Arduino Debugger

Author: Jan Dolinay; dolinay [at] fai.utb.cz   **Last revised: November 16, 2016**

Table of contents

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 can be used as a graphical frontend for debugging.

Release notes

June 2015

First release. Arduino Uno supported.

Abbreviations and terms used in this document

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

Arduino software library – see Arduino MCU Framework.

Debug driver – 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.

MCU – microcontroller

Introduction: How to use this document

This document should help you use the Arduino debugger. It assumes use of Eclipse IDE. Setting up Eclipse and other tools is quite complicated, but currently I am not aware of any easier way to achieve this.

The chapters in this document 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.

Basically there are these main tasks 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 section 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.

Unfortunately, there is no direct support for Arduino in Eclipse. Eclipse is not a "single purpose" IDE; it can be used for all kind 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 (1.6.5-r2)	No	Yes	No	Yes	Yes	Needed for uploading the programs.
Atmel AVR toolchain	No	Yes	No	No	Yes	In 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 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 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.

Important limitations of the debugger

One pin must be reserved for the debugger

When debugger is used, one pin must be reserved for its use. This can be either digital pin 2 or 3 (PD2 or PD3 pin of the MCU). By default, pin 2 is used. To change this, change the value of `AVR8_SWINT_SOURCE` define in avr8-stub.h file.

Arduino software library is in conflict with this use of INT0 or INT1. There is small modification of the Arduino code required. Please see Step 5 section for details.

Reason: The debugger needs to generate software interrupts to work. Unfortunately, on ATmega328 there is no dedicated instruction for this and the interrupt must be generated using an I/O pin.

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.

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 (June 2015) is 1.6.5-r2. 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-1.6.5-r2.

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 (May 2015) is Luna.

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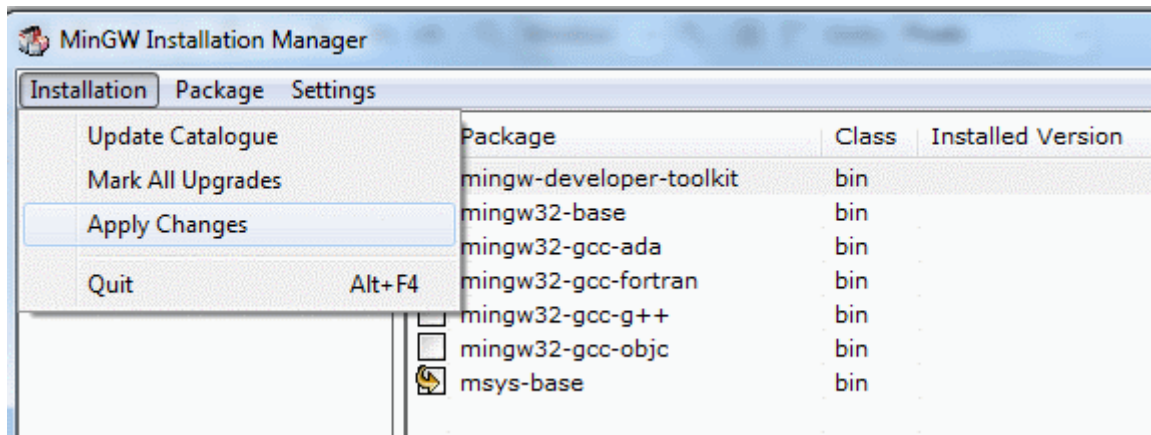
In the next step wait for the download of some files and click Continue.

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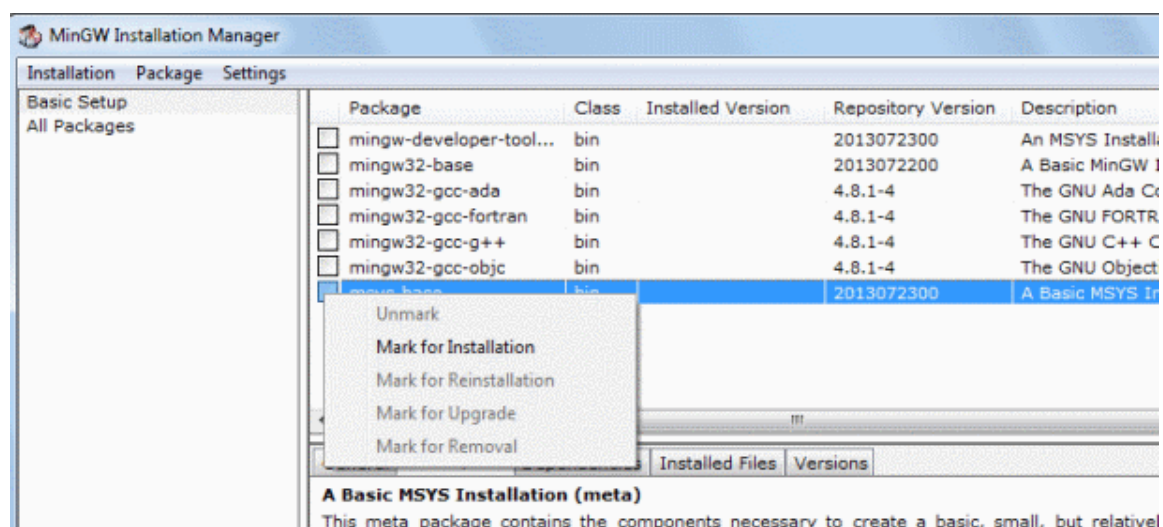
From the menu at the top of the window select Installation > Apply Changes.

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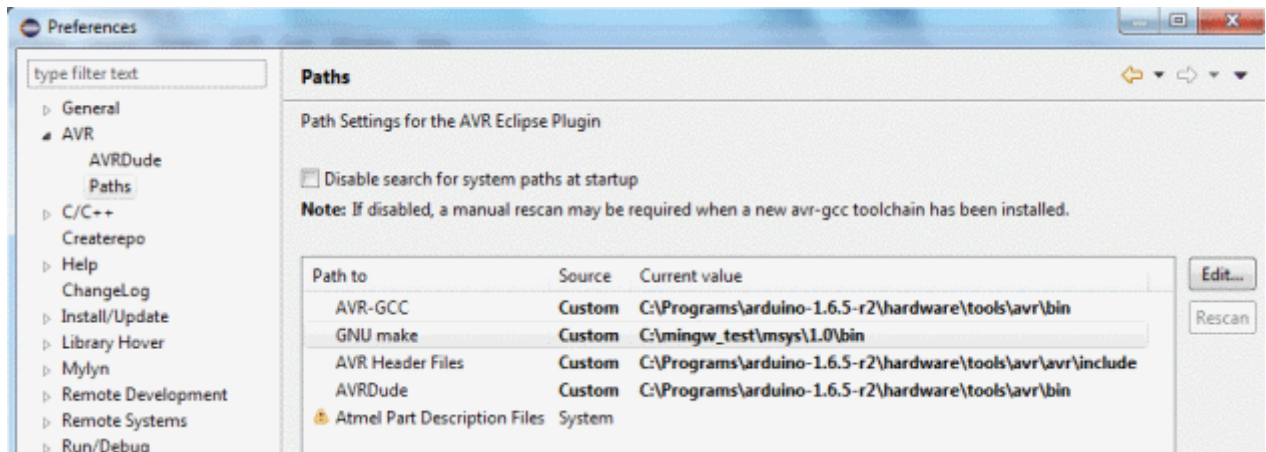
The paths may be different on your system, depending on where you installed the tools. Use the Browse button in the window for path selection and "Custom" Path source; see the picture below.
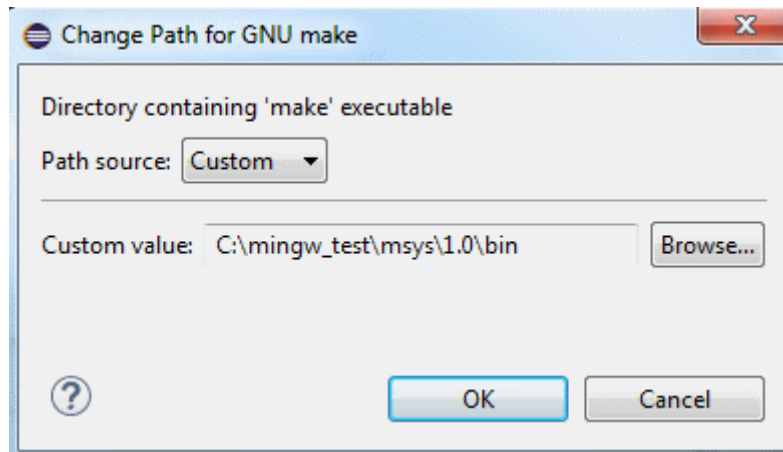
Here are the paths on my computer:

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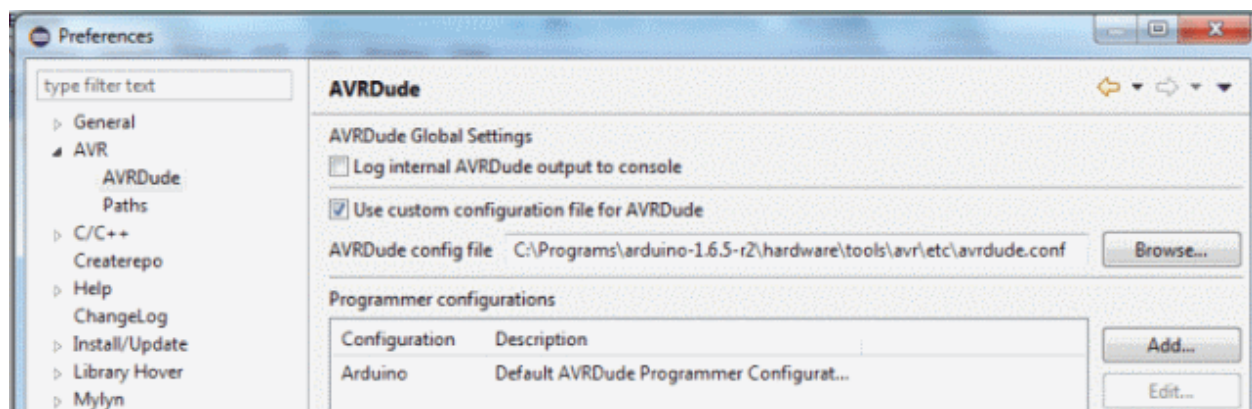
The path is, for example, C:\Programs\arduino-1.6.5-r2\hardware\tools\avr\etc\avrdude.conf.

