Università degli Studi di Catania

Using Open Source Tools for STR7xx Cross Development

Authors:

Giacomo Antonino Fazio - Antonio Daniele Nasca

November 2007

Read carefully

Information furnished is believed to be accurate and reliable. However, we assume no responsibility for the consequences of use of such information nor for any infringement of patents or other rights of third parties which may result from its use. Specifications mentioned in this publication are subject to change without notice.

Table of contents

Chapter 1 – Introduction and target of the work

The idea from which our work had origin was to develop, run and debug embedded software on a class of ST microcontrollers (STR710, STR730, STR750, based on the core ARM7TDMI), using only open-source solutions, and after that to write a tutorial on how to use those solutions.
Nowadays there are many proprietary IDEs (IAR, RealView, and so on) which allow building and debugging of produced code on microcontrollers easily and quickly, but the licences for those IDEs are very expensive. Moreover, the interfaces used by those IDEs for connecting the target are usually proprietary and expensive.
Even if this problem isn't so important for industries where there's enough money, for individuals and universities the price to pay can be very high.
So using free solutions that allow to compile and download code on many microcontrollers using cheap interfaces can be the best solution for that problem. Moreover, if you consider that all those solutions are open-source, you can understand how these programs can be improved and could have in the future features that today aren’t present or aren't working properly.
The downside of these solutions is that sometimes they are still a less mature and supported environment than proprietary solutions, although surely in expansion; so their installation and configuration is a fundamental (and sometimes not so easy) step if you want them to work.
From all those aspects derived the idea to explore current solutions to find the best one, to help the development of specifications and missing parts and finally to share the gained knowledge through a tutorial, similar to that for the family Atmel AT91SAM7S written by James P. Lynch (the title of this document clearly recalls it).
Finally, considering the growing diffusion of the operating system GNU/Linux, we decided to make our tests both on Windows and Linux. So, our work was done following some steps:

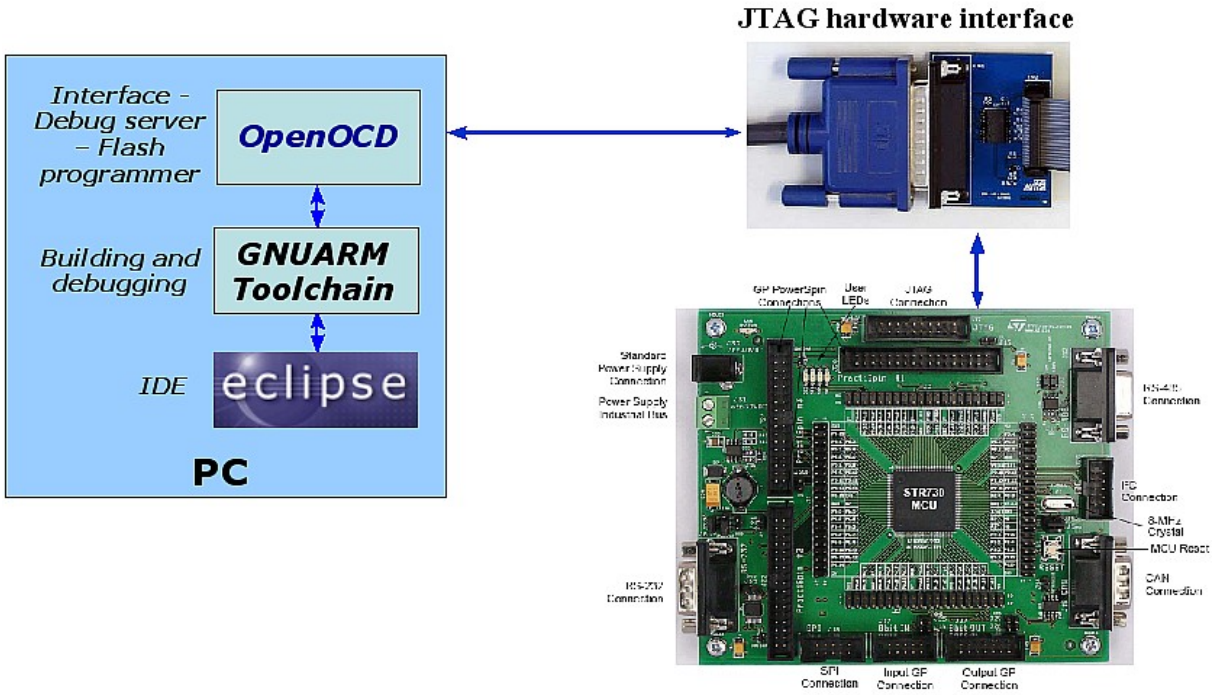
1. exploration and test of open-source solutions, both for Windows and Linux.
2. development and test of the missing or not working parts, through the help provided by datasheets and reference manuals.
3. writing of this document, that is on the one hand a relation on the job done, on the other a detailed tutorial, easily (we hope) readable also by a newbie.
4. english translation: we wrote the first version of this document in Italian, that is our native language, but since this work seems to have success also abroad and many people had asked for an English translation, here you have it and we apologize if our English is not that good (thanks to Eng. Davide Lombardo for his help in translation).

However please remember that this work has to be considered as a preliminary version because open-source tools are improving themselves more and more, but there are features we implemented that could be not entirely correct and other features still have to be implemented, although main features are present and seem to work correctly.
Moreover **this solution can be used with other microcontrollers, too**: obviously more tests are required to make it work properly with them.
To show you (and us) this is true, and also because we liked what we did, we created the configuration files and the template for the STR72x microcontroller (we didn’t test it) and we decided to start the tests with the **STR9** family, using a board with the STR912 microcontroller: the results are good (although other work is needed) and we summarized them in the appendix C.
Now let's describe the main parts of this document: after this chapter, dedicated to introduce the work, the programs and the hardware we used, chapter 2 will deal with how to install the tools in both Windows and Linux; chapter 3 will analyse the configuration files, the modifications we did

and why we did them.
At the end there are the appendixes, the first one is an Insight and Eclipse tutorial, a bit more advanced than the one present in chapter 2; appendix B studies GCC extensions and the *Thumb mode* and appendix C describes our experience with the STR912 board.

1.1 Block Diagram

In the following picture you can see the block diagram of the implemented solution to provide a global overview which can be useful for you while reading the document.



If you consider only the hardware, you can see the scheme so: the user uses the PC that is connected to the board through the JTAG hardware interface, which converts the signals coming from the PC to JTAG signals to be sent to the board.
If you add software to that vision, the scheme becomes more complex: the user uses Eclipse to write the code, to compile it and to make the ARM executable (it uses GCC to do that), which has to be sent to the board: to do that OpenOCD is used, whose task is, as we will specify better later, to act as interface to the board, so you can see it as the access door to the board, from which all the communications PC/board (and viceversa) have to pass.
For debugging, Eclipse uses GDB (GNU Debugger), which sends the commands to OpenOCD, which acts also as GDB server.
So every command that has to be sent to the board, must pass from OpenOCD, that has to convert it to a JTAG command. However, this JTAG command is a parallel/serial/USB port signal, not suitable to the board, so it's necessary to use something, between OpenOCD and the board, which converts PC signals to JTAG signals for the board: that's the hardware interface job. To send commands back to PC, for example the debug answer, the reverse procedure is used. We will see better OpenOCD function later.

1.2 Necessary Hardware

So the necessary hardware is a board mounting one of the microcontrollers here analyzed, a JTAG cable, a JTAG hardware interface and obviously a PC. Now we deal with each of these components, analysing the various alternatives.

1.2.1 Board

There are many boards that have one of the analysed microcontrollers and that can be used for our work. Since the microcontroller can handle several peripherals, each board has some of them, like LEDs, LCD screen, CAN, UART, etc. Obviously the higher is the number of I/O devices present on the board, the more expensive the board will be.
The first choice concerns the selection of the microcontroller to use, because depending on the microcontroller, you can choose the board.
Here we show a box that summarizes the features of each microcontroller:

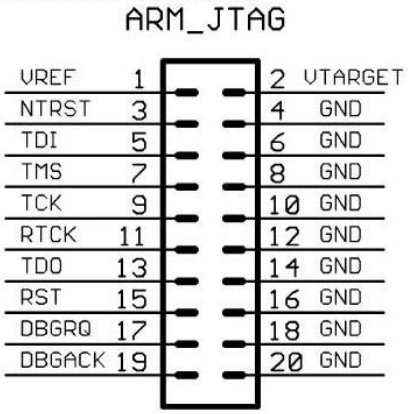
Comparative table of microcontroller features

	STR71x	STR73x	STR75x
Core	- ARM7TDMI 32 bit RISC - 59 MIPS@66 MHz for SRAM - 45 MIPS@50 MHz for Flash	- ARM7TDMI 32 bit RISC - 32 MIPS@36 MHz	- ARM7TDMI-S 32 bit RISC - 54 DMIPS@60 MHz
Memory	- 256 Kb program. flash - 16 Kb data flash - 64 Kb RAM - EMI (External Memory Interfaces): 4 banks	- 256 Kb program. flash - 16 Kb RAM	- Till 256 Kb program. flash - 16 Kb Read-While-Write flash - 16 Kb high speed RAM
Communication Interfaces	2 I2C 4 UART Smart Card on UART1 2 BSPI 1 CAN 1 USB 1 HDLC	2 I2C 4 UART 3 BSPI Till 3 CAN	1 I2C 3 HiSpeed UART 2 SSP till 12 Mbit/s 1 CAN 1 USB

1.2.2 JTAG Interface

The connection to an ARM target requires the use of the JTAG protocol, whose port is clearly visible on each board. Let's see more about this protocol:
JTAG standard
JTAG, acronym of *Joint Test Action Group*, is a consortium of 200 companies that produces integrated circuits and PCBs, born to define a standard protocol for the functional test of the devices. Among the main members there were IBM, AT&T, Siemens, DEC, Nixdorf, Texas Instruments and Philips.
This consortium was active from 1985 to 1990 and it gave life to the future IEEE 1149.1 standard, known colloquially as JTAG standard from the name of the promoter consortium.
JTAG is nowadays primarily used for accessing sub-blocks of integrated circuits, and is also useful as a mechanism for debugging embedded systems.
Here you have the layout of the JTAG connector with a schematic description.

JTAG connector layout:



Now let's explain the meaning of each connector pin:

- **VREF**: Voltage Reference, used to indicate the target's operating voltage to the debug tool.
- **VTARGET**: Voltage Target. It may be used to supply power to the debug tool.
- **NTRST**: JTAG TAP reset. This signal should be pulled up to Vcc in target board .
- **Ground**.
- **TDI**: JTAG serial data in. This signal should be pulled up to Vcc on target board.
- **TMS**: JTAG TAP Mode Select. This signal for level of voltage should be pulled up to Vcc on target board.
- **TCK**: JTAG clock.
- **RTCK**: JTAG retimed clock. Implemented on certain ASIC ARM implementations, the host ASIC may need to synchronize externals inputs (such as JTAG inputs) with its own internal clock.
- **TDO**: JTAG serial data out.
- **RST**: Target system reset.
- **DBGRQ**: Asynchronous debug request. This signal allows an external signal to force the ARM core into debug mode. It should be put down to ground.
- **DBGACK**: Debug acknowledge. The ARM core acknowledges debug-mode in response to a previous DBGRQ input.

However PC doesn't have a JTAG port, but it usually has parallel, serial and USB ports then, as we have said at first, it's necessary to use the JTAG hardware interface which converts the signals sent from the PC through those ports in JTAG signals to send to the JTAG port on the board.
There are many types of JTAG hardware interface, sold by several producers. Now let's see the most famous ones.

JTAGKey and JTAGKey Tiny

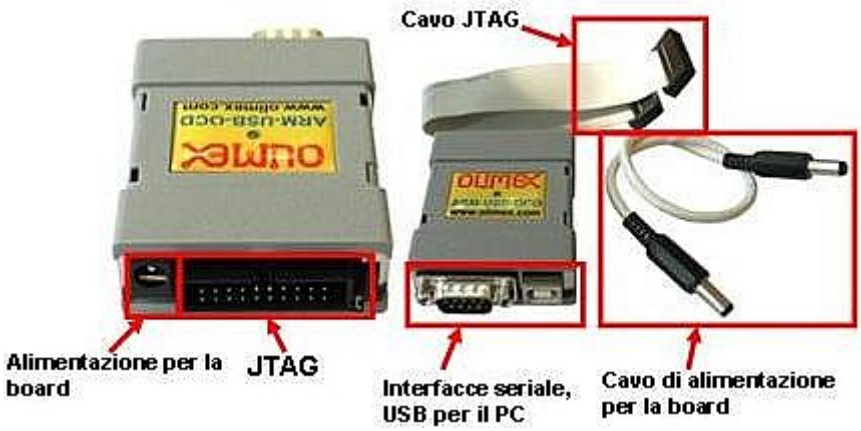
They are two types of JTAG interfaces that connect to the PC through the USB port.
They are both produced from Amontec and the differences between them are the price (about 100 euros the first, 30 euros the second) and the features they offer.
For more details visit www.amontec.com

JTAGKey

JTAGKey Tiny

ARM-USB-OCD e ARM-USB-OCD Tiny

Another solution comes from Olimex that offers two alternatives, different for the price, called respectively *ARM-USB-OCD* and *ARM-USB-OCD Tiny*. The connection to PC can be through USB or serial port.



ARM-USB-OCD

ARM-USB-OCD Tiny

Wiggler

Macraigor Wiggler is a simple device which connects the parallel port of the PC to an OCD (On-Chip Debug) port, in our case the JTAG port on the board. It's very simple as circuit, uses the parallel port and supports clock frequencies of the JTAG interface between 60 and 380 kHz. PC simulates the target interface switching on-off signals, using a technique called *bitbanging*. This technique provides a signal level translation between PC (5V TTL) and the target (1.5V-5V).
Here you have a list of target microprocessors supported by Wiggler.

Supported Processors

AMCC:	PPC440
AMD:	SC520, Athlon, Duron, AU1000, AU1100, AU15x0
ARM:	7TDMI, 710T, 720T, 740T, 9TDMI, 920T, 922T, 940T, 946T, 1136EJ-S
Broadcom:	BCM1250, BCM7115
Freescale:	MC9328MX1, 56300, 56600, 56800, StarCore, 683xx,MPC603e, MPC8xx, MPC5xx, MPC5554, MPC740,MPC745, MPC750,MPC755, MPC8240, MPC8245, MPC8247/8248, MPC8250, MPC8255, MPC8260, MPC8264/65/66, MPC8270/71/72, MPC8280, MPC8540, MPC8560, MPC8541, MPC8555
IBM:	PPC603e, PPC740, PPC750
IDT:	RC323xx
Intel XScale® Technology:	PXA21x, PXA25x, PXA26x, PXA27x, IOP3xx, IXC1100, IXP42x, IXP46x, IXP24xx, IXP28xx, 80200, 80219, 8032x, 8033x
MIPS:	4Kc/p/m/e, 5Kc
NEC:	VR5432, VR5500
Net Silicon:	NetARM+10, NetARM+40, NetARM+50, NS7520, NS9750
PLX:	IOP480
Phillips:	PR1900
Toshiba:	TX49
Triscend:	E5, A7

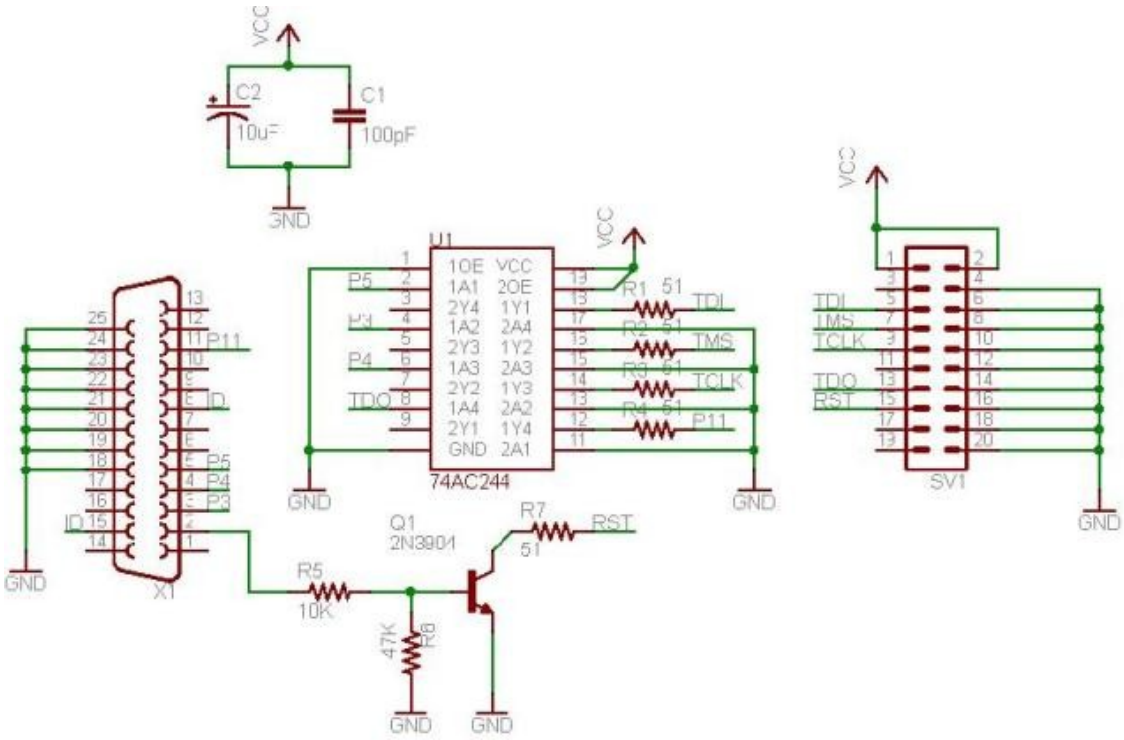
The following picture shows the original Wiggler:

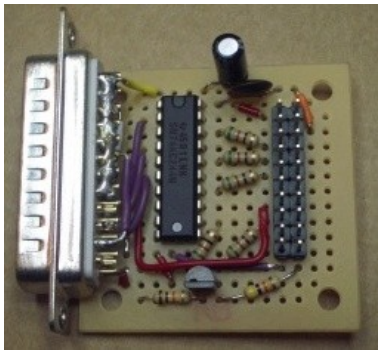
Wiggler

However there are also many Wiggler clones made by other producers, such as Olimex, which did the following dongle:

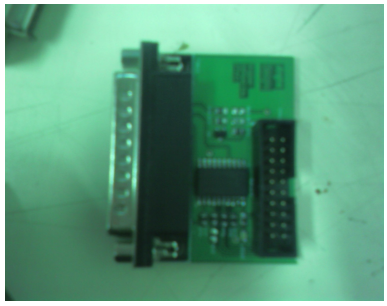
Olimex Wiggler compatible

In Internet there are also many schematics for Wiggler clones, they are freely downloadable and can be modified for special requirements, for example adding or deleting signals that are optional in JTAG standard, but required by some targets.
The following pictures show an example of Wiggler clone schematic and a photo of it.





As you can see from the picture, it isn't a very complex circuit. The following is instead the Wiggler clone used by us:



Below you can see a comparative box of the various solutions:

Vendor	**Price**	**Com Port**	**Necessary Software**	**Comments**
Olimex ARM JTAG	19.95 $	Printer Port	OpenOCD	Programmer called Wiggler, slow code download
Olimex ARM-USB-OCD	69.95 $	USB	OpenOCD	Serial port extra and power to 5V
Olimex ARM-USB-Tiny	49.95 $	USB	OpenOCD	Version for students
Amontec JTAGKey	131.78 $	USB	OpenOCD	It has an extra ESD protection
Amontec JTAGKey-Tiny	38.60 $	USB	OpenOCD	Version for students

1.3 Hardware used in our project

The hardware we used is all the necessary to make our tests: three boards (one for each analysed microcontroller), a JTAG cable, a JTAG parallel interface (Wiggler compatible), a parallel cable and obviously, a PC with a parallel port.
We chose Wiggler because it's the cheapest interface (although the performances are lower than

almost every alternative solution), so the tutorial considers only Wiggler and not the other interfaces, but looking at either Yagarto's webpage (www.yagarto.de), or producers sites, or simply by searching on Internet, you will certainly find lots of documents about them and their use.
If you decide to use Wiggler but you don't have the parallel port on your PC, there are USB/parallel or serial/parallel adapters that can solve the problem.
Now let's see more closely the components, starting from the board:

ST STR710 Evaluation Board

The STR710 Evaluation Board is a complete development platform containing a microcontroller STR710FZ2T6; it is a flexible board, available at a low price, suitable to show the features of these microcontrollers with the I/O devices.

Main features:

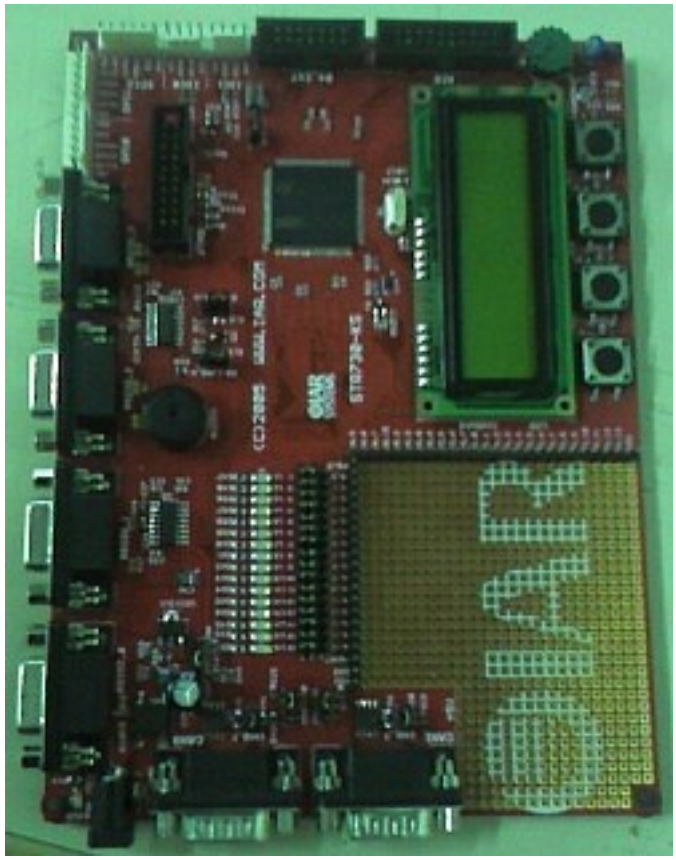
- EMI SRAM 4 MB (2Mx16)
- EMI FLASH 4 MB (2Mx16)
- SPI Serial FLASH
- I2C EEPROM
- LCD Display (2x16)
- Support for USB, CAN, RS232
- LED displays
- Buzzer
- Test buttons
- JTAG connector

IAR STR730-KS

This development board is based on the STR730 microcontroller and is sold from IAR together with its IDE and JTAG interface, as shown in the following picture:

For our purposes, obviously, we used only the evaluation board, that has the following main features:

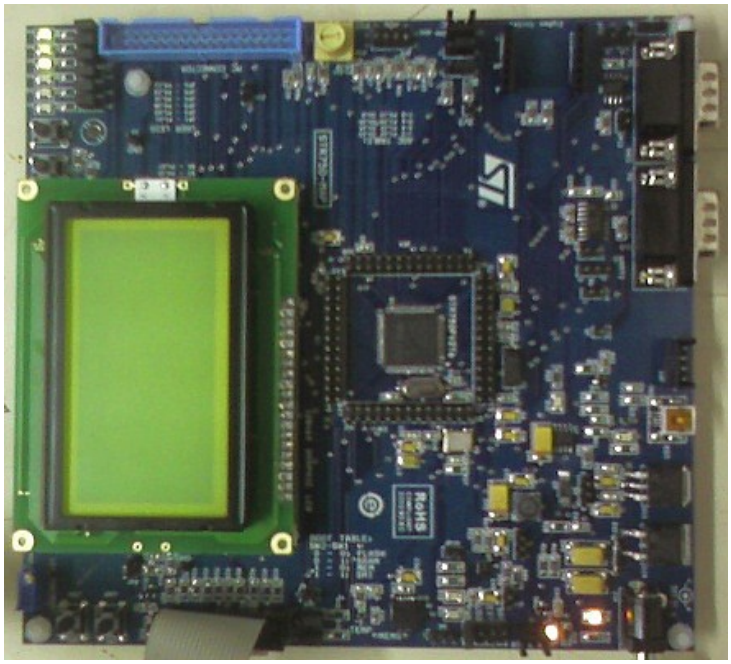
- 20-pin JTAG interface connector
- Power-on LED
- RESET button
- RS232 drivers
- Four UART DB9 connectors
- Two CAN ports
- Two I2C ports
- Three BSPI ports
- PWM
- 16 user LEDs
- Four push buttons
- LCD: 16x2
- Potentiometer connected to ADC
- Powered via 400 mA through USB port via IAR J-Link



STR750-MAP (Multi Application Platform)

The I/O devices available for this board are:

- UART
- I2C
- CAN
- LCD
- LEDs
- Connector for Zigbee
- Connector for engine
- Accelerometer
- Temperature sensor



1.4 Software used in our project

Now let's talk about the software we used, omitting the installation and configuration phases, phases, which will be described later.

OpenOCD

OpenOCD is an open-source program developed from Dominic Rath for his diploma thesis. As we already said before, it works both as TTY interface (via Telnet) for the connection to a target and as GDB server. Let's try to clarify both these functionalities.

About the connection to an ARM target, OpenOCD does it by means of an intermediate component which converts the signals sent from OpenOCD through the USB/parallel/serial port to JTAG signals that have to be sent to the JTAG interface of the target. This component is the JTAG hardware interface with which we have already dealt.

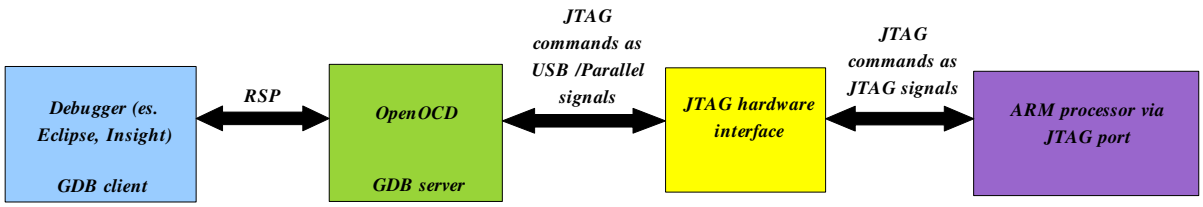
About the function of GDB Server, first of all it's necessary to explain how GDB works.

GDB (GNU Debugger) is the standard debugger of the GNU software system and allows the user to trace the program execution by allowing him to set breakpoints to block code execution at any moment; moreover GDB allows to monitor each variable and register values, that are easy to view and modify.

Another feature present in GDB is the so-called "remote debugging", which allows to monitor from a PC (GDB client) a program execution that is on another PC (GDB server); the communication between client and server is based on a message-passing protocol called RSP (Remote Serial Protocol) that uses a connection, for example TCP: in this case the GDB server listens on a TCP port, waiting for incoming commands from the GDB client. It should be clear at this point the way used for debugging embedded devices: the application is executed on the embedded device and controlled from the PC that works as GDB client. But, to do that, it's necessary a GDB server for the embedded device, and here the circle closes: OpenOCD works also as GDB server.

These two features are obviously tied, since when OpenOCD works as GDB server receiving debugging commands from a GDB client, then has to send those commands to the ARM processor (through the JTAG port on the board) and here it uses its feature of interface with an ARM target.

We show now an application to clarify and summarize what we have just said, supposing we want to debug a program on our embedded device and we also want to see a certain variable value: the debugger (GDB client) sends the request through RSP to OpenOCD (GDB server) which, as we said, also works as interface to the device, so it converts the instruction to commands of JTAG protocol and sends them through the normal USB/parallel ports to the JTAG hardware interface; this one converts the received signals to 3.3 V JTAG signals to be sent to the port on the board; the requested value is read and sent back following the reverse path. The following picture shows what we have just said:



About the ARM processor, ARM7 and ARM9 versions are provided with a special Embedded-ICE unit, that is a hardware circuit implementing on the chip the main debug operations, by means of two internal breakpoint/watching circuits: that's why if your application works on FLASH, you can debug it, read the memory and use two (and not more) breakpoints, without using expensive (and maybe with limited performances) solutions.

OpenOCD now supports Wiggler clones, the FTDI FT2232 based on JTAG interface, the Amontec JTAG Accelerator and the Gateworks GW1602.
It allows debugging on cores based on ARM7 (ARM7TDMI and ARM720t), ARM9 (ARM920t, ARM922t, ARM926ej-s, ARM966ej-s), XScale (PXA25x, IXP42x) and Cortex-M3 (Luminary Stellaris LM3).
Writing on FLASH is supported for external memories compatible with the CFI (Common Flash Interface) created from Intel and AMD and also for several internal memories (LPC2000, AT91SAM7, STR7x, STR9x, LM3).

GNU ARM toolchain
It contains some GNU packages that can be useful to build and debug programs. These packages are:

- **binutils**: collection of tools useful for the management of linker files, object files, libraries and archives.
- **newlib**: C library created to be used on embedded devices; it's a collection of some open-source libraries that can be easily used on embedded devices.
- **GCC compiler**: acronym of **G**nu **C**ompiler **C**ollection (we think a presentation of it is useless!)
- **Insight Debugger**: Insight is a version of GDB (**G**nu **D**e**B**ugger) which uses Tcl/Tk to implement a full GUI with buttons, scrollbars and textboxes.

Eclipse + Zylin Embedded CDT plugin
Eclipse was born as an answer of IBM to Microsoft's Visual Studio, but later IBM donated the whole project to the open-source community.
Eclipse can be used to produce software of various kinds in many programming languages.
It was created as IDE for Java (JDT, "Java Development Tools"), but there are plugins for many other languages: for example, C/C++ (through the CDT plugin, "C/C++ Development Tools"), PHP and Python, and the list is growing more and more. Eclipse is suitable for many activities, from XML management to graphic design of a GUI for a JAVA application (Eclipse VE, "Visual Editor"), so Eclipse can be considered a RAD ("Rapid Application Development") environment.
Eclipse doesn't work on platforms like Windows ME and Windows 98 and, since it's created partially in JAVA, **it's necessary to have the JVM (Java Virtual Machine)**, so make sure you have it before installing Eclipse.
Our purpose is to use Eclipse to develop firmware for embedded devices; since many microcontrollers can be programmed using C/C++ language, the suitable plugin for this purpose is CDT. Initially it didn't work properly with debuggers that work on embedded devices, but the problem was solved from the Norwegian company Zylin that, together with the CDT team, rewrote the plugin to make it able to debug on embedded devices both in idle and in step-by-step modes (the Zylin Embedded CDT plugin was created).

Java
To make Eclipse work properly, it's necessary to have at least the version 1.4.2 of the JVM (*Java Virtual Machine*), but our advice is to have the latest version, that is the version 1.6.0.
For people that don’t know Java, it’s necessary to do a short introduction on Java acronyms.

The Java platform is divided into different solutions oriented to different users; the most important are J2SE (Java 2 Standard Edition is the standard version thought for a normal desktop user and server), J2EE (this solution was thought for enterprises) and J2ME (this solution has thought for mobile devices).
We are interested in J2SE, but inside it there are some packages thought for the different interests of users : the main ones are the JRE (Java Runtime Environment, called also JVM, created for people who only want to execute Java programs) and JDK (Java Development Kit, also called J2SE SDK, created for people who want both to execute and to develop Java programs).

For our purpose the JRE is enough, since we have to only execute Eclipse, but maybe you need in the future to develop something in Java, so you should consider the possibility to download the JDK. We will see how to install it in the chapter 2.

Chapter 2 – Tutorial

In this chapter we will explain the necessary steps to install and configure properly all the required tools. The installation might seem to be a bit difficult, but we will explain it in details, so it will be very simple to use all the tools properly. We will divide the tutorial in two parts, each one will cover all the needed steps for one of the two most used operating systems: on the one hand Windows (simple but less customizable than Linux and surely more affected by crashes), on the other Linux (surely more complex than Windows, but more customizable, we think it gives the user more satisfactions in using what we are going to present!). You will see that some concepts are repeated more than once, this is done in order to make you take them always in account, since they are very important.

