Setting up AVR development environment

Including support for Linux, MacOS and Windows

Version 1.0

Annex Document A.0
Gerard Marull Paretas
July 2007

Index

1. Introduction

Since setting up a working development environment for AVR microcontrollers has taken me so many hours and reading from lots of websites, I decided to write this document explaining the process as clearly as possible. I will only make use of open source software since it can be accessed by everybody without any cost and offers a very good quality. That process will be explained for the three operating systems mainly used: Linux, MacOS and Windows (note that the process between Unix systems is quite similar). The following topics will be treated:

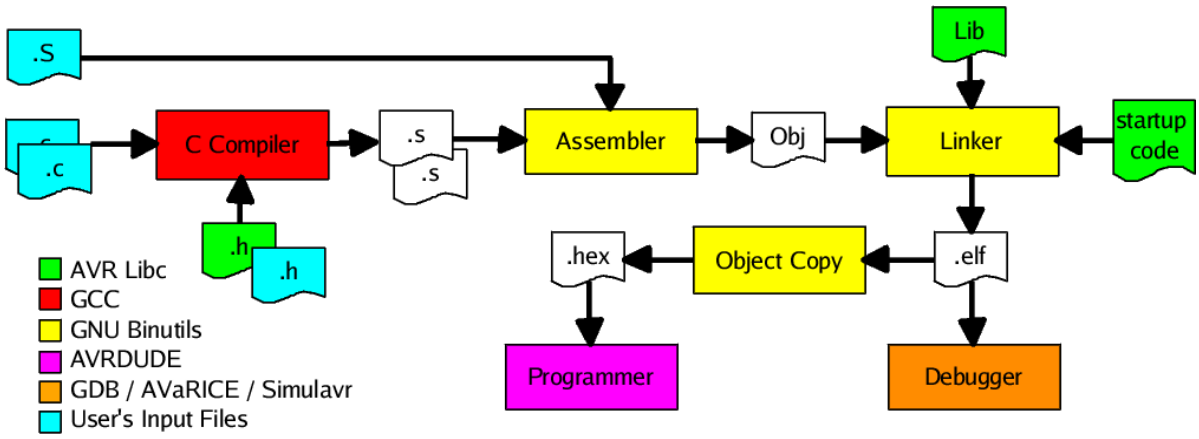
- Installing AVR GNU Toolchain (*gcc, binutils and avr-libc*)
- Installing the programming tool *avrdude* and learning how to use it with JTAGICEmkII and a home-made Parallel port ISP.
- Introduction on how to create a Makefile
- Installing Eclipse IDE and integrating with AVR development tools
- Graphical debugging with JTAGICEmkII using *avarice* and *gdb* through Eclipse

If you find any mistakes in this document, please, report them at gerardmarull@gmail.com

2. AVR GNU Toolchain

2.1. Introduction

The AVR GNU Toolchain is the fundamental part of our development environment, and it is compounded by *GCC*, *Binutils* and *avr-libc*. The following graphic shows a summary of what AVR Toolchain can do:



Graphic 2.A

Except for programming and debugging which will be explained later, all these tasks (and more) can be performed by the AVR toolchain. I will give a brief explanation of each program/process in the following paragraphs.

2.1.1. GCC (*GNU Compiler Collection*)

Programming in assembler is a difficult task, especially on large projects. However, *GCC* supports the translation of C and C++ code into target assembly only. As you see in *Graphic 2.A*, there are more processes to execute apart from compiling: assembling and linking, but this is not done by *GCC*. This work will be done by *Binutils*.

2.1.2. Binutils

Binutils are a set of programs wherein GNU assembler and linker are included. It also contains many other useful tools to work with binary files generated by the toolchain programs. A full list of included tools is shown below:

Tool name	Description
avr-as	Assembler for producing AVR code
avr-ld	Linker to link AVR object files
avr-ar	Create, modify, and extract from libraries (archives)
avr-ranlib	Generate index to library (archive) contents
avr-objcopy	Copy and translate object files to different formats
avr-objdump	Display information from object files including disassembly
avr-size	List section sizes and total size
avr-nm	List symbols from object files
avr-strings	List printable strings from files

avr-strip	Discard symbols from files
avr-readelf	Display the contents of ELF format files
avr-add2line	Convert adresses to file and line
avr-c++filt	Filter to demangle encoded C++ symbols

Some of these will be explained later in this document.

2.1.3. avr-libc

Programming AVR devices in C is a great thing, but we need something more: a C Library. Without it, we would have to refer to registers by their addresses or add startup code. As you know this would be quite a difficult task, but you can forget this nightmare: *avr-libc* is here. Apart from including specific device headers, it even includes built-in functions, most of them which are equal or similar to the ones found in a C standard library, and also AVR specific functions. You can refer to *avr-libc user's manual* if you want to get more information.

2.2. Installation of AVR GNU Toolchain

After having an idea of the toolchain components, we will start installing it. I will divide it in two parts: one for Windows and the other for Linux/MacOS. The second process is very similar for both operating systems.

2.2.1. Windows

The installation on Windows is very easy: Just install WinAVR. This package is always up to date and even includes many necessary and useful patches. You can download the most recent version of WinAVR at their official website: http://winavr.sourceforge.net/. It also includes programming and debugging tools, and even the Programmer's Notepad to manage your projects. You can read the full list of features in their website.

2.2.2. Linux / MacOS

Since both Linux and MacOS are Unix based operating systems, the installation process is very similar. However some things change a little, but they will be explained. You have two options for the installation: getting binary packages or compiling from sources. I personally recommend compiling it from sources, basically for two reasons: the first is to have the latest versions of the programs, and the second is that you can include patches that could be necessary (in my case for example, I need support for Atmega1281, which now is only available applying a patch). In this document we will focus on compilation from sources.

A. About patches

As I said, when you compile from sources you can add patches that can bring new features to the program in question or fix some bugs. The first thing you will probably think is where will I get these patches. Well, the easy way, if you do not want to search or know which ones are available and recommended, is to take them from WinAVR. Since WinAVR is always up to date and uses most of the available patches for the AVR Toolchain, it is a good source where to take them from. You can browse them by accessing their Sourceforge CVS at the following address:
http://winavr.cvs.sourceforge.net/winavr/patches/
** Note that some of the patches can be for a specific version of a program or only necessary for Windows platform. So ensure that they are suitable for your program version.*

You will see that they have a number before their name (ex. 00-patchname.patch). This indicates the order in which they have to be applied.
After collecting the ones that you need, you will have to apply them. This is a very easy task, since it only requires one command for each patch. You will have to make use of *GNU patch* program, which will do that line replacing job for us. The command to apply any patch is:

```
# patch -p0 <patchname.patch
```

There is one flag that you should know a bit more about: '*-p0*'. It indicates the patch level. If you open a patch file, you will see some lines like these ones:

```
--- gcc/config/avr/avr.c.orig   Thu Jul 28 00:29:46 2005

+++ gcc/config/avr/avr.c        Mon May 14 07:18:20 2007
```

What this means is that it will expect to find a folder called *gcc*, then *config*, then *avr* and a file called "*avr.c*" inside, where to apply patch. Patch level will indicate the level of our current working directory in relation to patch file path. For example: when you indicate '*-p0*', it means that it will expect to find all folder levels: *gcc*, *config* and *avr*. But if you put '*-p1*', means that it will only expect to find *config* and *avr*, ignoring the first one (*gcc*).
I hope that you now know how to apply patches well, but if you want to know more about them, you can visit this website:
http://www.cpqlinux.com/patch.html

B. Before starting installation

Two programs are needed before proceeding: *flex* and *bison*. These are the *gcc* compiler for your platform and building utilities such as *GNU make* are needed. You can find them in your Linux package repositories or in the case of MacOS in the first CD-ROM of MacOSX and also in Apple Developer Connection at http://www.apple.com. *Flex* and *bison*, are also available in package repositories, but in the case of MacOS, I recommend that you install *Fink*, an APT like interface that makes many packages available from a repository (*flex* and *bison* between them). You can download it at http://fink.sourceforge.net/ and installing a program is done by just typing this command:

```
# sudo fink install nameofpackage
```