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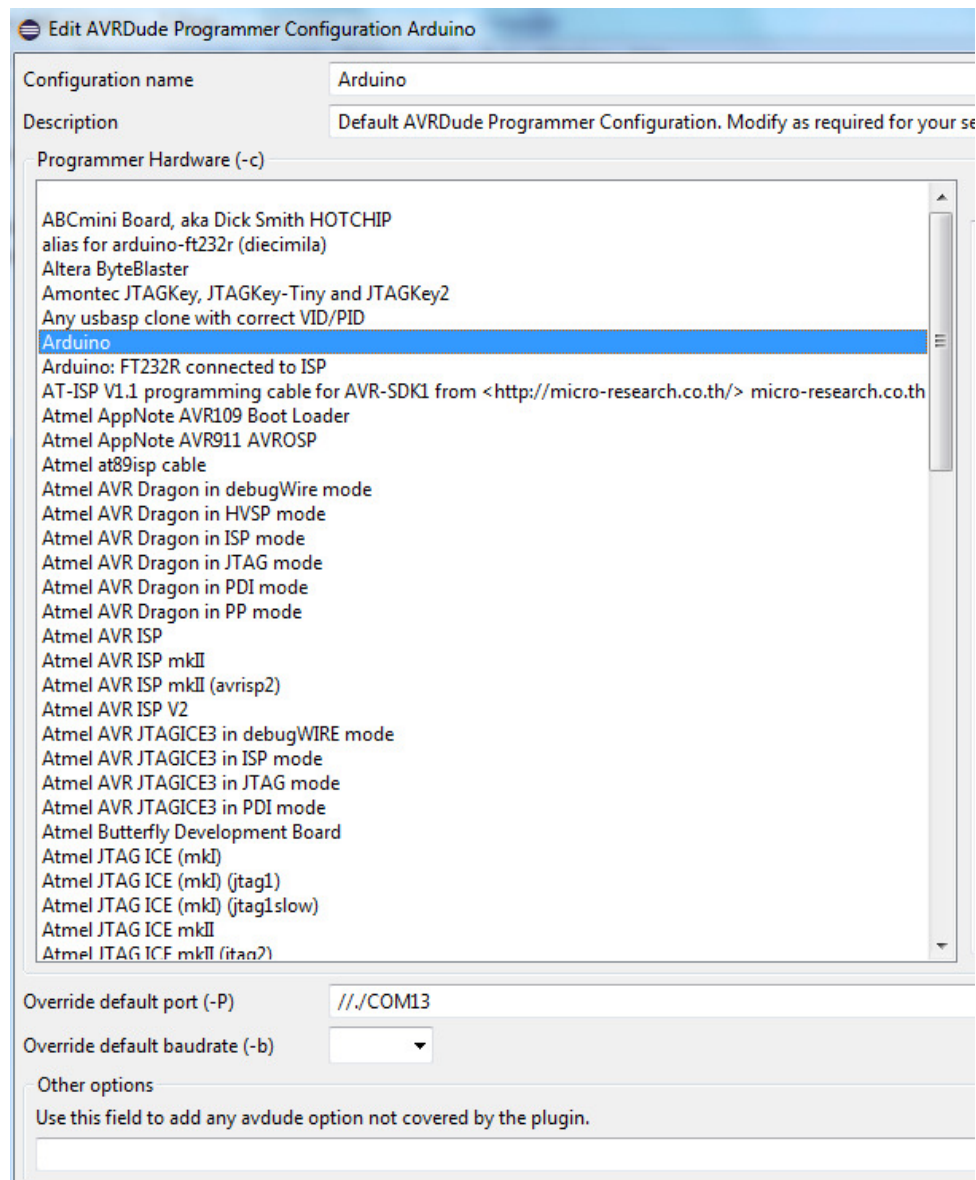
Close the window with OK and then close the Preferences window also.

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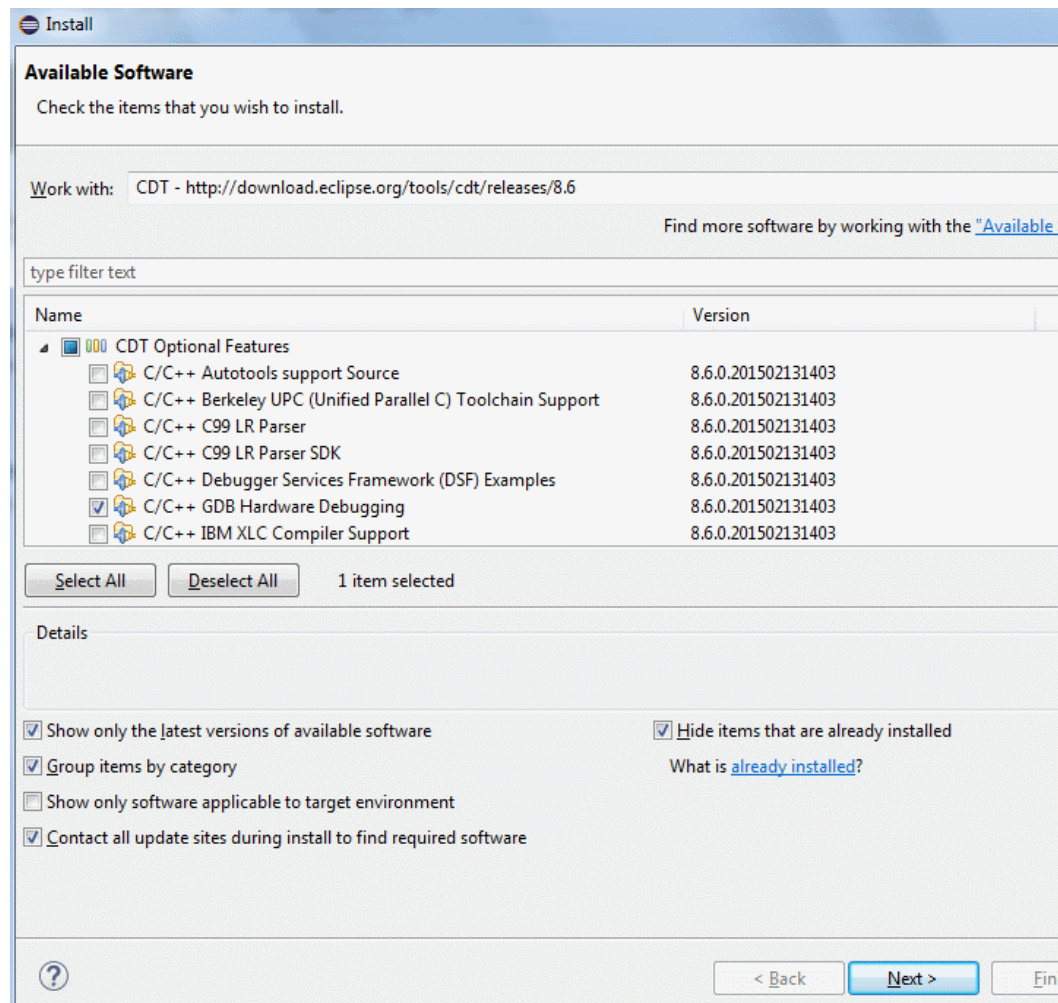
Follow the wizard to install this feature and restart Eclipse when prompted to finish the installation.

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.atmel.com/tools/atmelavrtoolchainforwindows.aspx.

Download the package Atmel AVR 8-bit Toolchain 3.4.5 – Windows.

Run the installer and install the toolchain. The following text assumes you install it into the default folder: c:\Program Files\Atmel\Atmel Toolchain\AVR8 GCC.

