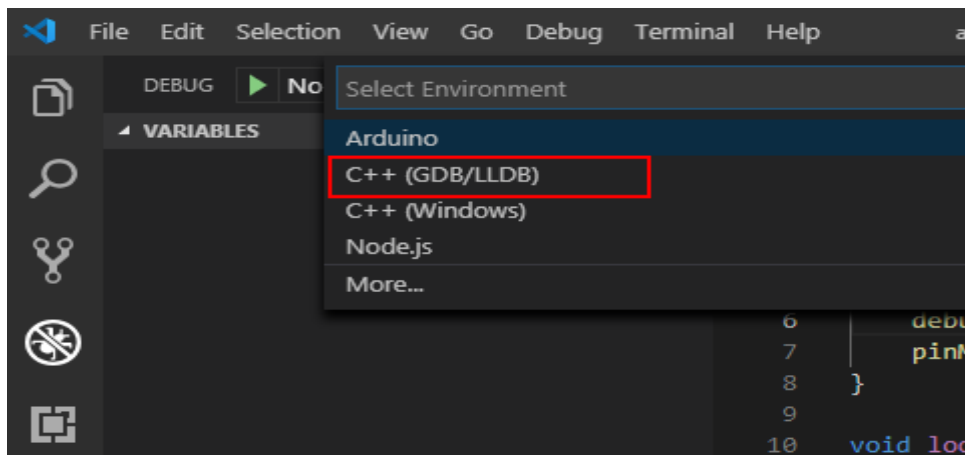
At the top left you will see a green “play” button and a gear wheel next to it with Configure or fix launch.json tooltip shown when you hover your mouse there.

Click this gear wheel button.



A list with options will appear at the top. From this list select the **C++ (GDB/LLDB)**.

Important: don't select the Arduino, this will not work for us. It works only for the boards with integrated debug interface.



A launch.json file will appear in the .vscode subfolder of your folder and it will be also opened in the editor.

Change the following options in the launch.json file (see the final result below):

program - set to "\${workspaceFolder}/build/app.ino.elf"

miDebuggerPath – set to the path to avr-gdb.exe in your Arduino IDE installation, for example:

"c:\\Program Files (x86)\\Arduino\\hardware\\tools\\avr\\bin\\avr-gdb.exe"

Add this line under the miDebuggerPath line – but change the COM port number to your port – see the bottom right of the status bar for the current COM port you are using to communicate with your Arduino:

"miDebuggerServerAddress": "\\\\.\\COM3",

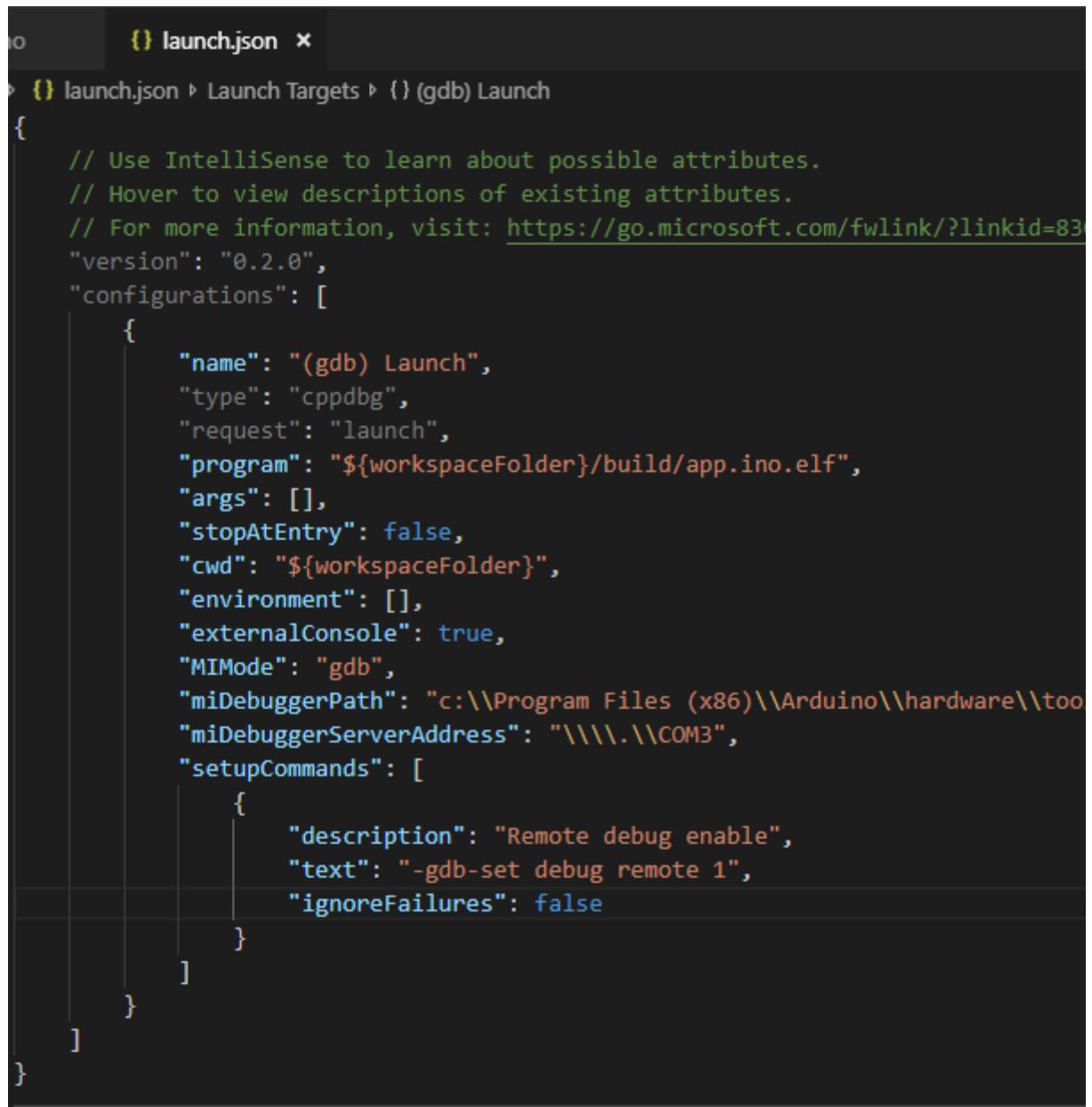
In the setupCommands section replace the default block with “Enable pretty printing” with this block:

```
"description": "Remote debug enable",  
"text": "-gdb-set debug remote 1",  
"ignoreFailures": false
```