2.1 Tutorial for Windows users

Copying the files

You have to copy the two folders *ARMProjects* and *openocd-configs* included in the folder *Code\Windows* and paste them where you prefer, we put them in *C:*

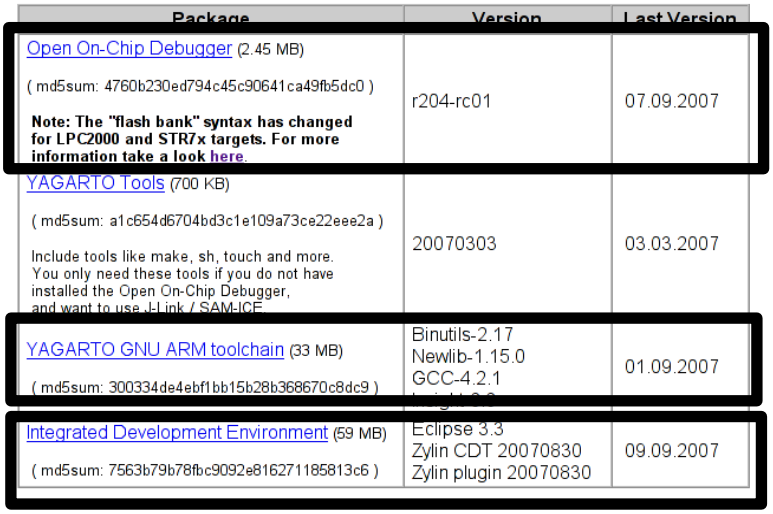
2.1.1 OpenOCD, GNUARM and Eclipse => YAGARTO

YAGARTO stands for *Yet Another Gnu ARm TOolchain* and is a relatively new GNU ARM toolchain created by Michael Fischer.

From this point of view, YAGARTO doesn't propose anything new if compared with existing toolchains, but the real innovation is that it isn't based on *Cygwin* (till now a must for people who want to use Linux programs on Windows) but on *MinGW* (**Min**imalist **G**nu for **W**indows): so it's a native version for Windows that doesn't need any Cygwin dll to work and this leads to a better efficiency but also to a worse compatibility if compared with Cygwin.

On YAGARTO's website (www.yagarto.de) you can find not only the toolchain (*Binutils*, *Newlib*, *GCC compiler* and *Insight*), but also all the other tools we described previously among the necessary software, i.e. OpenOCD and Eclipse + Zylin CDT.

You have to download all the packages showed in the squares in the following picture:



Package	Version	Last Version
Open On-Chip Debugger (2.45 MB) (md5sum: 4760b230ed794c45c90641ca49fb5dc0) **Note: The "flash bank" syntax has changed for LPC2000 and STR7x targets. For more information take a look here.**	r204-rc01	07.09.2007
YAGARTO Tools (700 KB) (md5sum: a1c654d6704bd3c1e109a73ce22eee2a) Include tools like make, sh, touch and more. You only need these tools if you do not have installed the Open On-Chip Debugger, and want to use J-Link / SAM-ICE	20070303	03.03.2007
YAGARTO GNU ARM toolchain (33 MB) (md5sum: 300334de4ebf1bb15b28b368670c8dc9)	Binutils-2.17 Newlib-1.15.0 GCC-4.2.1 Insight 6.8	01.09.2007
Integrated Development Environment (59 MB) (md5sum: 7563b79b78fbc9092e816271185813c6)	Eclipse 3.3 Zylin CDT 20070830 Zylin plugin 20070830	09.09.2007

Pay attention to the executable names, which indicate the package version, because Michael Fischer,

the creator of YAGARTO, uploads new versions, after that new versions of GNUARM appear. In this moment, the executables you can download are the following:

- *openocd-2007re204-setup-rc01.exe*
- *yagarto-bu-2.17_gcc-4.2.1-c-c++_nl-1.15.0_gi-6.6_20070901.exe*
- *yagarto-ide-20070909-setup.exe*

Update: Just before I put on-line this document, I saw on Yagarto's website that a new version of the Yagarto GNUARM toolchain was present, that kept the same versions of the included programs, with the exception of Insight, which has been downgraded to the version 6.5, because of some problems Michael Fischer had with the latest version 6.6. It's ok to download this version, but the only thing to take into account is that, if you want to try the STR91x microcontroller (see Appendix C), you must have Insight 6.6. Don't panic, you can still download the YAGARTO GNUARM toolchain with Insight 6.6: below the box where you can download all the packages, there's the sentence "*Older software versions can be found here*", following that link you will go to a page in which all the versions of the packages are present, included the one we search, that is the file *yagarto-bu-2.17_gcc-4.2.1-c-c++_nl-1.15.0_gi-6.6_20070901.exe*.

After downloading these packages, you can start the installation phase. The steps to correctly install YAGARTO are the same for each of its packages. These steps are:

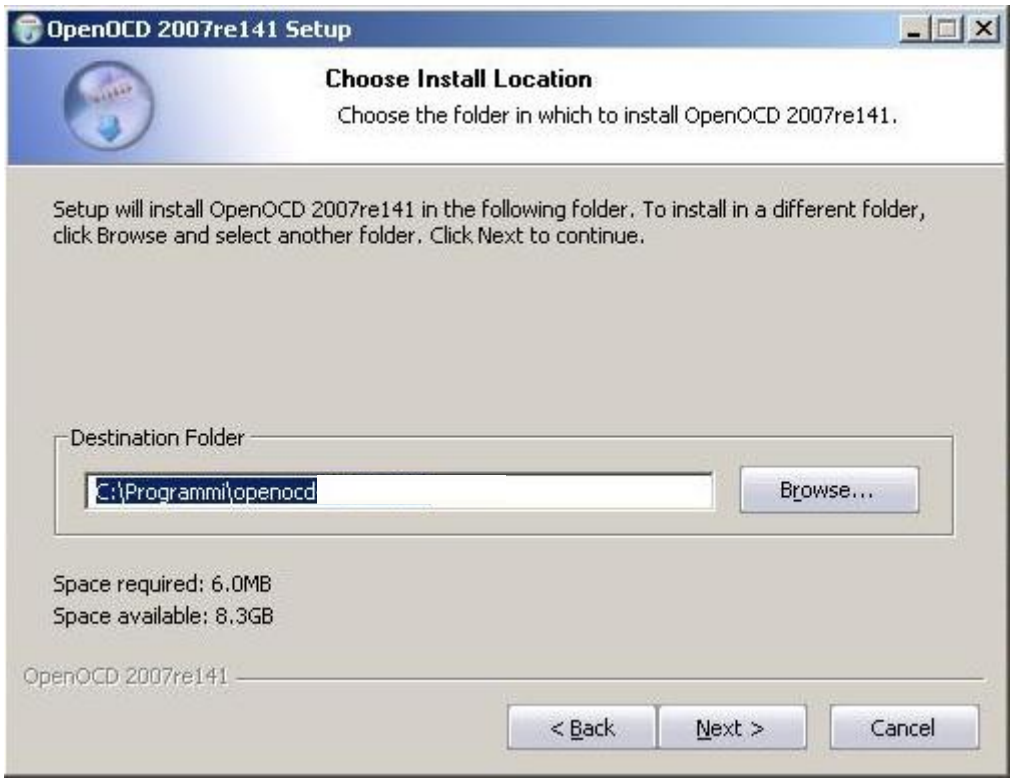
- Launch of the executable
- Acceptance of the GPL license
- Selection of the packages to install for each component (for example the make of OpenOCD)
- Choice of the installation path
- Installation

Now let's analyse the last three steps of each package installation:

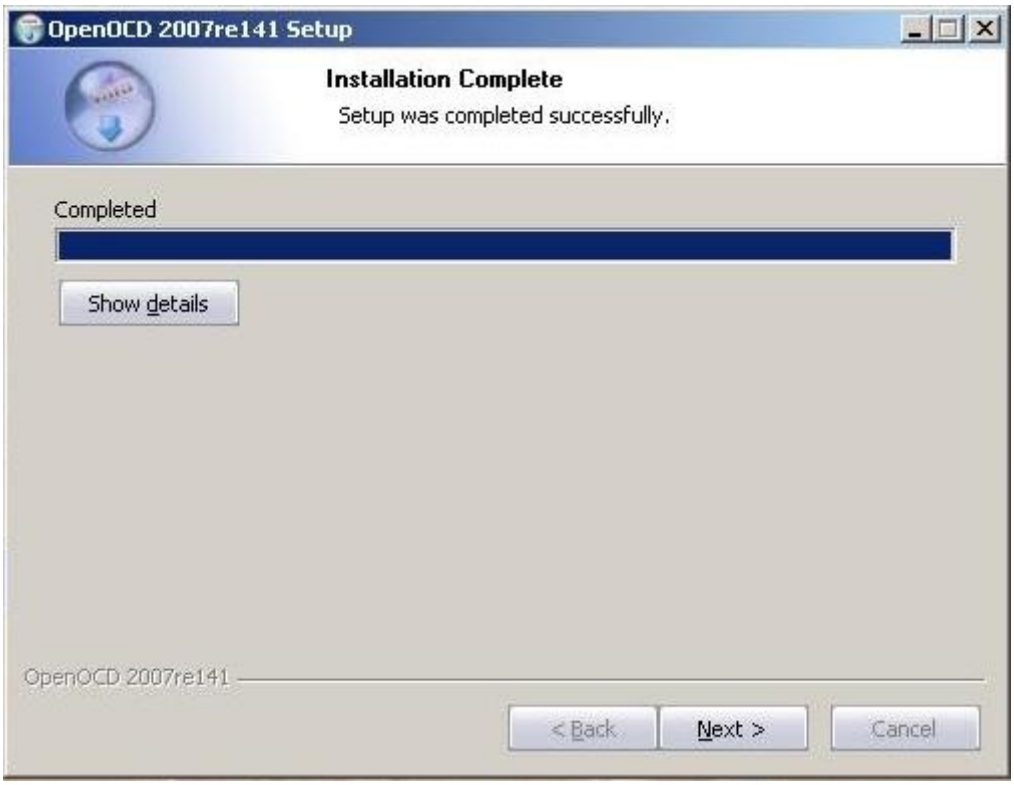
OpenOCD (openocd-2007re204-setup-rc01.exe)

Through this window you can choose which components of OpenOCD to install, we have selected all of them. Now you have to choose the installation path, the default directory contains the version number (for example in our case *C:\Programmi\openocd-2007re204*), but our advice is to make it not contain any number (so in our case it becomes *C:\Programmi\openocd*), so you won't have to

change the OpenOCD directory in the Makefile of your programs when you install a new version of OpenOCD.



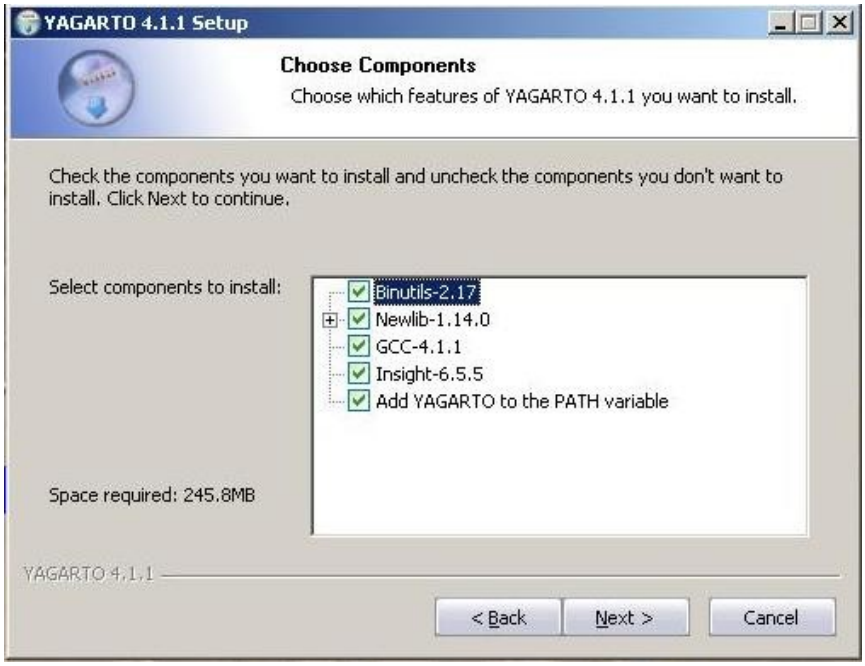
Now the installation will start and if all is ok you'll get this window:



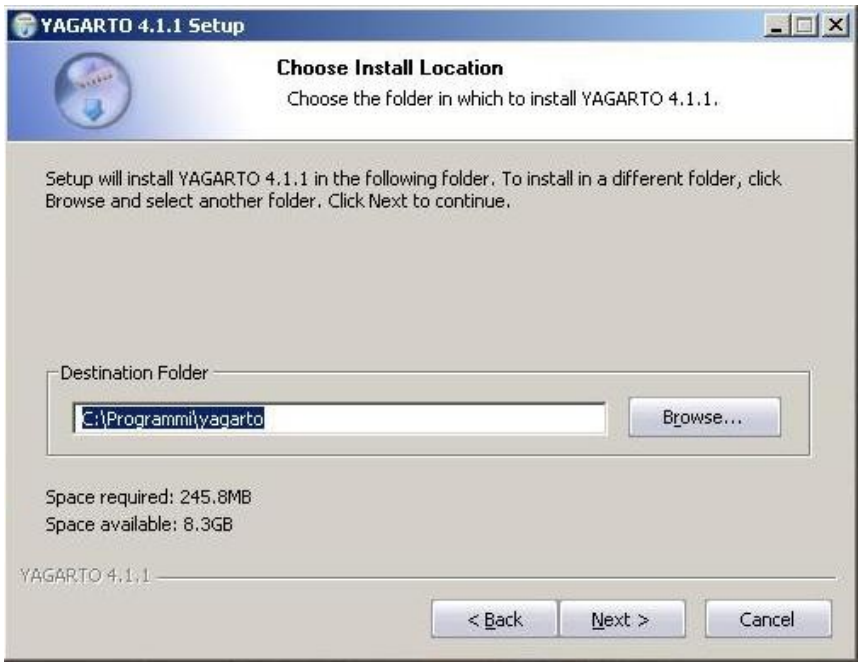
Don't mind if OpenOCD is not present in the *Start* menu, in that it will be called from Eclipse or

from the command prompt and not from the *Start* menu.

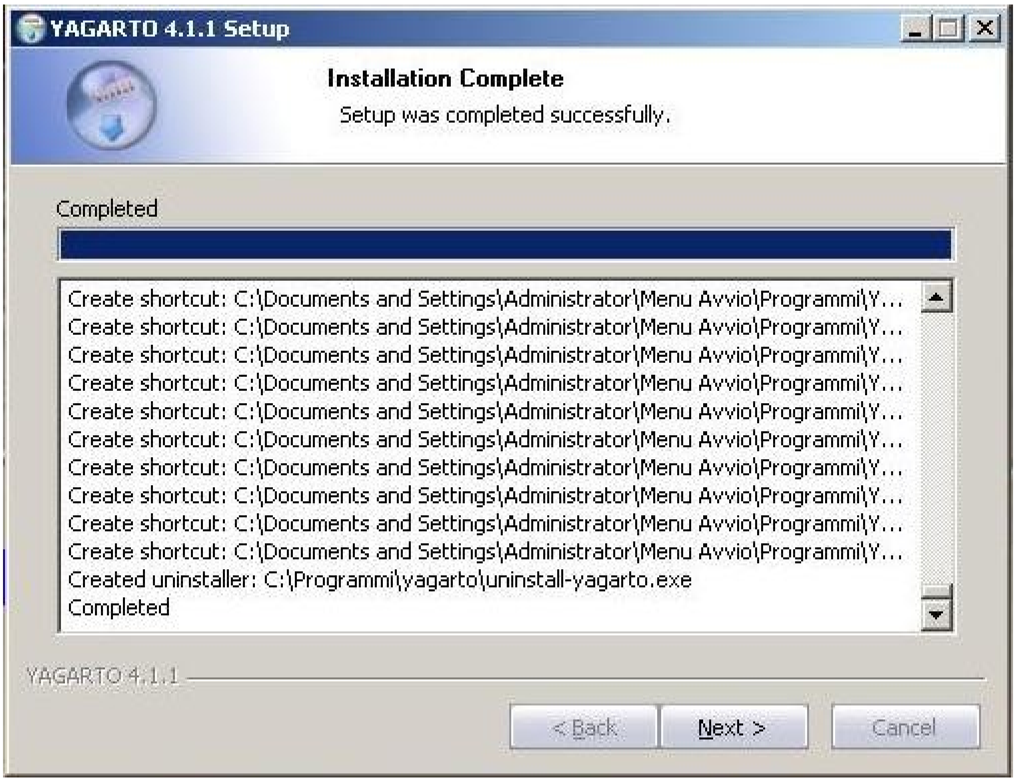
Toolchain (yagarto-bu-2.17_gcc-4.2.1-c-c++_nl-1.15.0_gi-6.6_20070901.exe)
The following window should appear:



Select all of them and click on the *Next* button. Now you have to choose the installation path.



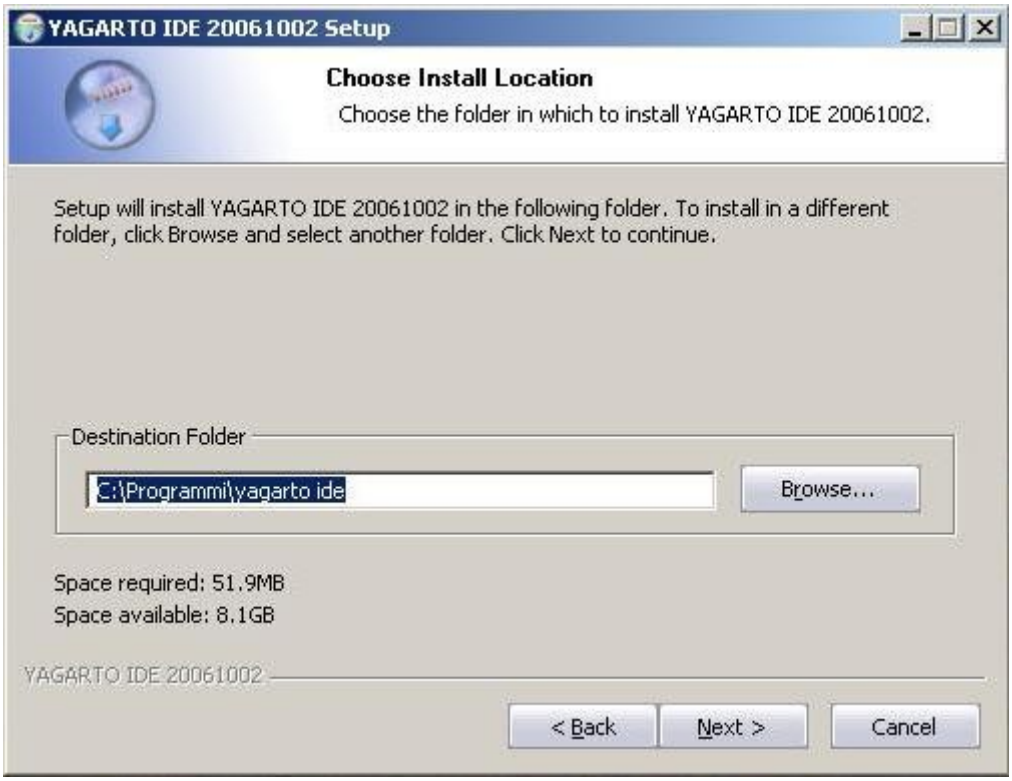
Choose it and click on the *Next* button. The installation will be executed:



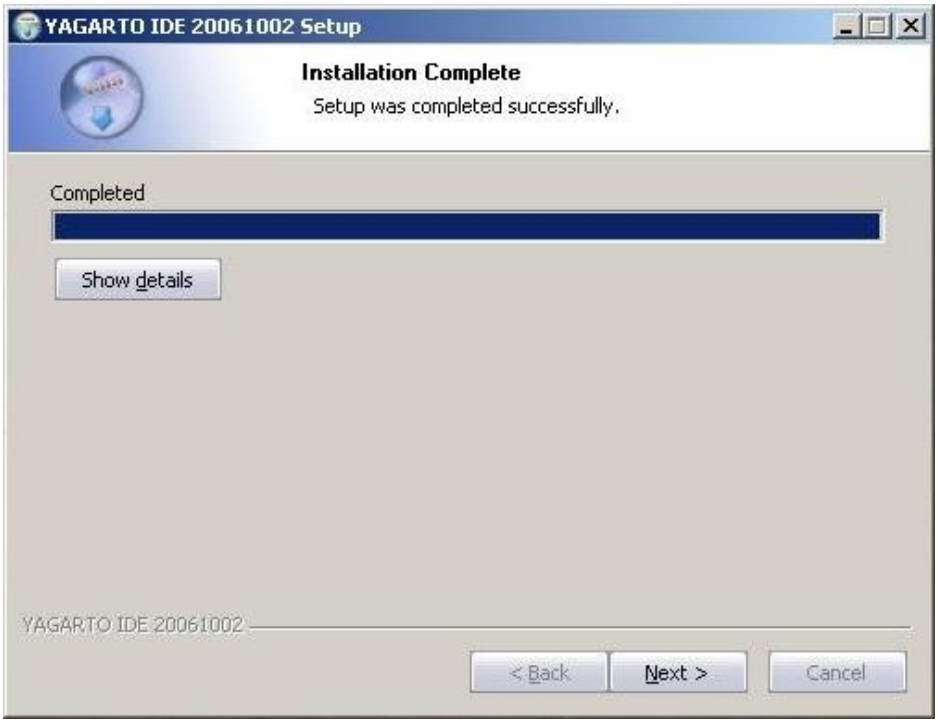
Now click on the *Next* button.

Eclipse (yagarto-ide-20070909-setup.exe)
A window in which you can choose the packages to install should appear:

Select all of them and click on the *Next* button. Now you have to choose the installation path.



Choose it and click on the *Next* button. The installation will be executed:



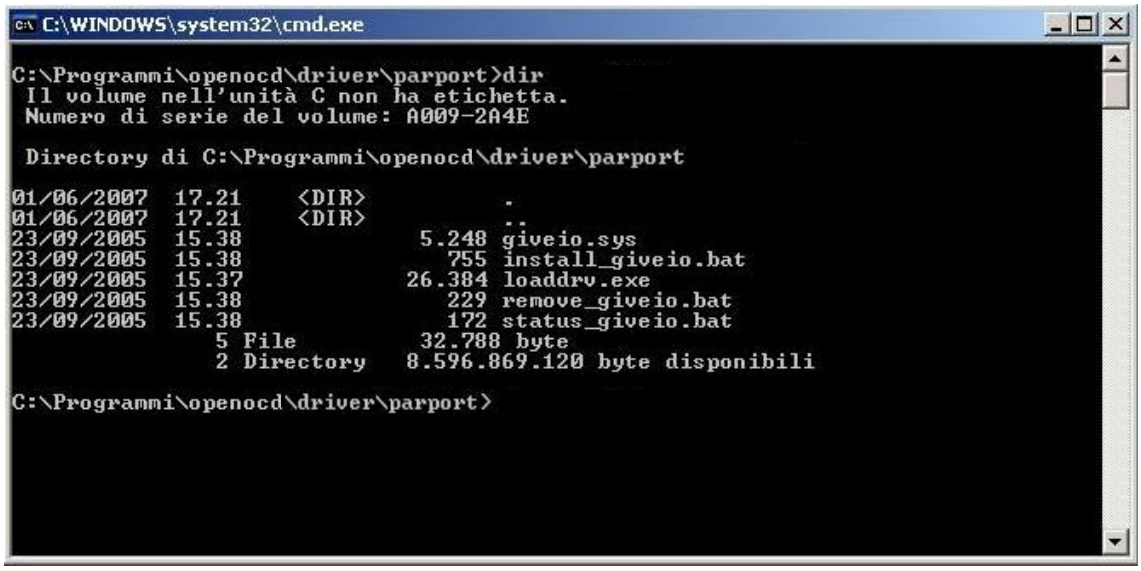
Now click on *Next*.

Parallel port driver installation

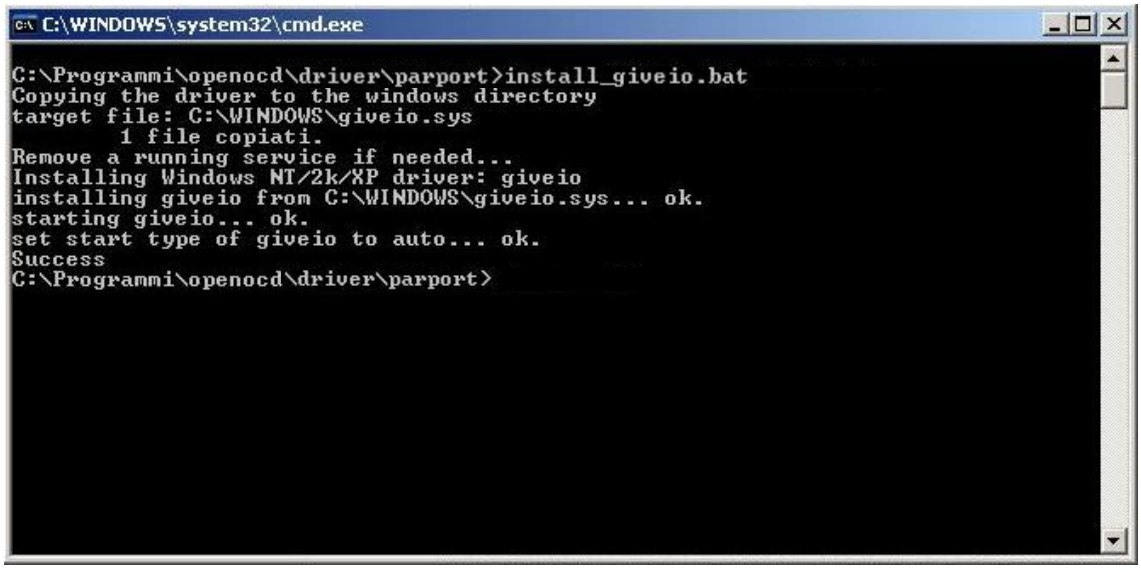
To use the parallel port with the Wiggler, you have to install the driver for this device.

To use another way to connect to the target and install the correct driver, refer to YAGARTO's site or to Jim Lynch's tutorial.

The parallel port driver is a file called *giveio.sys* and it is located in the subdirectory *driver\parport* of OpenOCD.
In the following picture you can see the content of this directory:



To install the driver you have to launch the installation batch file *install_giveio.bat* from the command prompt.
The file will install and load *giveio.sys* as a Windows driver and then the installation is permanent. You can see below the screenshot of this procedure:



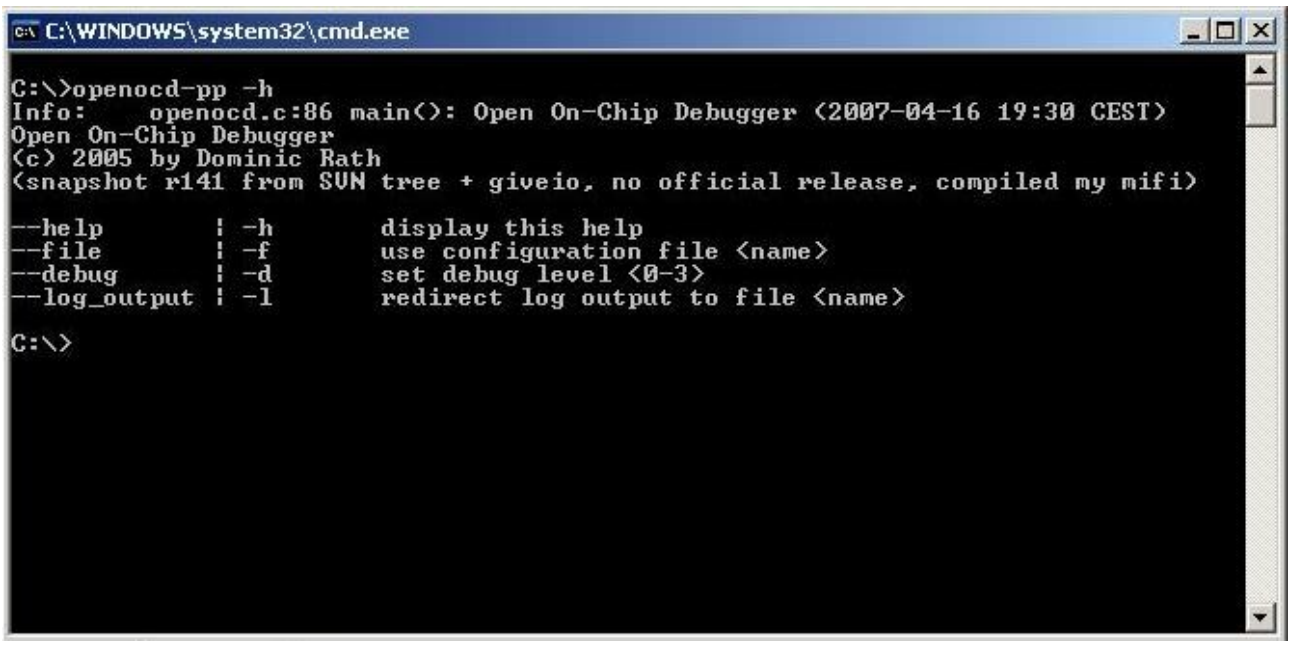
The driver can be uninstalled using the batch file *remove_giveio.bat*.

Installation test for OpenOCD, driver and utilities
Now you have to test if the *make* command works properly.

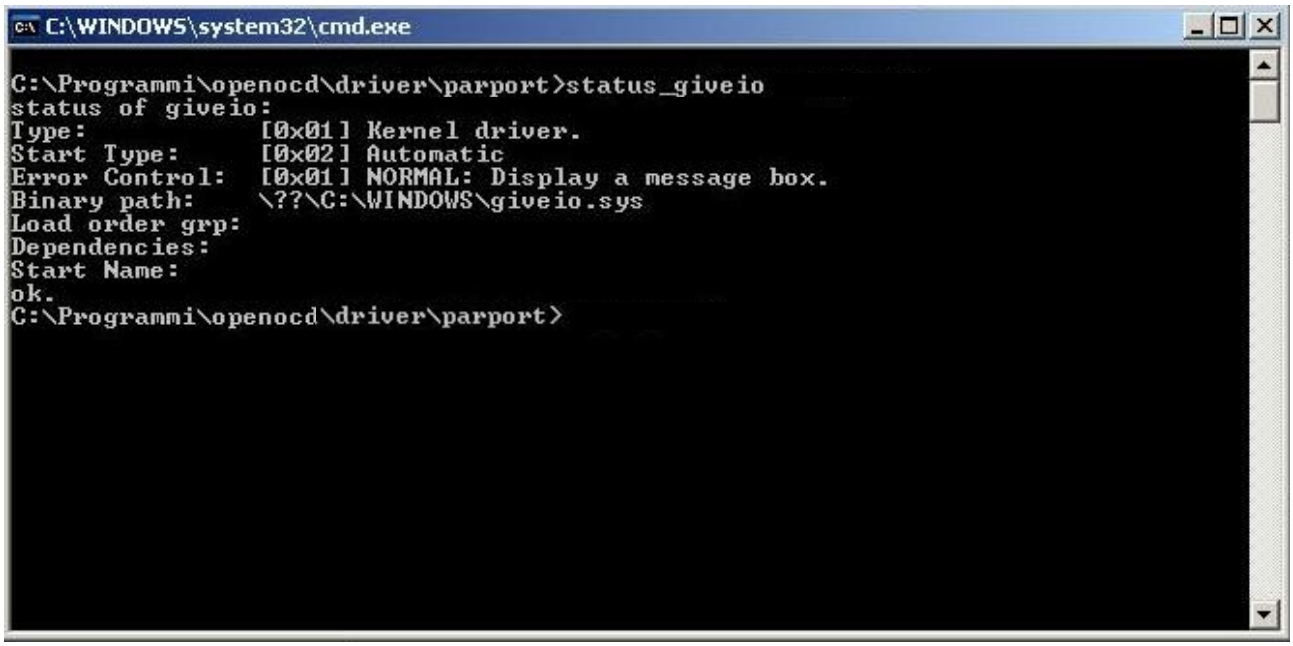
Make is an utility used in the operating system UNIX (in this case it has been ported on Windows by MinGW) to speed up the compilation of the source files and the linking of the object files generated by the compilation. To do that, it's necessary to use a file called “Makefile” that contains all the information about these two fundamental phases, so user won't have anymore to write all the commands whenever he wants to recompile the program, it is enough to call the command *make*. Moreover *make* uses a database in which it records all the information it grabs during its execution, so if you modify something in the source code and then you recompile, only the files really modified will be recompiled, saving so time.