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.
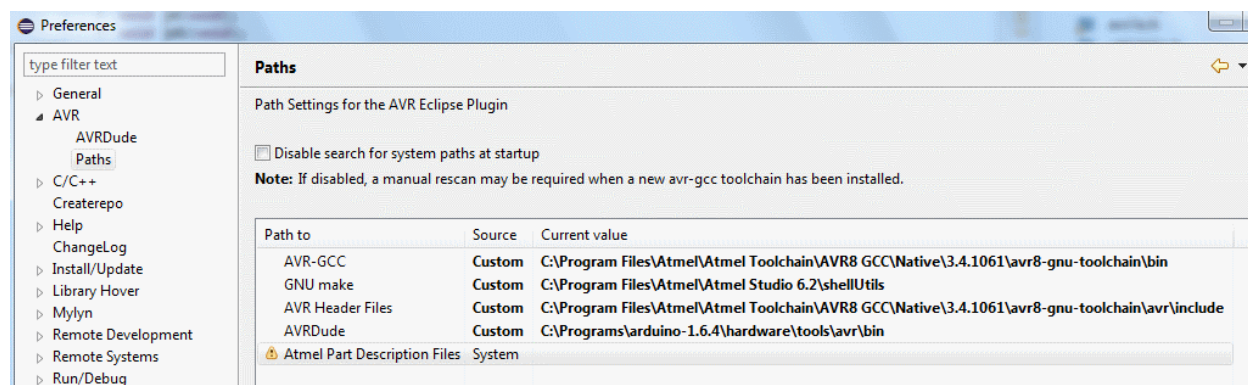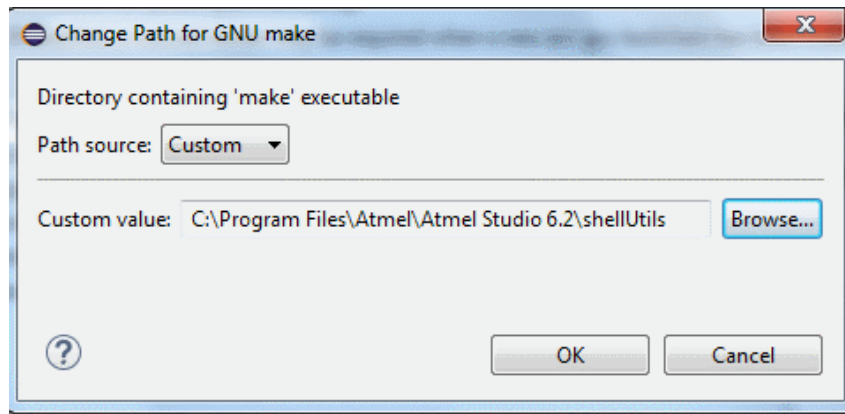
Here are examples of the paths:

AVR-GCC: C:\Program Files\Atmel\Atmel Toolchain\AVR8 GCC\Native\3.4.1061\avr8-gnu-toolchain\bin
GNU make: C:\Program Files\Atmel\Atmel Studio 6.2\shellUtils
AVR Header Files: C:\Program Files\Atmel\Atmel Toolchain\AVR8 GCC\Native\3.4.1061\avr8-gnu-toolchain\avr\include
AVRDude: C:\Programs\arduino-1.6.5-r2\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 MCU framework.
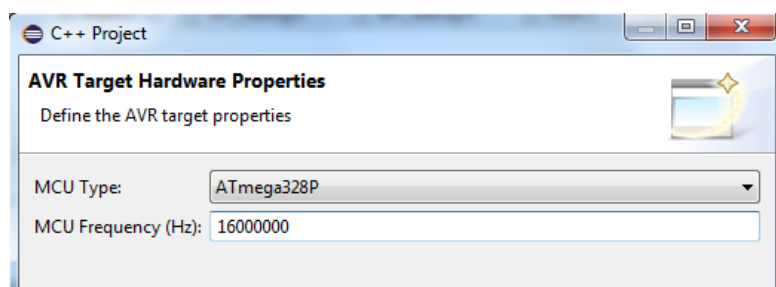
In Eclipse select New > C++ Project.

Enter a name for the project, for example, "test1".

Select AVR Cross Target application > Empty project.
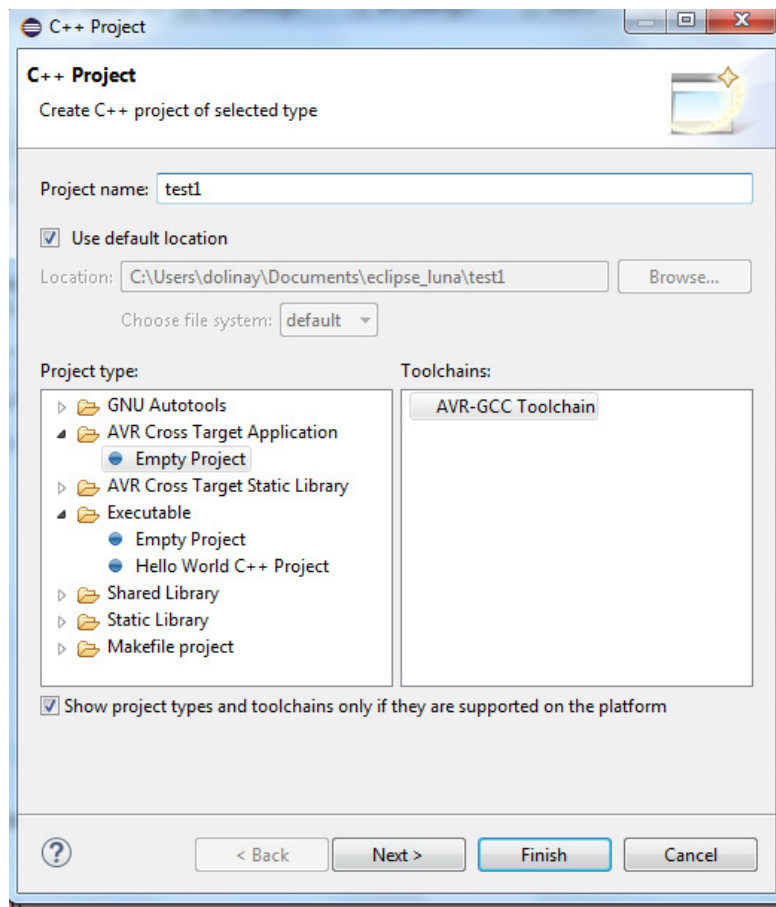
In the next step leave everything as offered (this will create Debug and Release configuration) for the project.

In the next step select the MCU Type: ATmega328P and MCU Frequency (Hz): 16000000.
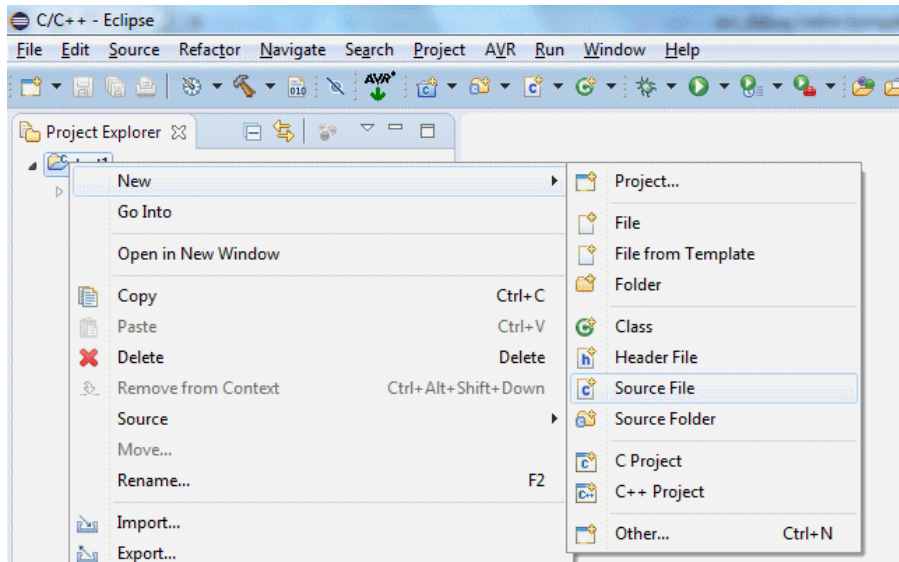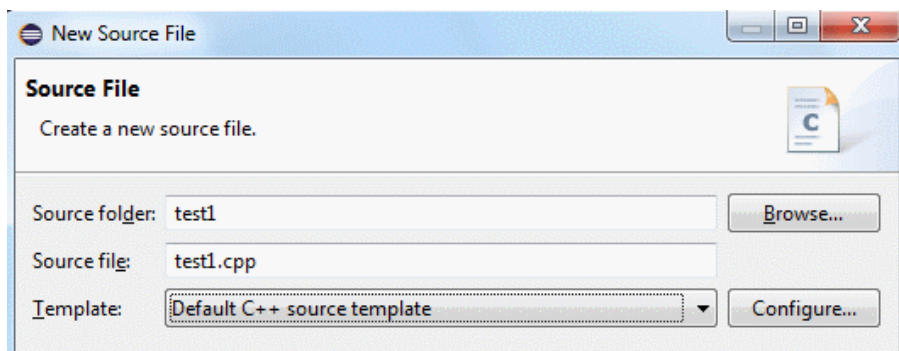
Click Finish to create the project.

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 in the right window.