The final launch.json file should contain this:

```
{  
  // Use IntelliSense to learn about possible attributes.  
  // Hover to view descriptions of existing attributes.  
  // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387  
  "version": "0.2.0",  
  "configurations": [  
    {  
      "name": "(gdb) Launch",  
      "type": "cppdbg",  
      "request": "launch",  
      "program": "${workspaceFolder}/build/app.ino.elf",  
      "args": [],  
      "stopAtEntry": false,  
      "cwd": "${workspaceFolder}",  
      "environment": [],  
      "externalConsole": true,  
      "MIMode": "gdb",  
      "miDebuggerPath": "c:\\Program Files (x86)\\Arduino\\hardware\\tools\\avr\\bin\\avr-gdb.exe",  
      "miDebuggerServerAddress": "\\\\.\\COM3",  
      "setupCommands": [  
        {  
          "description": "Remote debug enable",  
          "text": "-gdb-set debug remote 1",  
          "ignoreFailures": false  
        }  
      ]  
    }  
  ]  
}
```

Here is a screenshot:



```
{} launch.json ×
> {} launch.json ▸ Launch Targets ▸ {} (gdb) Launch
{
  // Use IntelliSense to learn about possible attributes.
  // Hover to view descriptions of existing attributes.
  // For more information, visit: https://go.microsoft.com/fwlink/?linkid=83
  "version": "0.2.0",
  "configurations": [
    {
      "name": "(gdb) Launch",
      "type": "cppdbg",
      "request": "launch",
      "program": "${workspaceFolder}/build/app.ino.elf",
      "args": [],
      "stopAtEntry": false,
      "cwd": "${workspaceFolder}",
      "environment": [],
      "externalConsole": true,
      "MIMode": "gdb",
      "miDebuggerPath": "c:\\Program Files (x86)\\Arduino\\hardware\\tools\\avr\\bin\\avr-gdb.exe",
      "miDebuggerServerAddress": "localhost:3333",
      "setupCommands": [
        {
          "description": "Remote debug enable",
          "text": "-gdb-set debug remote 1",
          "ignoreFailures": false
        }
      ]
    }
  ]
}
```

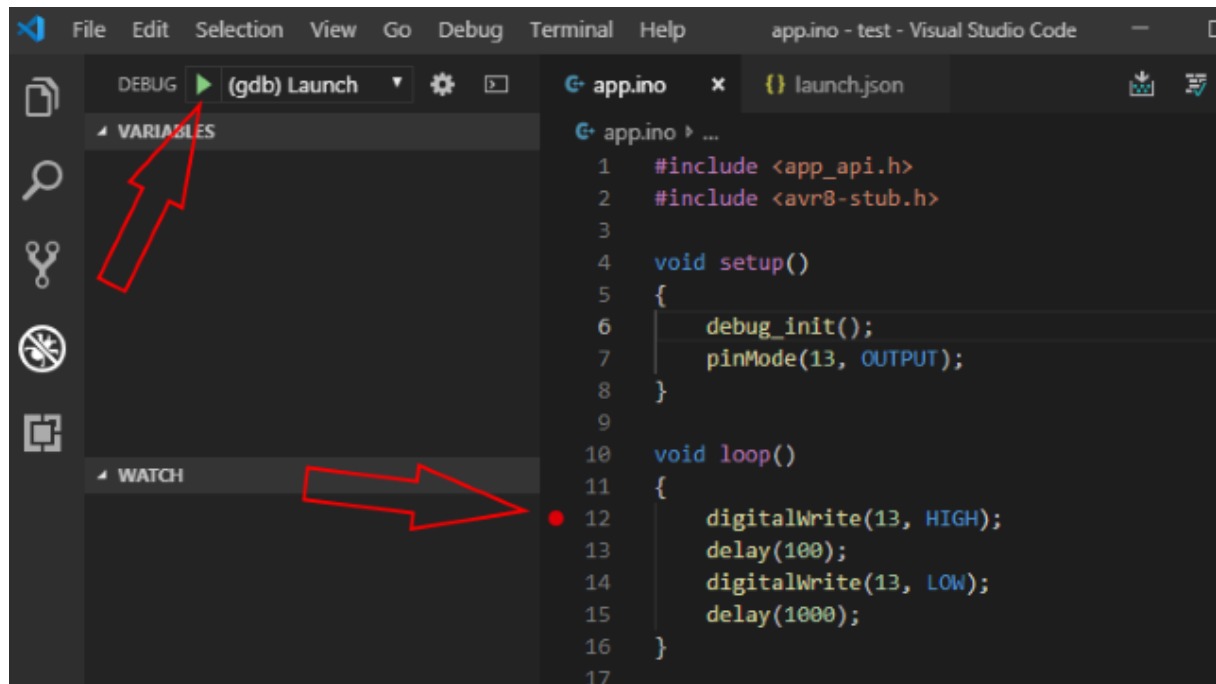
Now we are ready to start debugging.

Upload the program to your Arduino.