Open the command prompt and type the command *make --version*: if all is ok, the output should look like the one shown in the following picture:

If the output is not similar to the one shown, either *make* hasn’t been installed or it isn't in Windows PATH, so repeat Yagarto installation (only the toolchain part).
To check OpenOCD installation, type the command *openocd-pp –h*: if all is ok, the output should look like the following:



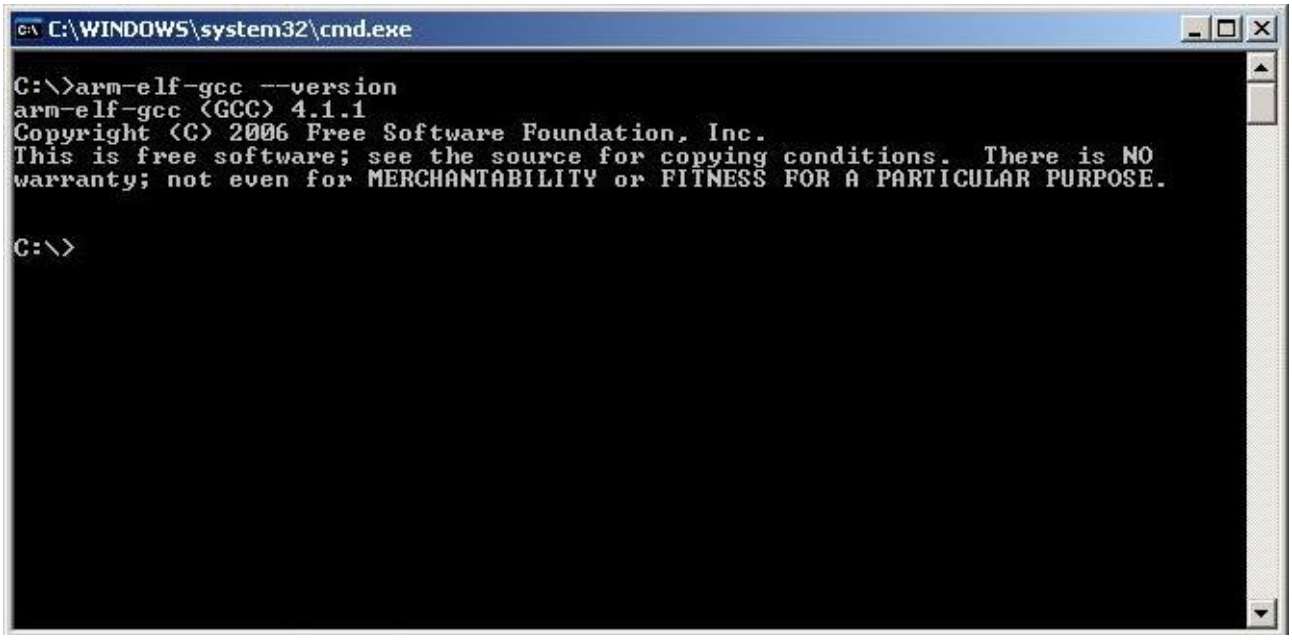
If the output is not similar to the one shown, either OpenOCD hasn't been installed or it isn't in Windows PATH, so repeat OpenOCD installation.
Take into account that OpenOCD installer installed two OpenOCD executable versions: *openocd-pp* for parallel port and *openocd-ftd2xx* for a FDTI FT2232C device.
Finally, if you installed the parallel driver, you have to check it works properly; to do that go through the command prompt to the subdirectory *driver\parport* of OpenOCD and launch the batch file *status_giveio.bat*.
If the driver is ok, the output should be the following:



If instead you installed another driver, you have to test it (refer to Yagarto's site or to Jim Lynch's tutorial).
Now let's try to launch OpenOCD using the just installed parallel port driver (or the driver you installed). It's necessary to launch it specifying a configuration file which describes the target

parameters: this is done by the command *openocd -f CONFIGFILE ,* where *CONFIGFILE* must be substituted by the proper configuration file: as we already said, those files are located in the folder *openocd-configs*, that you have already copied at the beginning of this tutorial; you have to choose this file among the many alternatives, that differs for the microcontroller and for the way to connect to the target (don't worry, the names are really intuitive), although we did our tests only through the parallel port.
For example, the file for the connection to a STR710 board using parallel port is called *str71x_pp.cfg*.
To use that file, after you have connected and powered on the board, you must type the command
`openocd -f OPENOCD_CONFIGS_PATH\str71x-configs\str71x_pp.cfg`
where *OPENOCD_CONFIGS_PATH* must be substituted by the path you have chosen for *openocd-configs*.
OpenOCD now should block waiting for an input. If the program exits (not for trivial reasons such as no power or configuration file not found or because of a jumper on the board that blocks debugging), it's possible to have more information by launching again OpenOCD the same way but adding the option *–d* which gives a more detailed output.
For some boards it might be necessary to modify some parameters in the configuration file: usually it's enough to lower *jtag_speed* value or to modify *trst_and_srst* parameter in *trst_only* or *srst_only* or in *none*.
If all is ok, you can terminate OpenOCD with the combination *Ctrl+C*.

YAGARTO installation test
The first thing to check is the availability of the GCC compiler for ARM: to do that you have to open the command prompt and type *arm-elf-gcc --version* to verify GCC version you're using.



If the output of the command isn't similar to that shown in the picture, then GCC compiler for ARM isn't correctly installed or isn't in the PATH, so you should install again Yagarto (only the toolchain).
Then you have to check if GCC works properly. To do it we'll use the example called *template71x*, a sample code which sums some integer numbers (it was created for the STR71x microcontroller, but

we provided a version for all the analyzed microcontrollers, so use the correct one for yours, for example *template73x* for the STR730 microcontroller, *template75x* for STR750, etc.).

2.1.2 Build of the first program and debug with Insight

Now you can start building your first program. Together with this work, we supply some simple programs (the files can be distinguished from their name); moreover, for each of them, a "template" is present, that is the base for building more complex programs (for more information you can read next sections and Chapter 3). We start from the template to do the tests.
Open the terminal and go to the template directory of your microcontroller. For example, for a STR710 board, go to the directory *template71x* (located in *ARMProjects\STR7*).
From now, we will take as example the STR71x microcontroller, but the operations we will describe are the same for the other microcontrollers, except for minor modifications in filenames.

Building the software library

You are in the template directory. Here there's a subdirectory that contains the ST software library files, that is all functions and variables that allow you to use completely all the functions and the peripherals embedded into the microcontroller.
In this case the directory is called *str71x_lib* and contains, besides other things, a *Makefile* which allows you to compile software library files in a library (file with extension *.a*), so you can then include this file in your programs.
In each program we already provided the *.a* file, but it may be a good idea to recompile it on your system. To do so, you have to move in *str71x_lib* and type the following commands:

```
make clean
make all
```

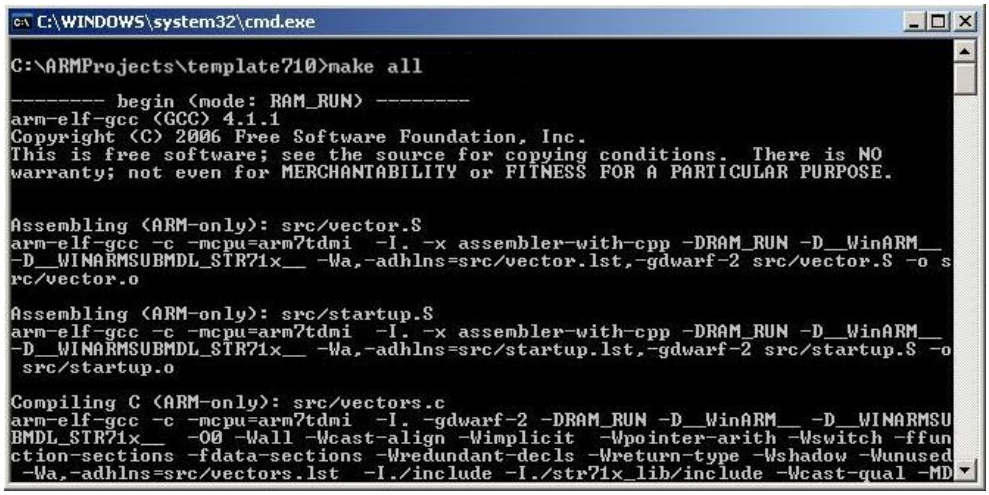
The first command cleans the files obtained from previous compilations (and also the file *.a* already present), the second one, instead, is the true compilation and you should obtain your *.a* file (in this case *libSTR71x_lib.a*).

Building the program

You have to come back in *template71x* directory and there you can see a *Makefile*, whose content defines how your compilation will be executed (for example there's a setting to include the *.a* file obtained from the previous software library compilation).
The Makefile section which interests you is the one about program execution in RAM or ROM (FLASH). You can see this in the screenshot on the left.
You have to edit Makefile, search this section and uncomment the line about the type of execution you want, commenting the other one.

```
## Create ROM-Image
#RUN_MODE=ROM_RUN
## Create RAM-Image
RUN_MODE=RAM_RUN
```

In this case we chose a RAM execution.
Then you have to exit the editor and compile your program using the command `make all`.
The output obtained should be similar to the following screenshot:



If you look at this output, you can see that at first there was each *.c* file compilation and each *.s* file assembling and finally all obtained object files were linked in the file *main.elf*.
Finally you can see a list of the *main.elf* file sections, each one has its dimension and start address.
Like in software library compilation, the command to delete the rests of previous compilations is:

```
make clean
```

If you chose a RAM execution, you can avoid reading the next rows and go directly to the Debug section, otherwise if you chose a FLASH execution (choose *ROM_RUN* in the *Makefile*), the operations to make are the same, but obviously the compilation result will change a little bit, especially in start addresses of *main.elf* sections.
Moreover you mustn't forget that in RAM it's always possible to write and read, instead **if you use FLASH memory, after the compilation and before debugging it's necessary to download the code in it.**

In fact FLASH is a ROM (Read Only Memory) memory so it can only be read and not written.
Actually it's possible to erase and write it, but these operations are done through an electric operation which doesn't work with a single byte, but with the entire block of memory which contains it.
Moreover this operation is really slow if compared to RAM writing and it can be done a limited number of times, about (10^5) times.
The *Makefile* was created to program FLASH in a simple way. However you have to edit it and change some parameters.

To do that, search in the *Makefile* the section that starts with the row *#FLASH Programming with OPENOCD*, as showed in the screenshot:

```
gccversion :
	@$(CC) --version

# FLASH Programming with OPENOCD

# specify the directory where openocd executable resides (openocd-ftd2xx.exe or openocd-pp.exe)
# Note: you may have to adjust this if a newer version of YAGARTO has been downloaded
OPENOCD_DIR = 'c:\Programmi\openocd\bin\'

# specify OpenOCD executable (pp is for the wiggler, ftd2xx is for the USB debugger)
OPENOCD = $(OPENOCD_DIR)openocd-pp.exe
#OPENOCD = $(OPENOCD_DIR)openocd-ftd2xx.exe

# specify OpenOCD configuration file (pick the one for your device)
#OPENOCD_CFG = C:\openocd-configs\str71x-configs\str71x_signalyzer-flash-program.cfg
#OPENOCD_CFG = C:\openocd-configs\str71x-configs\str71x_jtagkey-flash-program.cfg
#OPENOCD_CFG = C:\openocd-configs\str71x-configs\str71x_armusbocd-flash-program.cfg
OPENOCD_CFG = C:\openocd-configs\str71x-configs\str71x_pp-flash-program.cfg

program:
	@echo
	@echo "Flash Programming with OpenOCD..."
	$(OPENOCD) -f $(OPENOCD_CFG)
	@echo
	@echo
	@echo "Flash Programming Finished."
```

You must change OpenOCD path, the executable name and the configuration file name (*.cfg* file) you want to use with OpenOCD: there are many configuration files, each of them is used for one of the interfaces to connect to the board. As usual, we used only parallel port.
The *.cfg* file is used to connect to the board and to call a *.ocd* file, whose task is to write the FLASH. Probably you have to edit this *.cfg* file to change in it the path of the *.ocd* file. Let's explain this better. This is the content of the *.cfg* file:

```
#daemon configuration
telnet_port 4444
gdb_port 3333

#interface
interface parport
parport_port 0x378
parport_cable wiggler
jtag_speed 0
jtag_nsrst_delay 200
jtag_ntrst_delay 200

#use combined on interfaces or targets that can't set TRST/SRST separately
reset_config trst_and_srst srst_pulls_trst

#jtag scan chain
#format L IRC IRCM IDCODE (Length, IR Capture, IR Capture Mask, IDCODE)
jtag_device 4 0x1 0xf 0xe

#target configuration
daemon_startup reset

#target <type> <startup mode>
#target arm7tdmi <reset mode> <chainpos> <endianness> <variant>
target arm7tdmi little run_and_init 0 arm7tdmi
run_and_halt_time 0 30

working_area 0 0x2000C000 0x4000 nobackup

#flash bank <driver> <base> <size> <chip_width> <bus_width>
flash bank str7x 0x40000000 0x00040000 0 0 0 STR71x
flash bank cfi   0x60000000 0x00400000 2 2 0

#Script used for FLASH programming
target_script 0 reset C:\openocd-configs\str71x-configs\str71x_flashprogram.ocd
```

The last row is used to call the *.ocd* file: here you have to set the correct path.
That's all: come back in *template71x* directory and after compiling correctly the project for FLASH, you have to power on the board and execute the command:

```
make program
```

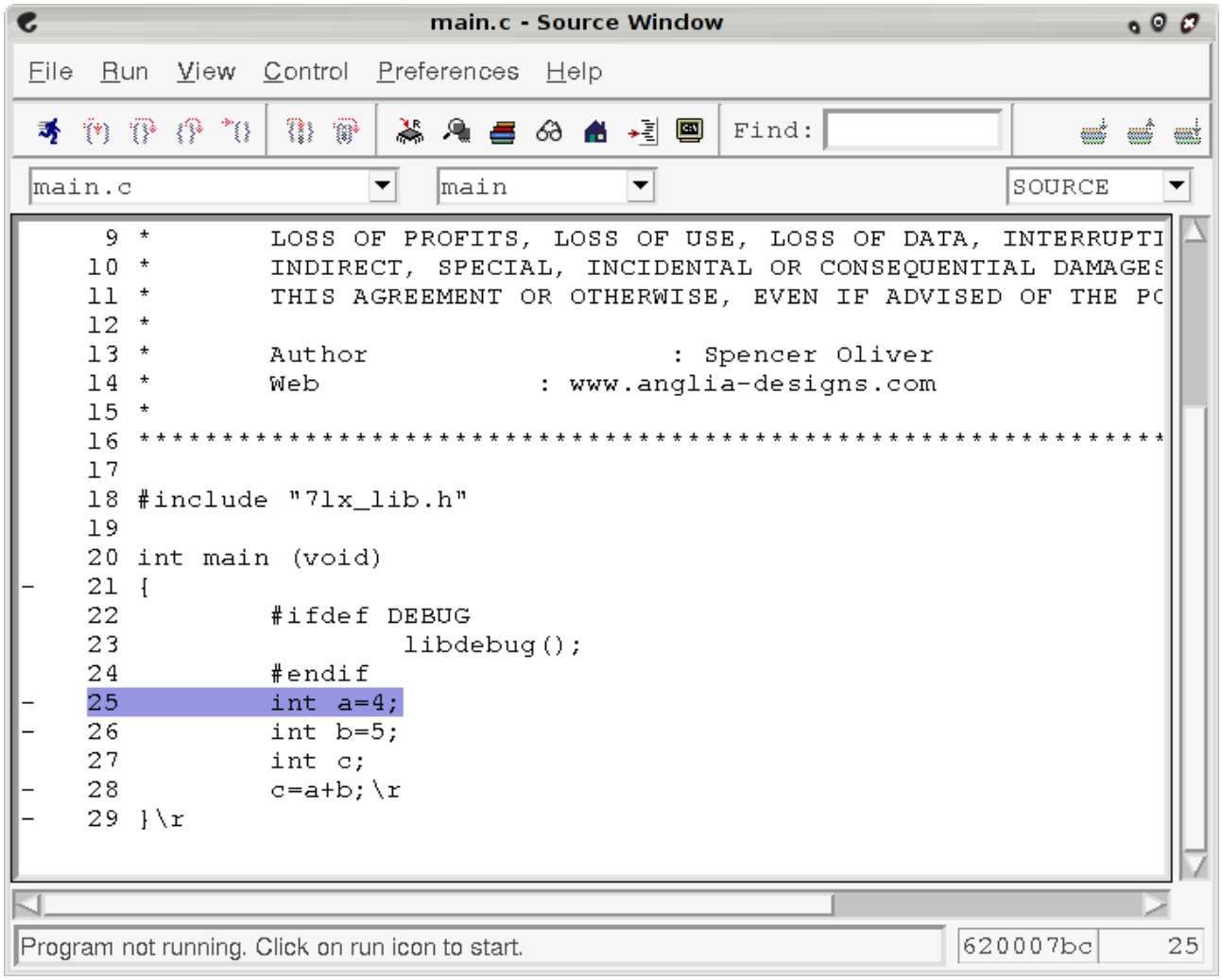
Now OpenOCD should be called, with the *.cfg* file as input, which will call the *.ocd* file that will write FLASH.
This operation could take some minutes depending on FLASH memory dimensions, then the sentence *Flash Programming Finished* should appear.
Now you are ready for debugging through INSIGHT.

Debug

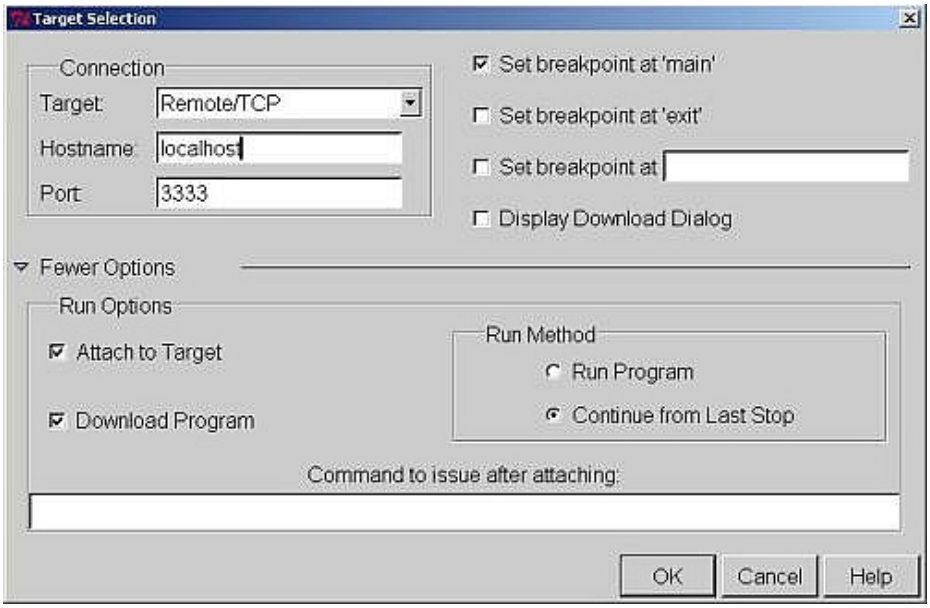
Insight is the famous GDB front-end, one of the most famous solutions used in UNIX for debugging.
You have to add some options to the command used to start Insight, so you have to call it this way: `arm-elf-insight -x GDBFILENAME ELFFILENAME` where you have to substitute the *.elf* file of the program to *ELFFILENAME*, in your case *main.elf*. *GDBFILENAME* instead must be substituted by a file which contains some commands to initialize GDB, for example for the connection to OpenOCD.
In the folder *openocd-configs* you copied before, besides the true configuration files there are also the *.gdb* files, distinguished by the microcontroller model and by the debug type (RAM or FLASH).
For example, if you want to debug in RAM a program for the STR710 microcontroller, the file to use is called *str71x_ram.gdb* , so this is the file to substitute to *GDBFILENAME*.
Before starting Insight, you have to launch OpenOCD, following what we said in the proper section of 2.1.1 paragraph.
OpenOCD will block, leave this session opened and open another command prompt session, then go to the folder *template71x* that you already compiled.
Let's launch Insight through the following command:

```
arm-elf-insight -x OPENOCD_CONFIGS_PATH\str71x-configs\str71x_ram.gdb main.elf
```

where *OPENOCD_CONFIGS_PATH* must be substituted by the path you chose for *openocd-configs*.
This is Insight main window:

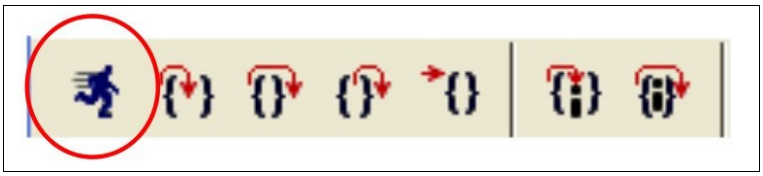


Click on the menu “File” and choose the “Settings” item.
In the window that will appear you have to set the parameters as specified in the following screenshot:

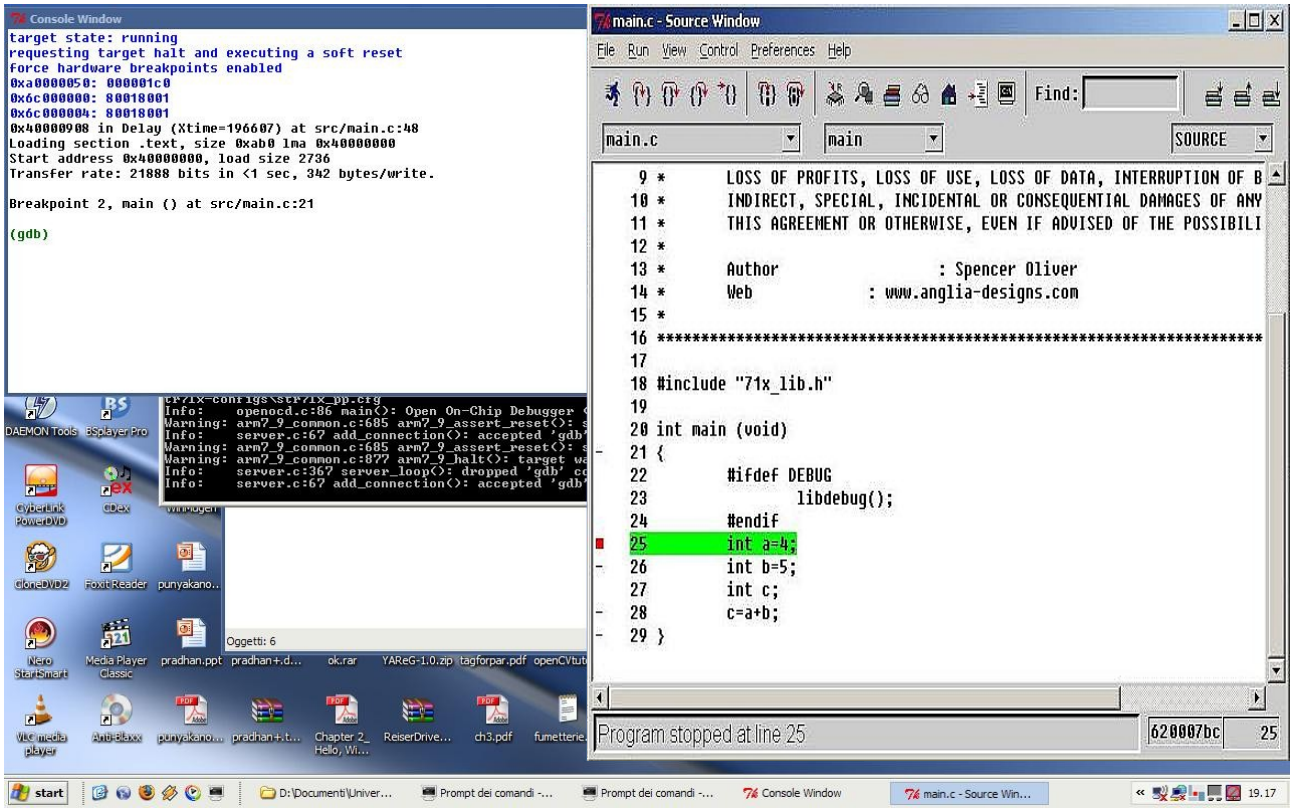


Through these parameters you establish how to connect to the target (via OpenOCD), to set a breakpoint at *main* (that is the program is loaded and then it’s stopped on the starting point waiting that the user decides to continue) and that when the *Run* button is pressed, both the operations of connecting to the target and downloading of the program in memory are automatically executed.
This is right what you are going to do: after you have pressed the OK button, you'll come back in the

main window of Insight where you have to press the *Run* button (circled in next picture).
This button, as we explained before, will execute automatically the operations of connection and download.



If all is ok, the program should be downloaded in memory and start, blocking on the first instruction, waiting for a decision by the user on how to continue.
The next picture shows what we have said:

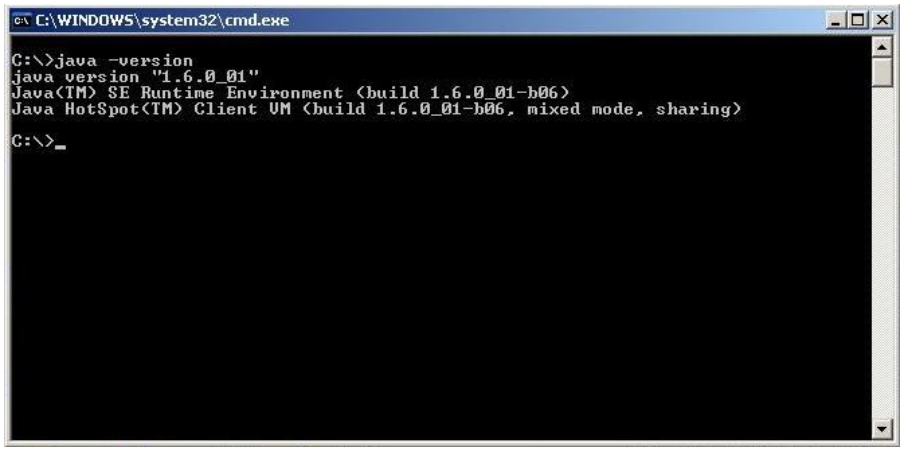


Please look at the console on the left that can give you useful information.
If the console isn't present, it's possible to call it by clicking on the button shown in the picture:

If Insight, instead of loading the program, blocks or has an anomalous behaviour, it could mean that the addresses are wrong in linker files or in *.gdb* files.
For more details refer to chapter 3.
Insight is a very powerful debugger that allows to do the most important operations, as step by step execution of instructions, watching and changing of variables and registers (we will deal with all these functions in the appendix A), but now let's pass to an IDE that manages compilation and debugging functions, and result to be at the same time simple and useful: Eclipse.

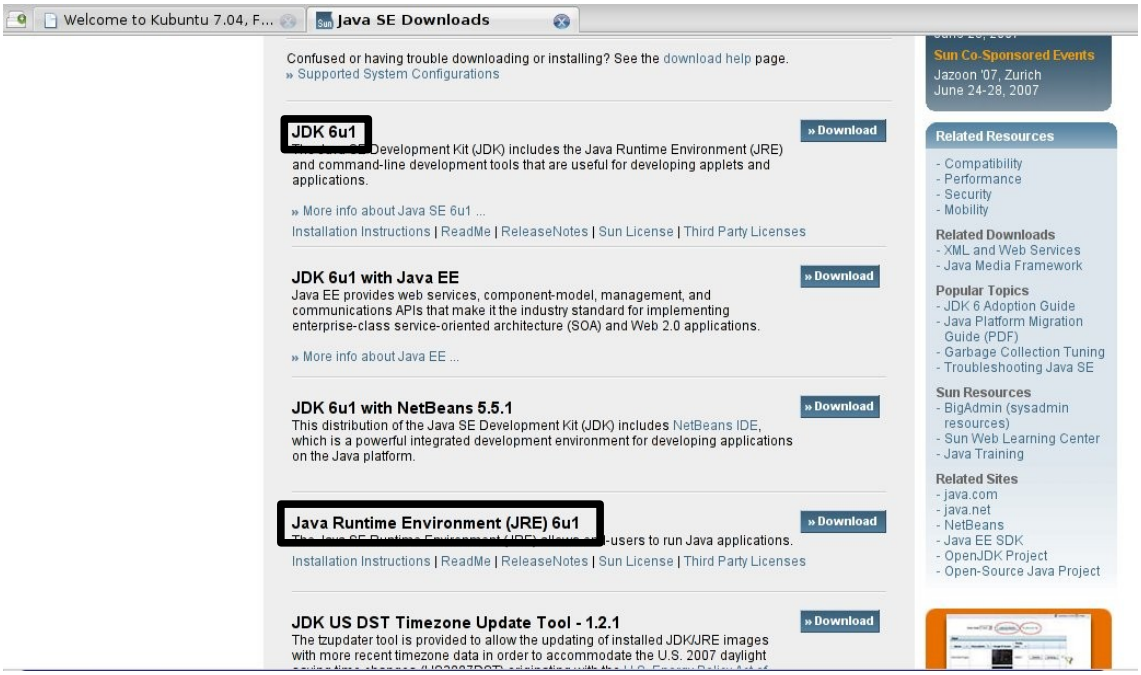
2.1.3 How to check Java presence

To make Eclipse work properly, it's necessary to make sure you have the right version of the JVM (*Java Virtual Machine*): to do it, open the command prompt and type *java –version* . You should obtain an output similar to the one shown below:



Note that the version of the installed JVM is the 1.6.0_01 and it's ok for your purposes since the Zylin plugin for Eclipse is compatible with versions of Java from 1.4.2.
If the screenshot output is different, then the JVM isn't correctly installed or the binary JVM files aren't in the Windows PATH and so you have either to install JAVA or to set the path. Let's see how to install JAVA.
For our purpose the JRE is enough, since we have to only execute Eclipse, but maybe you need in the future to develop something in Java, so you should consider the possibility to download the JDK.
After you have connected to Sun's web page, you have to go to Downloads – Java SE section, as you can see from the picture.

A window similar to the one shown below will appear, in which you can choose among several versions. You have to choose between the two versions shown into the box, which are JDK and JRE.



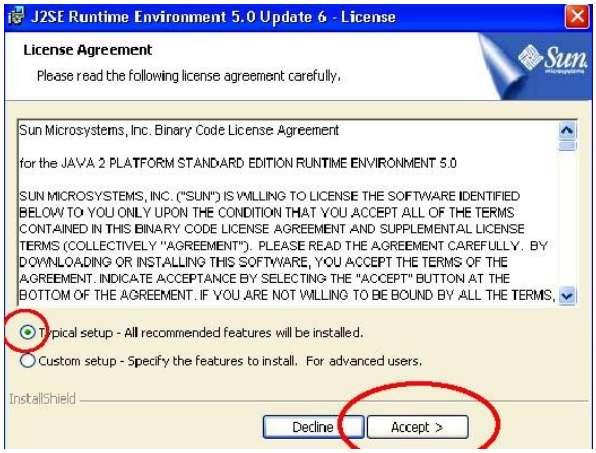
We have chosen JRE, but the operations are equivalent for JDK. Now a window should appear, in which you can download the requested package depending on the operating system you have. For Windows you have to choose between the two versions in the red box in the following picture: the first one allows to download the whole package and then to install it, the second one instead downloads only the installer and the files are downloaded from Internet during the installation.
We chose the first option.

Windows Platform - Java(TM) SE Runtime Environment 6 Update 1			
	Windows Offline Installation, Multi-language	jre-6u1-windows-i586-p.exe	13.16 MB
	Windows Online Installation, Multi-language	jre-6u1-windows-i586-p-iftw.exe	361.65 KB

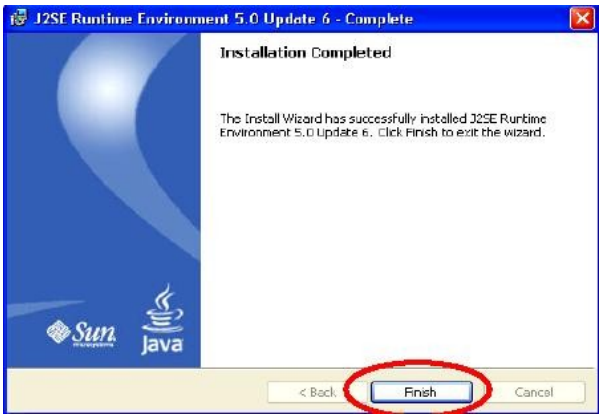
Linux Platform - Java(TM) SE Runtime Environment 6 Update 1			
	Linux RPM in self-extracting file	jre-6u1-linux-i586-rpm.bin	17.67 MB
	Linux self-extracting file	jre-6u1-linux-i586.bin	18.15 MB

After you have downloaded the *.exe* file, you have to double-click on it to launch the installation process.