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

```c
#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: I recommend enabling 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, without these 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.
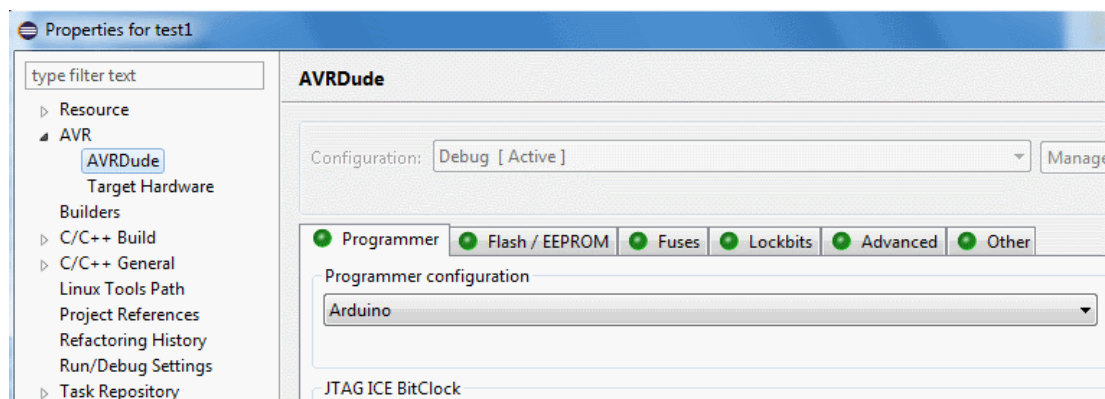
Still in the Properties window expand the C/C++ Build > Settings.

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

Check the "Generate HEX file for Flash memory" box.
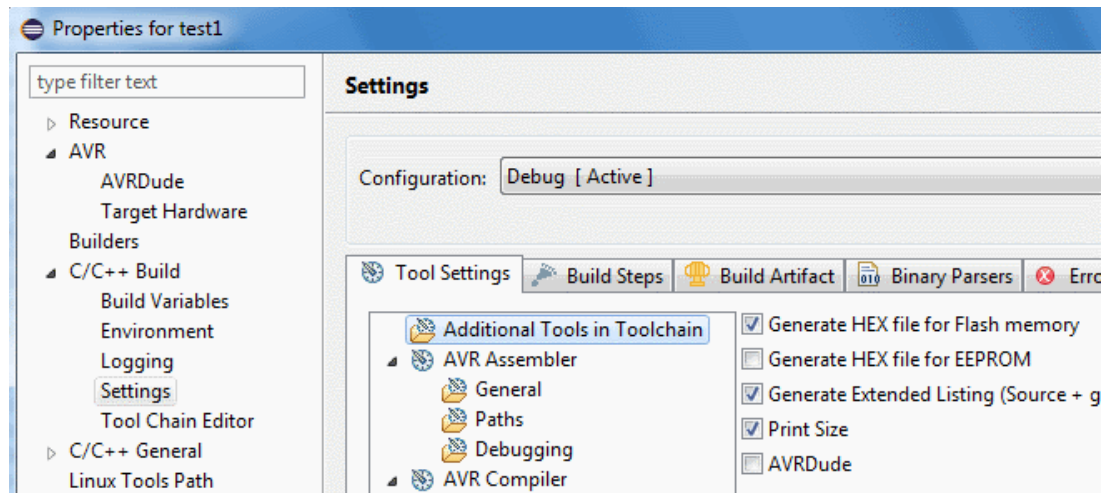
Close the Preferences window with OK button.

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."
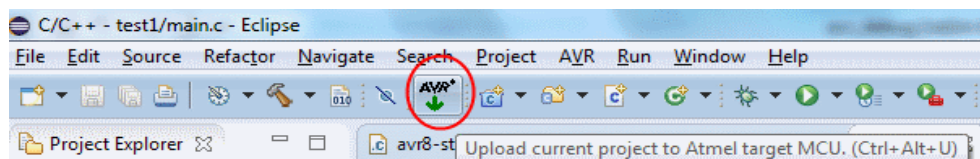

You should now see that the LED on your Arduino is blinking fast (100 ms on, 100 ms off). This means our program is running.

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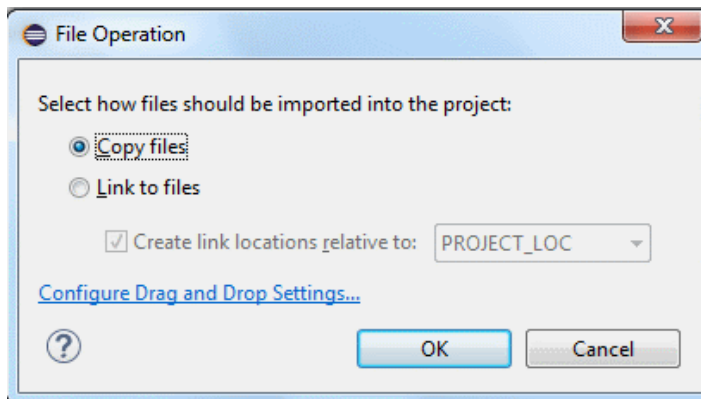
Open Properties of your project (Alt + Enter) and go to C/C++ Build > Settings.

Select AVR C++ Compiler > Debugging. In the "Debug Info format" select dwarf-2.
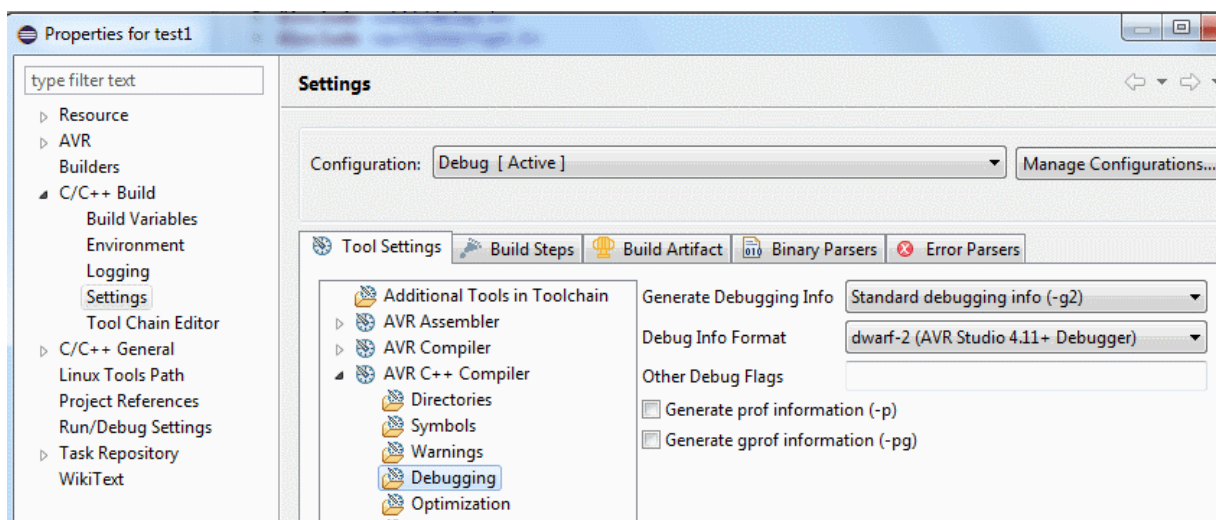
Do the same for the AVR C Compiler. It is not necessary now but can come in handy later when we also have some .c files in the project.

Close the Preferences window with OK.

Replace the program with the following one:

```c
#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 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 cannot be seen blinking because the speed is too fast. We will 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.

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.

22

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.

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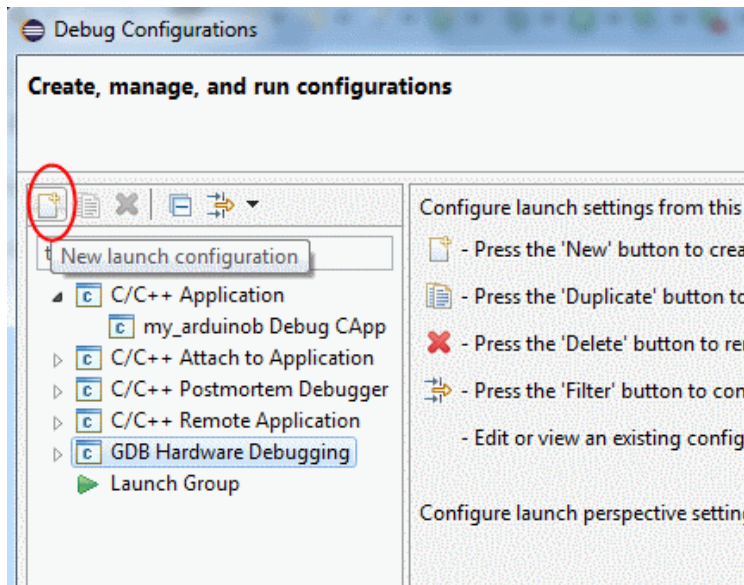
This will create new launch configuration under the GDB Hardware debugging item.

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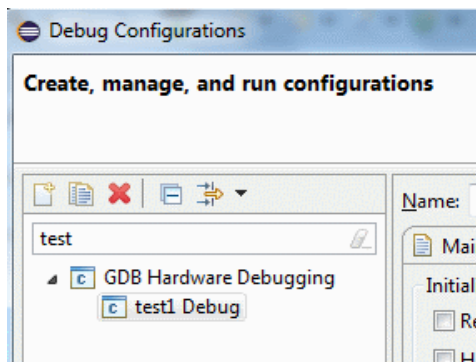
In the Debugger tab in the "GDB command" field enter (or browse to) the path to the GDB executable avr-gdb.exe, followed by the path to your "executable" file (.elf).