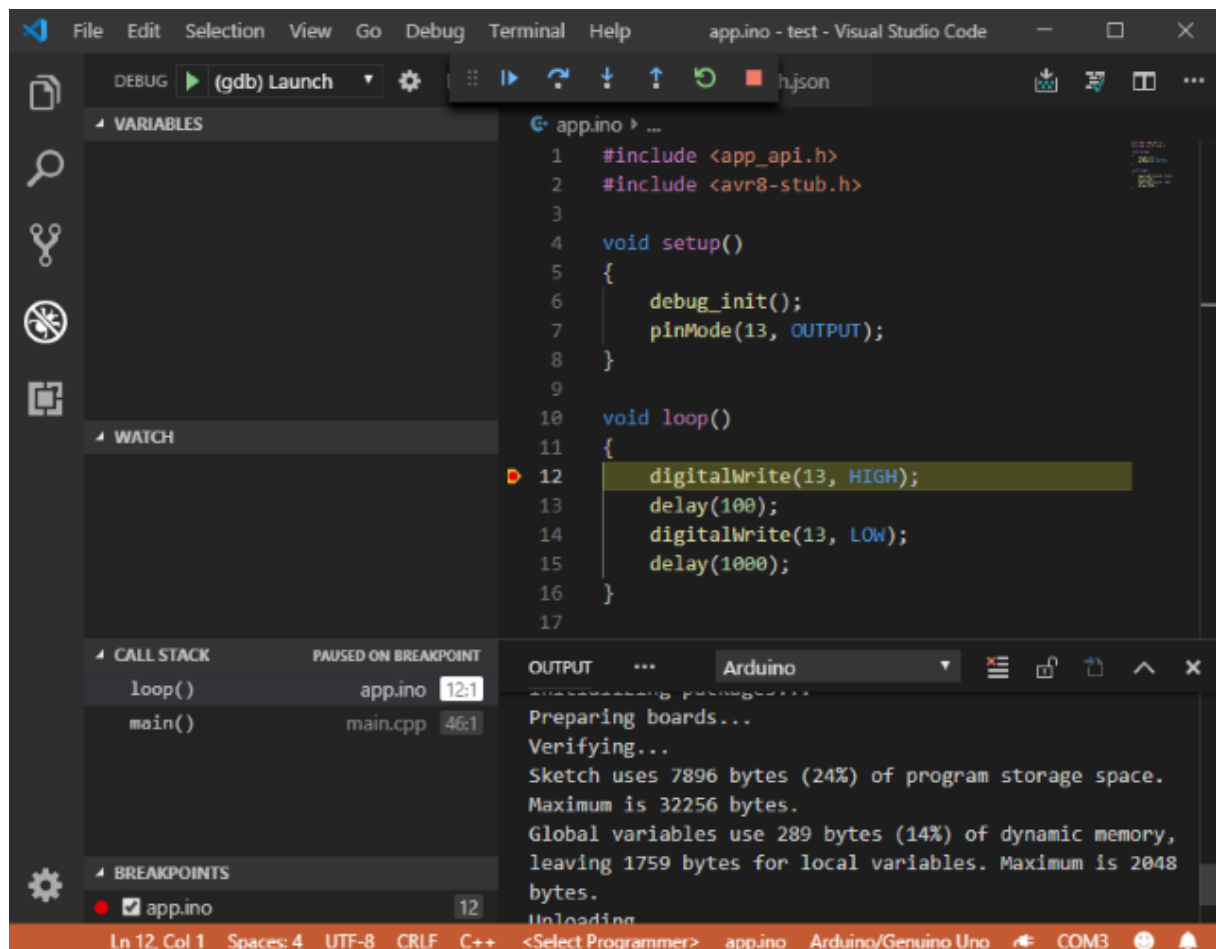
You may want to switch to the app.ino file to be able to use the Upload button in the top-right of the window.

Click to the left margin on the line of `digitalWrite(13, HIGH);` in the `loop()` function. This will insert breakpoint at this line and the program should stop there. You will see a red dot.

Click the Start Debugging (green Play) button in the top left.



If everything goes well, after few seconds you should see buttons to control the program at the top, the status bar should turn orange and the line with the breakpoint should be highlighted. See the picture.



Click the Step Over button in the toolbar above, or press F10 to execute one line of the program. The LED on Arduino board should now be on. Step through the program by repeatedly clicking the Step Over button and watch the LED blink.

After stepping over the last `delay()` you will find yourself in the `main.cpp` file. To get back to your `loop()`, click Step Into (F11) on the `loop()` line.

You can also let the program run by clicking the Continue button or pressing F5. It will stop again on the breakpoint.

Play with the program, remove and insert breakpoints and step through the code.

When you are ready, press the Stop red square button to stop debugging the program.

Viewing variables

Let's add some variables to the program and see what we can do with them in the debugger.

Change the code as follows:

```
int globalVar;
void setup()
{
    debug_init();
    pinMode(13, OUTPUT);
}

void loop()
{
    int localVar = 5;
    globalVar++;
    localVar++;
    digitalWrite(13, HIGH);
    delay(100);
    digitalWrite(13, LOW);
    delay(1000);
    localVar++;
}
```

We've created two variables – one is global, visible in the whole program and one is local, defined inside loop. The local variable is visible only in the loop function and it exists only as long as the loop function is executed. When loop finishes the variables is removed and then re—created when loop is executed again.

Verify and upload the program again.