Another thing is where to install the toolchain. I recommend that you keep it separate from a default system paths, for example at */usr/local/avr*. So create that folder and store it in an environment variable:

```
# sudo mkdir /usr/local/avr
# export TOOLCHAINPATH="/usr/local/avr"
```

C. Starting the process: Binutils

The first package to install is *Binutils* downloadable at:
http://sources.redhat.com/binutils/
After downloading it, we will have to decompress it and create a folder called "*obj-avr*" inside where you can store files generated by the compilation process:

```
# tar -zxvf binutils-<version>.tar.gz
# cd binutils-<version>
# mkdir obj-avr
# cd obj-avr
```

D. Finally, we will have to configure, compile and install it. For MacOS users '*--enable-install-libbfd*' flag must be added when executing configure:

```
# ../configure --target=avr --prefix=$TOOLCHAINPATH --
disable-nls
# make
# sudo make install
```

Now, if you look at */usr/local/avr* you will notice the new files there. If you enter to '*bin*' folder, you will see all programs mentioned in section 2.1.2. But if you try to access them from your shell typing the program name, you will get something like this: "*bash: avr-xxx: command not found*". The reason is that they are not in a default place, but it is easy to solve: just add the '*bin*' directory to your PATH environment variable:

```
# PATH=$PATH:/usr/local/avr/bin
# export PATH
```

E. GCC

First of all, download *GCC* from the official website: http://gcc.gnu.org/
The steps to compile *GCC* are essentially the same as for *Binutils*:

```
# tar -zxvf gcc-<version>.tar.gz
# cd gcc-<version>
# mkdir obj-avr
# cd obj-avr
# ../configure --target=avr --prefix=$TOOLCHAINPATH --enable-
languages=c,c++ --disable-nls --disable-libssp --with-dwarf2
# make
# sudo make install
```

F. avr-libc

Finally, we will install the AVR C Library *avr-libc*. You can get it from:
http://savannah.nongnu.org/projects/avr-libc
And the steps are again similar to those in the previous packages:

```
# tar -jxvf avr-libc-<version>.tar.bz2
# cd avr-libc-<version>
# ./configure --prefix=$TOOLCHAINPATH --
build=`./config.guess` --host=avr
```

That is all: now you can compile any code and set it ready to upload into your AVR microcontroller. Steps on how to compile will be explained later.

3. Programming tools

Being able to compile your programs but not to upload them into your microcontroller is not very useful, but of course programmers exist for that. In this document we will use *avrdude*, which is a very powerful tool that will let us program our device, read/write fuse settings and more. To program your microcontroller you need a tool like STK500 or JTAGICEmkI/II. They are really good tools, but if you are just starting with AVR, it can be a bit expensive. But do not worry: AVR microcontrollers have ISP (*In System Programming*). ISP is a method to program your chip without having to remove it from the circuit and just using 5 wires. The good thing is that making an ISP tool is very easy, you just need a few wires, resistors and a parallel port connector. I will also explain all the steps on how to make your own programmer.

3.1. Installing avrdude

First we need to install our programming software tool. You can download it from its website: http://savannah.nongnu.org/projects/avrdude/, and the installation is done as follows:

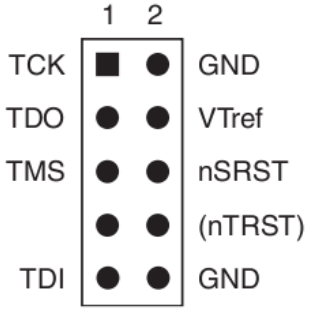
```
# tar -zxvf avrdude-<version>.tar.gz
# cd avrdude-<version>
# ./configure
# make
# sudo make install
```

3.2. Choosing a programmer

Before starting to use *avrdude*, you will need to have a programmer. *Avrdude* accepts a wide list of programmers, just refer to avrdude man page to see the full list of ones available. In this document I will explain how to use JTAGICEmkII and create/use a Parallel ISP programmer.

3.2.1. JTAGICEmkII

I bought this device a few weeks ago and I really love it! It is expensive: it costs about 299$, but what it can do is priceless. Programming the microcontrollers is only a little taste. You can choose between ISP or JTAG programming mode. I will only explain JTAG programming, and this way we will be able to debug using the same cable, too. Before starting to program you will need to make the following connections between JTAGICEmkII and your microcontroller:



Vtref is connected to VCC and nSRST to the RESET pin of your microcontroller. Pins 7 and 8 are not connected. Find in your microcontroller's datasheet the corresponding pins for TCK, TDO, TMS and TDI.

A. Known issues when using USB

I would also like to talk about known issues about using JTAGICEmkII through USB. I have seen in AVRFreaks.net forums that in Windows some people cannot get it work due to the following error:

```
AVRDUDE: usbdev_open(): did not find any (matching) USB device "usb"
```

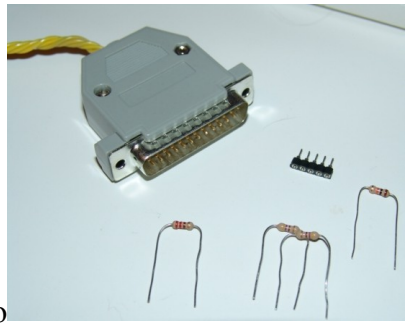
The solution is to install libusb-win32, available at:
http://sourceforge.net/projects/libusb-win32
And in my case, I also had a problem under Linux about permissions. I solved that adding a new udev rule, that can be achieved by creating a new file with *.rules* extension at */etc/udev/rules.d* named *10-avr.rules* for example, and adding the following line inside:

```
SUBSYSTEM=="usb", ATTR{product}=="JTAGICE mkII", MODE="0666"
```

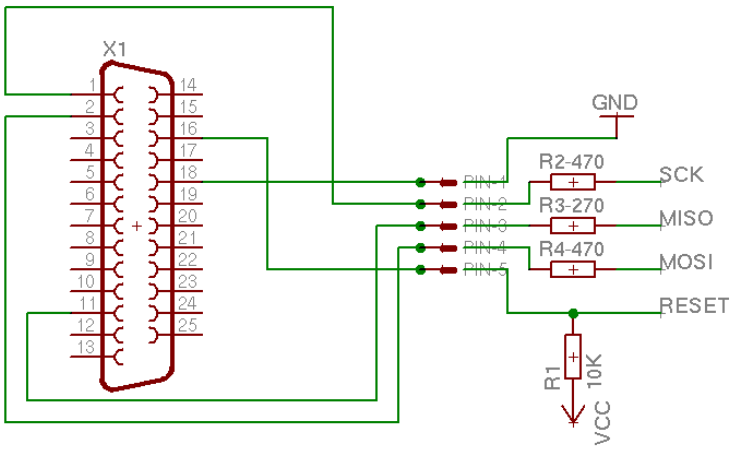
3.2.2. Parallel port ISP programmer

Although JTAGICEmkII is a good option, it is quite expensive, and specially for beginners. However AVR microcontrollers have ISP which will let us program our chip with only 5 wires and without removing it from the circuit. For that you will need the following components:

a) 2 - 470Ω resistor
b) 1 - 220Ω resistor
c) 1 - 10KΩ resistor
d) 1 - Parallel port connector (DB-25)
e) 1 - 5-pin connector (see photo)
f) 5 long wires or the connector wire



And the cable plus circuit connections you will have to make:



For the connections SCK, MISO, MOSI and RESET look at your microcontroller datasheet to see which the corresponding pins are.

Just one more thing before ending this point: in assembled parallel cables, sometimes all pins are not soldered. If that is the case you will have to disassemble it and solder an unused pin cable to the correct pin.

3.3. Working with avrdude

After getting everything ready, it is time to learn how *avrdude* works. It is very easy, and just requires a few free bytes of memory in your brain. First I will list the *avrdude* flags that we will use. Note that more are available.