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:\Program Files\Atmel\Atmel Toolchain\AVR8 GCC\Native\3.4.1061\avr8-gnu-toolchain\bin\avr-gdb.exe "${project_loc}/Debug/${project_name}.elf".
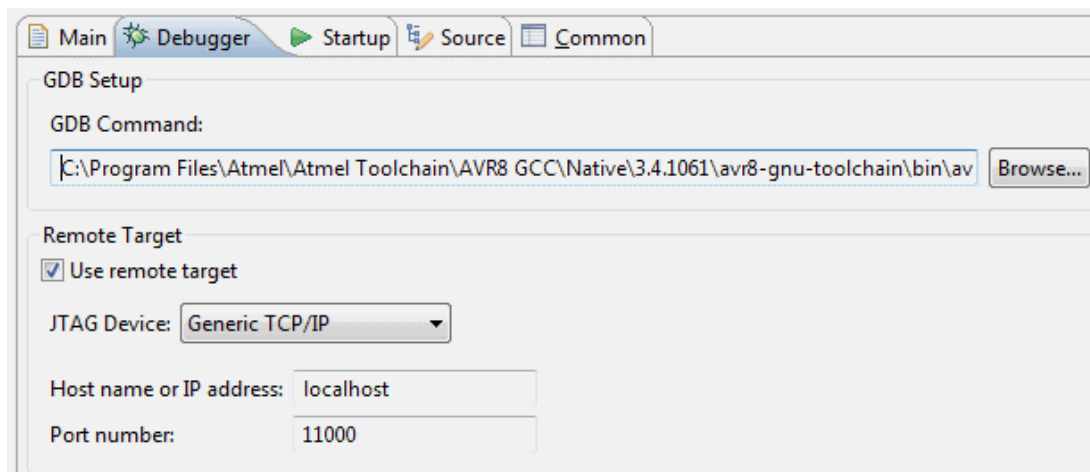
TIP: Use the Browse button to select the avr-gdb.exe. Then enter space after the path into the edit box and then paste the following line: "${project_loc}/Debug/${project_name}.elf".

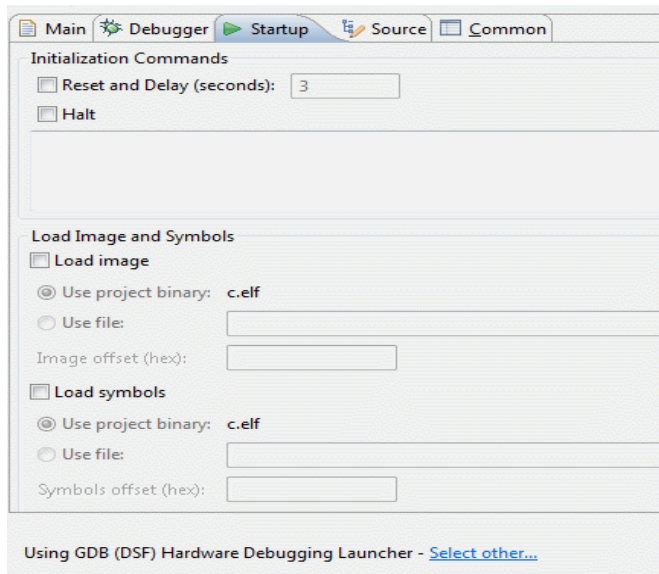Check the" Use remote target" box.

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

Host name or IP address: localhost

Port number: 11000.

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.

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.

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.

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 was just stopped at any point in its run when the debugger connected.
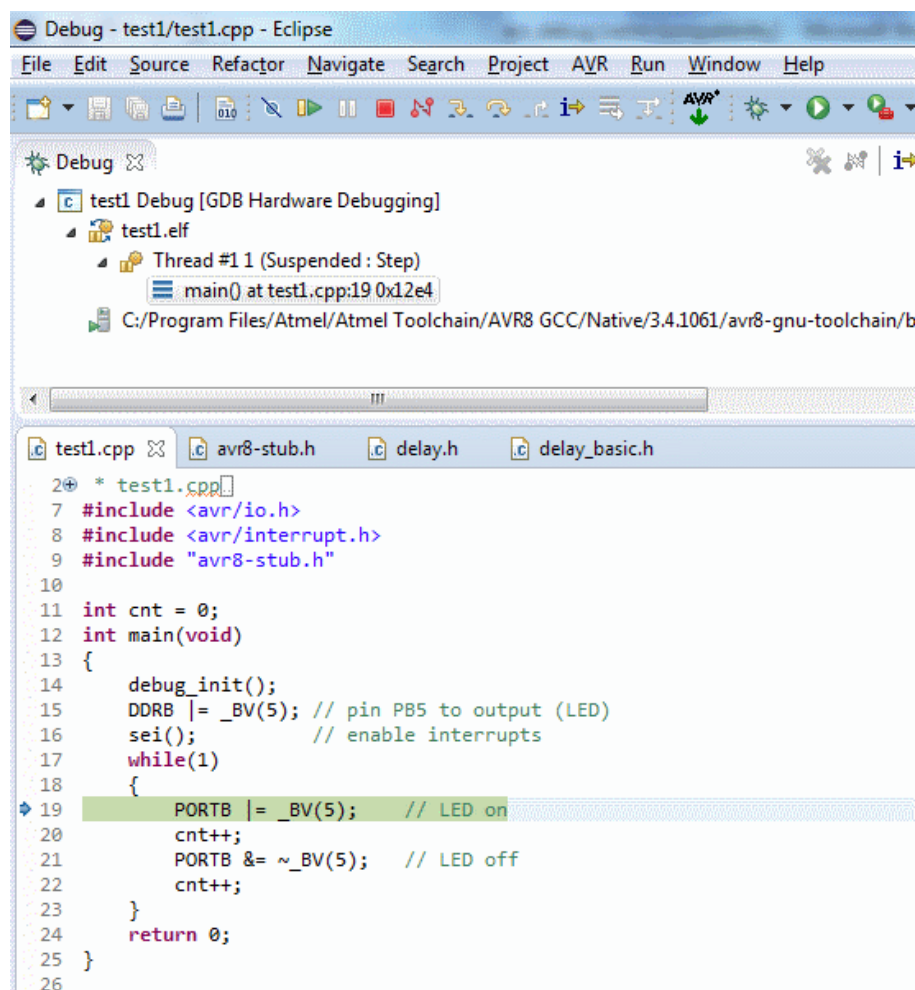
You can now single step the program using the Step over button in the toolbar. This will advance the program by one line. Note that when you step over the PORTB |= _BV(5); line, the LED on Arduino board will turn on.
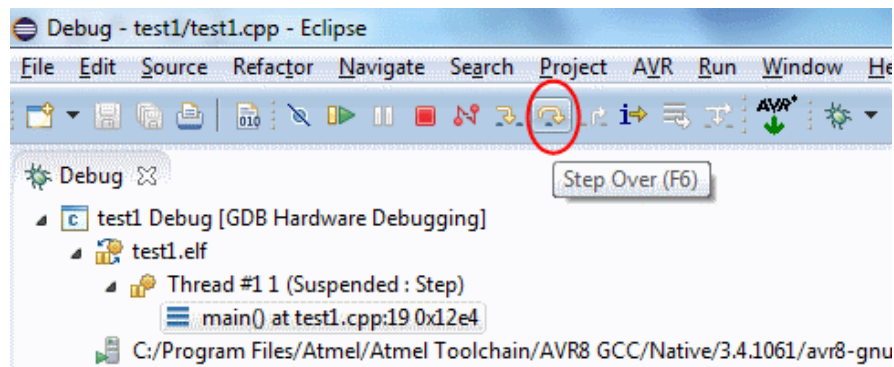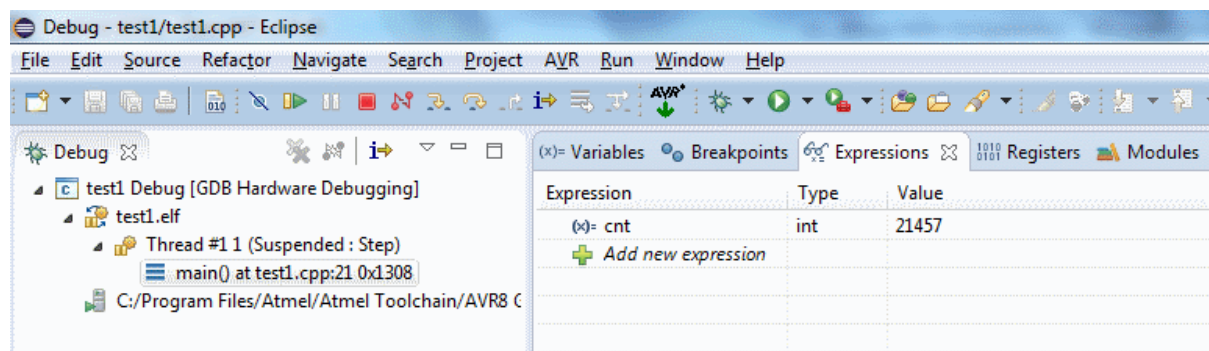

We can also look at the value of the "cnt" variable. Just hoover the cursor at the cnt++; code and the value will be displayed. Don't be surprised the value is high, remember, we interrupted the program at random moment.