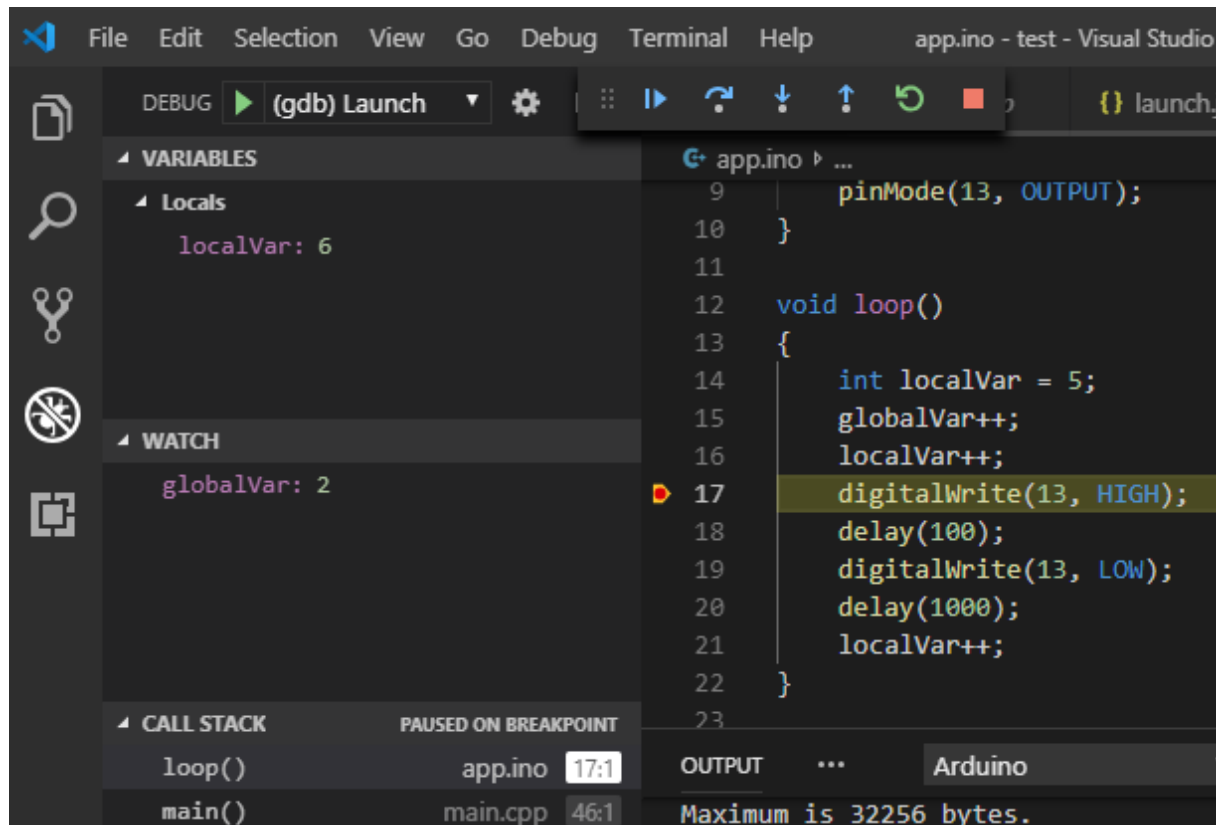
Important note: you need to upload the program every time you make some changes in the code. It is not uploaded automatically when you click the Start debugging button.

Click the Start debugging button.

After the program stops at the breakpoint in loop (I hope you still have it there), look at the Variables view on the left-hand side. There is the local variable with value 6.

The global variable is not automatically shown. You need to add it to the Watch view. Just move your mouse to the Watch window and click the plus (+) button which appears. Then type globalVar into the 'Expression to watch' box which appears.

Here is how it all looks.



Step through the program and see how the values change. The globalVar should grow while the localVar is reset to 5 every time we enter the loop function.

You can also set the value of the localVar – just click the value, enter new number and press Enter key.

Unfortunately, it's not possible to change the value of the globalVar in the Watch view. But you can still change the value by executing gbd command.

To change the value of global variable

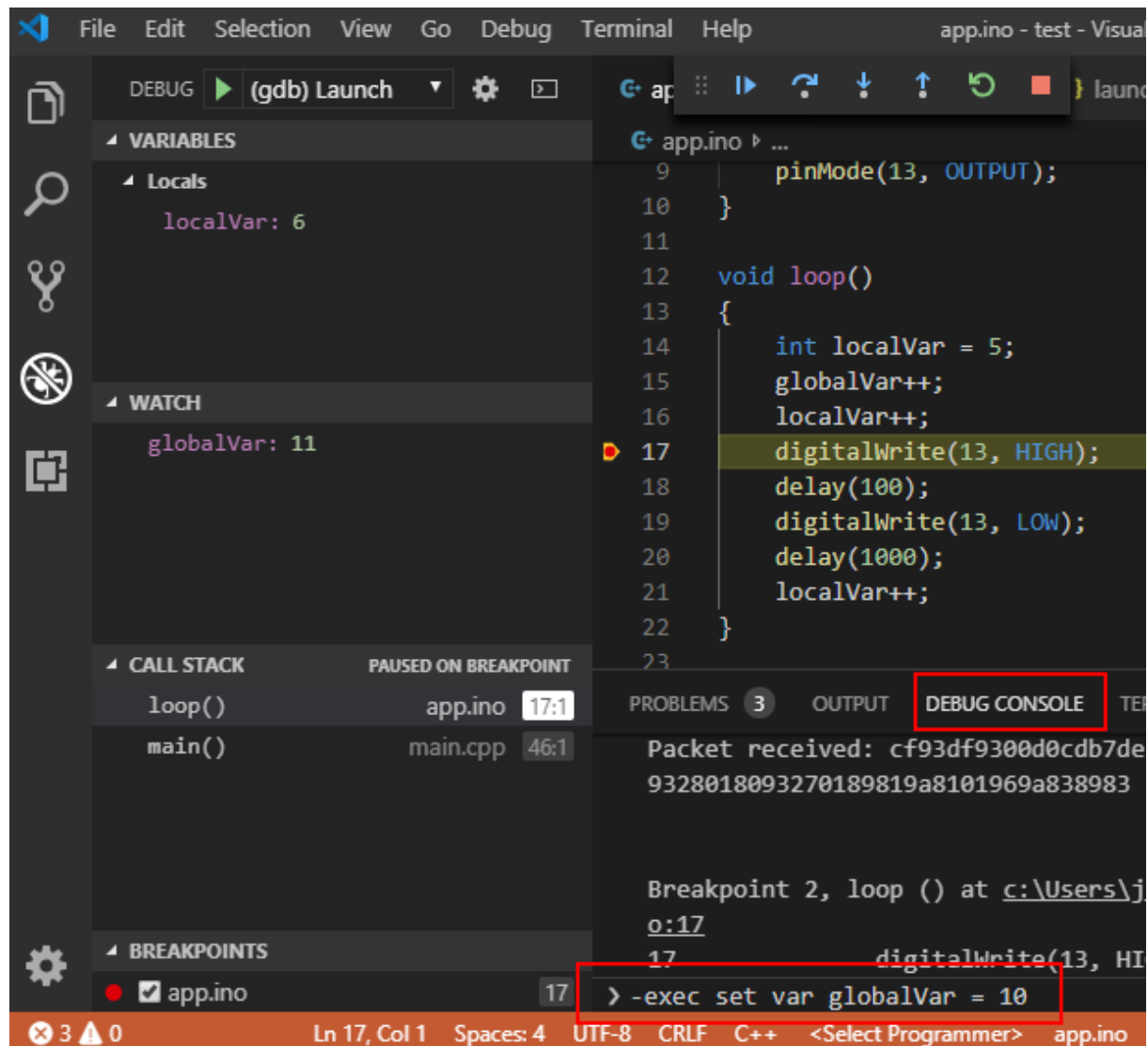
Click Debug console below the code of your program. There should be a line with command prompt at the bottom.