Flag	Description
-p	**Part id**: You have to specify the model of the microcontroller you are using. An example of this flag when using Atmega8 would be: *-p m8*. See a full list of supported devices and its tag in *avrdude* man page.
-c	**Programmer type**: You have to specify your programmer. An example of this flag when using parallel port ISP programmer would be: *-c dapa*.
-P	**Programmer port**: Specify the port where programmer is attached. An example of this flag when parallel port is used: *-P /dev/parport0*
-e	**Erase**: Perform a chip erase (flash and eeprom)
-U	Perform a memory operation. Structure: **memtype:operation:filename:format** **memtype** → The memory you want to access. Available ones: *flash, eeprom, hfuse, lfuse, efuse, fuse, calibration, lock, signature*. **operation** → Operation type: *w* (write), *r* (read), *v* (verify) **filename** → File where to read or write **format** → [optional] Specify format of the file. Look at avrdude man page to see available formats. An example of this flag when uploading your program called *main.hex* into the flash space: *-U flash:w:main.hex*

Finally, let us see some examples of how you would use avrdude:

1. **Erasing the chip contents of an Atmega16 using JTAGICEmkII (usb conn.)**

```
# avrdude -p m16 -c jtagmkII -P usb -e
```

2. **Uploading a program to Atmega8 using Parallel ISP**

```
# avrdude -p m8 -c dapa -P /dev/parport0 -e -U
flash:w:program.hex
```

3. **Reading the contents of flash from an Atmega 1281 using JTAGICEmkII (serial conn.) in fast mode and store in Intel Hex format**

```
# avrdude -p m1281 -c jtag2fast -P /dev/ttyS0 -U
flash:r:output.hex
```

4. Automation using a Makefile

Compiling or uploading a program makes use of several shell commands, and occasionally it takes a long time to write them every time we update our program. However we can automate this process by creating a Makefile. Makefiles are no more than a file that stores a list of these commands, but it even offers an organization of your compilation steps and dependency between them, the creation of variables where to store parameters, and more. There is a program that processes these Makefiles: *GNU Make*, and it is accessed by typing '*make*' in your shell. We will not focus on how to create Makefiles in this document, I will just explain the parts of a simple template and how to use that.
WinAVR users have Mfile, a program that lets you create Makefiles with just a few clicks selecting your device, files to compile, etc. So they can step over this process and use Mfile if they want. Users from other operating systems may find useful to take a look at WinAVR Mfile template, since it is very complete and full of interesting features. However little things can change, since it is made for Windows. *Note: Mfile for Unix systems does not seem to be updated.*

4.1. A simple but flexible Makefile

When I started in the AVR world, I did not know about Mfile, so I decided to learn a bit more on how to create Makefiles. With the help of some tutorials, I made my first Makefile. However the first version is obsolete now. I added new features and corrected some things when I discovered Mfile. I will copy my entire Makefile below, adding some comments to make it easier to read. An explanation of how it works and how to use it will be given in the next paragraph.

```
# Available targets:
# make all          /Do everything (main.elf and main.hex)
# make load         /Program the device
# make dload        /Program the device and start debugging
# make debug        /Start debugging
# make clean        /Clean
# make size         /Get information about memory usage
# make filename.s   /Generate assembler file from a C source
# make filename.o   /Generate object file from a C source


#Source files
SRC_FILES=\
          main.c \
          two.c \

#Object files
OBJ_FILES=$(SRC_FILES:.c=.o)

#Directories where to look for include files
INC_DIRS=\
          -I. \
          -Iincludes \

#Output file name
OUTPUT=main

#Programmer and port
PROG=jtag2fast
```

```
PORT=usb

#Debugging host and port
DHOST=localhost
DPORT=6423

#Compiler related params
MCU=atmega1281
CC=avr-gcc
OBJCOPY=avr-objcopy
CFLAGS= -mcall-prologues -std=gnu99 -funsigned-char -funsigned-bitfields
-fpack-struct -fshort-enums -mmcu=$(MCU) -Wall -Wstrict-prototypes
$(INC_DIRS)
#Optimization level
CFLAGS+=-Os
#Debug info
CFLAGS+=-gdwarf-2


#Generate hex file ready to upload
all: $(OUTPUT).elf
          $(OBJCOPY) -R .eeprom -O ihex $(OUTPUT).elf $(OUTPUT).hex
          @echo "--------------------------------"
          @echo "        BUILD FINISHED          "
          @echo "--------------------------------"

#Link output files
$(OUTPUT).elf: $(OBJ_FILES)
          $(CC) $(CFLAGS) $(OBJ_FILES) -o $(OUTPUT).elf -Wl,-
Map,$(OUTPUT).map

#Create object files
$(OBJ_FILES): %.o : %.c
          $(CC) -c $(CFLAGS) $< -o $@

#Create assembler file of a C source
%.s: %.c
          $(CC) -S $(CFLAGS) $< -o $@

#Loads the program to the avr device
load:
          avrdude -p $(MCU) -c $(PROG) -P $(PORT) -e -U
flash:w:$(OUTPUT).hex

#Starts debugging
debug:
          avarice -2 -j $(PORT) $(DHOST):$(DPORT)

#Loads the program to the avr device and starts debugging
dload: $(OUTPUT).elf
          avarice -2 -j $(PORT) -e -p -f $(OUTPUT).elf $(DHOST):$(DPORT)

#Get information about memory usage
size: $(OUTPUT).elf
          avr-size -C --mcu=$(MCU) $(OUTPUT).elf


#Cleans all generated files
clean:
          rm -f $(OBJ_FILES)
```

```
	rm -f $(OUTPUT).elf
	rm -f $(OUTPUT).hex
	rm -f $(OUTPUT).map
```

4.2. Understanding and using the Makefile

Before explaining how to use it, I will give you a brief explanation of things you should know about this Makefile.

A) Syntax

1. All lines starting with '#' character are considered comments, so they will be ignored by GNU Make.
2. All declarations like *NAME=List of words* are variables. The name which we will use to refer to it is the part before the equal sign. The text preceding the equal sign is the content that a variable stores.
3. To get the content of a variable, we will write its name preceded by a '$' character and put it in brackets: $(NAME).
4. An inverted slash at the end of a line, means that the next line is the continuation of the actual one. Example:
   The following line:
   TEST=some.c test.c
   Would have the same effect if it was written as follows:
   *TEST=some.c *
   test.c
5. Every line starting with a TAB is understood as a command.

B) The Make targets

Every line with a format like '*targetname: other*' is considered a target. This means that this will be accessible by typing '*make targetname*'. For example: when you call '*make clean*', the commands below the clean target will be executed. However, some targets have names such as '%.s' or a variable content like '$(OUTPUT).elf'. The first one means that it will be accessible by any name followed by '.s' extension. And in the second case, it is just as if it was a normal target, just replace the $(OUTPUT) by content of the variable, that would give access to '*make main.elf*'.
The second part of the target (after the colon) means which targets have to be processed before the actual one or in some cases the files needed, so we could call them dependencies. Therefore, when you call '*all*', '*$(OUTPUT).elf*' has to be processed first, and even when '*$(OUTPUT).elf*' is called, '*$(OBJ_FILES)*' is needed before.

C) Usage of this Makefile

1. **Adding new files:** Add an space followed by an inverted slash to the last file in the SRC_FILES variable. Then add a new line and write the new file name.
2. **Adding directories where to look for include files:** The same as described above but in INC_DIRS variable and preceded by '*-I*'.
3. **CFLAGS**: This variable contains the flags that will be passed to the compiler. You can learn the meaning of each one by reading the *avr-gcc* man page.
4. **Other modifications**: Most variables are commented on explicitly in the same Makefile (ex. the programmer - PROG), so modify them according to your needs.
5. **Shell access**: Read the header of the Makefile.

I know that there is a lot more to talk about Makefiles and maybe in this section they are not explained as they should be, but I hope that you have got a general idea and you have basically understood how to use and modify the template shown here. If you want to learn more about Makefiles, just visit http://www.gnu.org/software/make/manual/make.html

5. Choosing an IDE: Eclipse

You have almost all the tools necessary to develop, but something is missing: an IDE. IDE (*Integrated Development Environment*) is a program that will let you manage your projects, edit source files, debug, and more. I have chosen Eclipse, basically for a few reasons: it is cross-platform (works in Linux, MacOS and Windows), offers integrated debugging, has support to manage C/C++ projects, ability to add plugins, and a wide list of other features. It uses ~200mb of RAM while running, which could be a problem for old computers, but you have other alternatives such as *Programmer's Notepad* as an IDE and *Insight* for graphical debugging which use less memory.