We can also change the value of the cnt variable:

In the upper right corner switch to the Expressions tab. If it is not open, go to the main Eclipse menu Window > Show View and select Expressions. This will open the Expressions tab.

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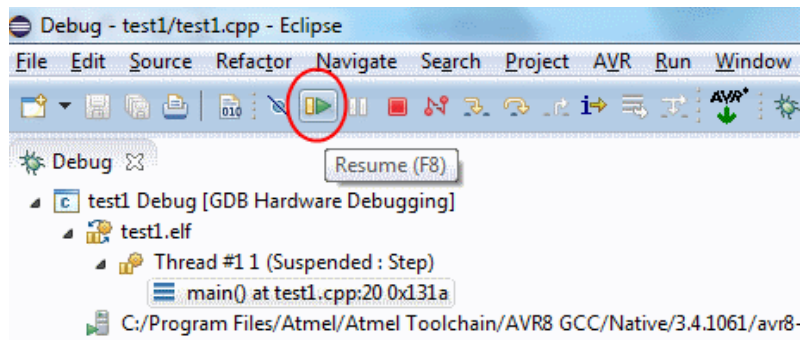
Interrupt the program at any time using the Suspend button (pause) which is just next to the Resume button.

Breakpoints

You can also stop the program at any line. To do so, right click at the line where you want the program to stop. Make sure you click into the gray left margin, not into the code.

From the context menu select Toggle breakpoints. 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 plan to modify your program, make sure you rebuild and upload it to the board before debugging again. For this, the COM to TCP proxy server 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 COM to TCP server (use the start_proxy.bat file)
- Start the debug in Eclipse (Expand the Debug button in the toolbar and select 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. The 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 the COM to TCP proxy server is running and that your program calls the debug_init() function.

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.

```c
#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 big 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 MCU Framework (the software library), 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: C++ Project type, C++ Cross Target Application > Empty project, MCU Type Atmegas328P and MCU Frequency 16000000.

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 hardware\arduino\avr\cores\arduino in your Arduino installation. See the picture below.
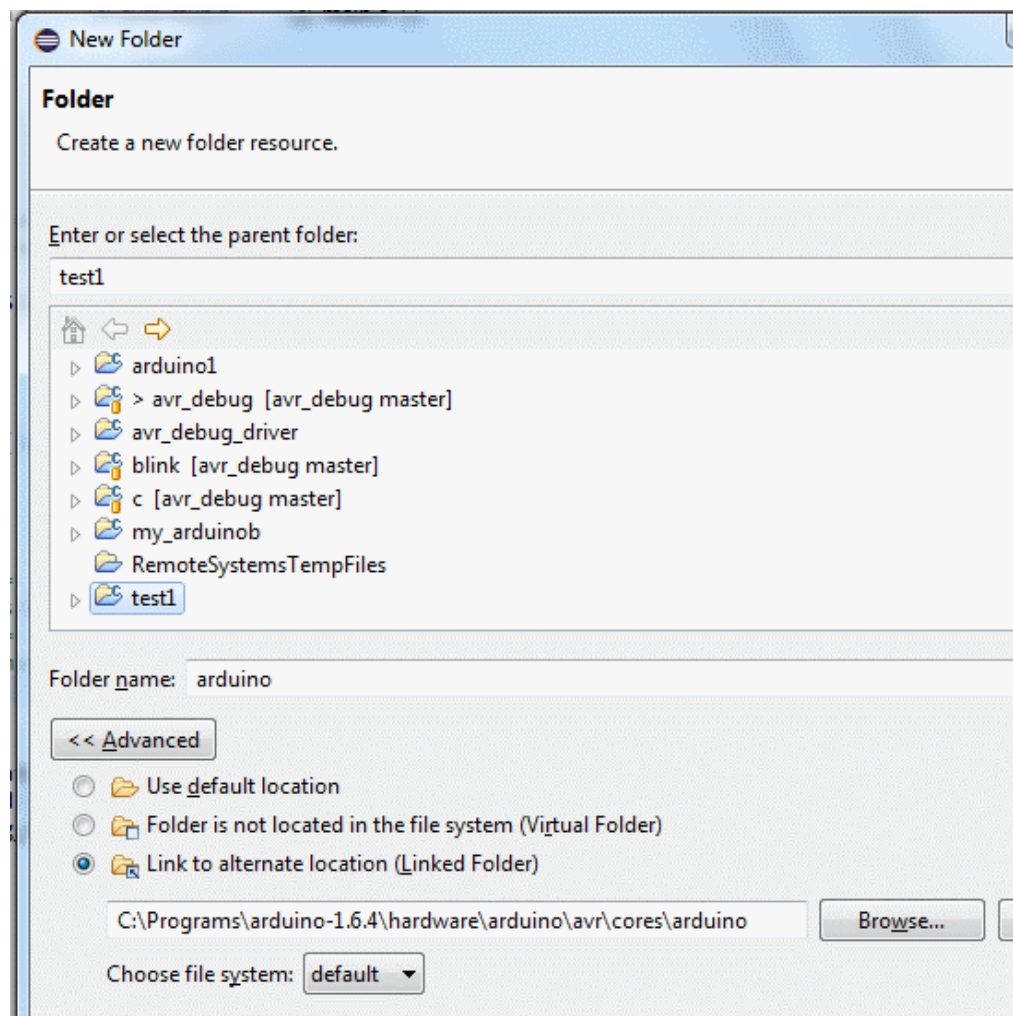
Click Finish to create (link) the folder.

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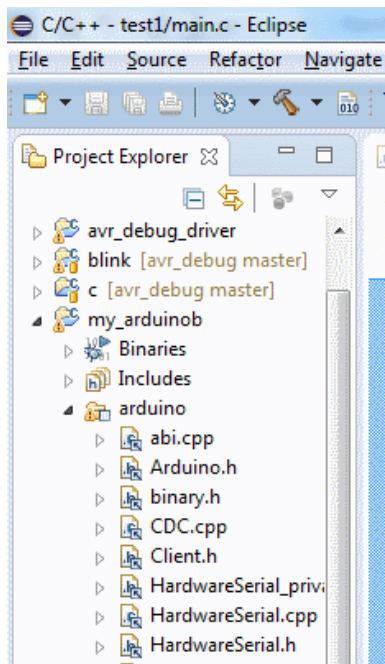
Note that these folders are linked to the location in your Arduino installation; they are not copied into the project. If you modify or delete the files under these folders; you could damage your Arduino installation.

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:

```cpp
#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.

NOTE: Before uploading, enable the HEX file output 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 (if you haven't done it yet).

 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 your program cannot use the Serial functions (the hardware serial) and it cannot use external interrupt INT0. 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 required files and set the debug format in Preferences to dwarf2.