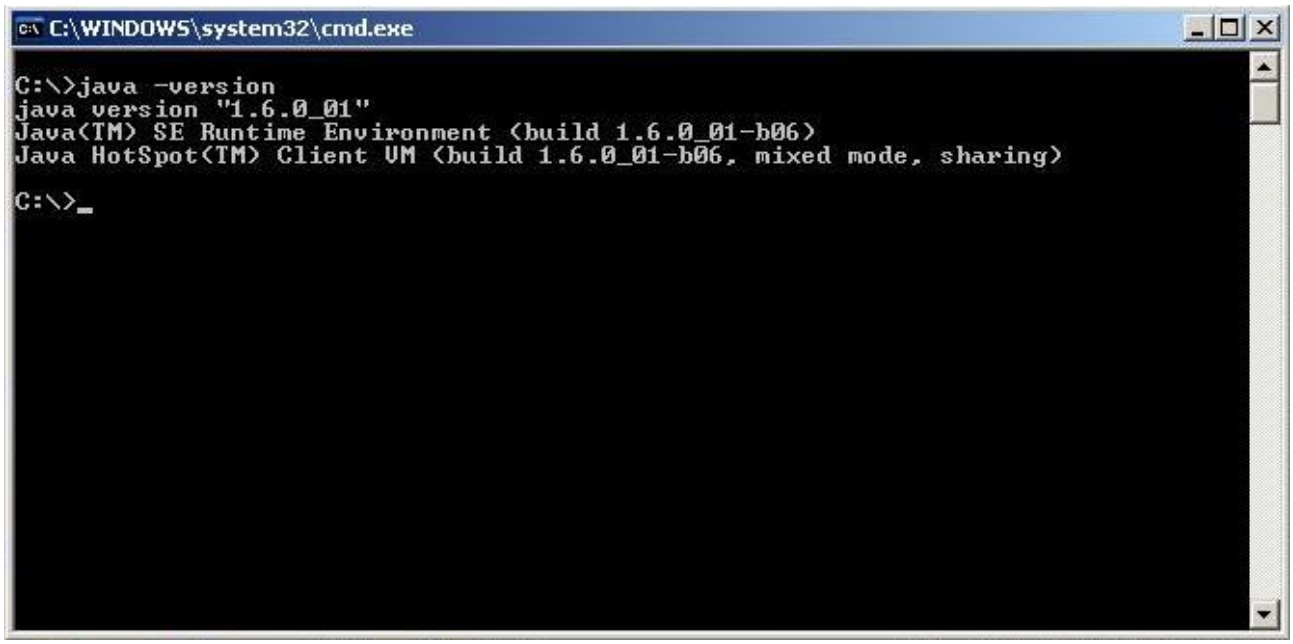
You can choose *Typical* setup and click then on *Accept*, as shown in the following window:



Now the installation starts. At the end the following picture will appear:



Click on *Finish*. If now you try to open the command prompt and to type *java –version* , you should get an output as the following one:

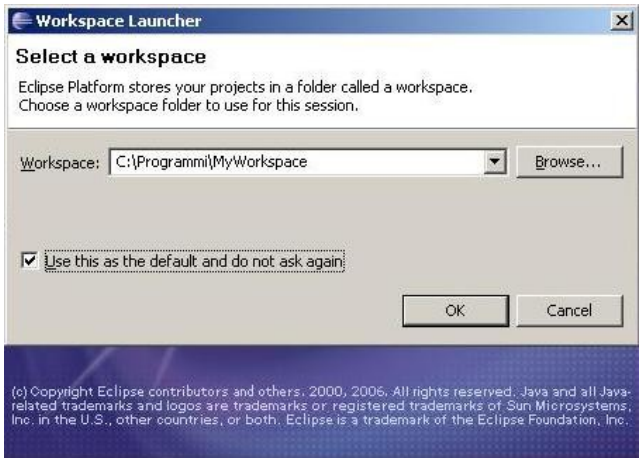


Now Java is correctly installed.

2.1.4 Eclipse: first run and preliminary configuration

Click on the link to Eclipse you did on the desktop.
As soon as you have launched this IDE, a Workspace Launcher will appear, where you have to specify the path and the name of the folder that will contain the workspace, which contains all the files of your projects.
If the specified folder doesn't exist, Eclipse will create it for you.

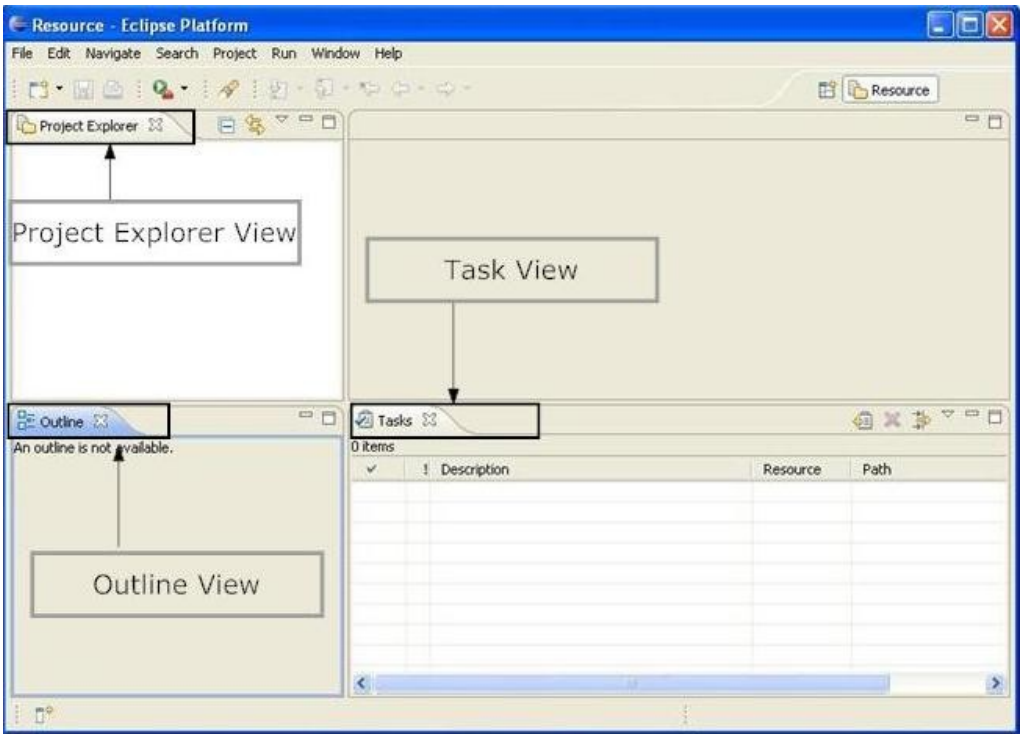
> It's important to use always the same workspace, because when you change it, your configuration settings aren't anymore visible.



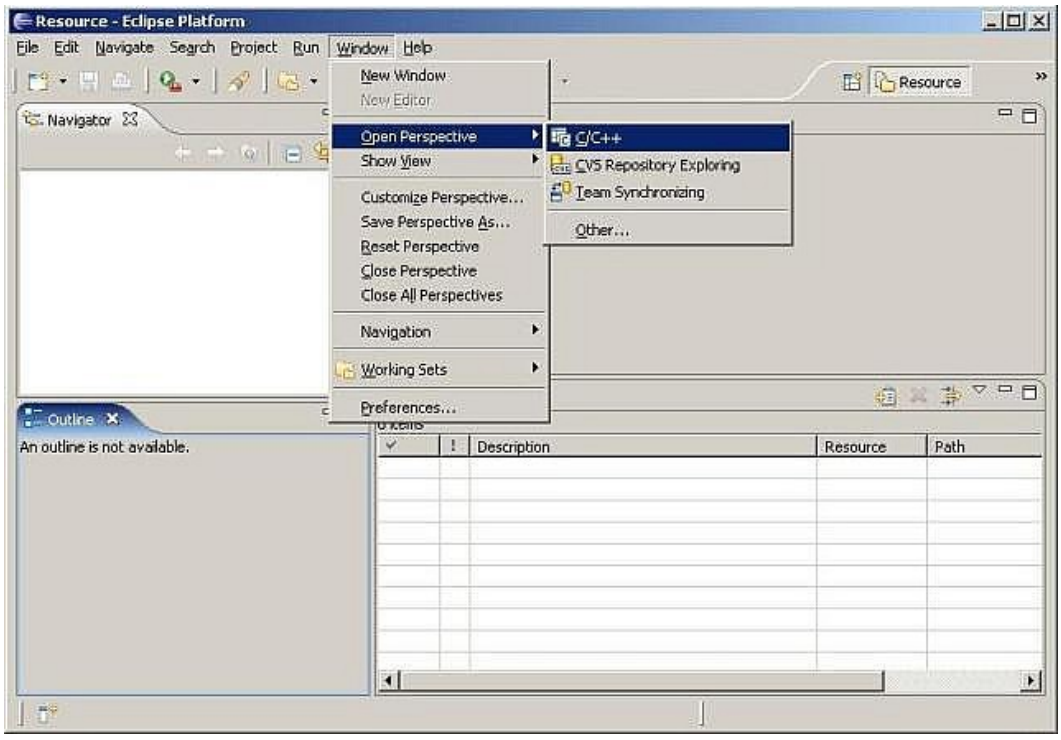
Then the Eclipse welcome page will appear:

Since many information you can access through the shown icons refer to Java programming, for our purposes this window isn't so useful and so you can close it.
An important aspect is that Eclipse is made from many perspectives, each of which allows to access different resources managed in views: when you set one perspective instead of another one, the

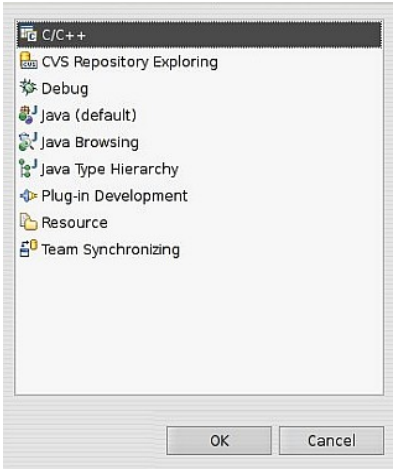
physical aspect of the IDE will change because new views will appear.
Now close the welcome window to access automatically the *Resource Perspective* (it's the default perspective), where there are the *Project Explorer* view, the *Outline* view and the *Task* view, as you can see in the following picture:



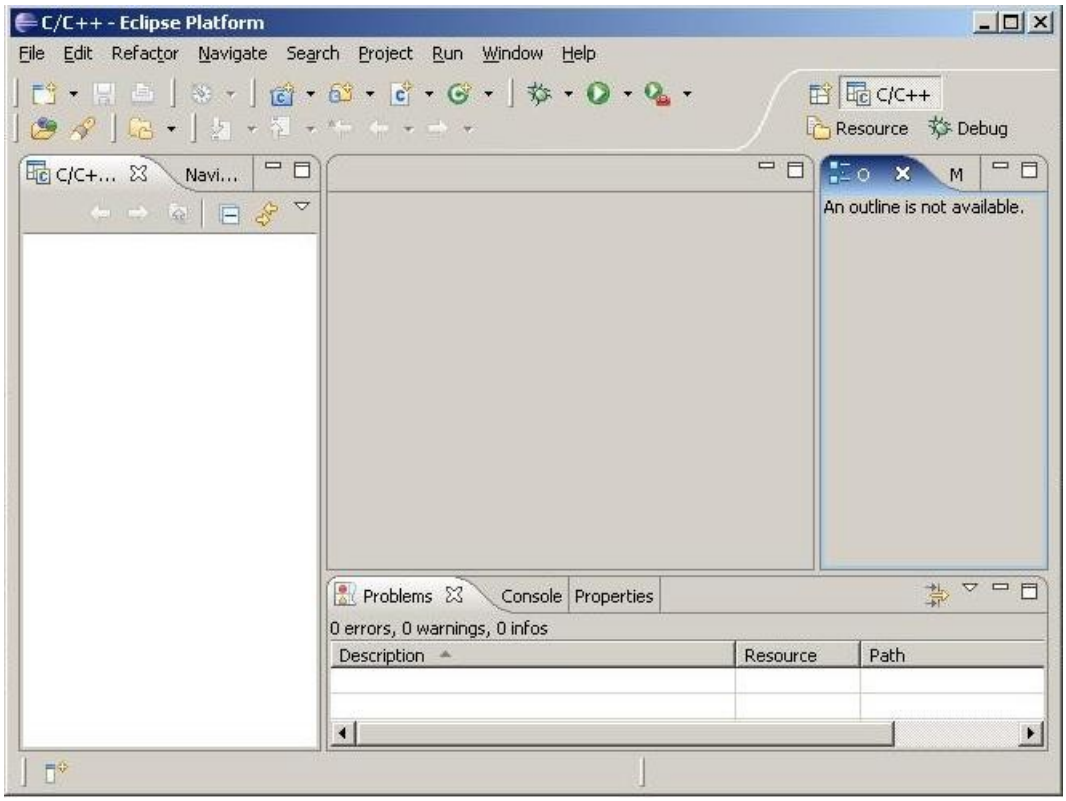
Now it's necessary to switch to the C/C++ Perspective: to do that go to menu *Window->Open Perspective->C/C++*, as you notice in the following picture:



If the *C/C++ Perspective* is not in the list, click on *Other*, so a new window with all the perspectives will appear, in which you can choose the C/C++ one, as shown below:



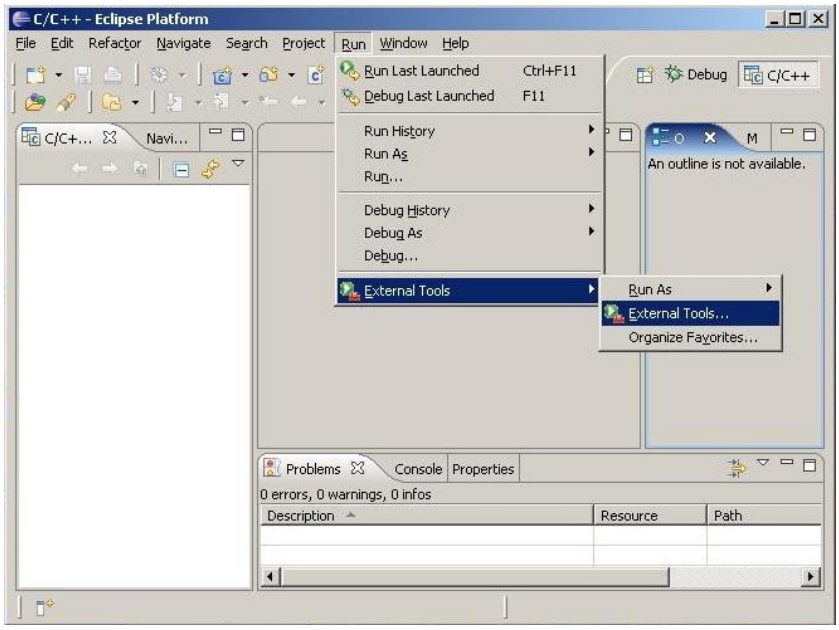
After you have selected the C/C++ Perspective, it will appear as shown below:



Inserting OpenOCD in the Eclipse external Tools

Now let's insert OpenOCD into the Eclipse external tools. Indeed, OpenOCD is necessary for debugging. Moreover it will also be used to erase the microcontroller FLASH memory. Let's see how.

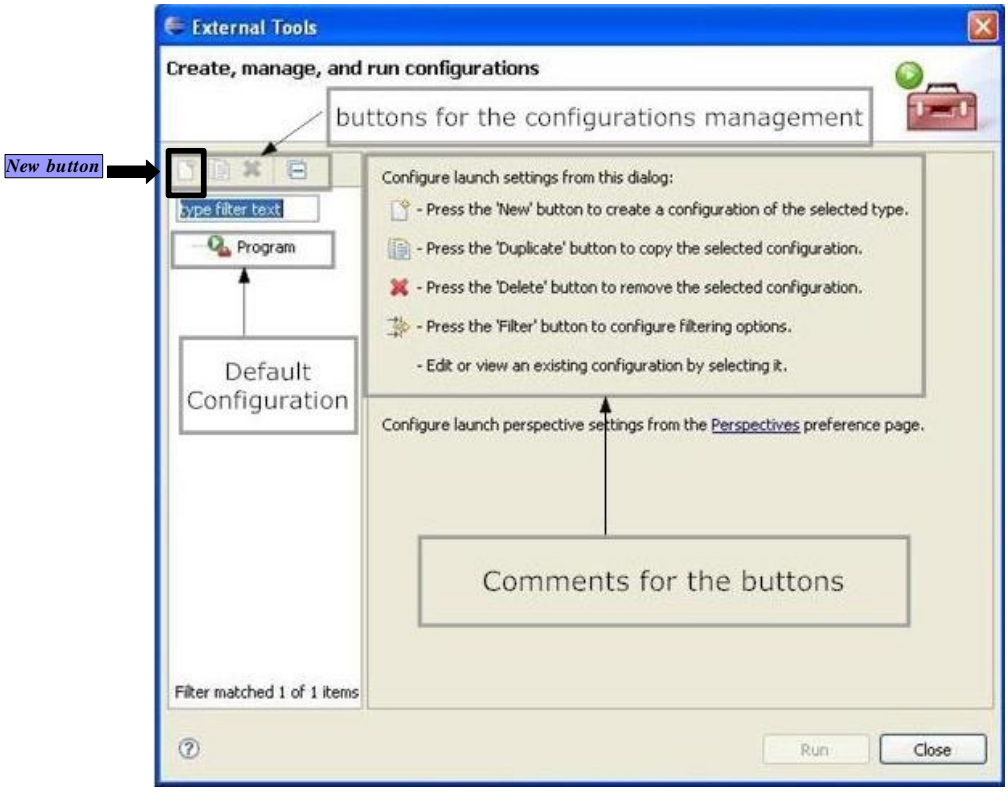
Eclipse has a very powerful feature that allows the user to include external programs. To use this feature it's necessary, as shown in next picture, to go to the menu *Run->External tools->External tools...*.



External tools window will appear.

In the following steps we will took as example STR71x microcontroller, but the same steps have to be done for each of the used microcontrollers, so, in our case, we will have in our list *OpenOCD (STR71x)*, *OpenOCD (STR73x)* and *OpenOCD (STR75x)*, each one with its own parameters.

Now click on *Program* and then on the *New* button to insert an external program in the list, as shown in the following picture:



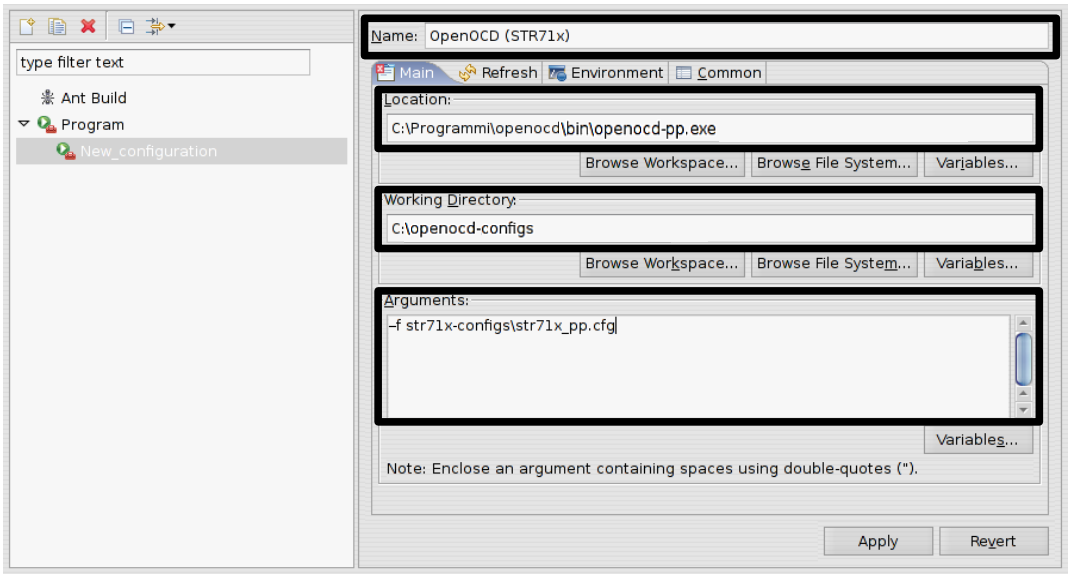
You have to fill the *External Tools* window this way:

- In the *Name* textbox insert *OpenOCD (STR71x)*.
- In the *Location* textbox insert the path of the OpenOCD executable. There are two OpenOCD versions: *openocd-pp.exe* that works with Wiggler and with parallel port (Olimex

ARM JTAG) and then *openocd-ftd2xx.exe* that works with Amontec and Olimex programmers based on USB connection. Using *Browse File System…* option it's possible to find this file that in our case is *C:\Programmi\openocd\bin\openocd-pp.exe*.

- In the *Working Directory* textbox insert the path of the folder *openocd-configs* (in our case *C:\openocd-configs*).
- In the *Arguments* textbox insert the option *–f str71x-configs\str71x_pp.cfg* (to specify OpenOCD configuration file for the chosen microcontroller; we used the parallel port version, but there are also the versions for the alternative connections).

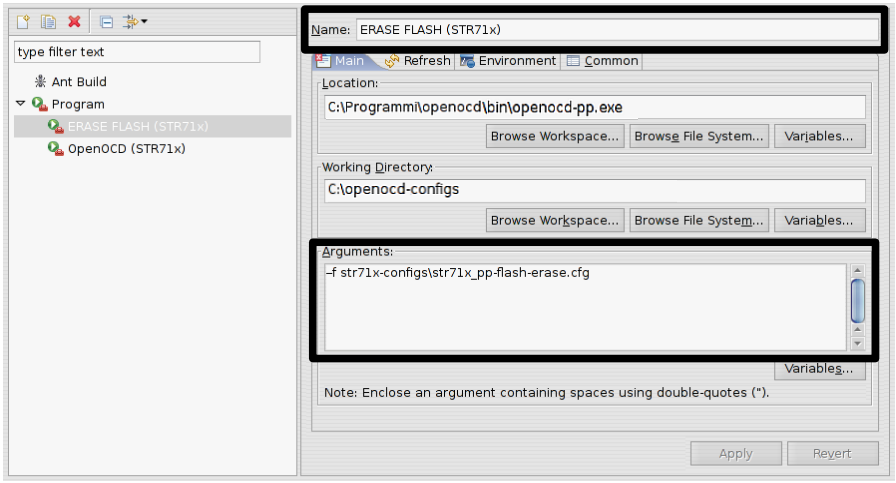
All these settings can be seen in the following picture:



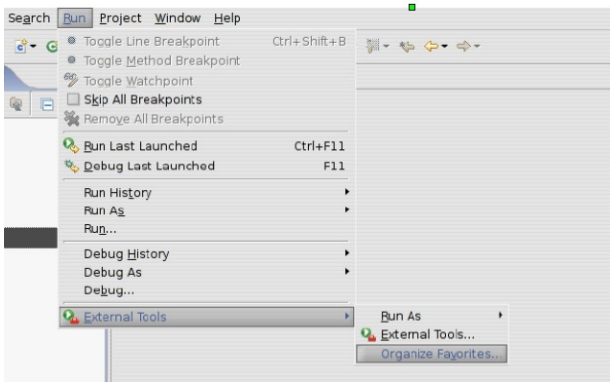
No other changes are necessary in the other tabs, that are *Refresh, Environment* and *Common.*
Then, by clicking on *Apply*, OpenOCD will be registered into Eclipse as an external tool.

Inserting OpenOCD as FLASH eraser among Eclipse External Tools.

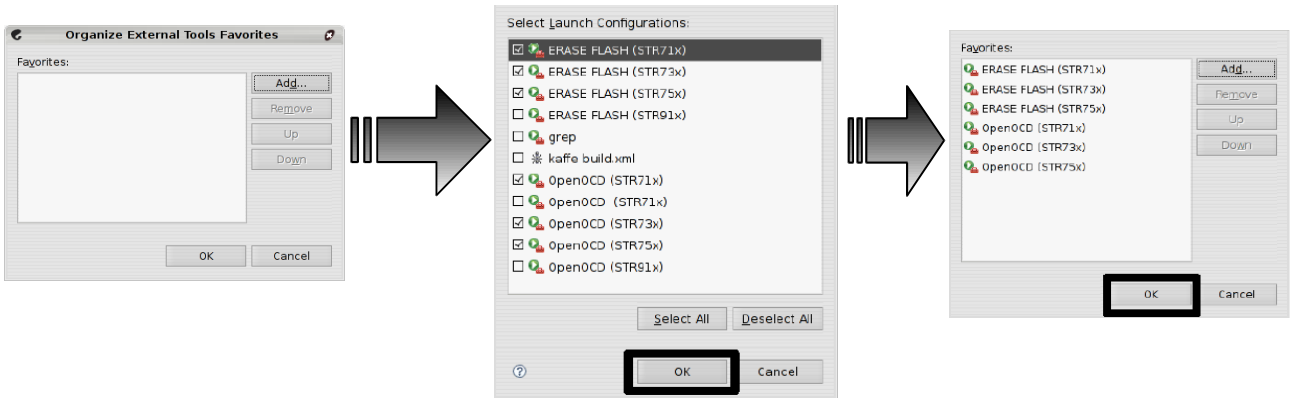
As we have just said, you also have to insert OpenOCD as eraser for the FLASH memory.
To do that, you have to insert a new program with the procedure just seen, inserting the following data.
In the *Name* textbox insert *ERASE FLASH (STR71x)*. The *Location* and the *Working Directory* textboxes are identical to the ones of *OpenOCD (STR71x)* shown before: in fact in the *Location* textbox you have to insert *C:\Programmi\openocd\bin\openocd-pp.exe*, while in the *Working Directory* textbox you have to insert your *openocd-configs* path (in our case *C:\openocd-configs*).
In the *Arguments* textbox insert *-f str71x-configs\str71x_pp-flash-erase.cfg*, to specify the file to give to OpenOCD for FLASH erasing.
These settings can be seen in the following picture:



Obviously the same operations have to be executed for the other microcontrollers, too.
Finally a little tip: don't you want to organize these “External Tools” to call them more quickly, since their use is frequent? This is possible: you have to go to the menu *Run->External Tools->Organize Favorites*, as shown in the following picture:

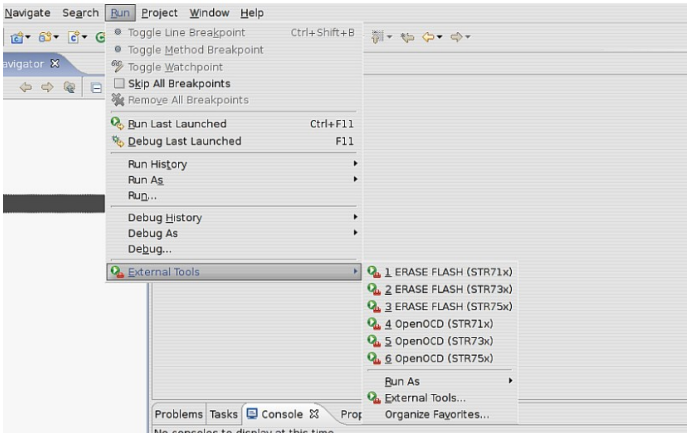


In the window that will appear, click on the *Add* button. A new window will appear where you can select the tools you want to have among the favourites: when you have done press *OK*. The three windows below show what we have just said:



Now, to call those programs, it's sufficient to go to the *Run->External Tools* menu and select the tool

you want.

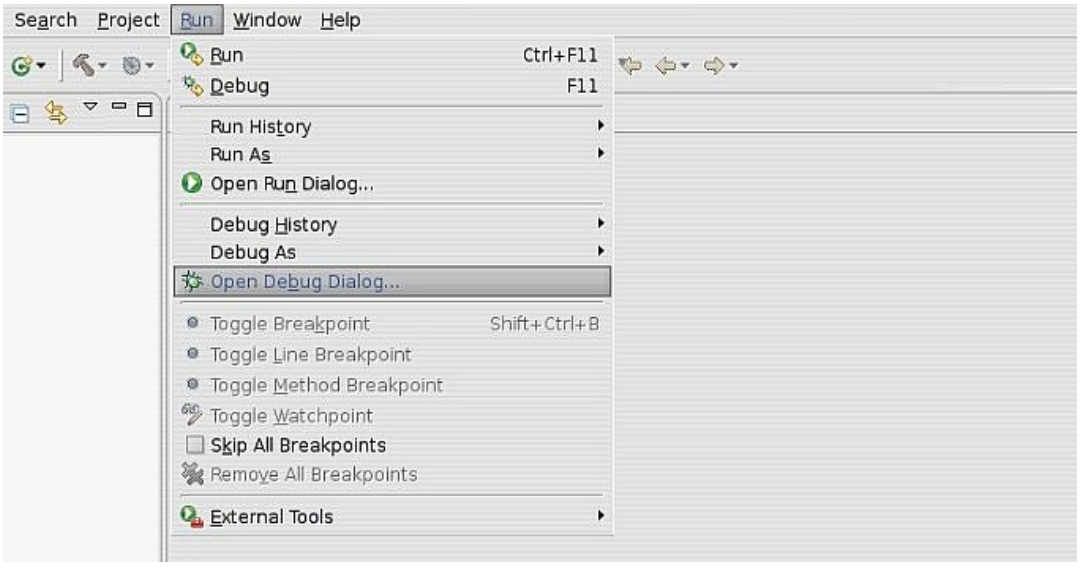


Debug configuration in Eclipse

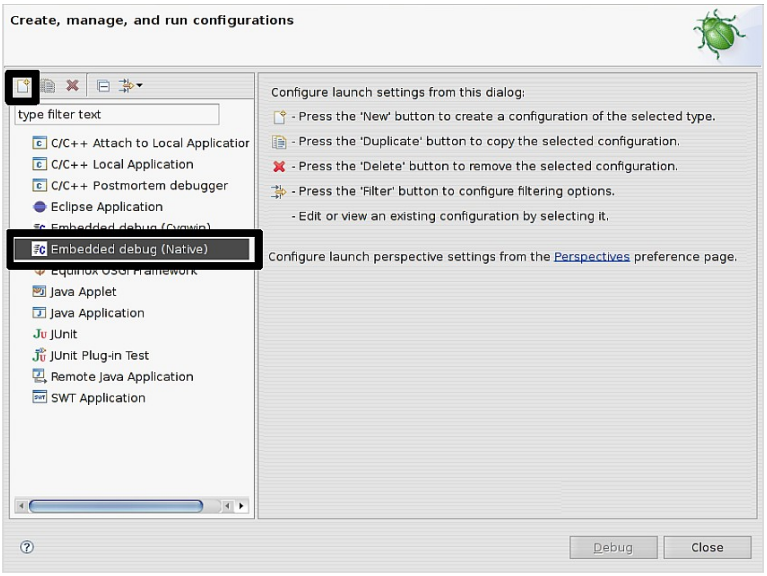
This paragraph explains how to configure Eclipse in order to correctly debug all our examples, it's a maybe annoying procedure, but you have to do it only once.

The objective is to create a configuration model for each used microcontroller, to debug both in RAM and in FLASH so, when you want to debug a project, you will only have to select the correct model and insert in it the project name, because the other parameters will be already present.