5.1. Installation of Eclipse

Installation will be divided in two steps: the installation of the Eclipse core, and the installation of the Zylin plugin that will add C/C++ features and embedded devices support.

5.1.1. The Eclipse core

Also known as Eclipse classic, it is the core of Eclipse environment. You have to install it in order to be able to add plugins that will give us features we are interested in afterwards. In this document I will use version 3.3, but the process should be similar for newer versions. You can download it at http://www.eclipse.org for free. Note that you must have Java installed to run Eclipse, so if you do not have it download from http://www.java.com, or if you are a Linux user it is probably available at your distro's repository. Let us see how to install it:

1. **Linux**: Decompress the file that you have downloaded. Then, you will have to create a directory to store these program files, for example */usr/local/share/eclipse*. Copy them there, and finally make a symbolic link to Eclipse program in */usr/bin* by typing:

```
# sudo ln -s /usr/local/share/eclipse/eclipse /usr/bin
```

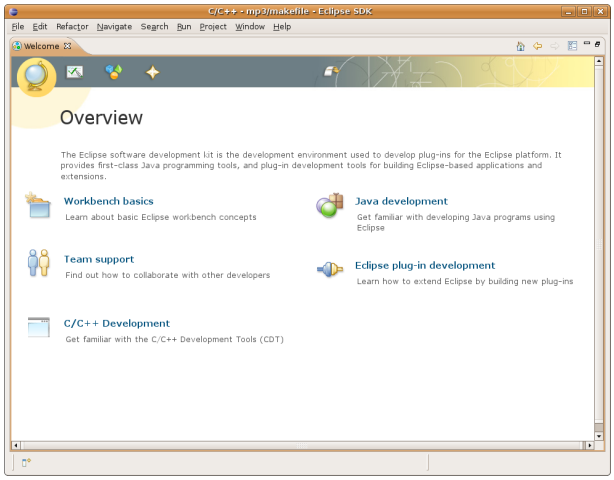
2. **MacOS**: Decompress the file, create a folder in your *Applications* directory and copy all the files there.
3. **Windows**: Decompress the file, create a folder in your *Program Files* directory and make a shortcut to *Eclipse.exe* in your Desktop or wherever you want.

5.1.2. The Zylin plugin

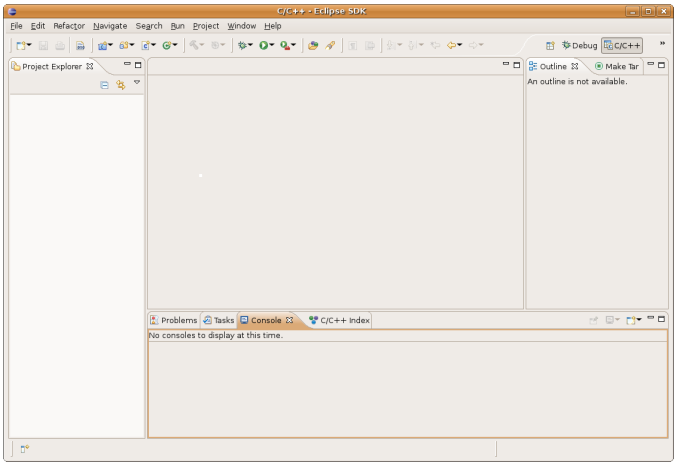
If you run Eclipse you will realize that it is an IDE for Java developers, so we must install a plugin to make it suitable for our purposes. Zylin Inc. creates a plugin that is basically CDT (a famous C/C++ plugin) but modified to add support for embedded devices. It will also let us debug through JTAG interface. First, you have to download it from http://www.zylin.com/ . It is composed of two files. Just extract both contents in the Eclipse root directory.

5.2. First steps with Eclipse: Setting up a project

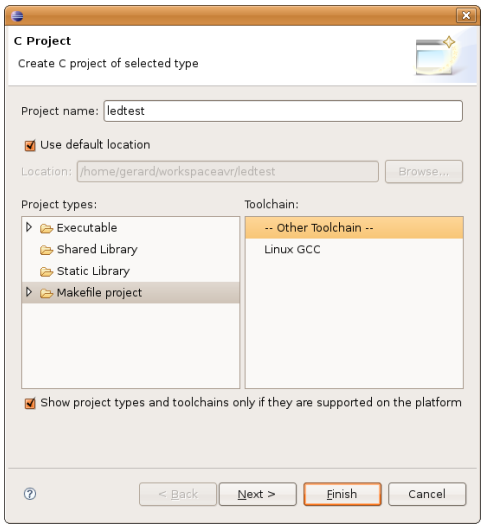
When you run Eclipse for the first time, it will ask you to set the Workspace folder where the projects and their files are stored. Choose the path you prefer. After that you will see a Welcome screen like this one:



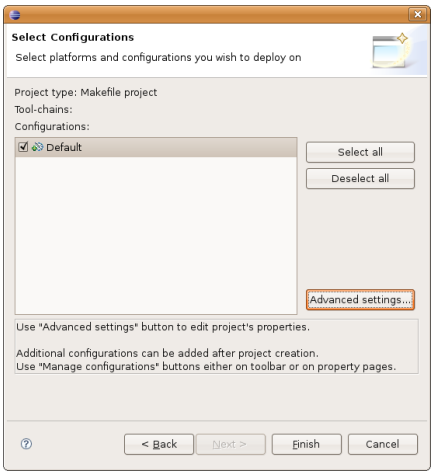
Welcome screens are very nice but not a place to work, so go to *Window* > *Open perspective* > *Other...* and select C/C++ from the list. You will see some changes:



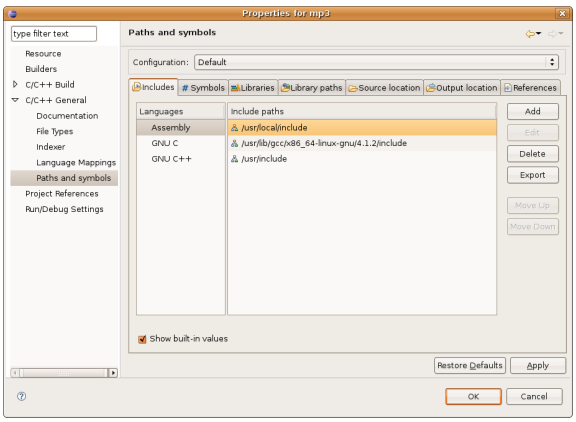
This is the window that you will become familiar with from now on. There are many sub-windows: The *Project Explorer*, where projects and files appear, the centre window where source files are edited, and other little windows that will be explained later if necessary.
We will start by creating a new test project, let us call it '*ledtest*'. So start by going to *File* > *New* > *C Project*. A wizard will be opened:



Write the project name and select *Makefile project / -- Other toolchain --* from the list. Now, click on *Next >* button, and then on *Advanced Settings* from the next wizard step:

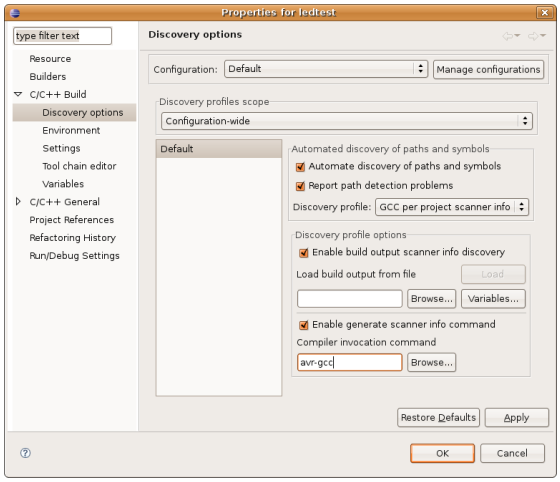


Now we must configure the compiler used in order to make Eclipse find the headers from *avr-libc* automatically. First enter in *C/C++ General / Paths and symbols* and delete the default Include paths from the list (note that they are defined for Assembler, C and C++ so delete all of them from each language):