Enter the following command there:

-exec set var globalVar = 10

You will not see the change in the Watch window until you do a step in the program to force refresh of the view, but the variable value is already set to 10.

This is not very comfortable but you can use the up/down arrows on your keyboard to quickly select previous commands - no need to type it again and again.



Breakpoint in main

The VS Code debugger always places breakpoint into main() function and so far I didn't find any way to stop it from doing this. The program never stops at this breakpoint because we actually connect to a running program which is already running in the main function but while this breakpoint is present, the program runs at lower speed than normal - see point 3 above in the Limitations sections. You can remove the breakpoint by executing the following command in the Debug Console after the debugging starts:

-exec clear main

The debugging solution described here was tested on Windows 10 with Arduino Uno and Mega and with Nano on Windows 7. It should also work on Linux and Mac. On Win 7 and older you may need to use a “proxy” program instead of direct serial connection. Please see the steps for eclipse in this document for more info.

Create assembly listing

If there is some problem connecting to the debugger, you can enable logging the communication by adding the following block into the setupCommands section in launch.json.

```
{
  "description": "Log file enable",
  "text": "-gdb-set remotelogfile gdb_logfile.txt",
  "ignoreFailures": false
}
```

You will then find gdb_logfile.txt file in your program folder and you can examine this file to see what went wrong.

If there is some strange behavior when you step through the program, like jumping over several lines, this is most likely because the code generated by the compiler is different from the code you wrote. See the talk about optimizations in the Step 3 section above. This sometimes happens even if the optimizations are off. To examine the problem, it is useful to see the code generated by the compiler. To do this, run the following command from the command line in the build sub-folder of your project:

```
"c:\Program Files (x86)\Arduino\hardware\tools\avr\bin\avr-objdump.exe" -S app.ino.elf > list.txt
```

Substitute your path to Arduino IDE if it is different.

This command will produce list.txt file with your program code together with the resulting assembly code generated by the compiler. You can search this file for the <main> or <loop> symbols to see how your program translates to the actual operations of the processor.

Using this debugger with PlatformIO

This chapter describes how to use this debugger with PlatformIO. It uses PlatformIO IDE in Visual Studio Code (VSCode). For information on PlatformIO please see <https://platformio.org/>.

Step 1 – Install PlatformIO into VSCode

Follow the instructions in PlatformIO documentation here:
<https://platformio.org/install/ide?install=vscode>.

Step 2 – Create PlatformIO project for your Arduino

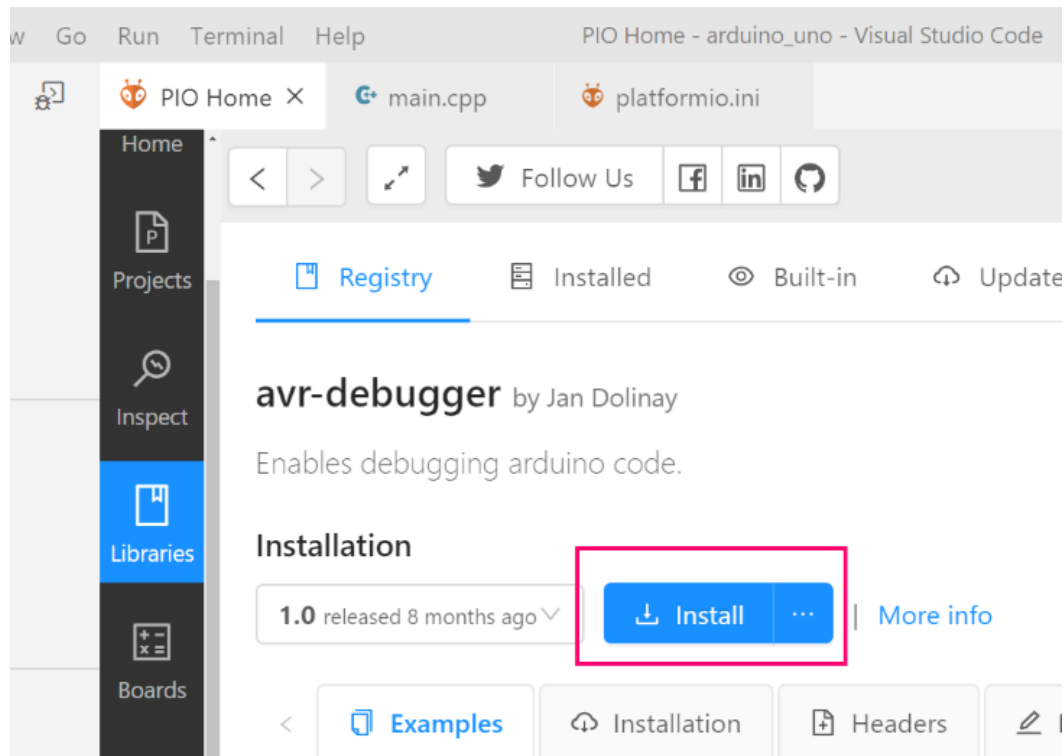
Again please follow the instructions in PlatformIO documentation, the Quick start section here:
<https://docs.platformio.org/en/latest/integration/ide/vscode.html#quick-start>

Step 3 - Install avr-debugger library

In the PlatformIO sidebar on the left click the Libraries button.

In the Libraries window on the Registry tab enter avr-debugger into the search field and find the avr-debugger library.

Click on the Install button to install this library.



Step 4 – Update the project configuration for the debugger

In this step we modify the platformIO project configuration as follows:

- Add the debugger library avr-debugger dependency to your project
- Configure the debug configuration to work with this debugger
- Configure the build options – turn off compiler optimizations.
- Set your configuration as debug so the PIO does not build other version of the program when you use the Debug button than when you build the program with the build button.