In the program, add the #include "avr8-stub.h" and call to debug_init() and breakpoint(). The code could look like this:

```
#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 this:

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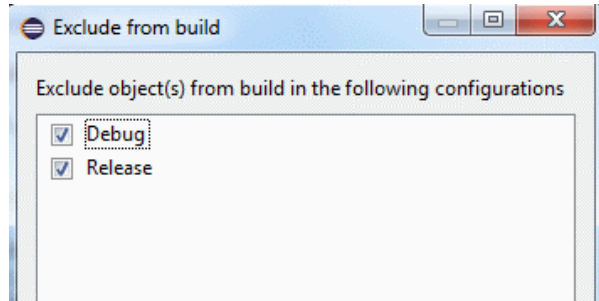
Now this file will not be built with your program. This will solve the multiple definition for vector 18 (UART) but it also means the Arduino Serial functions will not work. Note that this applies only to this program (project), not to other programs you create either in eclipse or in the Arduino IDE.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

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

This solves the multiple definition 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 note 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 duplication it (the button next to New launch configuration). Then you just need to change the Project and C/C++ Aplication fields on the Main tab. Use the Browse and Search Project… buttons to select the proper project and the corresponding .elf file.
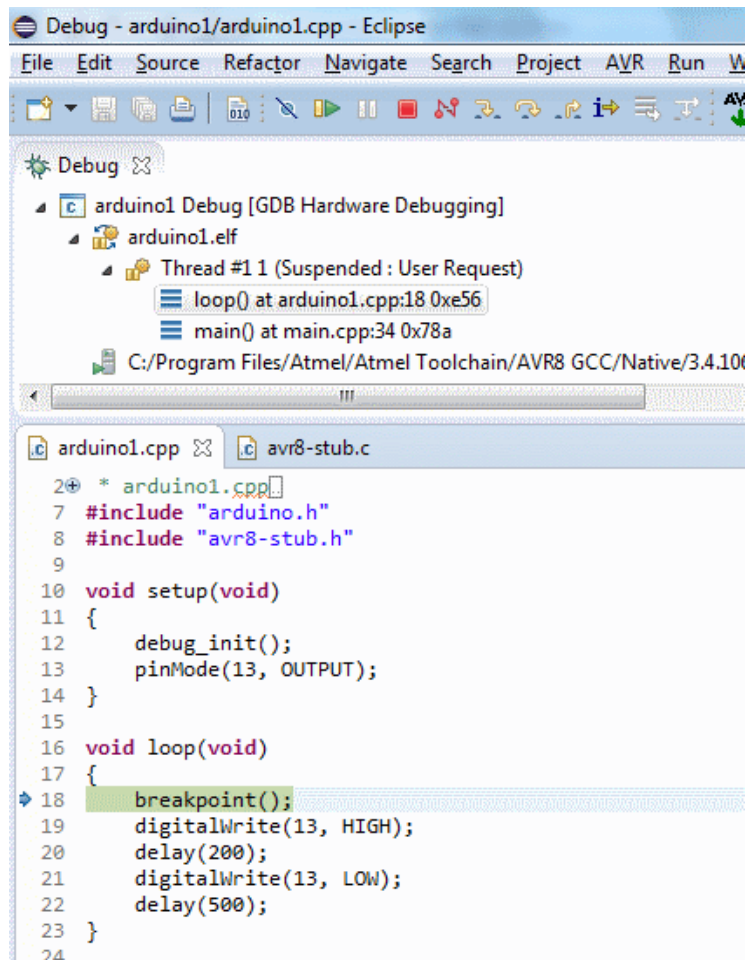
Close the Debug Configurations window.

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

Start the COM to TCP proxy server (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:
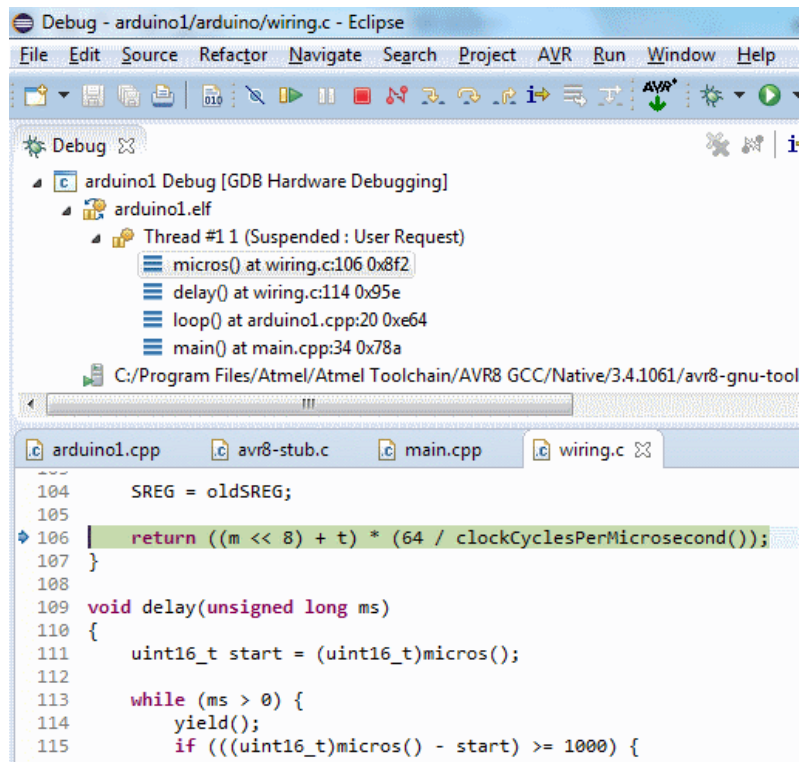
The program is stopped at the breakpoint line.

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.

Of course, 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 we connect with the debugger, it will stop at any random place; most likely somewhere in the delay() code. You may see this:

```
Debug - arduino1/arduino/wiring.c - Eclipse

File  Edit  Source  Refactor  Navigate  Search  Project  AVR  Run  Window  Help

Debug

  arduino1 Debug [GDB Hardware Debugging]
    arduino1.elf
      Thread #1 1 (Suspended : User Request)
        micros() at wiring.c:106 0x8f2
        delay() at wiring.c:114 0x95e
        loop() at arduino1.cpp:20 0xe64
        main() at main.cpp:34 0x78a
      C:/Program Files/Atmel/Atmel Toolchain/AVR8 GCC/Native/3.4.1061/avr8-gnu-tool