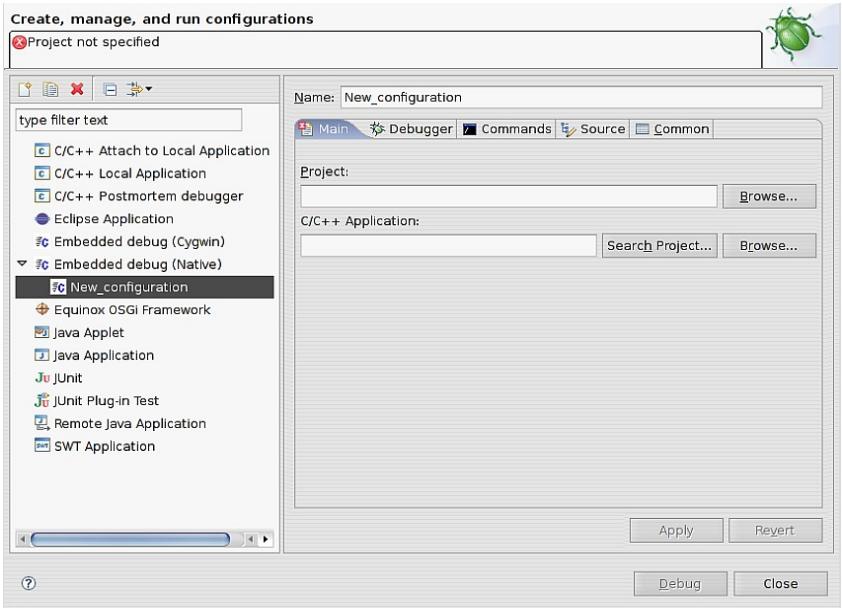
Let's start. Go to *Run->Open Debug Dialog...* menu as shown in the picture:



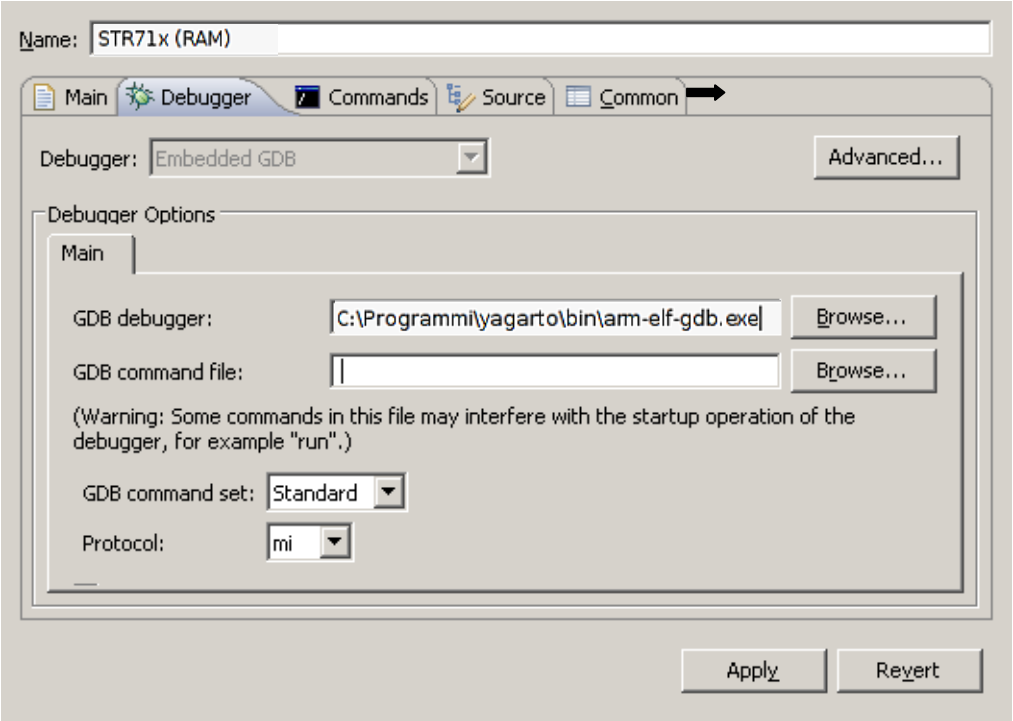
A window will appear, where it's possible to set the debug configurations for the programs in the workspace.
Click on *Embedded Debug (Native)* and then on the *New* button, as shown below:



The result will be the following:



On the right, you have all the properties of your project, let's set them.
First of all, the name: insert *STR71x (RAM)*. Below, in the *Main* section, don't modify anything.
Go to the *Debugger* section and fill it as shown in the following picture:



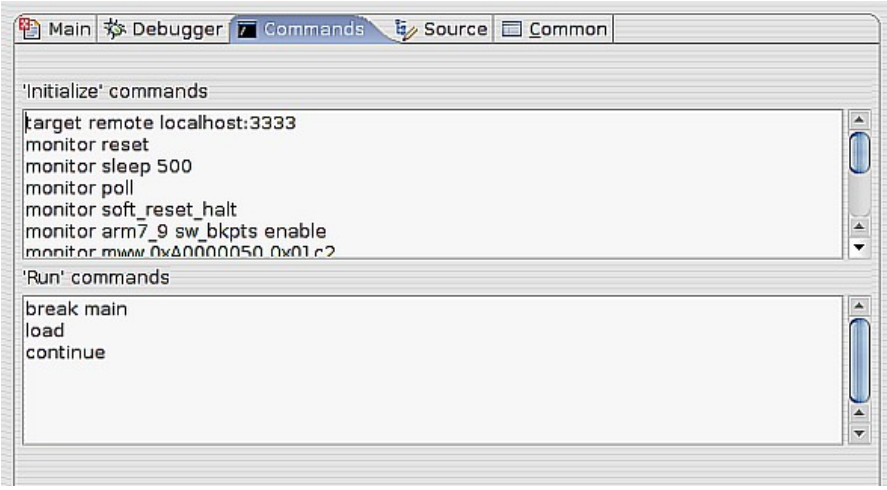
Insert in the section *GDB debugger* the *arm-elf-gdb* path (in our case *C:\Programmi\yagarto\bin\arm-elf-gdb.exe*) and remove *.gdbinit* from the section *GDB command file*. You continue with the *Commands* tab where there are two textboxes, *Initialize commands* and *Run commands.* In the first one it's necessary to insert the contents of the corresponding *gdb* file.
In this case go to the folder *openocd-configs\str71x_configs* and edit the file *str71x_ram.gdb* that has the following content:

```
target remote localhost:3333
monitor reset
monitor sleep 500
monitor poll
monitor soft_reset_halt
monitor arm7_9 sw_bkpts enable
monitor mww 0xA0000050 0x01c2
monitor mdw 0xA0000050
monitor mww 0xE0005000 0x000F
monitor mww 0xE0005004 0x000F
monitor mww 0xE0005008 0x000F
monitor mww 0x6C000004 0x8001
monitor mdw 0x6C000004
```

As we have just said, this content has to be copied in the *Initialize Commands* textbox.
In the *Run Commands* textbox, instead, you have to insert the following commands:

```
break main
load
continue
```

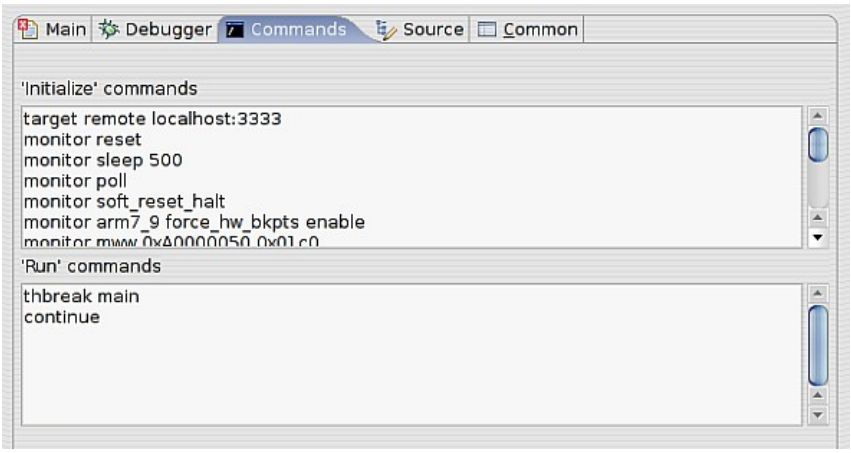
So finally this section will appear as shown below:



Now click on *Apply*.
Now you will do the same steps to configure this microcontroller for FLASH use.
Here, again, select *Embedded debug (Native)* and press the *New* button. Then insert *STR71x (FLASH)* in the *Name* textbox inside the *Main* section, the *Debugger* textbox has to be modified as shown for the previous example.
About the *Commands* section, in the *Initialize Commands* textbox insert the content of the file *str71x_flash.gdb* that is in the same folder of the file previously analysed, while in the *Run commands* textbox insert the following rows:

```
thbreak main
continue
```

So finally this section will appear as shown below:



Obviously it's necessary to repeat the above described operations also for the other microcontrollers, configuring both RAM and FLASH and using each time the proper files.
Some words about **the simulator**. What's that? Would you like to have a simulator that allows you to do a preliminary debug of your programs without using your board? There's a way to do that, although it can't be considered a complete simulator.
In fact, in the list of targets you can choose, GDB also offers the option *Simulator*, but it simulates correctly only ARM cores and not the other peripherals. Then you can't watch LEDs switching on and off or try a serial communication, but you can watch registers and variables and the values they

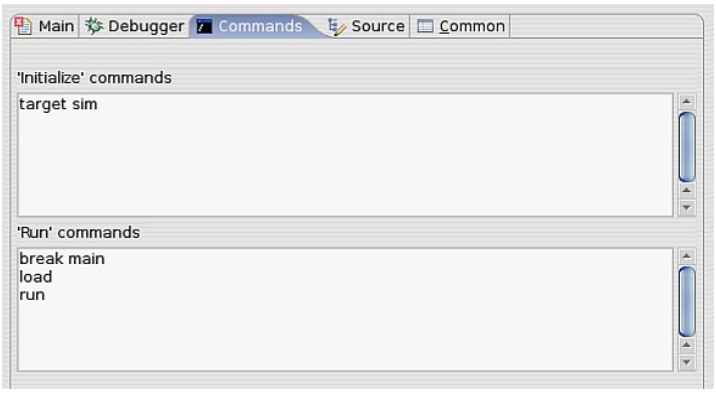
assume during program execution and you can also set all the breakpoints you like.
To configure the simulator you have as usual to select *Embedded Debug (Native)* in the *Debug* window and press then the *New* button.
In the *Name* textbox insert *Simulator*, don't modify the *Main* section and modify the *Debugger* section as seen in the previous configuration.
In the *Commands* section, in the *Initialize Commands* textbox, insert the following line:

```
target sim
```

while in the *Run Commands* textbox write:

```
break main
load
run
```

The window will be, finally, similar to the following:



Click on *Apply*. That's all, so close the *Debug* window by clicking on the button *Close*.

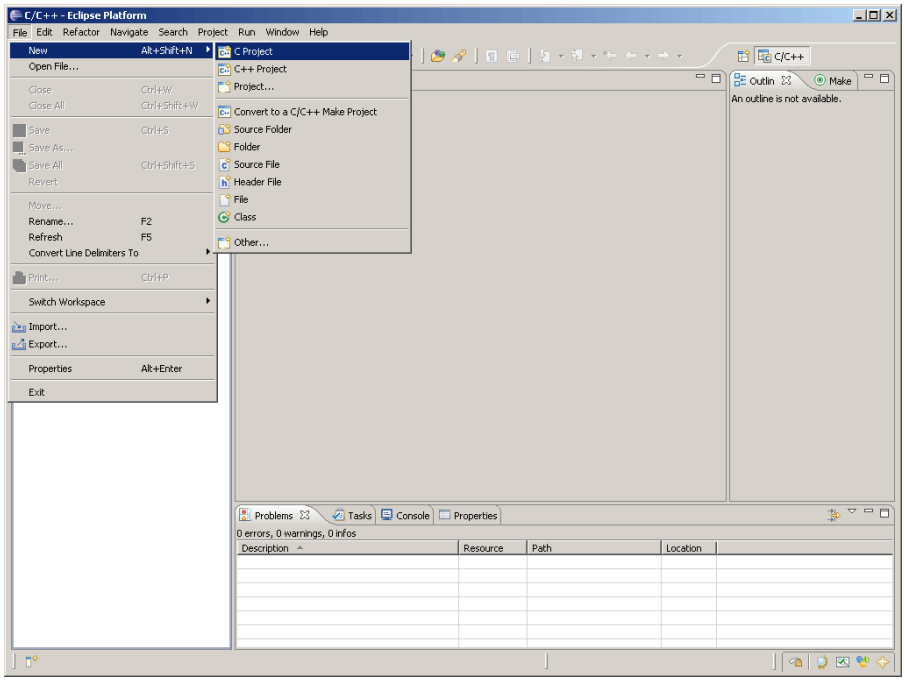
2.1.5 Eclipse: creation, compilation and debugging of a project

Now let's see how to create, compile and debug a new project in Eclipse.

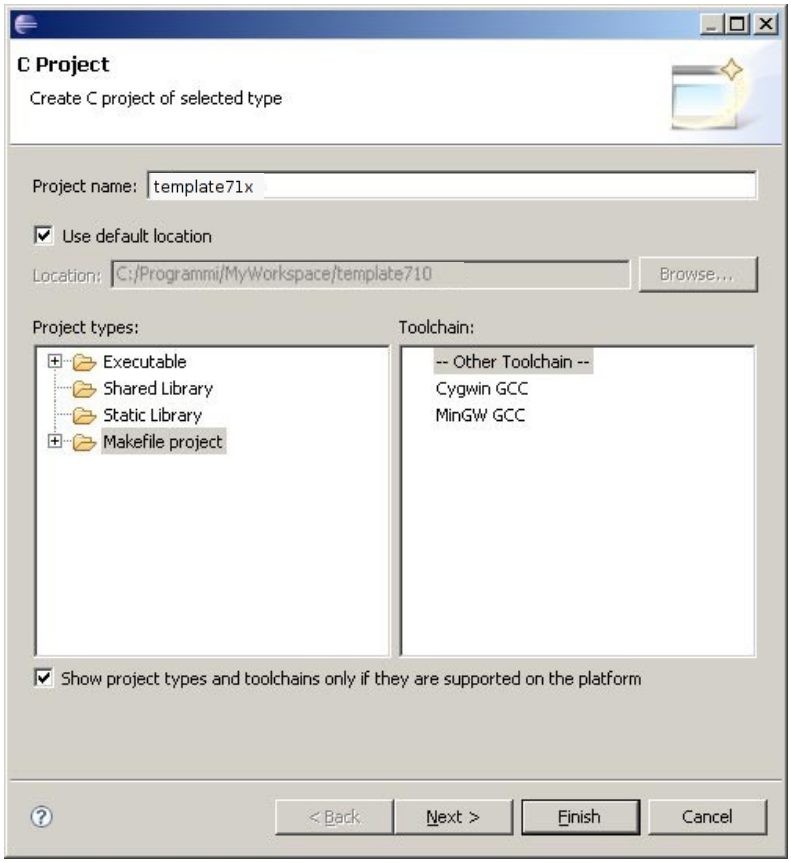
We will choose as usual a model (in this case the STR71x microcontroller), but the operations are the same for the other microcontrollers (except for some changes, such as file or folder names).

Creation of a new project

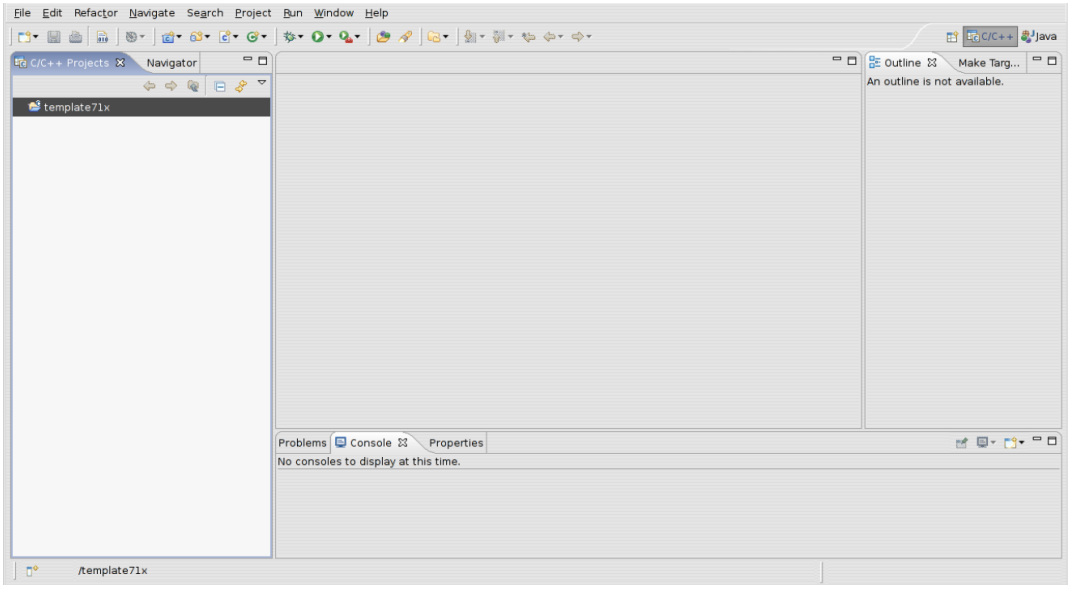
Before creating a new project, you have to open the menu *Project* and deselect the option *Start a build Automatically*, so Eclipse won't try to compile the project automatically, but only when asked.
To create a new project in Eclipse you have to go to the menu *File*, select *New* and finally choose *C Project* as shown in the following picture:



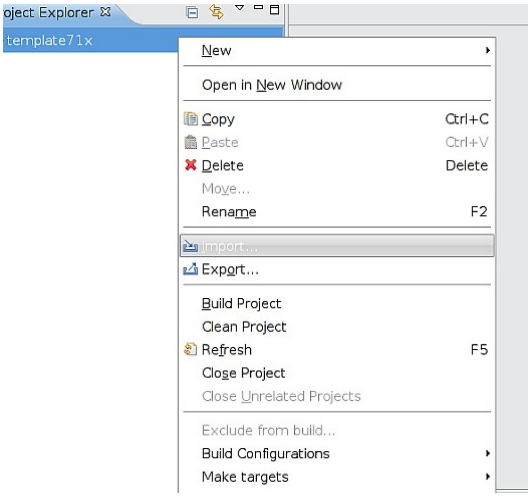
In the window that appears you have to insert the project name (in our case *template71x*) and the location, that is by deafult your workspace. Click on *Makefile project* in the *Project types* section and on -- *Other Toolchain* -- in the *Toolchain* section. The following picture shows what we have just said:



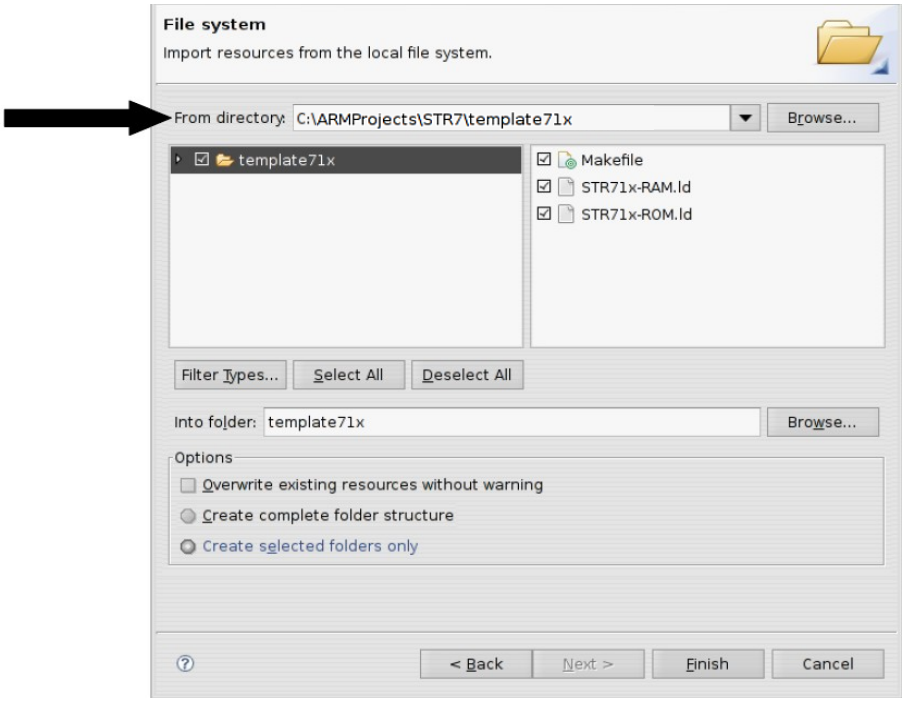
Then press the *Finish* button. In the following picture you can notice how the C/C++ perspective shows now the just created project:



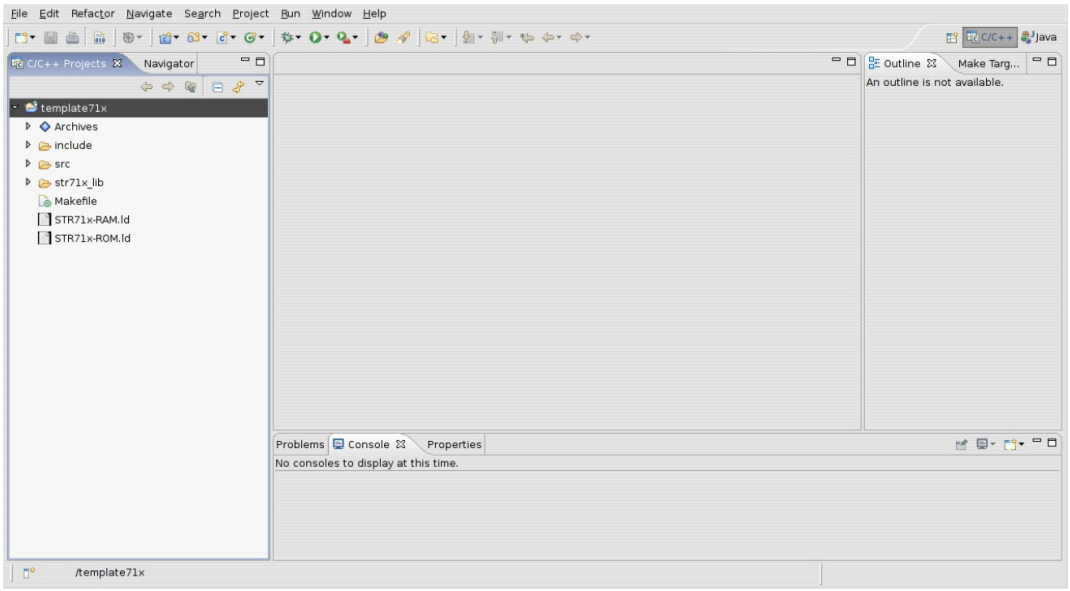
Now go to the menu Project and uncheck *Build automatically*.
To add a source file to the project it's necessary to go to *File->New->Source File*, but in this section we are going to import an already existing project.
To do that, right-click on the project name and select *Import*, as shown in the following picture:



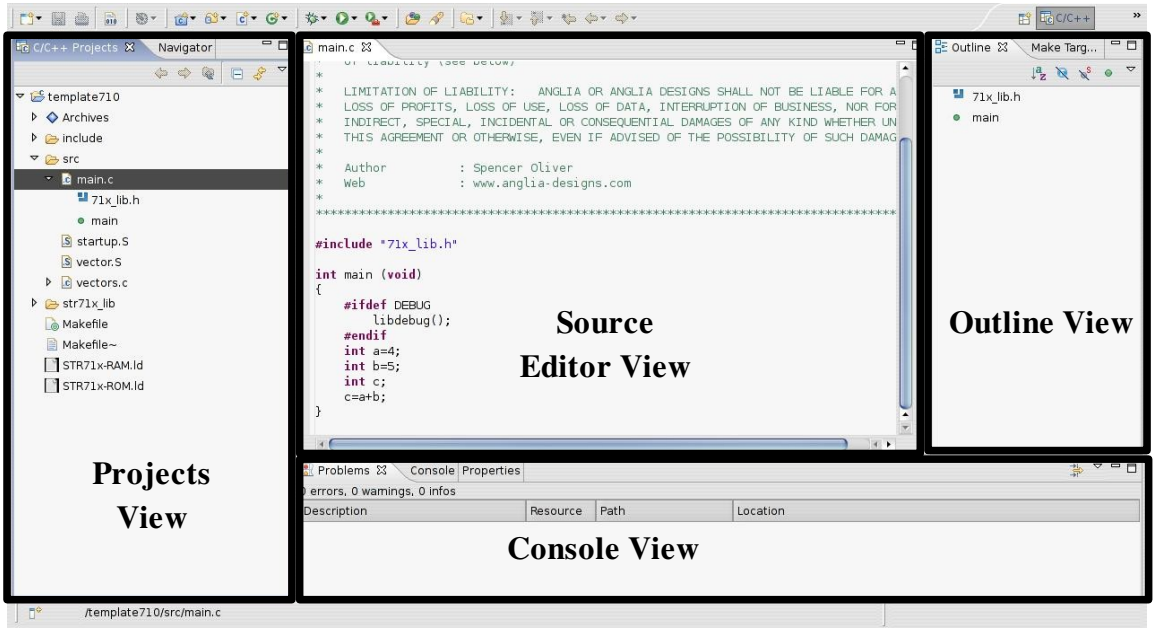
In the window that will appear you have to go to the folder *General*, select *File System* and press *Next*. The following window will appear:



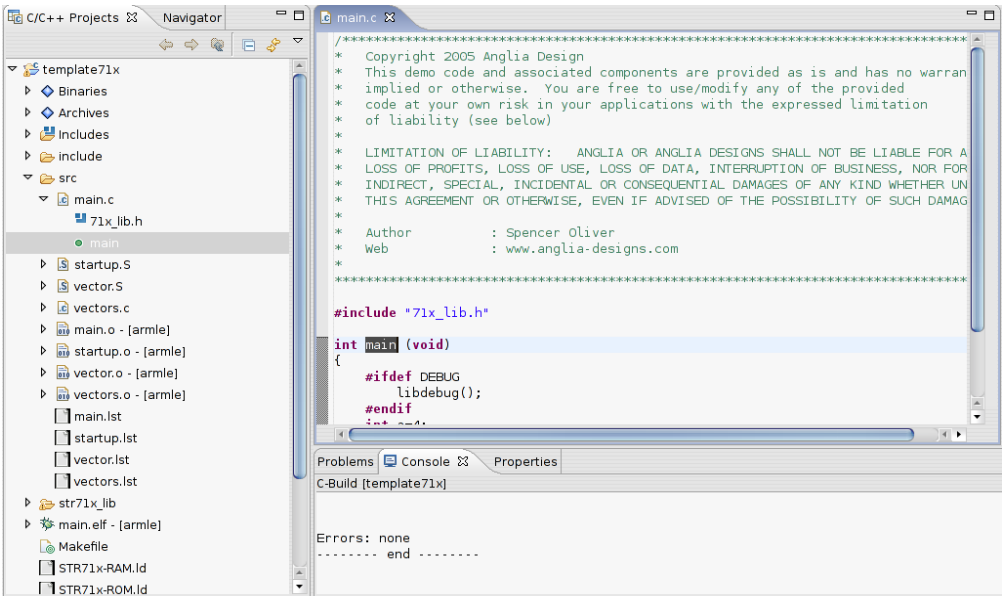
In the textbox *From directory*, as you can see from the picture above, it's necessary to insert the project folder, that you can search using the *Browse* button.
As soon as the folder of *template71x* has been selected (in our case *C:\ARMProjects\STR7\template71x*), you will be redirected to the window *Import*; in the left column, which is below the project path, check the box near the project folder name and all the files contained in it will be automatically selected. All these files will be show in the right column.
When you click on *Finish* , you will complete this phase.
Expanding the tree of the project *template71x*, you can see all the imported files, as shown in the following picture:



Now, it's possible to see the content of each file by double clicking on it. Source files are in the folder *src*, below you can see the content of the file *main.c*.



In the *Projects View* you can click on any source file and the *Source File Editor View* will be automatically updated.
Moreover, by seeing source files in the *Projects View* you can also see all the single variables and functions contained inside: by clicking on the symbol on the left of each file (it should be a little "+" or something similar), you will see a list of what is contained in the file (you can see the same list in the *Outline View*); for example, by clicking on the symbol on the left of main, you will automatically see in the *Source Editor View* the function *main*, as shown in the following picture:

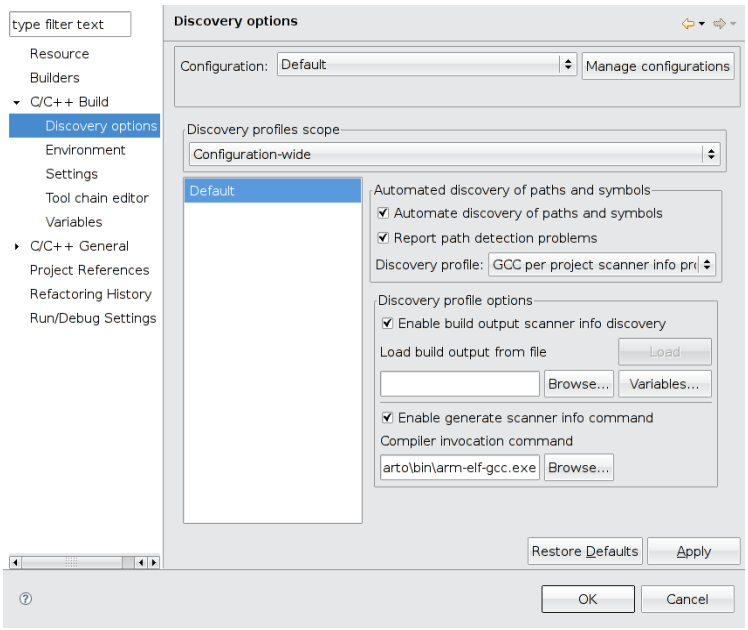


This feature is very useful to avoid scrolling very long and complex source files.

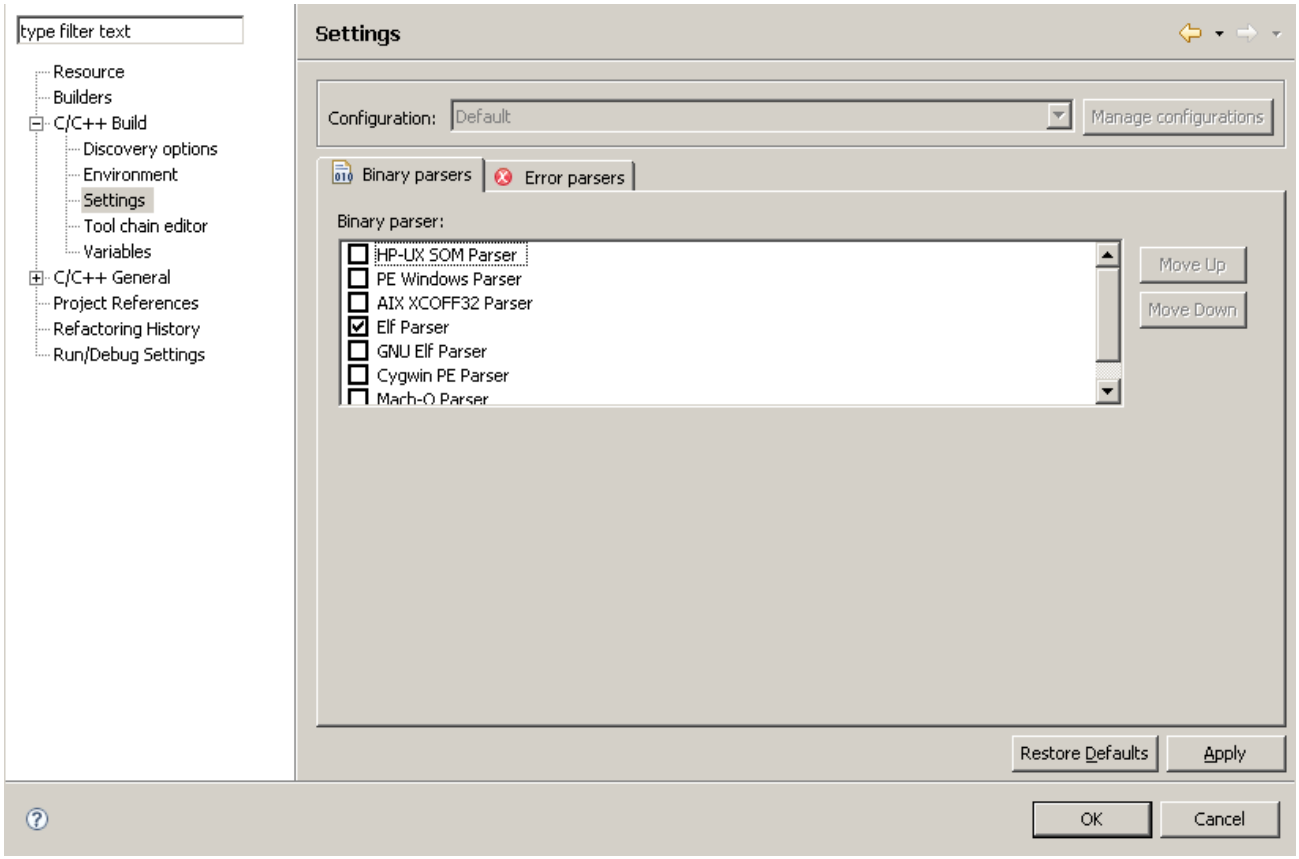
Project building

To build a project you need to tell Eclipse where it can find the right compiler: to do that, right-click on the project, choose *Properties* and, in the window which opens, go to *C/C++ Build->Discovery Options*.