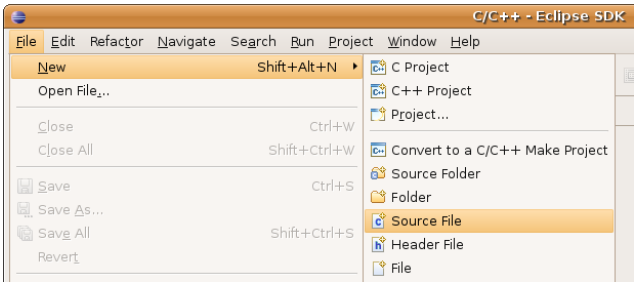


Now enter in the *C/C++ Build / Discovery options* section and change *Compiler invocation*

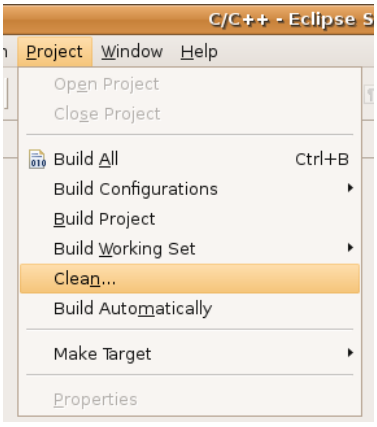
command field to '*avr-gcc*' and apply the settings by clicking on *OK*:



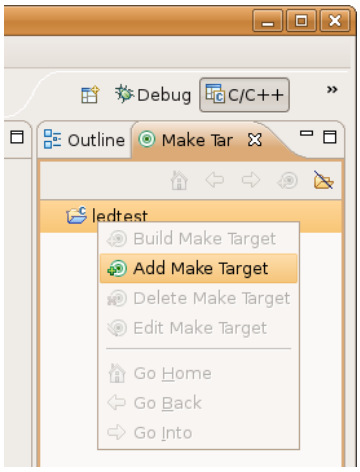
Now headers for AVR should be detected automatically, if not add the correct ones yourself, which are: *avr/include* and *lib/gcc/avr/<gcc-version>/include* (both preceded by toolchain installation path).
Before doing anything more, deselect the '*Build Automatically*' option located at *Project* menu. After that, you will be ready to add new files by going to *File > New > [Source file / Header File / File]*



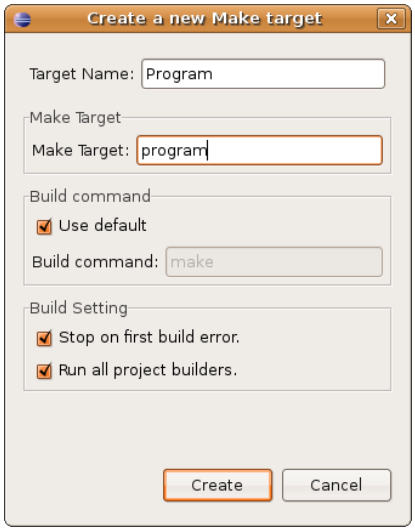
First, start adding a new file called '*Makefile*', and copy the template described in chapter 4.1 (or your own one, of course). I say this because I am going to explain how to compile/program the microcontroller through the same IDE. There are two options: using the *Project* menu items, or creating your targets in *Make Targets* window. The first one works without doing anything, just having the Makefile with standard targets such as *all* and *clean*:



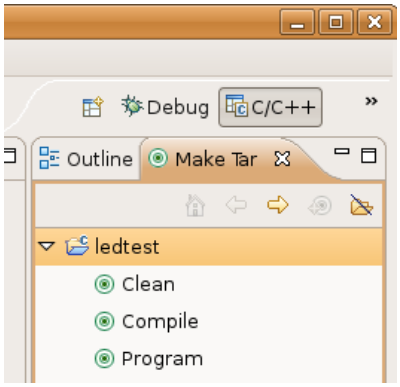
However, targets like *program* are not in the menu, so we will have to use the *Make Targets* window. You can open it by accessing *Window > Show View > Make Targets*. There, you can add your own targets just left-clicking on your project's nameand select *Add Make Target*:



A new window will appear, where you have to fill *Target Name* and *Target Make* fields. For the *program* target, you should put a name such as '*Program*' and the target '*program*':



After adding the most used Make targets, the window should look like this:

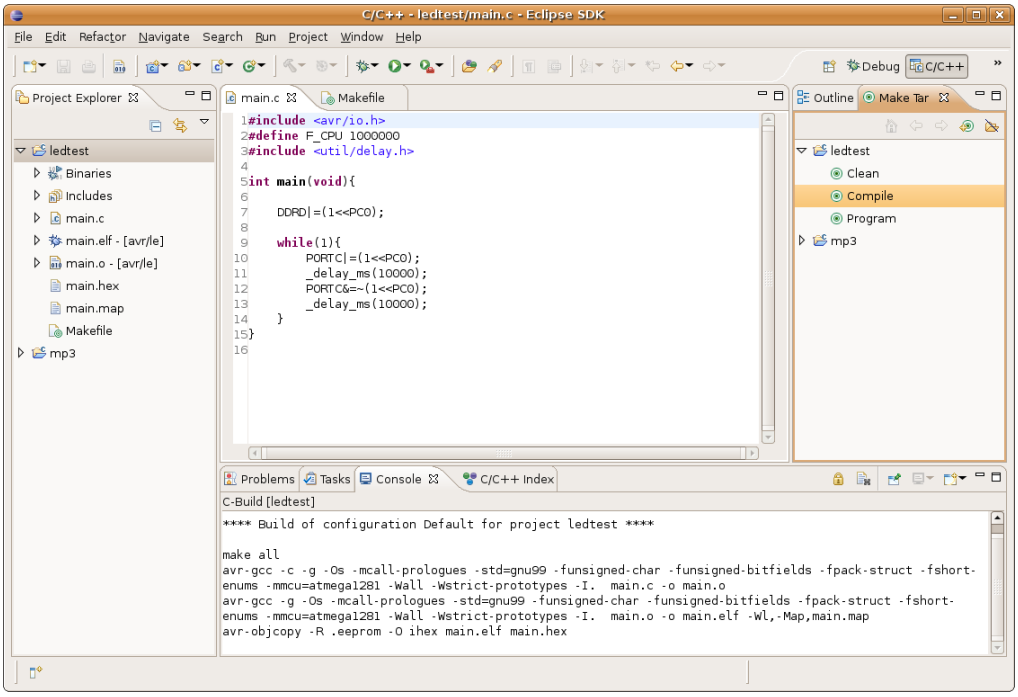


Now, to test that you have had success configuring your environment, add a new source file, and add the following code to test if everything is working properly:

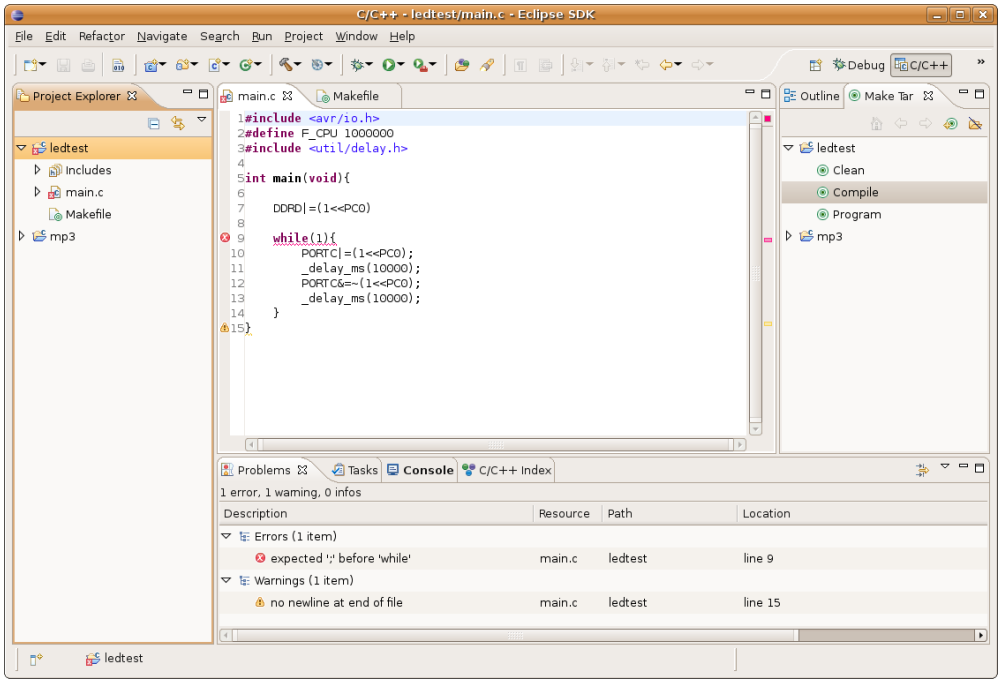
```
#include <avr/io.h>
#define F_CPU 1000000
#include <util/delay.h>
```

```
int main(void){

          DDRD|=(1<<PC0);

          while(1){
                    PORTC|=(1<<PC0);
                    _delay_ms(10000);
                    PORTC&=~(1<<PC0);
                    _delay_ms(10000);
          }
}
```

Add the new source file to the Makefile, and now click on 'Compile'. If everything is OK, you should see the new files created and the output text from the compiler in the Console window:



Note: if you want to hide object files from the *Project Explorer* window, click on the top-right arrow and select *Customize View*. Then select '*Object files*' from the list.
Any errors or warnings in the code will be shown in the *Problems* window, and marked in the corresponding lines:



That is all for this chapter. Eclipse offers many other features: you can learn more about them by accessing their wiki at: http://wiki.eclipse.org/index.php or for C/C++ developers: http://wiki.eclipse.org/index.php/CDT.

6. Debugging with JTAGICEmkII and Eclipse

Debugging is known as the task of finding and solving program errors, in this case with help from another program. I will explain how to debug using Atmel's JTAGICEmkII (introduced in chapter 3.2.1), a debugging hardware through JTAG interface. Emulation using *simulavr* will not be explained, but you may find the chapter on how to configure and use Eclipse useful, since it would be identical for *simulavr*. We will also use two programs in our computer: *Avarice* and *GDB (GNU DeBugger)*. The first one will communicate it with JTAGICEmkII and will open a local server where *GDB* will connect and interact with. *GDB* offers a command line interface, but it is not very comfortable. *Eclipse* offers a graphical interface for debugging which will be very useful. Setting breakpoints or reading/writing variable values will be made with only a few clicks. There is another good alternative for graphical debugging with *GDB* if your computer does not have enough memory to run Eclipse, or you do not like it: it is called *Insight* and is freely downloadable at http://sourceware.org/insight/. However it will not be covered in this document.

6.1. Installation of GDB and Avarice

As I said both programs are necessary for debugging. If you are a Windows user, you can step over the installation process since both programs are included in WinAVR. By the date (July 2007), *Avarice* is in version 2.6, but the CVS version has some important bug fixes and USB management has been completely rewritten, so I will also explain how to compile it from CVS. Note that some patches could be available for *GDB* (refer to chapter 2.2.2.A)

6.1.1. GDB

Before installing *GDB* you must have the *termcap* library installed, freely available at ftp://ftp.gnu.org/gnu/termcap. After that, you will have to download *GDB* from http://ftp.gnu.org/gnu/gdb/ and follow these steps to compile and install it in your toolchain's directory:

```
# tar -zxvf gdb-<version>.tar.gz
# cd gdb-<version>
# mkdir obj-avr
# cd obj-avr
# ../configure --prefix=$TOOLCHAINPATH --target=avr
# make
# sudo make install
```

6.1.2. Avarice

If you want to have the most recent code of *Avarice*, use the CVS installation method that requires a few more steps, but if you want to use the latest stable version follow the second method. Note that the CVS version is not considered stable.

A. **CVS Installation**

B. You must have the CVS package and even Automake >= 1.9 and Autoconf >= 2.59 in order to compile the program. For MacOS users, the following flag must be added when executing configure (included quotation marks):
"LDFLAGS=-L/usr/local/i386-apple-darwinX.X.X/avr/lib -lbfd" "CPPFLAGS=-I/usr/local/i386-apple-darwinX.X.X/avr/include"

```
# cvs -z8 -d
:pserver:anonymous@avarice.cvs.sourceforge.net:/cvsroot/avari
ce checkout avarice
# cd avarice
 # ./Bootstrap
# ./configure --prefix=$TOOLCHAINPATH
# make
# sudo make install
```

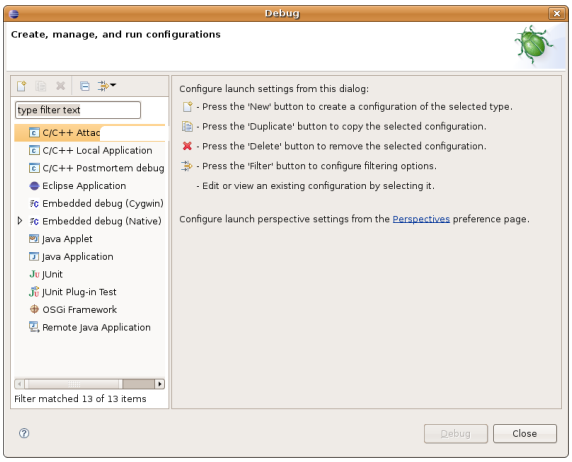
C. **Latest stable release**

The latest stable release can be found in their SF.net project page:
http://sourceforge.net/projects/avarice/
The installation is done as always, but MacOS users must add the following flag when executing configure (included quotation marks):
"LDFLAGS=-L/usr/local/i386-apple-darwinX.X.X/avr/lib -lbfd" "CPPFLAGS=-I/usr/local/i386-apple-darwinX.X.X/avr/include":

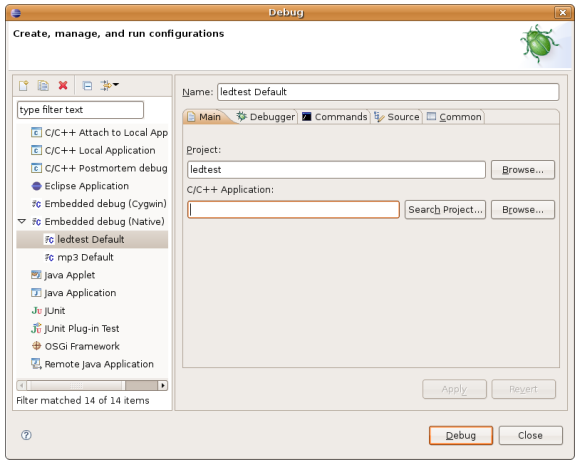
```
# ./configure --prefix=$TOOLCHAINPATH
# make
# sudo make install
```

6.2. Getting Eclipse ready to debug

A few things have to be configured in *Eclipse* in order to start debugging. Just follow the instructions described in the next lines.
First, go to *Run > Open Debug Dialog*. A new dialog like this one will be opened:



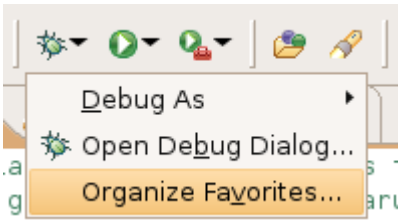
Then, left-click the *Embedded debug (Native)* item on the list and select *New*. A new debug profile will be created and a section with some fields to fill will be opened:



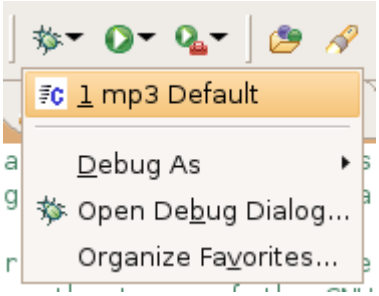
Fill the fields with these values:

Tab	Fields content
Main	C/C++ Application: *main.elf*
Debugger	GDB debugger: *avr-gdb* GDB command file: Clear this field and leave it blank
Commands	Initialize commands: *target remote [host]:[port]*

Note: [host]:[port] must be replaced by the host where Avarice is running (not necessarily localhost) and the port where it is listening to. An example would be: localhost:6423 (this configuration will be used in the document as default). Now click on Apply button and close the dialog.
Finally, we will add a shortcut to our new debugging profile by clicking on the Debug button in the toolbar and select *Organize Favorites*:



Then click on *Add* and select your debugging profile from the list. Now, the Debug menu should appear with the new profile:



That is all for *Eclipse*. Read the next chapter to learn how to start debugging!