Open the **platformio.ini** file (you can find it in VS Code explorer view under your project) and copy-paste the text below the already existing text.

```
; added for avr-debugger
build_type = debug
debug_tool = avr-stub
debug_port = \\.\\COM19
```

```

; GDB stub implementation
lib_deps =
  jdolinay/avr-debugger @ ~1.1

debug_build_flags =
  -Og
  -g2
  -DDEBUG
  -DAVR8_BREAKPOINT_MODE=1

```

Note that for ATmega1284(P), platformIO might not yet know that `avr-stub` is a valid `debug_tool` for these MCUs. In this case, you have to change the board definition and include `avr-stub` as a possible debug tool.

Change the **debug_port** option to **your serial port** – depending on where is your Arduino connected.

Note that we added dependency to the `avr-debugger` library with the `lib_deps` option.

Here is how it looks when done:

```

; Please visit documentation for the oth
; https://docs.platformio.org/page/proje

[env:uno]
platform = atmelavr
board = uno
framework = arduino

; added for avr-debugger
build_type = debug
debug_tool = avr-stub
debug_port = \\.COM19

; GDB stub implementation
✓ lib_deps =
  |   jdolinay/avr-debugger @ ~1.1
✓ debug_build_flags =
  |   -Og
  |   -g2
  |   -DDEBUG
  |   -DAVR8_BREAKPOINT_MODE=1

```

Step 5 – Modify the code to work with the debugger

Add these two includes at the beginning of your code, below the `#include "Arduino.h"`:

```

#include "avr8-stub.h"
#include "app_api.h"

```


Add the call to `debug_init()` to the setup functions. Here is the complete code:

```
/**
 * Blink with debugger
 */
#include "Arduino.h"
#include "avr8-stub.h"
#include "app_api.h" // only needed with flash breakpoints

void setup()
{
    pinMode(LED_BUILTIN, OUTPUT);

    // initialize the avr-debugger
    debug_init();
}

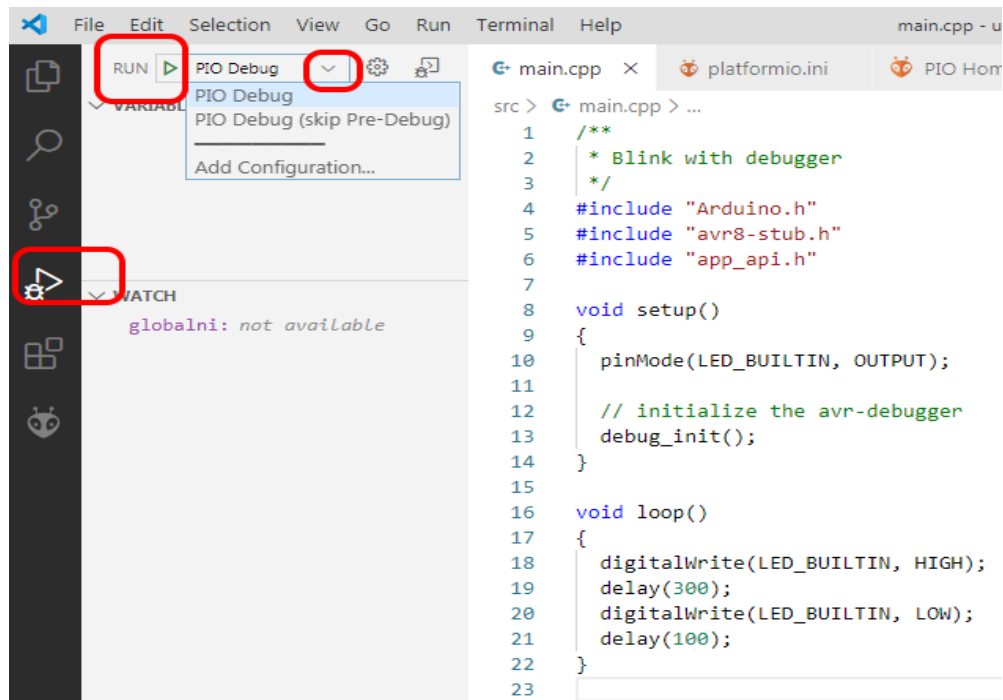
void loop()
{
    digitalWrite(LED_BUILTIN, HIGH);
    delay(300);
    digitalWrite(LED_BUILTIN, LOW);
    delay(100);
}
```

Step 6 – Debug your program

In the VS Code left side bar click the Debug button.

At the top expand the combo box with debug configurations and select the **PIO Debug**.

Now click the Run (Start debugging) button (with Play icon). See the picture below.