Then you have to insert the compiler path in the box *Compiler invocation command* (in our case *C:\Programmi\yagarto\bin\arm-elf-gcc*), as shown in the following picture:



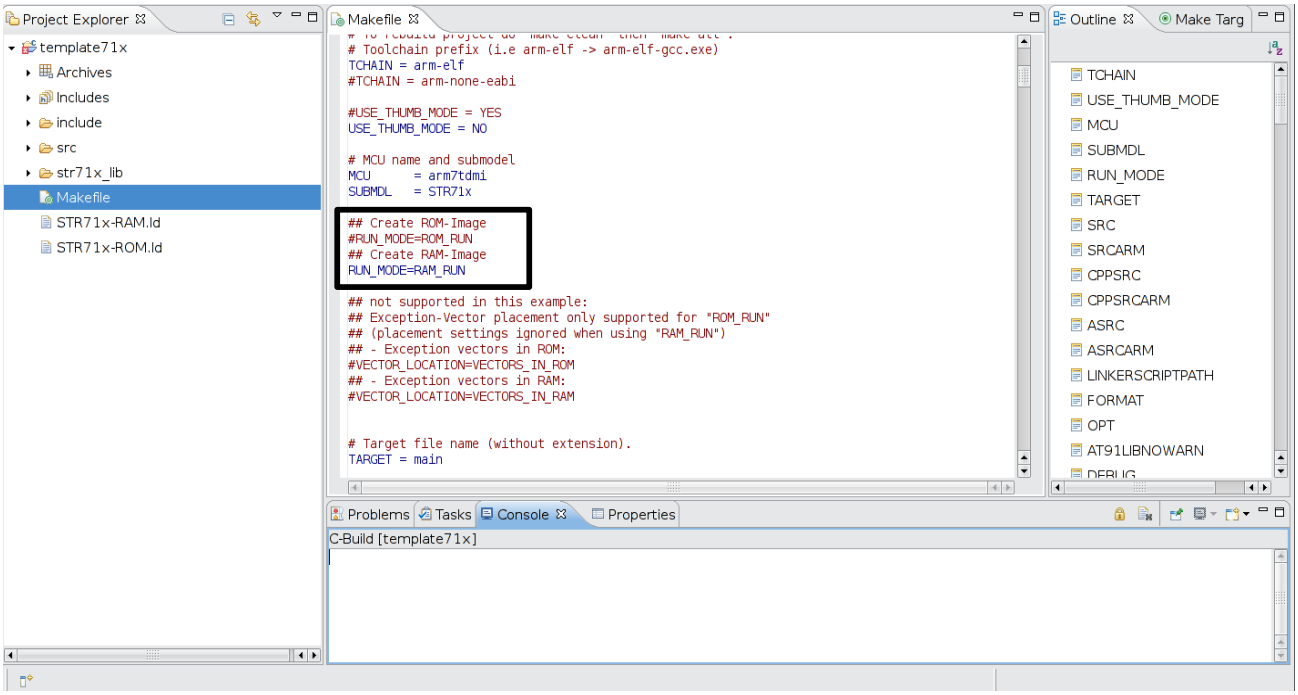
After that, move to *C/C++ Build->Settings*. In the window which opens select *Elf Parser*, as shown below:



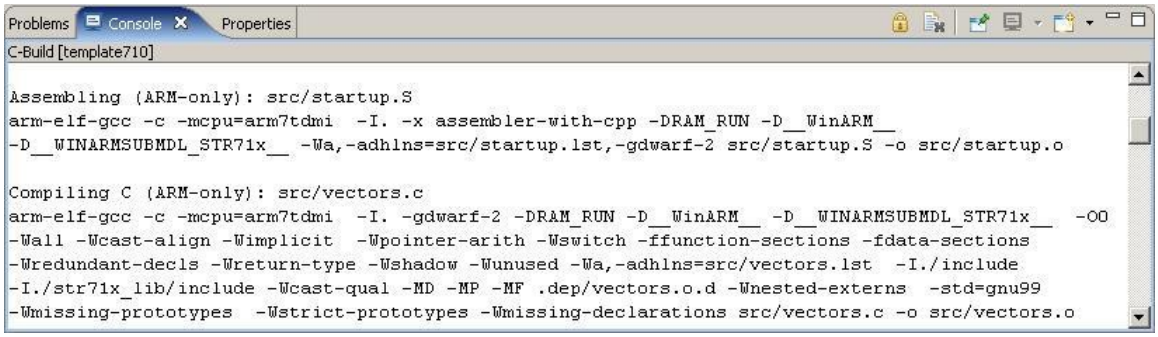
Then press the OK button.
Now let's clean the project: right-click on the project name and choose *Clean Project*. So the

Console View will appear (in particular look at the *Console* tab) after the *make clean* execution.

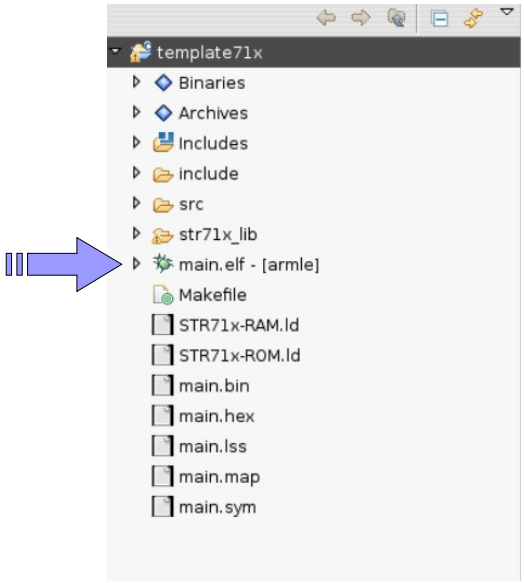
Before building the project, you have to choose if your program will execute in RAM or in FLASH. To do that, you have to modify the right section in *Makefile*, as you did in 2.1.2 paragraph; you can edit the *Makefile* directly from Eclipse through a double click on it.
In the section shown in the picture, you have to uncomment your choice, commenting the other one: for example in this case we chose RAM.



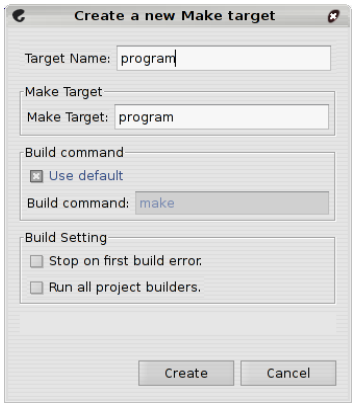
Save changes in Makefile and then build the project by doing right-click on the project name and choosing *Build Project*.
The *Console* tab of the *Console View* shows the *make* execution, which contains the phases of assembling, compiling and linking of the project.



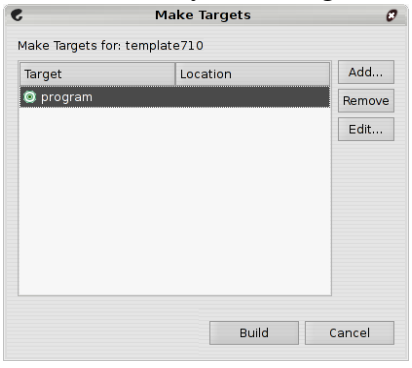
If there are problems, it's possible to see them both in the *Console* tab and in the *Problems* tab which gives more information about what went wrong.
After building has been correctly executed, a file *main.elf* is created, as shown in the following picture:



If you chose FLASH in the Makefile, you have to download the program in it (you have to launch *make program*).
To do this operation, you have to create a "*Make target*": this is done by right-clicking on the project name and by choosing then *Make Targets->Create*.

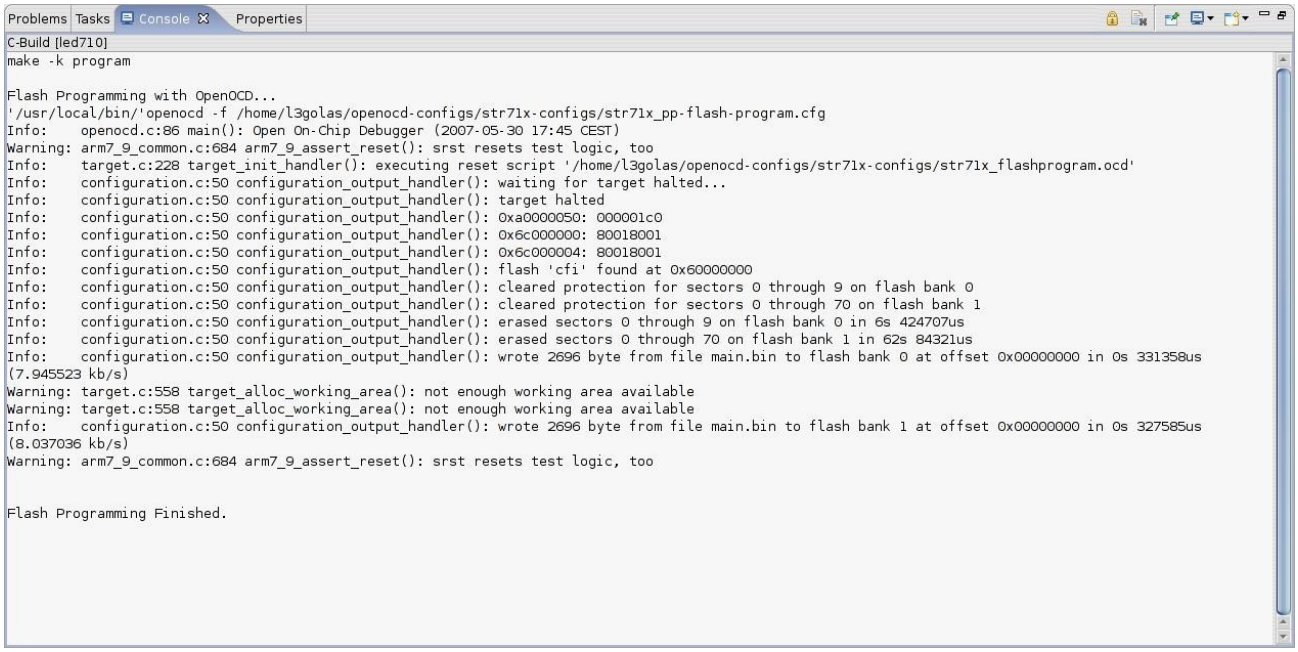


One window will appear, where you have to insert the options shown in the picture on the left.
Don't forget to deselect the checkbox *Run all projects builders*.
Finally press *Create*.
Now, by right-clicking on the project name and by choosing *Make Targets->Build* you will have the window on the right where you have to select your target.

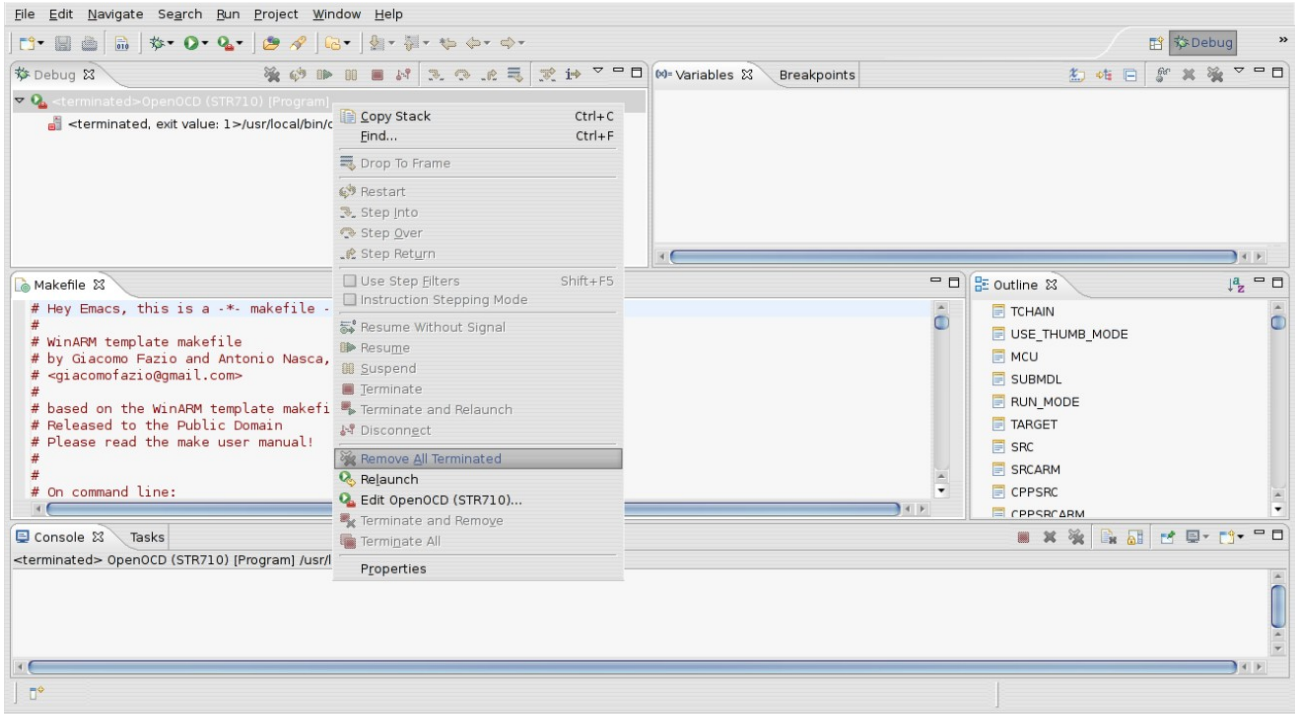


Before pressing the *Build* button, choose *Edit* and make sure all the settings are the same as those you put when you created the target (Eclipse sometimes decides to restore the checkbox *Run all project builders*). Now you can finally click on the *Build* button and in the console you will see the FLASH programming

result, similar to the following:

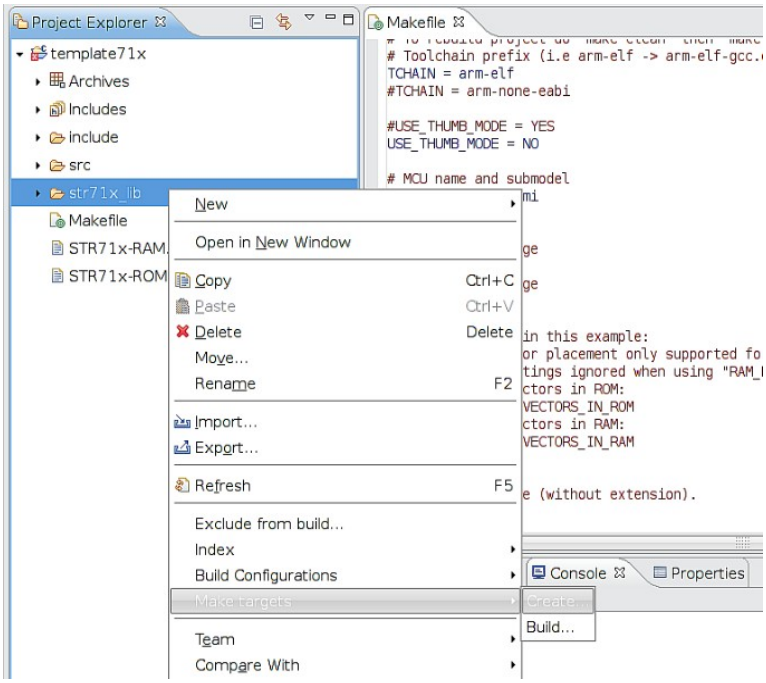


To finish, after having written successfully FLASH memory, switch to the *Debug* perspective by going to *Window->Open Perspective->Debug*.
The following window will appear, right-click on "*<terminated>OpenOCD...*" in the *Debug* section and choose *Remove All Terminated*, as shown in the following picture:

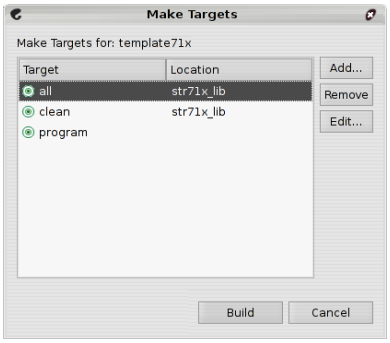


Now come back to the *C/C++* perspective.
Last note about software library: probably you will rarely modify and recompile the software library, however it could happen, for example if you modify some settings in the *71x_conf.h* file (for more details see chapter 3).

In that case you can create two “Make Targets” on software library folder, one for the *clean* and the other one for the actual *make*.
To do that, right-click on the *str71x_lib* folder and choose *Make Targets->Create*, as shown in the following picture:



Now insert the two targets *clean* and *all*, following the same steps specified before for the target *program*. So the two new targets will be added to the target *program* already present in the main project. Now when you want to perform a clean and then a recompilation of the software library, it will be sufficient to right-click on the project name and choose *Make Targets->Build*. You will see the following window



where there will be both the two targets of the software library (you can notice that under *Location* there's *str71x_lib*, which helps you in distinguishing the targets) and the target *program* of the main project.

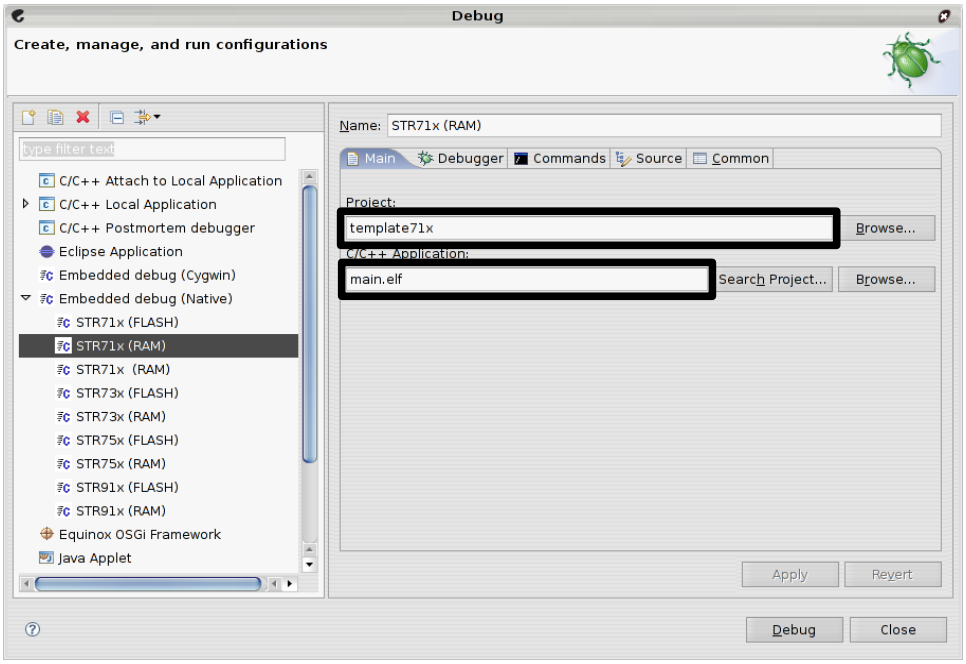
Project debug

Now let's see how to debug the project you have just built (and eventually, already downloaded in FLASH). As usual, we take as example *template71x*, but the operations are the same also for the

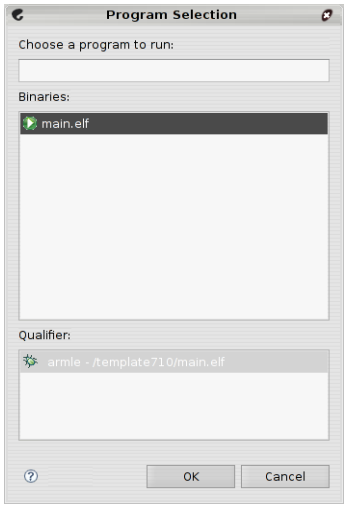
other projects of the other microcontrollers.
Let's start doing the debug in RAM: the first thing to do is to set the right section in the Makefile to make the program run in RAM and rebuild it.

In the debug settings you have created the various models of microcontrollers, where you have to add only the project name. Then whenever you have to debug a project, you have to use the correct model for your microcontroller and add the project name, as you will see by reading below.

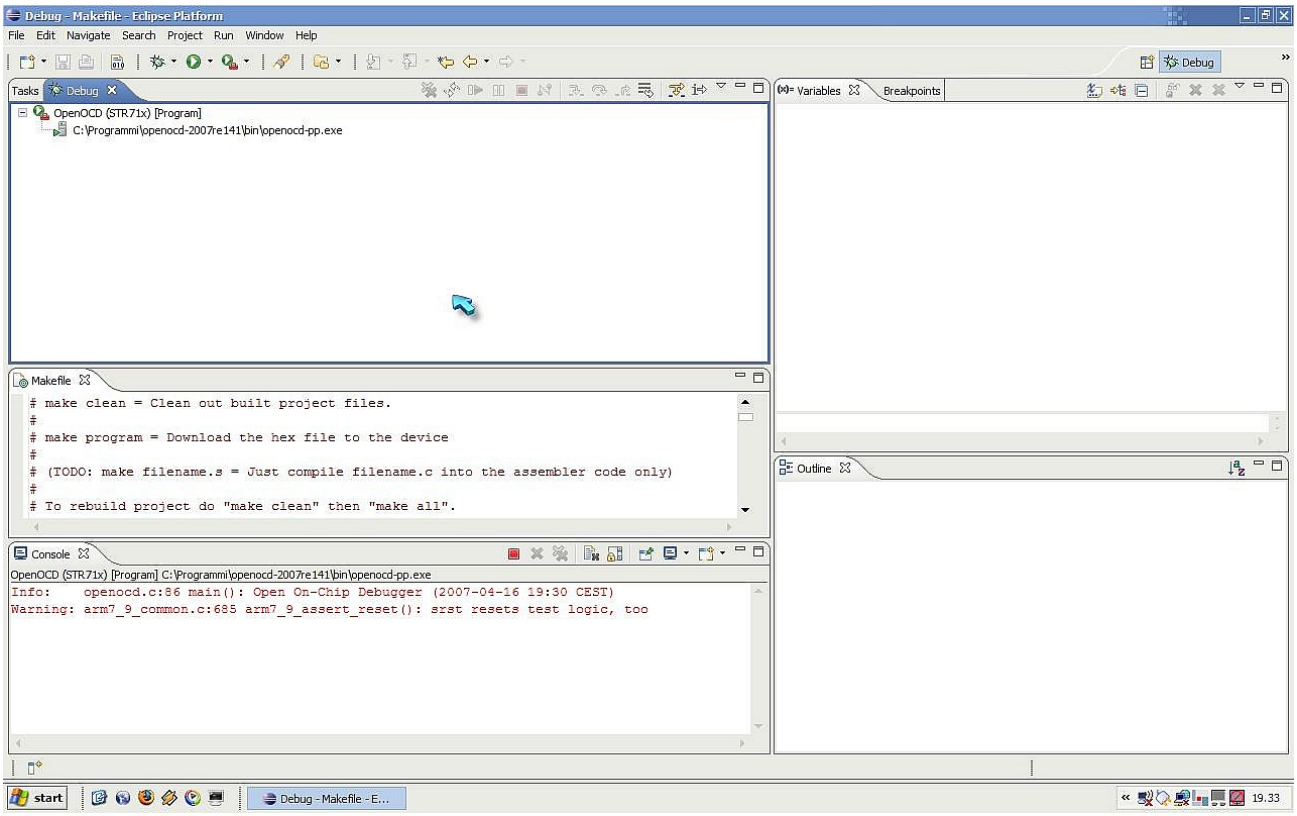
To debug your program we will use *STR71x (RAM)* model, to which we will add the project name: go to the *Run->Open Debug Dialog...* and select *STR71x (RAM)* from the window that will appear. On the right side, go to the *Main* section.
Here you have to fill the two textboxes as shown below:



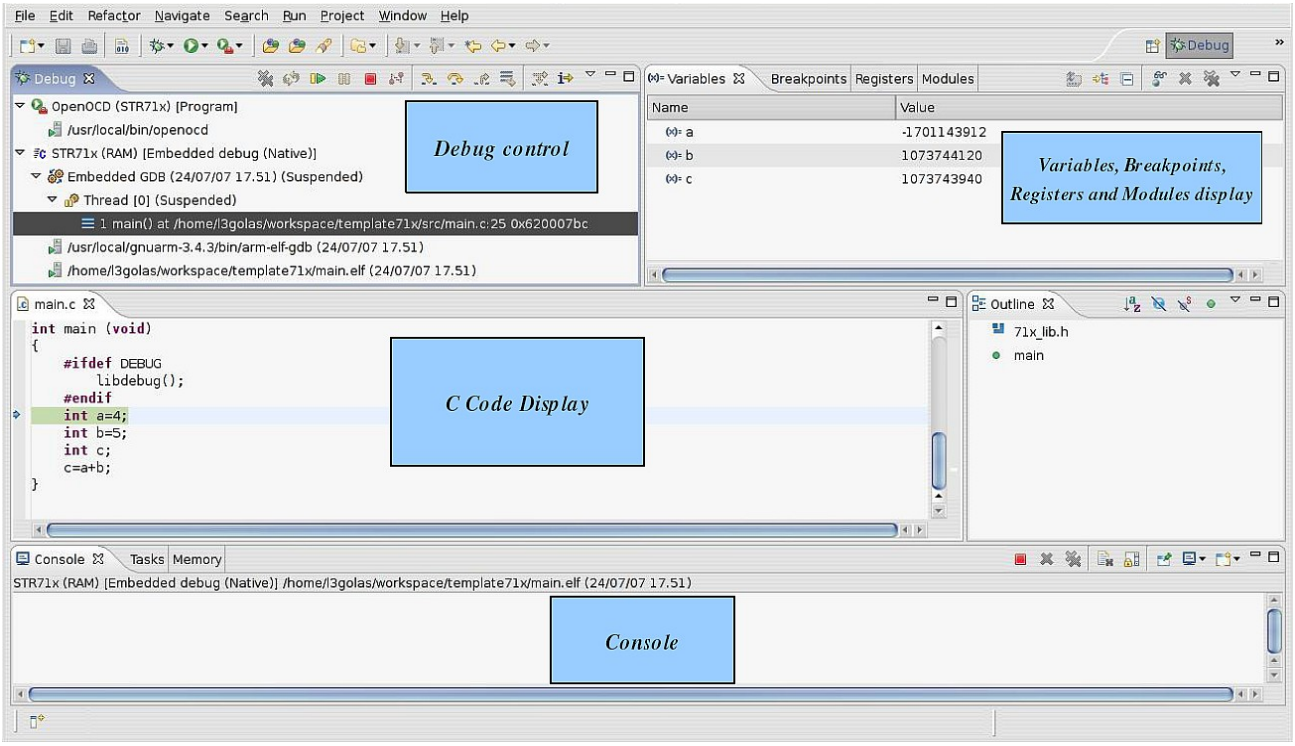
In particular, to fill the first one, click on the *Browse* button located on the textbox's right side and select the project you want to debug; to fill the second one instead, click on the *Search Project* button and select the file *main.elf*, as shown in the following picture:



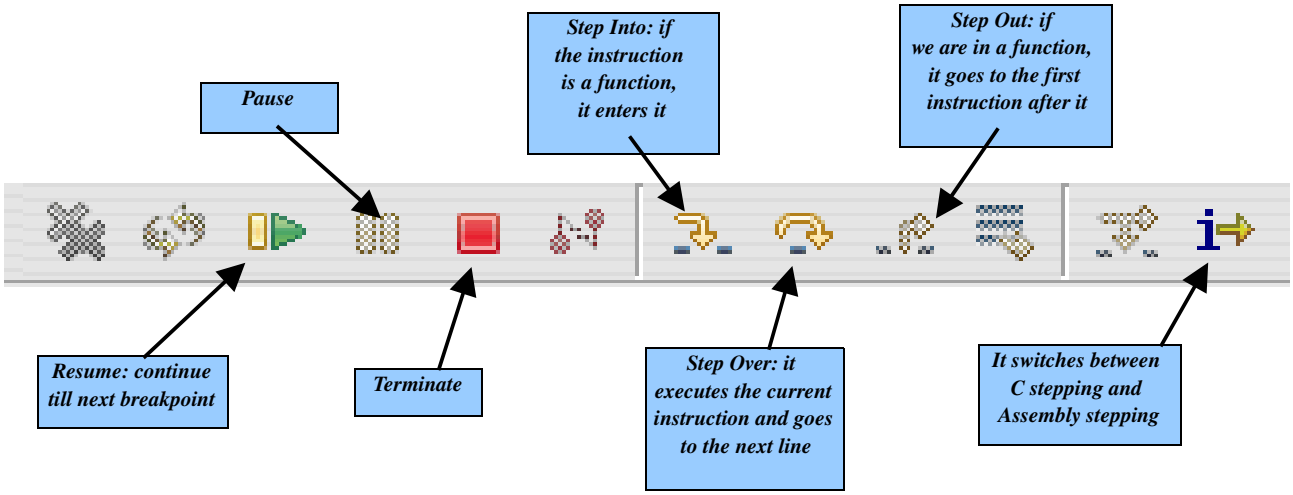
Click on *Apply* and finally on *Close*.
Now you can start debugging. After having powered on the board, switch to *Debug* perspective, go to *Run->External Tools* menu and select *OpenOCD (STR71x)*. If all went fine, in the Console you will see that OpenOCD has blocked waiting for an input, while in the Debug section on the upper-left side, OpenOCD executes, as you can see in the following picture:



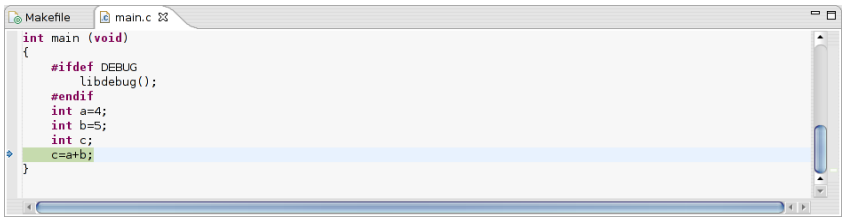
Now that OpenOCD is running, let's start the actual debug phase, which is based on GDB.
Go to *Run->Open Debug Dialog...*, select *STR71x (RAM)* and click on the *Debug* button. The program should block on the first instruction after the *main*, waiting for input.
To clarify what explained, look at the following picture, where the fundamental parts of the Debug perspective are indicated, to help the debugging operations:



Now you can control the debug through the buttons situated in the Debug section:



So if you click on the *Step over* button, you will execute only one instruction (in the shown example also *Step Into* will do so because the instructions aren't functions) and the program will block on the following instruction.

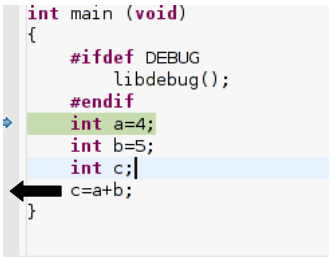


In the source code box, the next instruction to execute is always green-colored with an arrow on the left side.

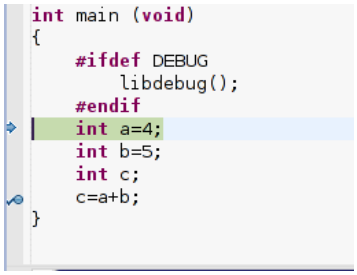
If the code calls a function, the Step over button will execute it, without letting you debug the inner code of the function, while the Step Into button will enter it, debug its code and return back after completing the execution; if the function is defined in another file, the Step Into button will automatically open it.