6.3. Starting to Debug

Before starting to debug, you need to know something more, which is how *Avarice* works, and how to enable debugging information in your ELF file. And of course, you will need to have your JTAGICEmkII connected as described in chapter 3.2.1. and the real circuit where your program will run.

6.3.1. Adding debug information to an ELF file

We need to add something to our ELF file to make it ready for debugging. It is very easy: we need to indicate it to *GCC* by adding a new flag. So, modify your Makefile adding this line below the first CFLAGS entry (already in the template given in chapter 4.1):

```
CFLAGS+=gdwarf-2
```

6.3.2. Working with Avarice

First I will explain the *Avarice* flags that we and then how we use it to debug together with *GDB* and *Eclipse*. Note that some flags have a shorter equivalent, also listed in the table.

Flag	Description
--mkII / -2 --mkI / -1 --dragon / -g	Debug devices available: JTAGICEmkI/II and AVRDragon. One of them has to be indicated
--jtag / -j	Port attached to the JTAG device. If it is through serial use device's path, ex: */dev/ttyS0*. If you use USB, just write '*usb*', and if more than one device is connected, write '*usb*' followed by a colon and the device serial number you want to use (ex. 'usb:xxx').
--erase / -e	Erase the target
--program / -p	Program the device. *--file / -f flag* required indicating the corresponding binary file.
--file / -f	File used for the *--program / --verify* flags
[host]:[port]	Specify the host and port to listen to for a *GDB* connection. Ex: *localhost:6423*

Let us see how we would use Avarice. I will assume that we are using JTAGICEmkII connected through USB. If you want to start debugging with a version of the program that is newer than the one already inside of the microcontroller, you will have to run avarice with the programming options. The command would be as shown below (both, long and short versions are indicated):

```
# avarice --mkII --jtag usb --erase --program --file
/your/project/path/main.elf localhost:6423
# avarice -2 -j usb -e -p -f /your/project/path/main.elf
localhost:6423
```

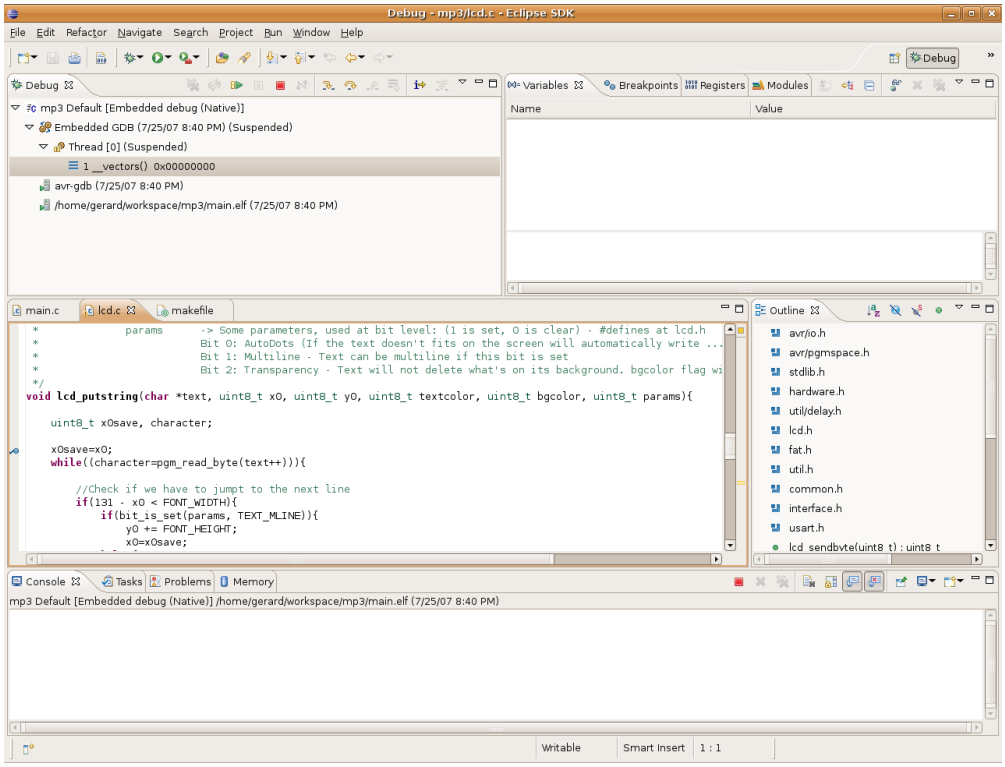
However, sometimes you will start a new debugging session with the program that is already uploaded, so in that case, you should run *Avarice* without program options:

```
# avarice --mkII --jtag usb localhost:6423
# avarive -2 -j usb localhost:6423
```

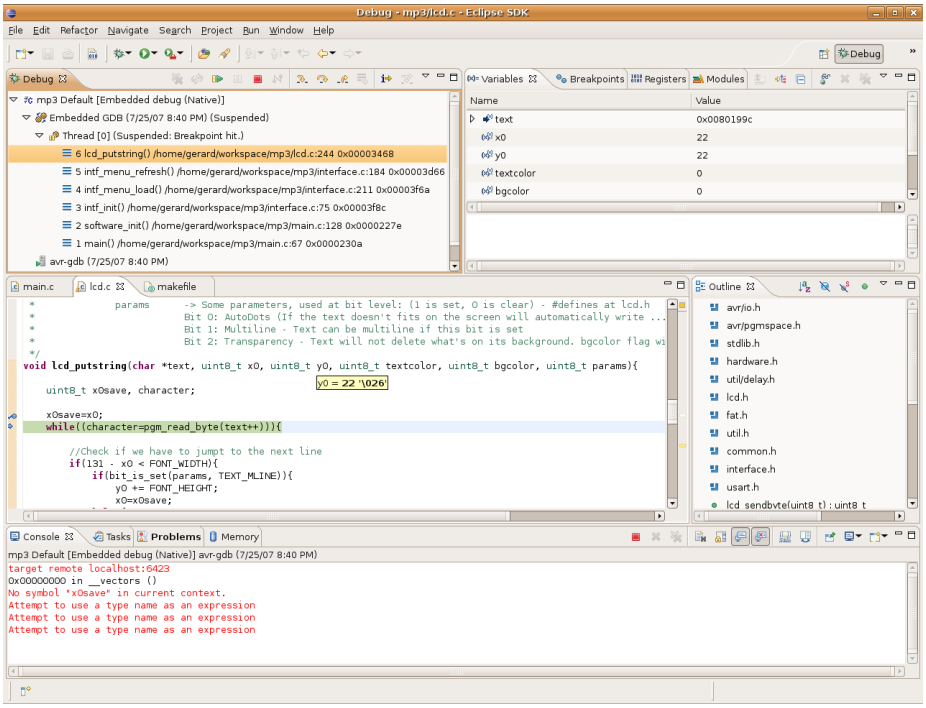
You will notice that after running *Avarice* it will stay waiting for a GDB connection by prompting *Waiting for connection on port 6423* on the screen. Now it is Eclipse turn.