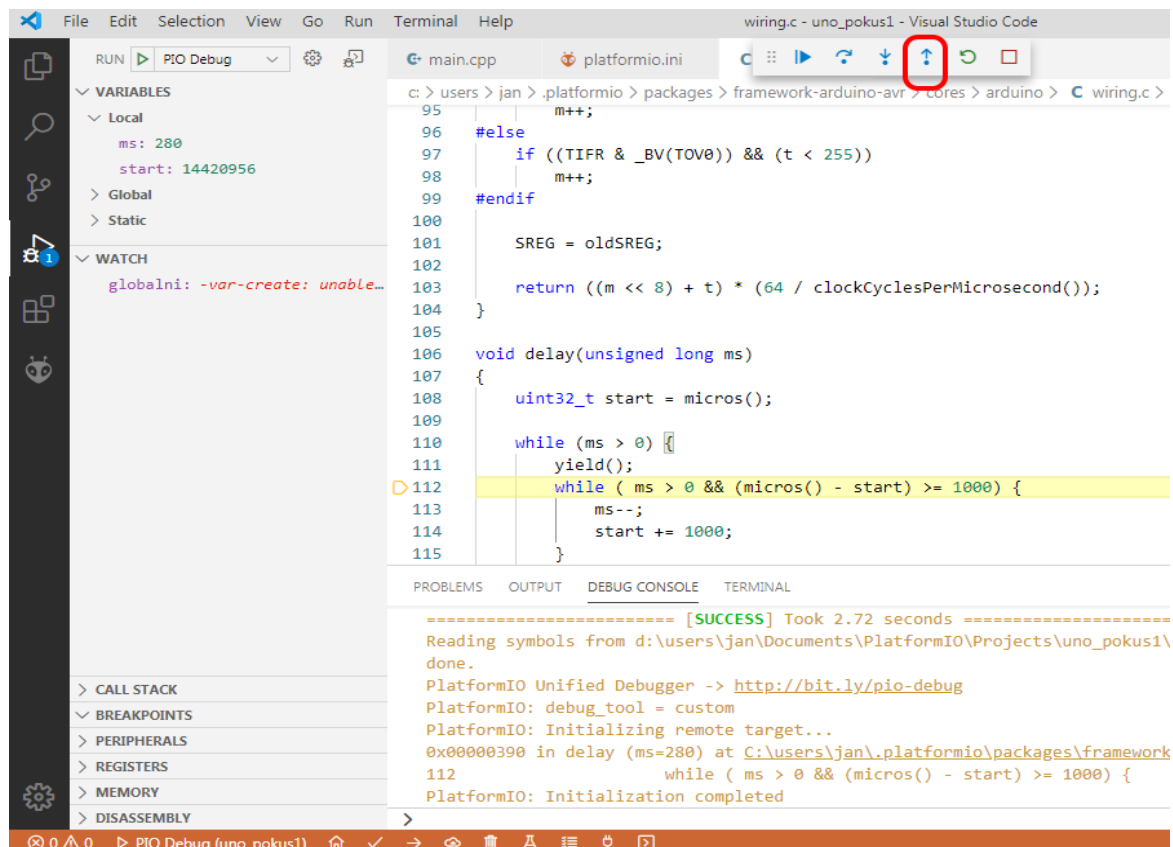


Your program will now be built, uploaded to Arduino and debugging will start.

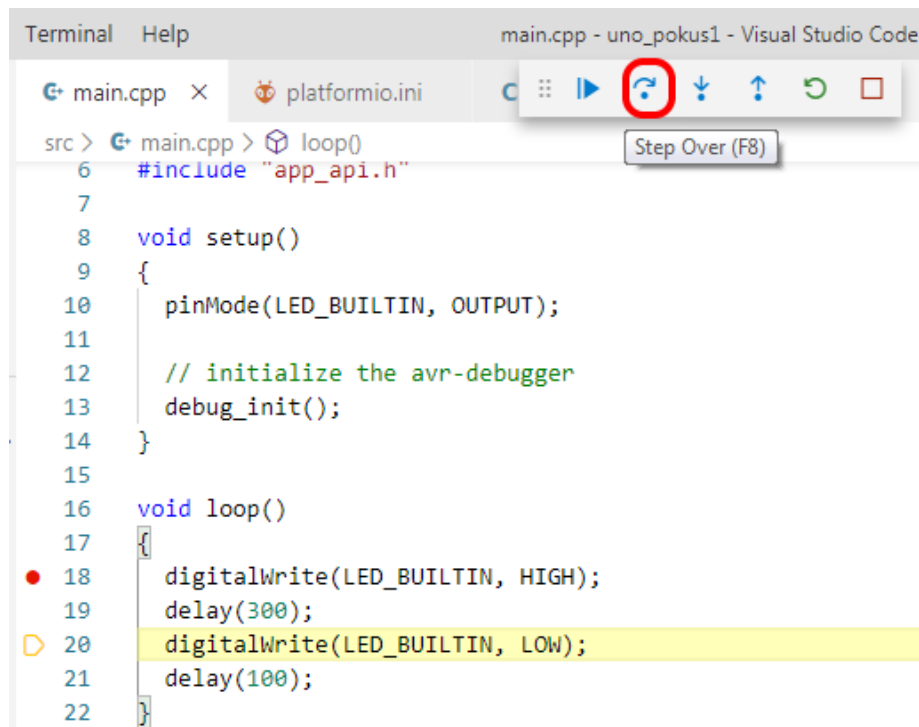
You will probably find yourself inside the delay() function, see the picture.

If the source code does not open automatically, you can Ctrl + click the link to file in the Debug Console window below (the line starting with 0x000... in delay (ms=200) at C:\users\...). Or expand the **Call Stack** area on the bottom left and click the loop line which should be there.

Use the **Step Out** button in the debug toolbar to return to the calling function. Now you should be in your loop() function. See picture below.



Now we are in the loop function:



Now you can step through the program line by line using the **Step Over** button, step into functions using the **Step Into** button, set breakpoints by clicking to the left margin of the editor and so on.

You can also work with variables if you have some in your program. Please see the previous chapter on [Using this debugger in Visual Studio Code](#) for more instructions and tips.

If the debugger disconnects/stops after short time or sometimes doesn't connect at all and if you are on Windows computer, it is possible that the direct serial connection with debugger does not work. Please see the section below on connecting through TCP-to-serial proxy.

If there is some other problem try enabling verbose output for the GDB by adding the following to the platformio.ini:

```
debug_extra_cmds = set debug remote 1
```

You should then see detailed information about communication between GDB and the AVR stub in the Debug console view which may help you identify the problem.

For other troubleshooting tips please see the chapter [Troubleshooting problems with debugging](#) earlier in this document.

Using TCP-to-Serial proxy (Windows only)

As mentioned earlier in this document, if the direct connection to serial port does not work you can use a "tcp to serial proxy" program. This seems to be needed for some Arduino boards on Windows 7 only; on Windows 10 the direct serial connection seems to work fine.

If you have to use the TCP to Serial connection do this:

You will need the complete package for the debugger, not just the avr-debugger library. Download it from here: https://github.com/jdolinay/avr_debug - use the Download ZIP option in the Clone or Download button.

Extract the folder somewhere on your computer, for example to c:\avr_debug.

Use your file manager to open the folder where the avr_debug package is located. You should see a **start_proxy.bat** file in this folder.

Open the start_proxy.bat file in Notepad or other text editor. Change the number of the COM port in this file. There is this line:

```
hub4com-2.1.0.0-386\com2tcp --baud 115200 \\.\COM15 11000
```

Just change the number after COM from 15 to the number of your COM port to which the Arduino board is connected.

Save and close the start_proxy.bat file.