You can also set breakpoints (since you are working in RAM, you use software breakpoints and so

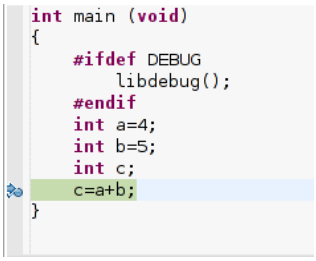
there is no limitation in their number) so, if you press the *Resume* button, Eclipse will continue the code execution and will block before to execute the instruction where you put the breakpoint. To set a breakpoint you have to double click on the chosen row's left, as shown in the following picture:



Set the breakpoint where indicated by the arrow, that is the instruction c=a+b

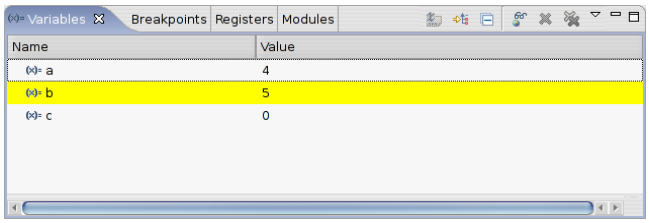


The blue little ball indicates the just set breakpoint. Now press the button Resume



As expected, the execution stopped just before the required instruction. Press Step Over to execute it.

If you look at the *Variables* tab, shown below, you will find the allocated variables with their value, that is updated in real-time.



To finish the execution, select *STR71x (RAM)* and press the button *Terminate* (or right-click and choose *Terminate and Remove*) and then do the same with OpenOCD.
At the end you can also right-click and choose *Remove All Terminated*.

Don't choose *Terminate All* instead of the just described procedure, because it would delete at the same time both GDB and OpenOCD and this can cause errors or slowdowns in Eclipse; instead you should at first terminate GDB and only then terminate OpenOCD.

Obviously the analysed code is deliberately very simple and it doesn't use any I/O device, then you will see nothing on the board during execution; however, if this attempt was successfully executed, this means that all the settings are ok and you will be able to build and debug more complex programs, too.
Now let's try for FLASH the steps you have done for RAM (remember that after you have built the code and before starting the debug, you have to download it in FLASH, as explained before).
The difference with RAM debug is obviously that you have to use the STR71x (FLASH) model and that you use now hardware (and not anymore software) breakpoints, then you can use a maximum of two breakpoints (or one if you want to use *Step Into* and *Step Over*, since those functions use one breakpoint to stop on the following instruction).

2.1.6 Conclusion

If you have arrived here without problems, then you have properly installed and tested all the parts of the solution. Now, with the same procedure, you can try to import and debug other and more complex examples that are situated in the subfolder *STR7* of the folder *ARMProjects* provided together with this work: you can find at least one example for each analysed microcontroller.

You can also modify or create new examples using the microcontroller functions, described very well in the Software Library manual of each microcontroller; you can download it from the website http://mcu.st.com/mcu. Make sure also to read the sections about how to develop a program in the chapter 3.

The provided examples are relative to the boards we used: for example the small program that switches on and off LEDs does that activating bits of one or more GPIO ports that, in our board, are connected to those leds; however in other boards they could do other things and the LEDs could be connected to different GPIO ports.
So check the code of each example and adapt it to your board, the things to modify shouldn't be so many.

2.2 Tutorial for Linux users

Copying the files
You have to copy the two folders *ARMProjects* and *openocd-configs* contained in the folder *Code/Linux* and paste them in your home.

2.2.1 OpenOCD

Installation and configuration

Now you have to install OpenOCD. To do that, you have to download it through a terminal window, in which you have to write

```
svn checkout svn://svn.berlios.de/openocd/trunk
```

If the command doesn't produce any result, maybe you have to install *Subversion*; for example, for Debian, Ubuntu and Debian-like distributions, you have it in your repositories, so you can easily install it through the command:

`apt-get install subversion` **(as root)**

For other distributions, there will surely be a similar procedure.
If you have other problems with OpenOCD installation, visit the website http://openocd.berlios.de/web/ and follow the instructions. If all is ok, you should have a directory called *trunk*. Now, run the following commands:

`cd trunk` (to enter the directory)
`./bootstrap` (to create the configuration script)
`./configure` (to generate the Makefile), followed by at least one of the following options:

- `--enable-parport`
- `--enable-parport_ppdev`
- `--enable-amtjtagaccel`
- `--enable-ft2232_ftd2xx`
- `--enable-ft2232_libftdi`
- `--with-ftd2xx=/path/to/d2xx/`

Both the first and the second option set OpenOCD to use parallel port, the difference is that the first one requires root privileges so, for a normal user, our advice is to choose the second one (pay attention that the second option must be used together with the first one, because it's only an option to the *parport* driver).
So, since we decided to use parallel port without having root privileges, the proper command is this:

```
./configure --enable-parport --enable-parport_ppdev
```

To use other solutions, you have to download the proper driver among the following sites, depending on your solution:

- ftdi2232: libftdi (http://www.intra2net.com/opensource/ftdi/)
- ftd2xx: libftd2xx (http://www.ftdichip.com/Drivers/D2XX.htm)
- Amontec JTAGkey: http://www.amontec.com

`make` (to compile files)
`make install` (**as root**, to install the files compiled in the previous step in the proper directories)

Even if you don't need to have root privileges to use OpenOCD, it's necessary to have reading and writing rights on parallel port: usually everybody has reading rights on it, but only root has writing access (for security reasons); so it's necessary to ask root to enable both the rights for people who want to use this solution; root can do that through the command

`chmod a+rw /dev/parport0`
(**Note:** the file which identifies parallel port could have a different name)

However this command enables the rights for everybody! To avoid that, it could be better for root to add the users to whom to give the rights to his own group and then to enable the rights only for the group and not for everybody. To insert the name *USERNAME* to the group *GROUPNAME*, the command is the following:

`adduser` *`USERNAME GROUPNAME`* **(as root)**

After that, to enable the rights only for the group:

`chmod g+rw /dev/parport0` **(as root)**
(**Note:** the file that identifies parallel port could have a different name).

Before using OpenOCD, it's necessary to unload the module *lineprinter*, through the command

`rmmod lp` **(as root)**

It could be also necessary to enable manually the module *ppdev*, through the command

`modprobe ppdev` **(as root)**

To launch OpenOCD, it's necessary to specify the configuration file that describes the target parameters, through the command *openocd -f configfile*
configfile is the configuration file you choose among the configuration files provided together with this work: there are files for all the microcontrollers we analyzed and, for each of them, there are many alternatives, different for the way to connect to the target, even if we did our tests only using parallel port.
All those files, whose names are really intuitive, are located in the folder *openocd-configs* that you already copied in your home at the beginning of this tutorial.
For example, the file for the connection to a STR710 board using parallel port is called *str71x_pp.cfg*.
To use this file, after you have connected and powered on the board, type the command

`openocd -f` *`YOUR_HOME`*`/openocd-configs/str71x-configs/str71x_pp.cfg`

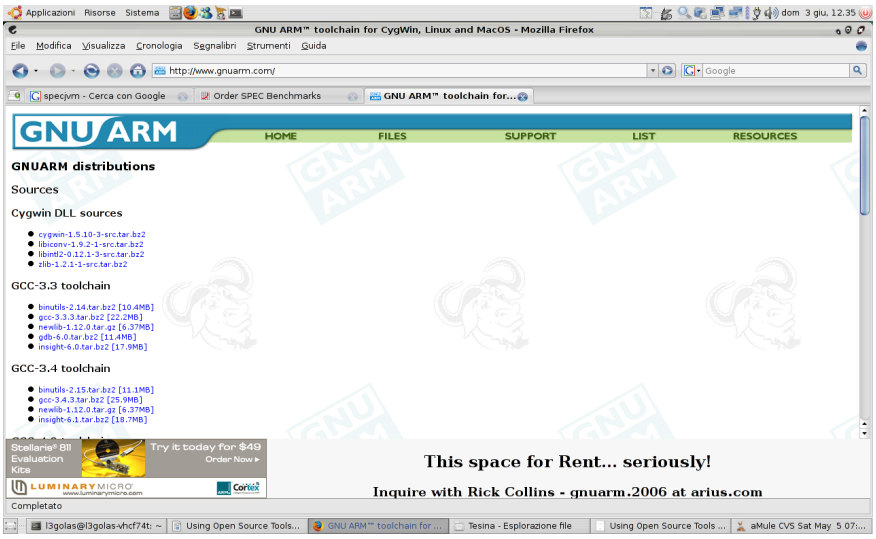
where *YOUR_HOME* must be substituted by your home actual path.
The program now should block waiting for an input. If the program exits (not for trivial reasons such as no power or configuration file not found or because of a jumper on the board that blocks

debugging), it's possible to have more information by launching again OpenOCD the same way, but adding the option *–d* which gives a more detailed output.
For some boards it might be necessary to modify some parameters in the configuration file: usually it's enough to lower *jtag_speed* value or to modify *trst_and_srst* parameter in *trst_only* or *srst_only* or in *none*.
If all is ok, for now you can terminate OpenOCD with the combination *Ctrl+C*.

2.2.2 GNUARM

Installation and configuration

The second step is to install the GNUARM toolchain, that you can download from the website www.gnuarm.com in the *Files* section, or from the site www.gnuarm.org in the same section: they are different mirrors and each of them seems to be updated independently of the other one. So check on both sites which one is the most updated version. In particular, there are two distribution types for the toolchain, both catalogued by version: *sources* (if you want to have the latest version and best performances, without worrying about manually compiling all the toolchain) and *binaries* (if you want to be ready to start without compiling anything, accepting at the same time that doing so, you won't perhaps have the most updated or optimized toolchain).



Toolchain compilation and installation

If you choose the first possibility, you have to download the source files of each toolchain part. It seems that the website that offers the most updated sources is www.gnuarm.org, which has already the latest version of each package, while the other mirror www.gnuarm.com has still a bit older versions. Our suggestion is to download latest versions, in that they could correct bugs and implement new features: in particular, if you want to try the microcontroller STR91x (see Appendix C) you must have the last version of Insight (6.6), which you can currently download only from www.gnuarm.org.
As follows you can see the necessary steps to compile and install the latest version of the toolchain (pay attention to the fact that the versions could be different, with minor changes in the filenames).
After you have downloaded the 4 required files, follow these steps:

```
1)  tar jxvf binutils-2.17.tar.bz2
2)  tar jxvf gcc-4.2.0.tar.bz2
3)  tar zxvf newlib-1.15.0.tar.gz
4)  tar jxvf insight-6.6.tar.bz2
5)  cd binutils-2.17
6)  ./configure --target=arm-elf --prefix=/usr/local/gnuarm --enable-interwork --enable-multilib
7)  make all
8)  make install (as root)
9)  export PATH="$PATH:/usr/local/gnuarm/bin"
10) cd ../gcc-4.2.0
```

```
11) ./configure --target=arm-elf --prefix=/usr/local/gnuarm --enable-interwork --enable-multilib
    --enable-languages="c,c++" --with-newlib --with-headers=../newlib-1.15.0/newlib/libc/include
12) make all-gcc
13) make install-gcc (as root)
14) cd ../newlib-1.15.0
15) ./configure --target=arm-elf --prefix=/usr/local/gnuarm --enable-interwork --enable-multilib
16) make all
17) make install (as root)
18) cd ../gcc-4.2.0
19) make all
20) make install (as root)
21) cd ../insight-6.6
22) ./configure --target=arm-elf --prefix=/usr/local/gnuarm --enable-interwork --enable-multilib
23) make all
24) make install (as root)
```

Now you can avoid reading next paragraph and go directly to the paragraph "*Make sure the toolchain is properly installed*".

How to use the toolchain binary version

We decided to use the simplest way (the binary version), because we considered the case of a newbie user who doesn't want to face eventual compilation problems.
In this case, visit both the sites www.gnuarm.com and www.gnuarm.org, then download the binary of the latest version you can, which is the version 3.4 for Linux x86 (32 bit processor) and the version 4.0 for Linux x86_64 (64 bit processor). Anyway, the only thing to take into account is that, if you want to try the STR91x microcontroller (see Appendix C), you must have Insight 6.6, but you can download and install it later, as explained below.
After having downloaded the binary toolchain file, extract it **(as root)**, through the command

```
tar xjvf bu-2.15_gcc-3.4.3-c-c++-java_nl-1.12.0_gi-6.1.tar.bz2
```

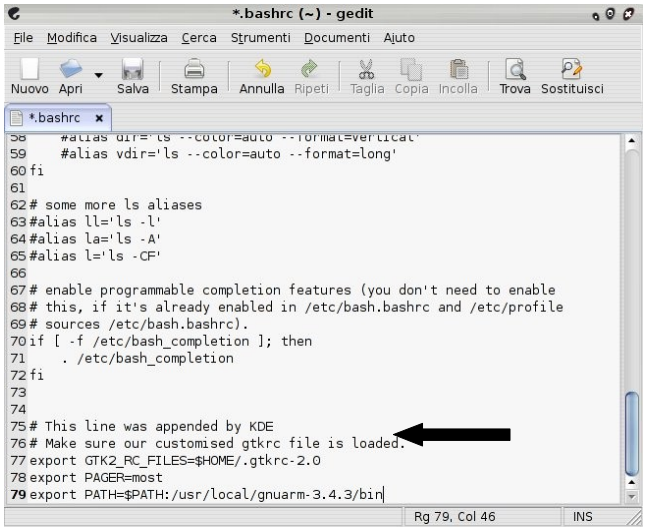
Note: The filename could vary a little bit, depending on the version you download.
Then you have to insert in the PATH the subdirectory *bin* of the toolchain, through the command

```
export PATH="$PATH:TOOLCHAIN_PATH/bin"
```

where *TOOLCHAIN_PATH* must be substituted by the absolute path of the toolchain (in our case for example the command becomes `export PATH="$PATH:/usr/local/gnuarm-3.4.3/bin"`).
This PATH insertion of the toolchain is temporary, it won't be anymore present when you open a new terminal session so, to make the PATH permanent, you can insert the command directly in the file `.bashrc` that should be in your home. For example in our case we edited the file *.bashrc* and we inserted the line

```
export PATH="$PATH:locazione_toolchain/bin"
```

at the end, as you can see in the picture on the right.



Optional step: as we already said, in order to use the STR91x microcontroller or only because you want so, you can install Insight most recent version and install it over your already present toolchain. Let's see how.
Download from www.gnuarm.org the package `insight-6.6.tar.bz2` from the section *Sources* and follow these steps:

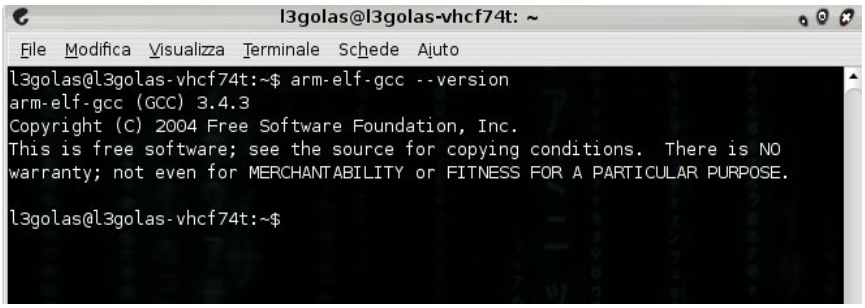
```
1) tar jxvf insight-6.6.tar.bz2
2) cd insight-6.5
3) ./configure --target=arm-elf --prefix=TOOLCHAIN_PATH --enable-interwork --enable-multilib
4) make all
5) make install (as root)
```

Make sure the toolchain is properly installed

Now make sure the toolchain has been successfully installed, through the command

```
arm-elf-gcc --version
```

The output should be similar to the one shown in the picture:



Congratulations, the toolchain is now installed. Let's go now to the following step.

2.2.3 Build of the first program and debug with Insight

Building

Now you can start building your first program. Together with this work, we supply some simple programs for each analyzed microcontroller (the files can be distinguished from their name); moreover, for each of them, a "template" is present, as a base for building more complex programs (for more information you can read next sections and Chapter 3). We start from the template to do our tests.
Open the terminal and go to the template directory of your microcontroller. For example, for a STR710 board, go to the directory *template71x* (located in *ARMProjects/STR7*).
Starting from now, we will talk about the STR71x microcontroller, but the operations we will describe are the same for the other microcontrollers, except for some little modifications in the filenames.

Building the software library

You are in the template directory. Here there's a subdirectory that contains the ST software library files, that is all the functions and the variables that allow you to use completely all the functions and the peripherals embedded into the microcontroller.
In this case the directory is called *str71x_lib* and contains, besides other things, a *Makefile* which allows you to compile software library files in a library (file with extension *.a*), so you can then include this file in your programs.

In each program we already provided the *.a* file, but it could be necessary to recompile it on your system. To do so, you have to move in *str71x_lib* and type the following commands:

```
make clean
make all
```

The first command cleans the files obtained from previous compilations (and also the file *.a* already present), the second one, instead, is the actual compilation and you should obtain your *.a* file (in this case *libSTR71x_lib.a*).

Building the program

You have to come back in *template71x* directory and there you can see a *Makefile*, whose content defines how your compilation will be executed (for example there's a setting to include the *.a* file obtained from the previous software library compilation).
The Makefile section which interests you is the one about program execution in RAM or ROM (FLASH). You can see this in the screenshot on the left.
You have to edit the Makefile, search that section and uncomment the line about the type of execution you want, commenting the other one.
In this case we chose a RAM execution.
Then you have to exit the editor and compile your program using the command `make all`.

```
## Create ROM-Image
#RUN_MODE=ROM_RUN
## Create RAM-Image
RUN_MODE=RAM_RUN
```

The output obtained should be similar to the following one:

```
-------- begin (mode: RAM_RUN) --------
arm-elf-gcc (GCC) 3.4.3
Copyright (C) 2004 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.


Assembling (ARM-only): src/vector.S
arm-elf-gcc -c -mcpu=arm7tdmi  -I. -x assembler-with-cpp -DRAM_RUN -D__WinARM__ -D__WINARMSUBMDL_STR71x__ -Wa,-
adhlns=src/vector.lst, -gdwarf-2 src/vector.S -o src/vector.o

Assembling (ARM-only): src/startup.S
arm-elf-gcc -c -mcpu=arm7tdmi  -I. -x assembler-with-cpp -DRAM_RUN -D__WinARM__ -D__WINARMSUBMDL_STR71x__ -Wa,-
adhlns=src/startup.lst, -gdwarf-2 src/startup.S -o src/startup.o

Compiling C (ARM-only): src/vectors.c
arm-elf-gcc -c -mcpu=arm7tdmi  -I. -gdwarf-2 -DRAM_RUN -D__WinARM__ -D__WINARMSUBMDL_STR71x__  -O0 -Wall -Wcast-
align -Wimplicit  -Wpointer-arith -Wswitch -ffunction-sections -fdata-sections -Wredundant-decls -Wreturn-type
-Wshadow -Wunused -Wa,-adhlns=src/vectors.lst  -I./include -I./str71x_lib/include -Wcast-qual -MD -MP -MF
.dep/vectors.o.d -Wnested-externs -std=gnu99 -Wmissing-prototypes -Wstrict-prototypes -Wmissing-declarations src/
vectors.c -o src/vectors.o

Compiling C: src/main.c
arm-elf-gcc -c  -mcpu=arm7tdmi  -I. -gdwarf-2 -DRAM_RUN -D__WinARM__ -D__WINARMSUBMDL_STR71x__  -O0 -Wall -Wcast-
align -Wimplicit  -Wpointer-arith -Wswitch -ffunction-sections -fdata-sections -Wredundant-decls -Wreturn-type
-Wshadow -Wunused -Wa,-adhlns=src/main.lst  -I./include -I./str71x_lib/include -Wcast-qual -MD -MP -MF
.dep/main.o.d -Wnested-externs  -std=gnu99 -Wmissing-prototypes  -Wstrict-prototypes -Wmissing-declarations
src/main.c -o src/main.o
In file included from ./str71x_lib/include/adc12.h:26,
                 from ./str71x_lib/include/71x_lib.h:29,
                 from src/main.c:18:
./str71x_lib/include/rccu.h:308: warning: function declaration isn't a prototype
In file included from ./str71x_lib/include/71x_lib.h:45,
                 from src/main.c:18:
./str71x_lib/include/eic.h: In function `EIC_IRQChannelPriorityConfig':
./str71x_lib/include/eic.h:138: warning: suggest parentheses around arithmetic in operand of |
In file included from ./str71x_lib/include/71x_lib.h:61,
                 from src/main.c:18:
./str71x_lib/include/i2c.h: In function `I2C_GetStatus':
./str71x_lib/include/i2c.h:318: warning: suggest parentheses around arithmetic in operand of |

Linking: main.elf
arm-elf-gcc  -mcpu=arm7tdmi  -I. -gdwarf-2 -DRAM_RUN -D__WinARM__ -D__WINARMSUBMDL_STR71x__  -O0 -Wall -Wcast-align
-Wimplicit  -Wpointer-arith -Wswitch -ffunction-sections -fdata-sections -Wredundant-decls -Wreturn-type -Wshadow -
Wunused -Wa,-adhlns=src/vector.lst  -I./include -I./str71x_lib/include -Wcast-qual -MD -MP -MF .dep/main.elf.d src/
vector.o src/startup.o  src/vectors.o src/main.o     --output main.elf -nostartfiles -Wl,-Map=main.map,--cref,--gc-
sections -lc  -lm -lc -lgcc   -L./str71x_lib -lSTR71x_lib -T./STR71x-RAM.ld
```

```
Creating load file for Flash: main.hex
arm-elf-objcopy -O ihex main.elf main.hex

Creating load file for Flash: main.bin
arm-elf-objcopy -O binary main.elf main.bin

Creating Extended Listing: main.lss
arm-elf-objdump -h -S -C main.elf > main.lss

Creating Symbol Table: main.sym
arm-elf-nm -n main.elf > main.sym

Size after:
main.elf  :
section            size         addr
.text              2028   1644167168
.ctors                0   1644169196
.dtors                0   1644169196
.vect                 0    536870912
.bss                  0   1644169196
.heap              1024    536870912
.stack             1024    536871936
.stack_irq          256    536872960
.stack_fiq          256    536873216
.stack_svc            0    536873472
.stack_abt            0    536873472
.stack_und            0    536873472
.debug_aranges      400            0
.debug_pubnames     705            0
.debug_info        1542            0
.debug_abbrev       172            0
.debug_line        1036            0
.debug_frame       1088            0
.debug_str           18            0
Total              9549


Errors: none
-------- end --------
```

If you look at this output, you can see that at first there was each *.c* file compilation and each *.s* file assembling and finally all the obtained object files were linked in the file *main.elf*.
Finally you can see a list of the file *main.elf* sections, each one has its dimension and start address.
Like in software library compilation, the command to delete the rests of previous compilations is:

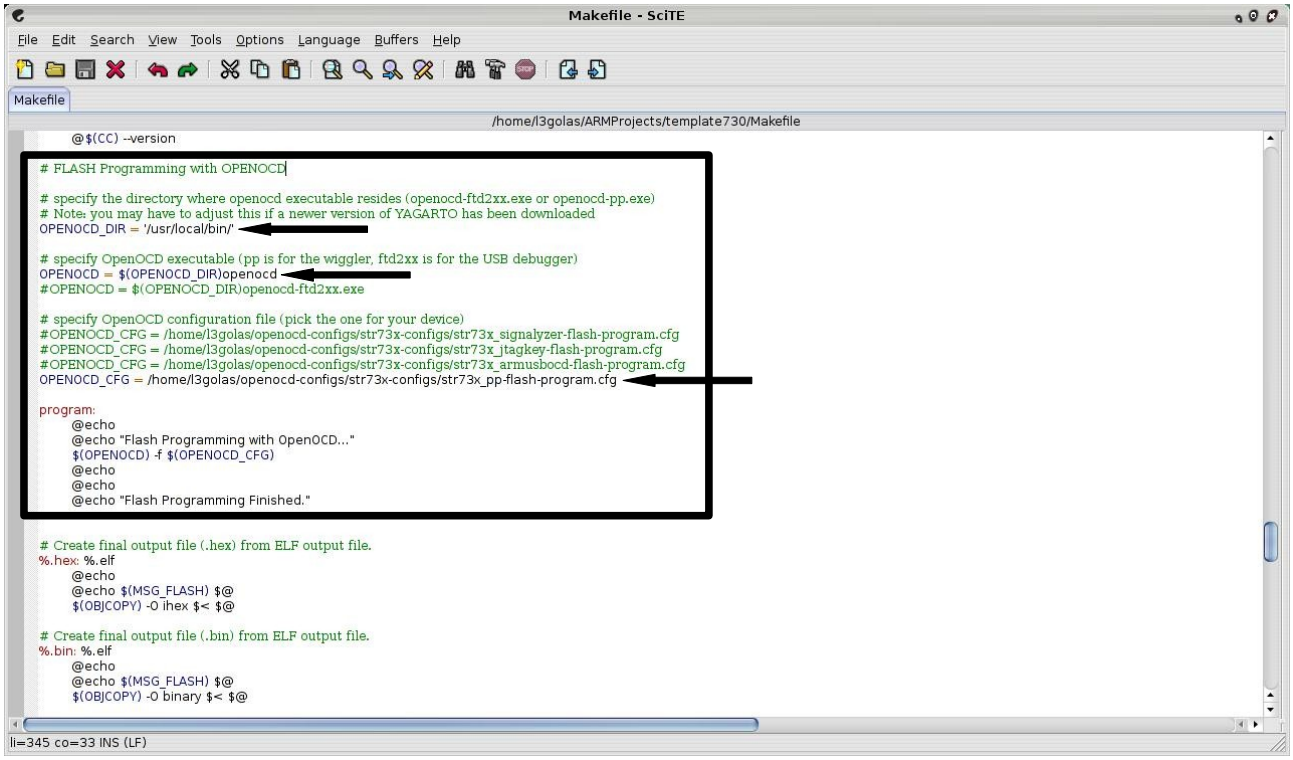
```
make clean
```

If you chose a RAM execution, you can avoid reading next rows and go directly to the Debug section, otherwise if you chose a FLASH execution (choose *ROM_RUN* in the *Makefile*), the operations to do are the same, but obviously the result of compilation will change a little bit, especially in start addresses of *main.elf* sections.
Moreover you mustn't forget that in RAM it's always possible to write and read, instead **if you use FLASH memory, after building and before debugging it's necessary to download the code in it.**

In fact FLASH is a ROM (Read Only Memory) memory so it can only be read and not written.
Actually it's possible to erase and write it, but these operations are done through an electric operation which doesn't work with a single byte, but with the entire block of memory which contains it.
Moreover this operation is really slow if compared to RAM writing and it can be done a limited number of times, about (10^5) times.

The *Makefile* was created to program FLASH in a simple way. However you have to edit it and change some parameters.
To do that, search in the *Makefile* the section that starts with the row *#FLASH Programming with*

OPENOCD, as showed in the screenshot:



You must change OpenOCD path, the executable name and the configuration file name (*.cfg* file) you want to use with OpenOCD: there are many configuration files, each of them is used for one of the interfaces to connect to the board. As usual, we used only parallel port.
The *.cfg* file is used to connect to the board and to call a *.ocd* file, whose task is to write the FLASH. Probably you have to edit that *.cfg* file to change in it the path of the *.ocd* file. Let's explain this better, looking at the content of the *.cfg* file:

```
#daemon configuration
telnet_port 4444
gdb_port 3333

#interface
interface parport
parport_port 0x378
parport_cable wiggler
jtag_speed 0
jtag_nsrst_delay 200
jtag_ntrst_delay 200

#use combined on interfaces or targets that can't set TRST/SRST separately
reset_config trst_and_srst srst_pulls_trst

#jtag scan chain
#format L IRC IRCM IDCODE (Length, IR Capture, IR Capture Mask, IDCODE)
jtag_device 4 0x1 0xf 0xe

#target configuration
daemon_startup reset

#target <type> <startup mode>
#target arm7tdmi <reset mode> <chainpos> <endianness> <variant>
target arm7tdmi little run_and_init 0 arm7tdmi
run_and_halt_time 0 30

working_area 0 0x2000C000 0x4000 nobackup

#flash bank <driver> <base> <size> <chip_width> <bus_width>
flash bank str7x 0x40000000 0x00040000 0 0 0 STR71x
flash bank cfi   0x60000000 0x00400000 2 2 0

#Script used for FLASH programming
target_script 0 reset /home/l3golas/openocd-configs/str71x-configs/str71x_flashprogram.ocd
```

The last row is used to call the *.ocd* file: here you have to set the correct path.
That's all: come back in *template71x* directory and after compiling correctly the project for FLASH, you have to power on the board and execute the command:

```
make program
```

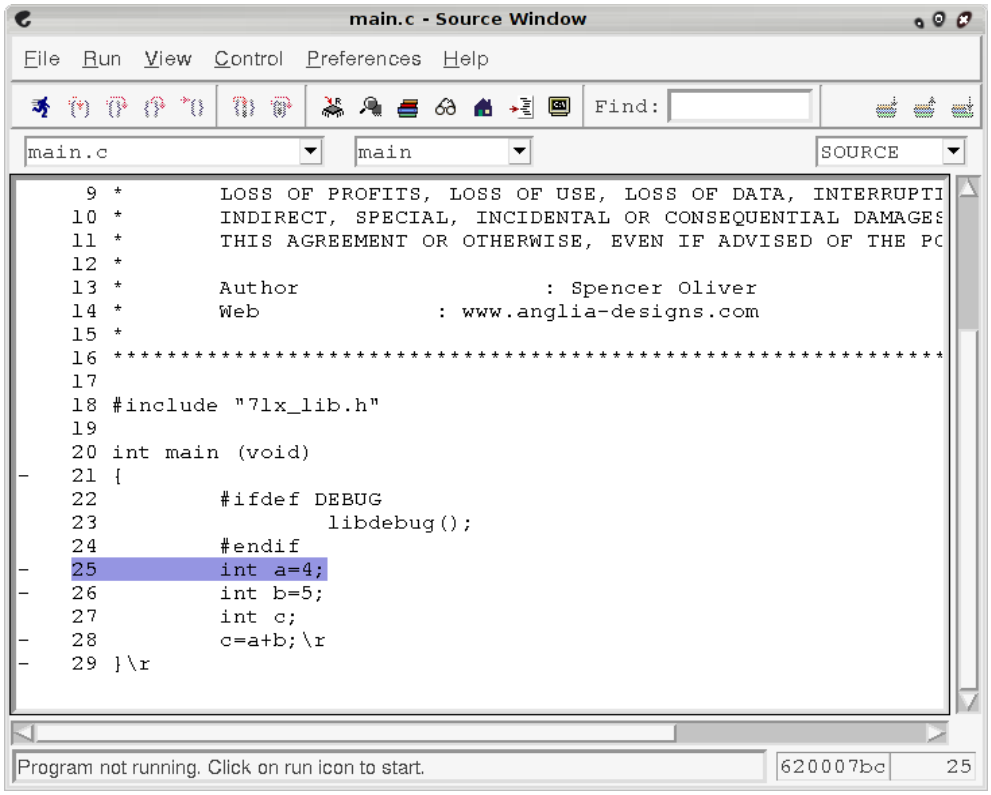
Now OpenOCD should be called, with the *.cfg* file as input, which will call the *.ocd* file that will write FLASH.
This operation could take some minutes depending on FLASH memory dimensions, then the sentence *Flash Programming Finished* should appear.
Now you are ready for debugging through INSIGHT.