6.3.3. Debugging with Eclipse

Before starting a new debugging session, you should put a breakpoint in any part of the program, for example in the middle of a function, that will let us test the variables features. How to add breakpoints is described in A section of this chapter. After that, run *Avarice* in your shell as described in the previous chapter and leave it waiting for a new connection. Now click on the Debug button in the toolbar, and select your debugging profile. If it is the first time you have run it, a new dialog will ask you to change to Debug perspective. Say 'Yes' and remember the decision to avoid future prompts. Now Eclipse should look like this:



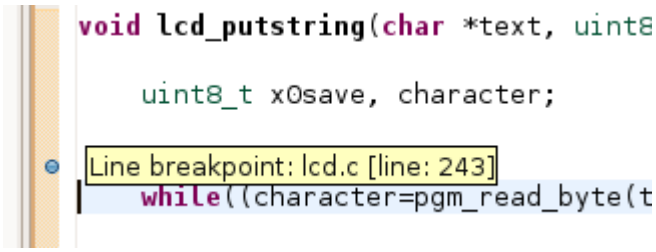
As you can see, execution is suspended. To continue with execution, just press the *Resume* button on the *Debug* window. Now, you will see that the program will start executing, and will stop at the first breakpoint it encounters. Then, the program is suspended again to let you do debugging tasks:



Next I will explain the basics of debugging: breakpoints, read/writing variables, viewing register values, etc. More features are available, but you will have to learn them yourself!

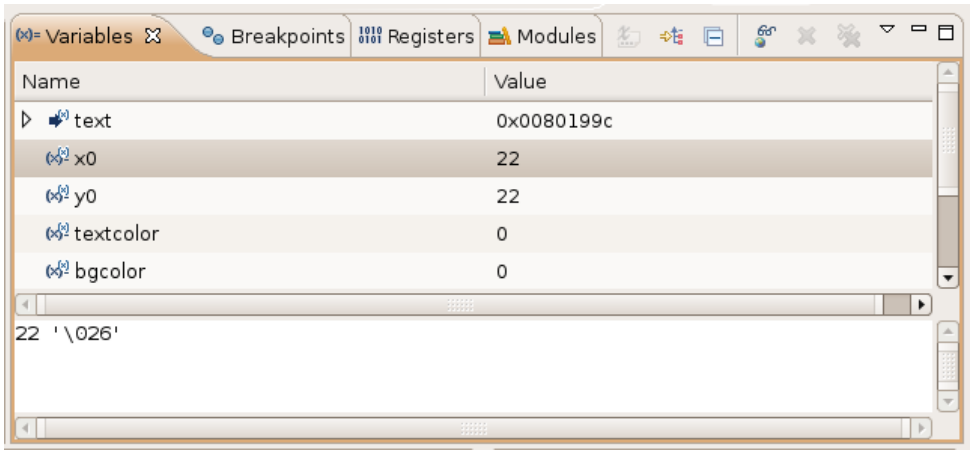
A. Breakpoints

Breakpoints are interruptions of execution at a specific point in the program. When they are matched by the program running, the debugger stops the execution and lets you check variable values and many other useful things for a programmer. In Eclipse setting up a breakpoint is very easy: double-click on the left of a line and a blue spot will appear. This means that the program will be interrupted at this line. To remove it, double-click again over it.

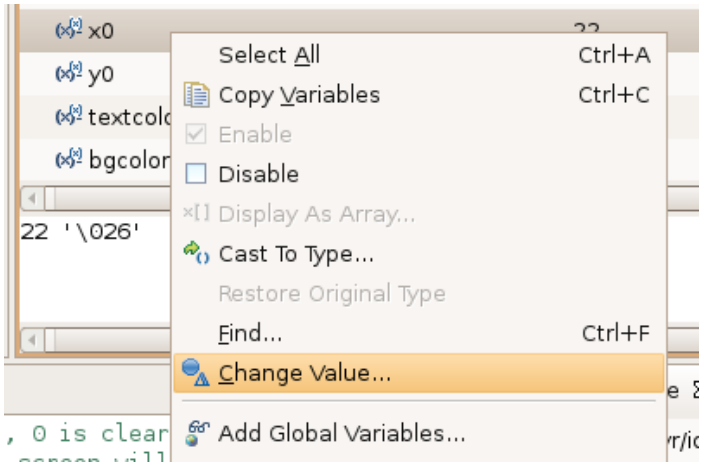


B. Working with variables

Another useful thing is to read, write or cast variables. When you are inside a function and the execution is suspended, you will see that in the Variables window a list of the current local variables with their value is shown:



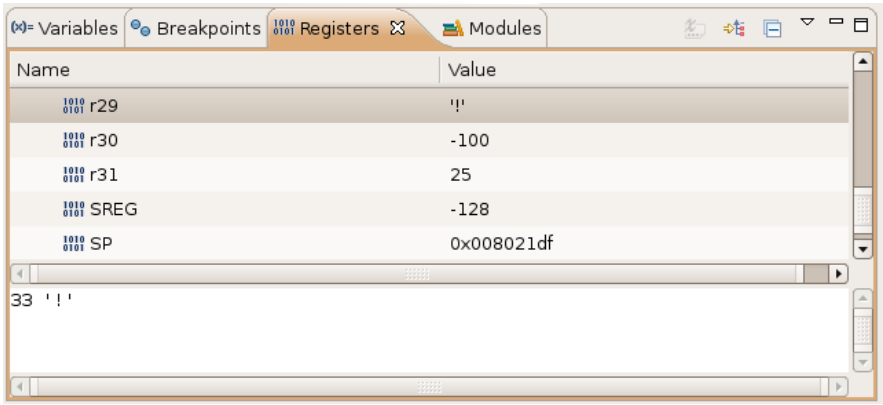
To modify the content of a variable, right-click over it and select *Change Value...*:



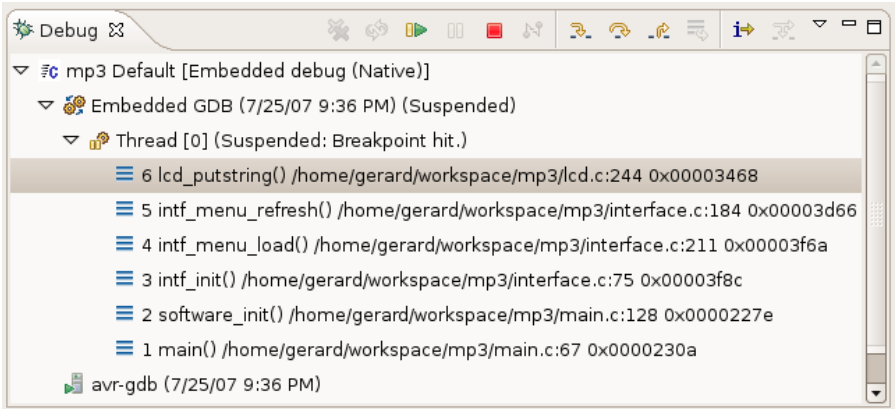
And then enter the new value for that variable. If you want to cast it, just click on *Cast To Type...* and write the new type.
As you can see, only local function variables are shown. However, global variables can be added to the list by right-click and select *Add Global Variables...*. A full list of the available global variables to be added will be shown.

C. Others

Other interesting things are the ability to see the current value of the microcontroller registers in the *Registers* tab:



Or watch the list of sub-sequent function calls:



As you can see in the picture, *lcd_putstring()* has been called by *intf_menu_refresh(),* this one by *intf_menu_load()* and so on.
Of course, debugging offers more possibilities, I have just explained the basic ones. You will have to discover the others!

7. Bibliography

http://electrons.psychogenic.com/modules/arms/art/6/SimulatingandDebuggingAVRprograms.php

http://lists.gnu.org/archive/html/avr-gcc-list/

http://winavr.sourceforge.net

http://www.atmel.com/avr

http://www.avrfreaks.net/wiki

http://www.eng.hawaii.edu/Tutor/Make/

http://www.nongnu.org/avr-libc/user-manual/install_tools.html

http://www.nongnu.org/avr-libc/user-manual/overview.html

http://www.reactivated.net/writing_udev_rules.html

http://www.tuxgraphics.org/electronics/200411/article352.shtml

http://www.yagarto.de/howto/yagarto2/index.html

Special greetings to AVR-GCC List members who are always helping people with their problems!

Thanks to *Ronnald Hedderwick* for gramatical corrections.