Run the bat file. This will start convertor between TCP/IP port used by the GDB (as configured above) and the serial port to which your Arduino is connected. You should see a console window with some information. This window will be opened all the time during the debugging.

```

C:\Windows\system32\cmd.exe
d:\Git\avr_debug>hub4com-2.1.0.0-386\com2tcp --baud 115200 \\.\COM15 11000
d:\Git\avr_debug>"hub4com" --baud=115200 --create-filter=pin2con --add-filter=rs=0:pin2con "\\.\COM15" --use-driver=tcp --reconnect=1000 "11000"
COM15 Open("\\.\COM15", baud=115200, data=8, parity=no, stop=1, octs=on, odsr=off, ox=off, ix=off, idsr=off, ito=0) - OK
Route data COM15(0) --> TCP(1)
Route data TCP(1) --> COM15(0)
Route flow control COM15(0) --> TCP(1)
Route flow control TCP(1) --> COM15(0)
Filters:
COM15(0) | \->{pin2con.IN}->
          | /
          /<-----
COM15(0) Event(s) [DSR] will be monitored
Started COM15(0)
Socket(0.0.0.0:11000) = 120
Listen(120) - OK
Started TCP(1)

```

Note: You may want to configure your firewall to block access to the port 11000 from other computers. You can also change the port number both in the .bat file and in the PIO debug_portoption.

In the platformio.ini file add the following line (it will prevent PIO from loading the program before debugging):

```
debug_load_mode = manual
```

And change the **debug_port** option to this:

```
debug_port = localhost:11000
```

See below for the complete platformio.ini file:

```

[env:uno]
platform = atmelavr
board = uno
framework = arduino

; added for avr-debugger
build_type = debug
debug_tool = avr-stub
debug_port = localhost:11000
debug_load_mode = manual

; GDB stub implementation
lib_deps =
    jdolinay/avr-debugger @ ~1.1

debug_build_flags =
    -Og
    -g2
    -DDEBUG

```

`-DAVR8_BREAKPOINT_MODE=1`

To debug the program with the TCP-serial proxy

First, build and upload the program as usual using the PlatformIO buttons in the toolbar.

Then start the start_proxy.bat file

Now start debugging in VSCode.

Then you are done debugging, close the proxy window. If you leave it open, you will not be able to upload your program to Arduino because the proxy blocks the serial port.

So, you need to use this sequence to update the program and debug it:

- change and upload code
- start TCP proxy with the .bat file.
- start debugging
- stop debugging
- close the proxy
- change and upload the code
- etc.

It is not very comfortable but it works. You could probably find a way to start the proxy automatically after upload if you play with the PIO configuration.

Debugging code with millis and micros

This section provides some info about using millis and micros Arduino functions with the debugger.

With Flash breakpoints there should be no problems with these functions other than inaccurate values returned (because the timer which provides the timing is stopped while the program is stopped in the debugger).

With RAM breakpoint millis and micros will not work as expected. To be more exact with RAM breakpoints:

- **millis** will return the same value (possibly 0) all the time.
- **micros** will return values which can differ from each other by about 1000, but the values are more or less random, they will not increase as you would expect.

This is because for these functions to work, timer interrupt must be executed to increment a counter. But in the RAM breakpoints mode no interrupts are executed except the one used by the debugger to step the program and check if a breakpoint was reached. The only exception is if there are no breakpoints set and you let the program run (with the Continue button), then the timer interrupt is executed and the functions will work somehow.

What can I do if I need to debug my program which uses millis or micros?

If possible, switch to the flash breakpoints.

If you cannot use the flash breakpoints you can still debug your program somehow but take into account the behavior of the functions described above. For example, you won't be able to step into or place a breakpoint into a condition based on polling the time with millis or micros. Here is example program which blinks a LED:

```
void loop() {

static bool on = false;
static unsigned long lastChange = millis();

  unsigned long now = millis();
  unsigned long diff = now - lastChange;
  if ( diff > 200 ) {
    if ( on ) {
      digitalWrite(13, LOW);
      on = false;
      debug_message("LED off");
    }
    else {
      digitalWrite(13, HIGH);
      debug_message("LED on");
      on = true;
    }

    lastChange = now;
  }
}
```

In this program if you set breakpoint inside the if or else block, the program will never stop there. If you remove all breakpoints and let the program run, it should blink the LED as expected. You can always break the program using the “pause” button.

You could also place hardcoded breakpoint into the condition using the breakpoint() function.

How come the delay() function works? (for those interested)

Yes, the delay() function works, the delay is just stretched by a factor of about 4. This is lucky because we can debug the simple blinky program with digitalWrite() and delay().

The code of the delay function looks like this (as of 3/2021):

```
void delay(unsigned long ms)
{
    uint32_t start = micros();

    while (ms > 0) {
        yield();
        while ( ms > 0 && (micros() - start) >= 1000) {
            ms--;
            start += 1000;
        }
    }
}
```

```

    }
}

```

It uses the `micros()` function and **if the micros function does not work, how come the delay works?**

It took me some time to figure it out, but the explanation is quite simple. In short, it is because of the assumption in the code that `micros` will always return higher value than in previous call. This makes sense (except for the situation when the value overflows), but it is not true when debugging. The `micros` will return “random” values which can change by about 1000.

In the inner while loop there is the condition `(micros() - start) >= 1000`

If the `micros()` returns value lower than the “start”, the result is a negative number. But since both the `start` variable and the return value of `micros()` are unsigned types, the result is interpreted as an unsigned integer- and a negative number turns into very big unsigned number. And the condition is true and the `ms` is decremented...

Basically, the delay works as follows under active debug:

- get some random number from `micros` and store it to “start” variable.
- try to wait for `micros` to return value either higher by 1000 than `start` or lower than `start`. The latter is much more likely, and I suppose it doesn’t take many tries for this to happen.
- Once the inner while condition is met, the `ms` variable is decremented and `start` is increased by 1000. Now the `micros` will always return lower value than “start” and the inner loop just decrements the `ms` in every run until it is 0.

So in fact the original `delay()` based on timer value from `micros` turns into a simple loop, decrementing the milliseconds count to 0 at full speed. Thanks to the degraded speed of the program under debugger this takes more than enough time to simulate some delay behavior.