  arduino1.cpp    avr8-stub.c    main.cpp    wiring.c

104        SREG = oldSREG;
105
106 |     return ((m << 8) + t) * (64 / clockCyclesPerMicrosecond());
107 }
108
109 void delay(unsigned long ms)
110 {
111        uint16_t start = (uint16_t)micros();
112
113        while (ms > 0) {
114            yield();
115            if (((uint16_t)micros() - start) >= 1000) {
```

In the upper window (Debug) there is so called call stack, the "chain" of calls which led us 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.

Note: Alternative way 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 instead:

Open the WInterrupts.c file.

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, but the disadvantage is that attachInterrupt will not work for the interrupt int.0 in any program. The change in the WInterrupts.c file affects all your Arduino projects.

If you are experienced programmer, 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 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.

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.
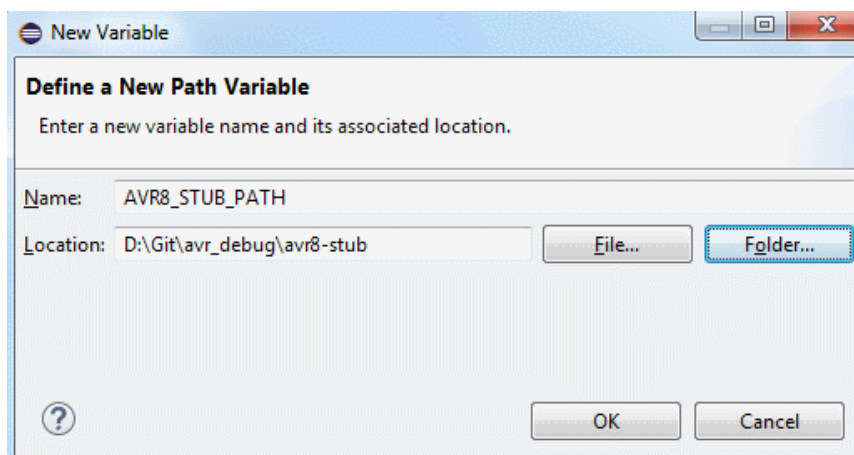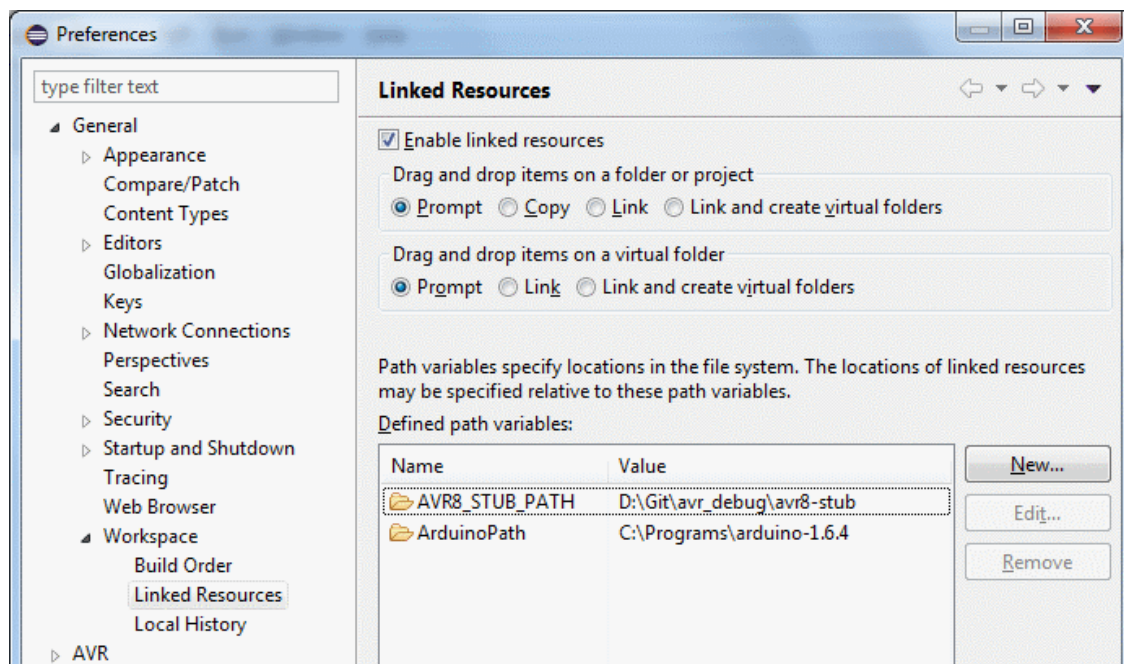

In the same way create another variable named ArduinoPath.

Set the location to the root of your Arduino installation, for example c:\Programs\arduino-1.6.4\.
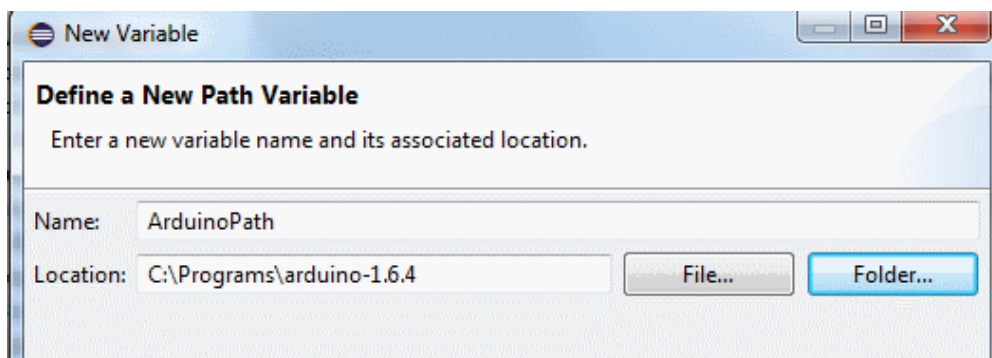
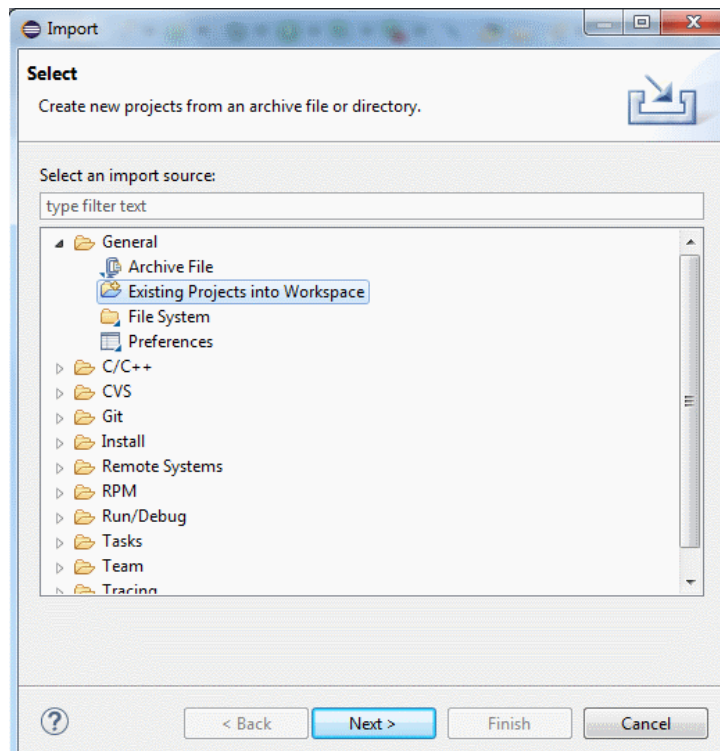There should now be these two variables in the list:

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.

Select one or more projects to import. Make sure the "Copy projects to workspace" option is NOT enabled.
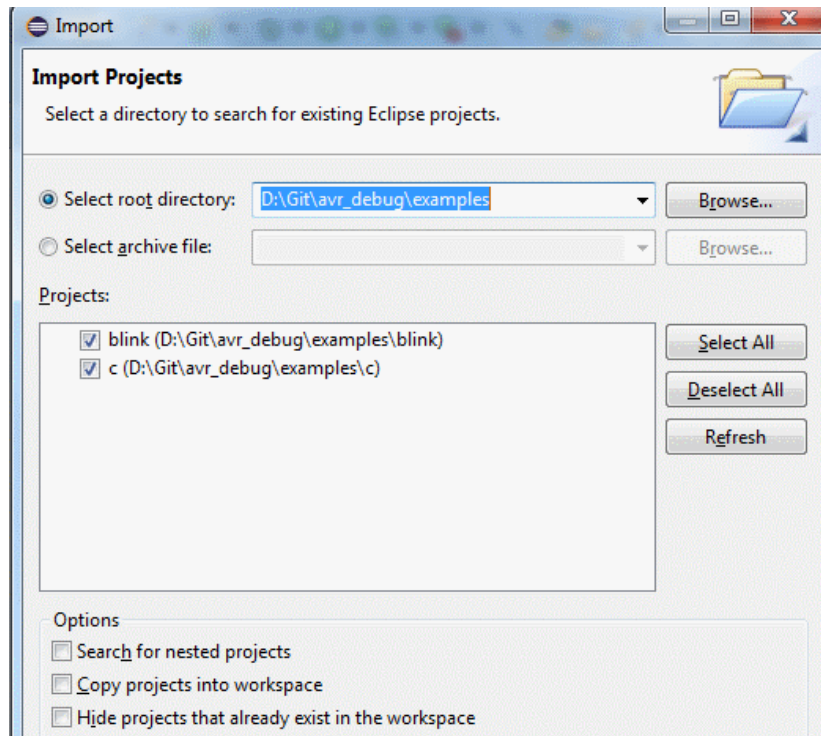
Build the project(s). There should be no errors.

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).
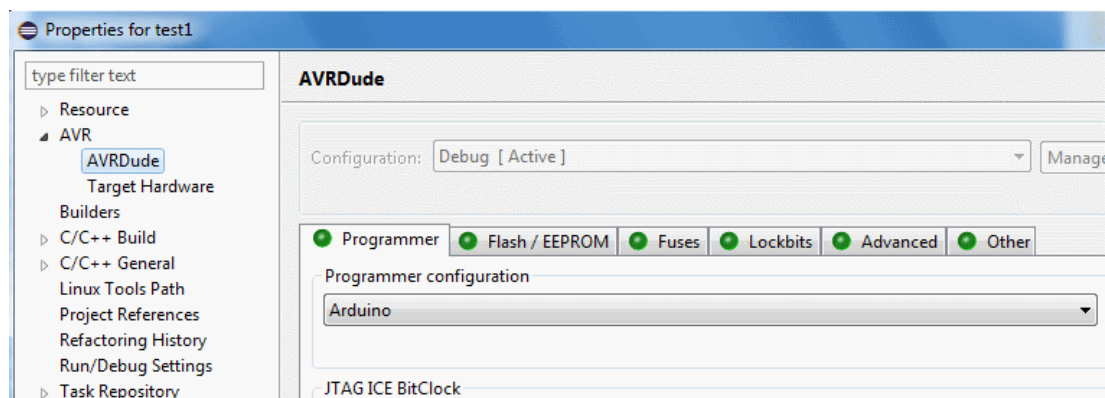
Close the Properties window with the OK button.

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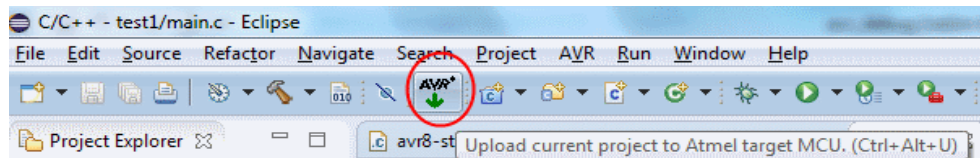
In the Eclipse console window you should see how your program is uploaded, ending with "avrdude done. Thank you."

Now we will connect to the program with debugger. But first, 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.

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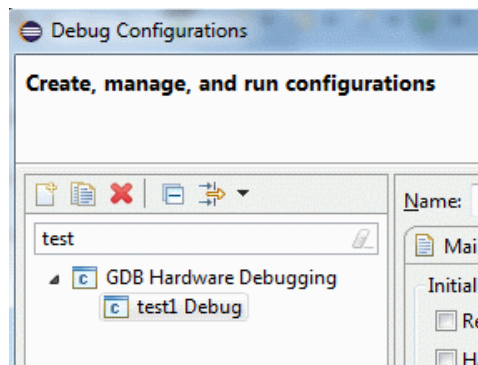
Select the configuration and in the right-side window select the Debugger tab.

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".
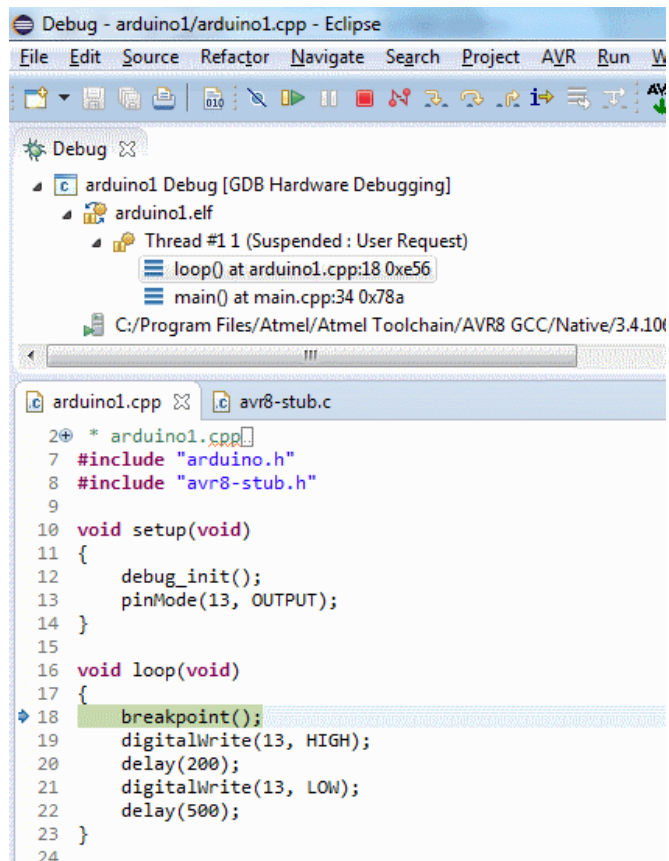
And from Atmel Toolchain:

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.

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.

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: 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.

Problem: The program is not jumping at some line or 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: substract 1 from i.

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

This can happen even with no optimizations (-O0 option). Things can go even "wilder" if you have optimizations enabled.

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 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 (in upper right cornenr of Eclipse).