Debug

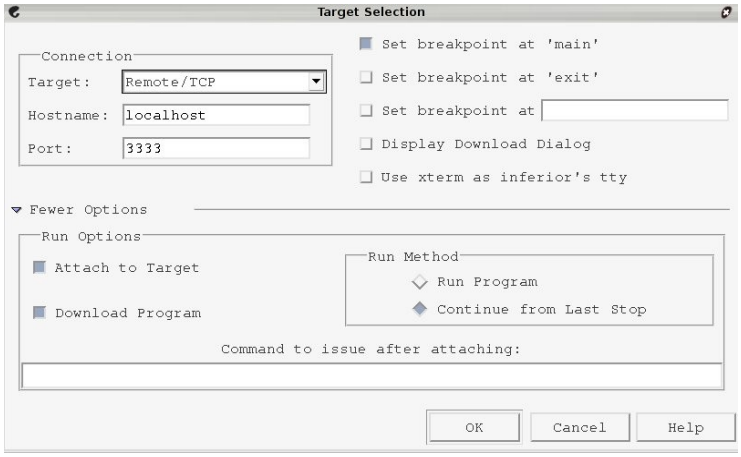
Insight is the famous GDB front-end, one of the most famous solutions used in UNIX for debugging.
You have to add some options to the command used to start Insight, so you have to call it this way: `arm-elf-insight -x GDBFILENAME ELFFILENAME` where you have to substitute the *.elf* file of the program to *ELFFILENAME*, in your case *main.elf*. *GDBFILENAME* instead must be substituted by a file which contains some commands to initialize GDB, for example for the connection to OpenOCD.
In the folder *openocd-configs* you copied before, besides the true configuration files there are also the *.gdb* files, distinguished by the microcontroller model and by the debug type (RAM or FLASH).
For example, if you want to debug in RAM a program for the STR710 microcontroller, the file to use is called *str71x_ram.gdb* , so this is the file to substitute to *GDBFILENAME*.
Before starting Insight, you have to launch OpenOCD, following what we said in the proper section of 2.2.1 paragraph.
OpenOCD will block, leave this session opened and open another command prompt session, then go to the folder *template71x* that you have already compiled.
Let's launch Insight through the following command:

```
arm-elf-insight -x YOUR_HOME/openocd-configs/str71x-configs/str71x_ram.gdb main.elf
```

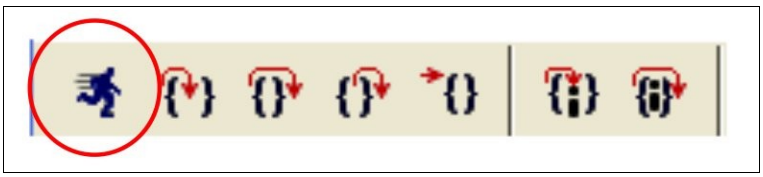
where *YOUR_HOME* must be substituted by your home path (where you should have put the folder *openocd-configs*).
This is the Insight main window:



Let's click on the menu “File” and choose the “Settings” item. In the window that will appear you have to set the parameters as specified in the following picture:

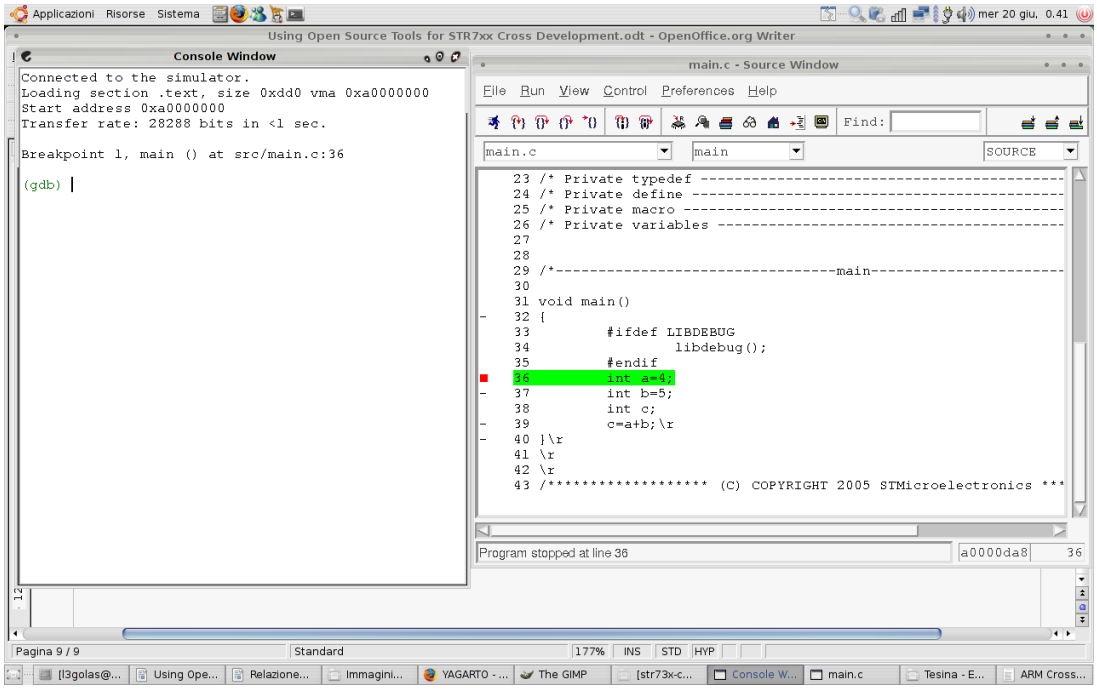


Through these parameters you establish how to connect to the target (via OpenOCD), to set a breakpoint at *main* (that is the program is loaded and then it’s stopped on the starting point waiting that the user decides how to continue) and that when the *Run* button is pressed, both the operations of connecting to the target and of downloading of the program in the memory are automatically executed.
This is right what you are going to do: after you have pressed the OK button, you'll come back in the main window of Insight where you have to press the *Run* button (circled in the next picture).
This button, as explained before, will execute automatically the operations of connection and download.



If all is ok, the program should be downloaded in memory and start, blocking on the first instruction, waiting for a decision by the user on how to continue. The next picture shows what we

have said:



Please look at the console on the left that can give you useful information. If the console isn't present, it's possible to call it by clicking on the button shown in the picture:

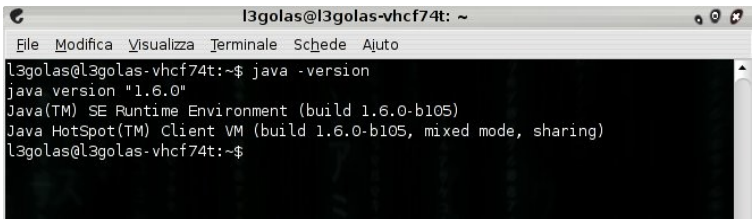
If Insight, instead of loading the program, blocks or has an anomalous behaviour, it could mean that the addresses are wrong in the linker files or in the *.gdb* files.
For more details refer to chapter 3.
Insight is a very powerful debugger that allows to do the most important operations, as step by step execution of instructions, watching and changing of variables and registers (we will deal with all these functions in the appendix A), but now let's pass to an IDE that manages compilation and debugging functions, and result to be at the same time simple and useful: Eclipse.

2.2.4 Java...where art thou?

To make Eclipse work properly, it's necessary to make sure you have the right version of the JVM (*Java Virtual Machine*): to do it, open the terminal and type

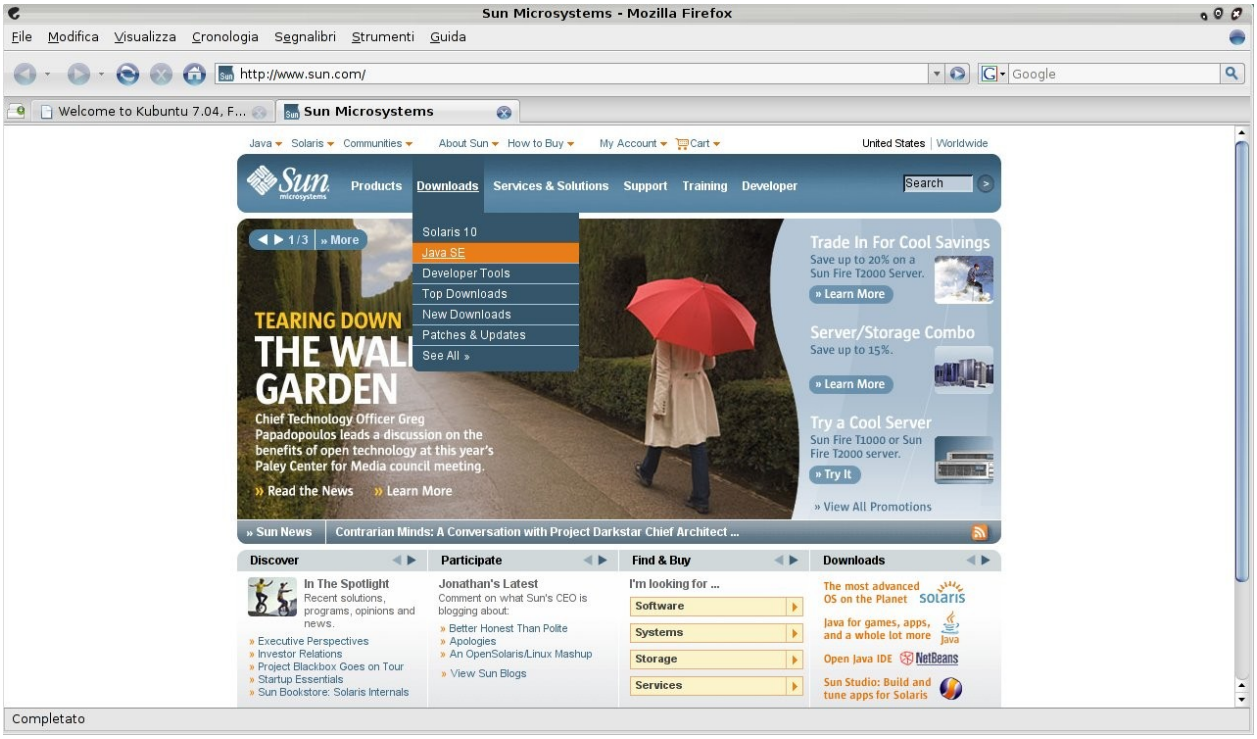
```
java -version
```

You should obtain an output similar to the following:

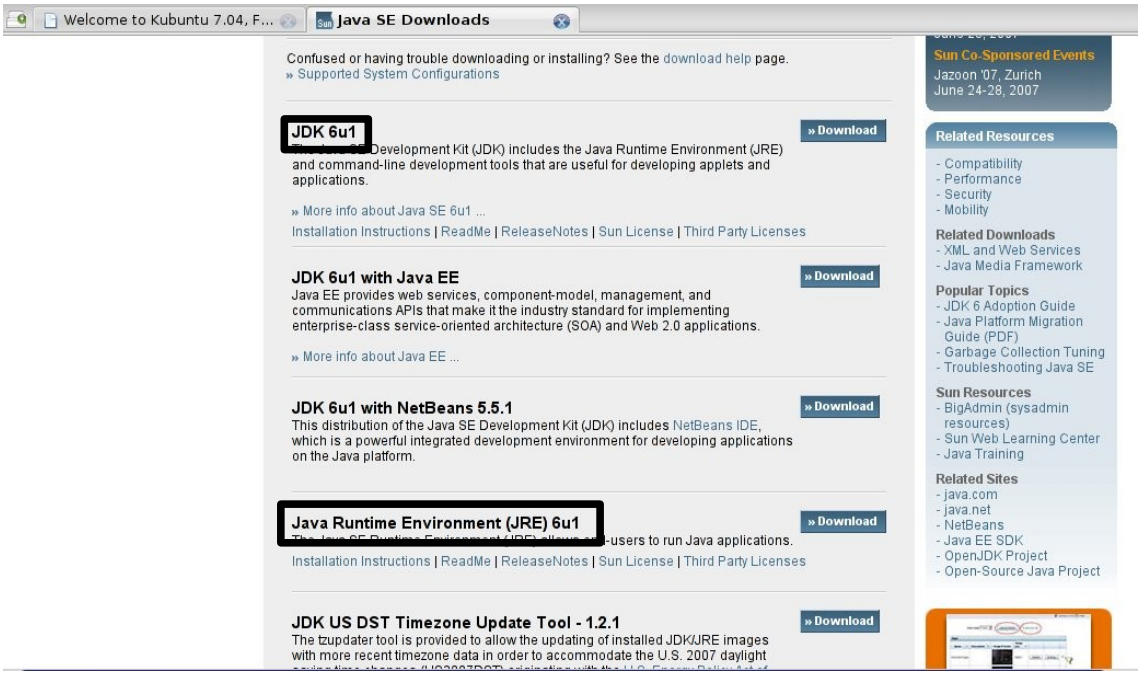


Note that the version of the installed JVM is the 1.6.0 and it's ok for your purposes since the Zylin plugin for Eclipse is compatible with versions of Java from 1.4.2.

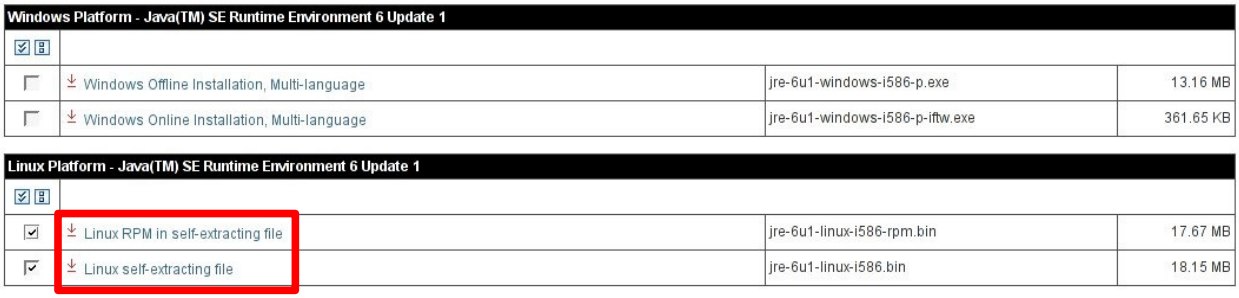
If the output is different, then the JVM isn't correctly installed or the binary JVM files aren’t in the PATH, so you have either to install JAVA or to set the path. Let's see how to install JAVA.
For our purpose the JRE is enough, since we have to only execute Eclipse, but maybe you need in the future to develop something in Java, so you should consider the possibility to download the JDK.
The installation could be simple in some distributions, in that both the JRE and the JDK could be present in the repositories, in that case installing them could be really simple: for more information refer to your distribution user manual.
If Java isn't present in your repositories or is too old, you should download it from Sun's site at www.sun.com.
After you have connected to it, you have to go to *Downloads – Java SE section*, as you can see from the picture below:



A window similar to the one shown below will appear, in which you can choose among several versions. You have to choose between the two versions shown into the box, which are JDK and JRE.



We have chosen JRE, but the operations are equivalents for JDK. Now a window should appear, in which you can download the requested package depending on the operating system you have. For Linux the choice is between the two versions shown in the red square in the following picture: the second one is ok for every Linux version, the first one should make installation easier if you have a RedHat-like distribution (a distribution which supports RPM). We opted for the second one, because it's suitable to everybody.



Make sure at first the file you downloaded is executable, by typing

```
chmod +x jre-6u1-linux-i586.bin
```

N. The *.bin* filename could be a little bit different, depending on the version.

Copy the file in the directory where you want to install it, for example */usr/local*, through the command

```
cp jre-6u1-linux-i586.bin /usr/local
```
(as root)

Launch the installation through the command

```
./jre-6u1-linux-i586.bin
```
(as root)

When the installer asks you for licence acceptance, type *Yes*.
Now the installation is complete, but perhaps you haven't finished yet: if on your system there were still other Java versions, you have to choose which one to use among the different alternatives. This step is impossible to explain in an exhaustive way, in that often each distribution has a different way to do that. We decided to explain the operation on the distributions that use *update-alternatives* (such as Debian, Ubuntu that are the most used); if your distribution doesn't use *update-alternatives*, there will surely be another way to do the same thing, that you can find quite easily on Internet: Google is your friend :-)
For *update-alternatives*, as follows you can find the right procedure, take into account that all the operations must be executed as root and that the lines about *javac* must be executed only by people who decided to have the JDK and not the JRE:

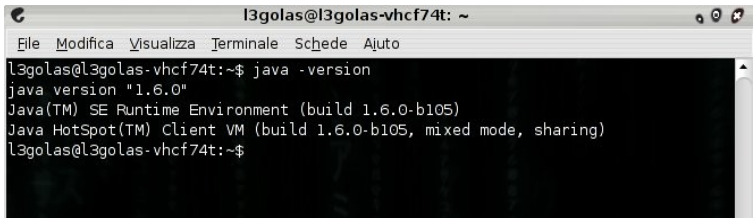
1. To insert the new version in *update-alternatives* you have to type:

```
update-alternatives --install /usr/bin/javac javac /usr/local/jdk1.6.0/bin/javac 30
update-alternatives --install /usr/bin/java java /usr/local/jdk1.6.0/bin/java 30
```

2. and to use the right version:

```
update-alternatives --config javac   (choose the number of the just installed version)
update-alternatives --config java    (choose the number of the just installed version)
```

Now, if you type in the terminal `java -version`, you should finally have:



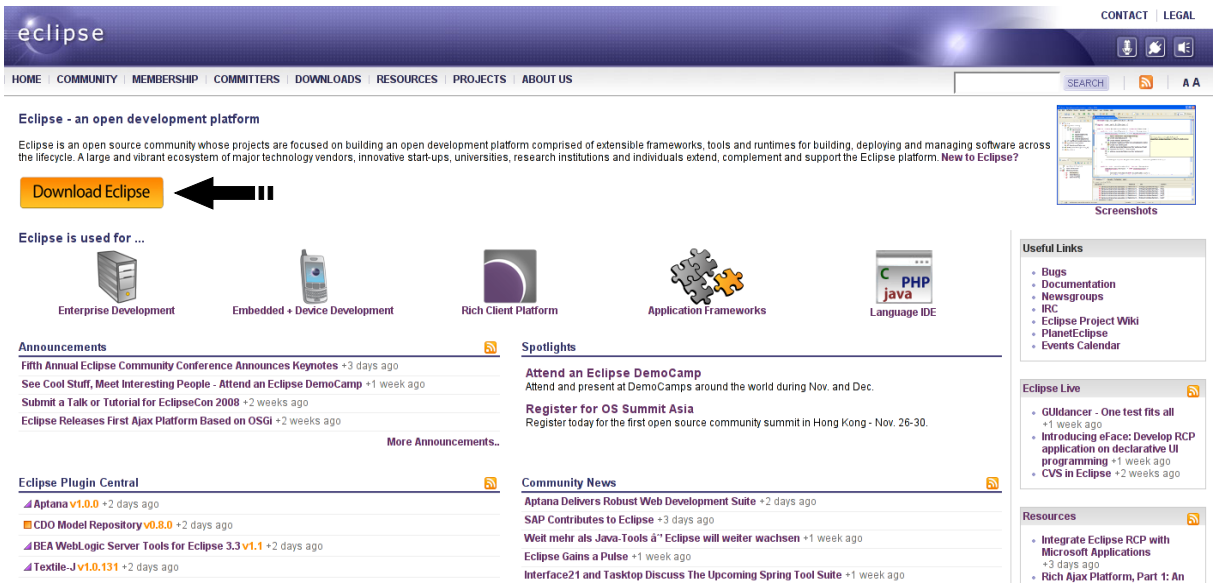
Our advice is to set also the environment variable *JAVA_HOME*. To do that type the command

```
export JAVA_HOME=/usr/local/jdk1.6.0/
```

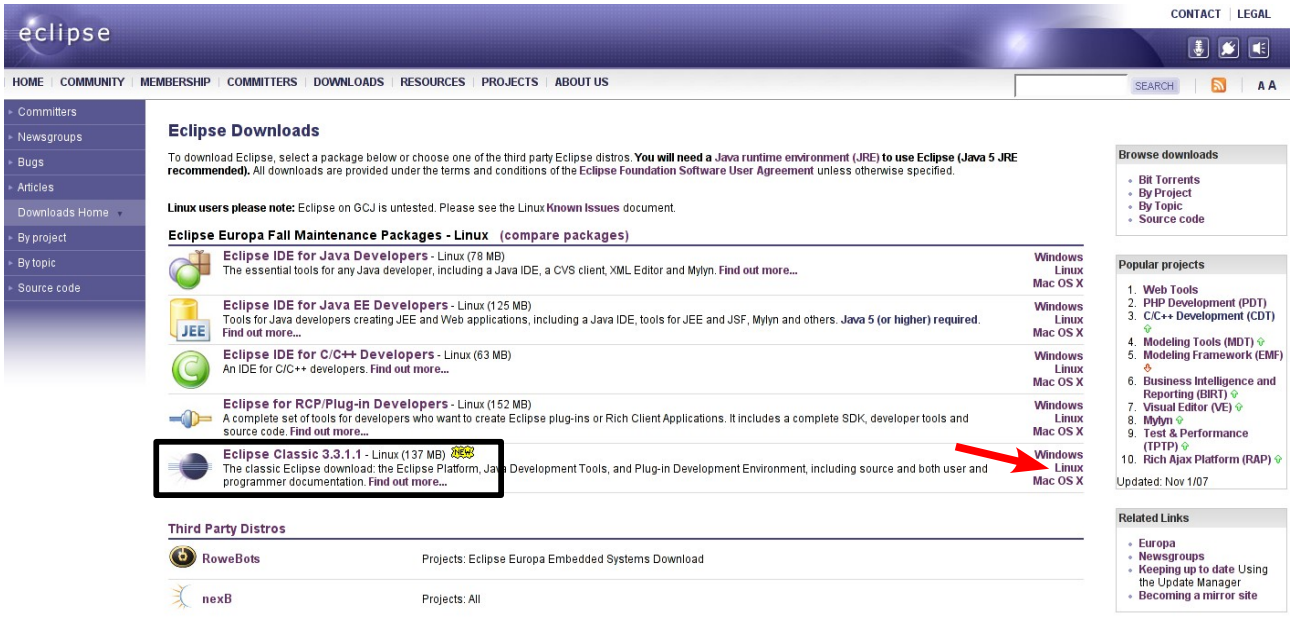
and, to make permanent this modification, insert that line at the end of the file *.bashrc*, that you can find in your home.

2.2.5 Eclipse installation

Probably you have already this wonderful IDE installed in your distribution or ready to install from your repositories. The important thing is that the version isn't older than 3.3. As already specified, for our purpose we will use the Zylin Embedded CDT plugin, that is a modified version of the CDT plugin for C/C++ programming in Eclipse. Its current version is 4.0, which works with Eclipse 3.3.
If you have already installed Eclipse or you can install it from the repositories (and the version isn't older than 3.3) do it and go directly to next paragraph about Zylin Embedded CDT plugin installation, otherwise continue reading.
Let's download Eclipse from the site www.eclipse.org. Once opened the site on your browser, click on the button "*Download Eclipse*" indicated below:



Now you should see a window similar to the following, where you have to choose the “Eclipse Classic” version, indicated in picture in a black square:



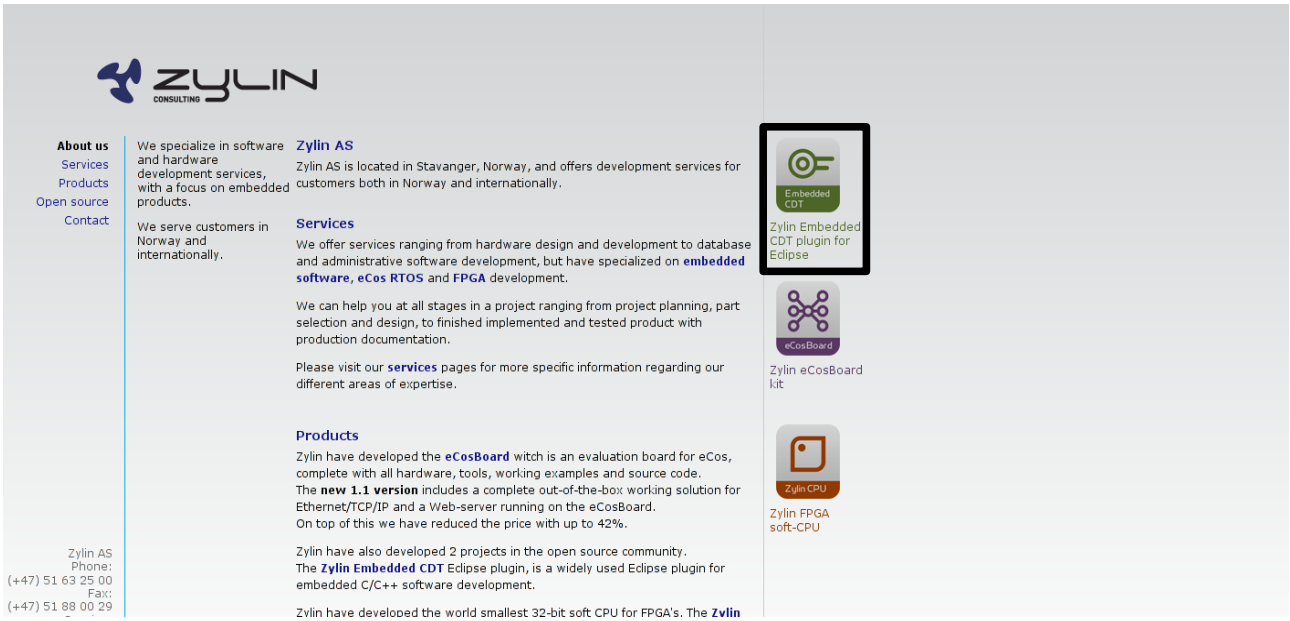
As you can see, the browser should understand automatically the operating system you are using, making you download the proper version, if not you can download Eclipse directly from the link indicated with the red arrow.
Then you have to choose the mirror from which to download Eclipse, choose the closest to you.
When you have downloaded the file, copy it in your home and extract it through the command

```
tar zxvf eclipse-SDK-3.3.1.1-linux-gtk.tar.gz
```

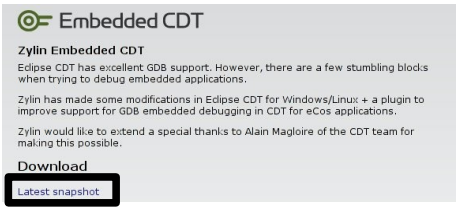
(**Note:** The name could change a bit depending on Eclipse version)
You'll get a folder called *eclipse* which contains the IDE. There is no need for installation, you'd better create only a shortcut on the desktop to the file *eclipse*, that is located in the just created folder.

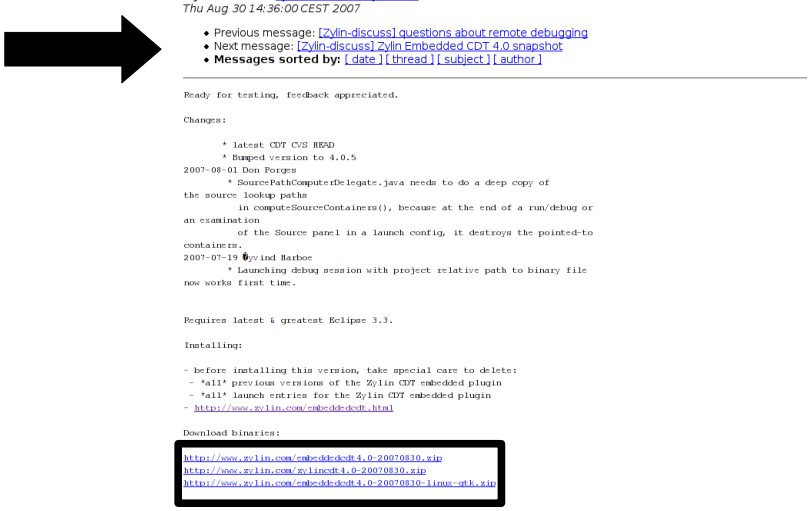
Zylin Embedded CDT plugin installation

You can download this plugin from the site www.zylin.com at the section showed below in the black square:



From the page which opens, click on *Latest snapshot* and download all the three links. The two pictures below show this:





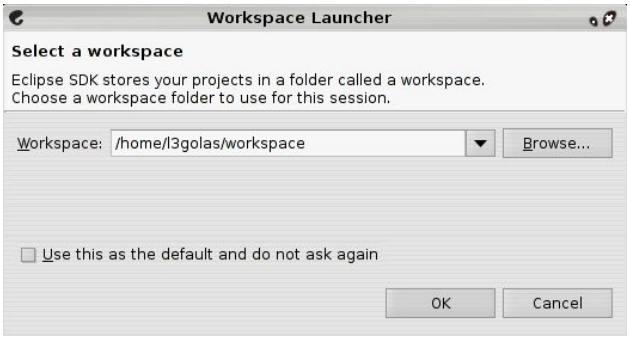
Open a terminal session, enter Eclipse folder, copy the three files and extract them through the following commands:

```
unzip embeddedcdt4.0-20070830.zip
unzip embeddedcdt4.0-20070830-linux-gtk.zip
unzip zylincdt4.0-20070830.zip
```

Now the Zylin Embedded CDT plugin should be installed, you will make sure it's so in the next paragraph.

2.2.6 Eclipse: first run and preliminary configuration

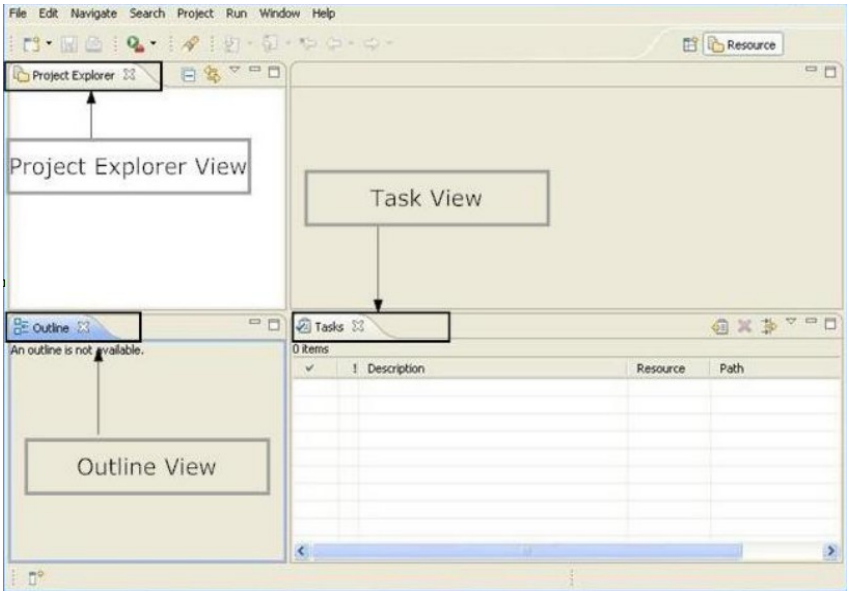
Click on the shortcut to Eclipse you did on the desktop. As soon as you have launched the IDE, a Workspace Launcher will appear, where you have to specify the path and the name of the folder that will contain the workspace, which contains all the files of your projects.
If the specified folder doesn't exist, Eclipse will create it for you.



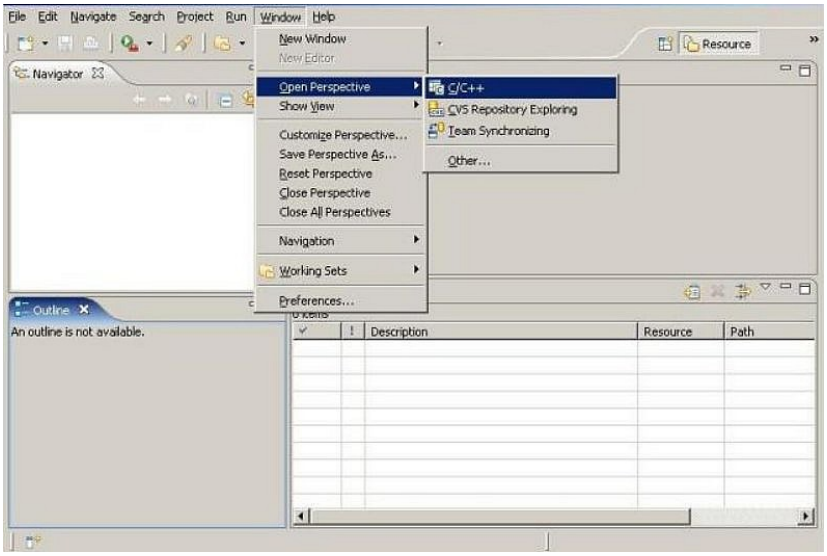
It's important to use always the same workspace, because when you change it, your configuration settings aren't anymore visible.

Then the Eclipse welcome page will appear:

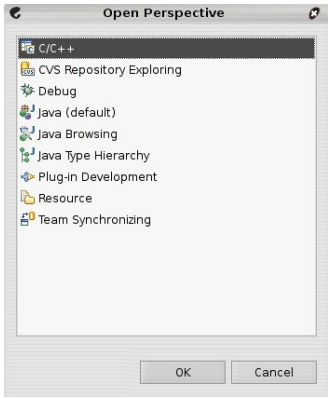
Since many contents you can access through the icons refer to Java programming, for our purposes this window isn't so useful and you can close it.
An important aspect is that Eclipse is made from many perspectives, each of which allows to access different resources managed in views: when you set one perspective instead of another one, the physical aspect of the IDE will change because new views will appear.
Now close the welcome window to access automatically the *Resource Perspective* (it's the default perspective), where there are the *Project Explorer* view, the *Outline* view and the *Task* view, as you can see in the following picture:



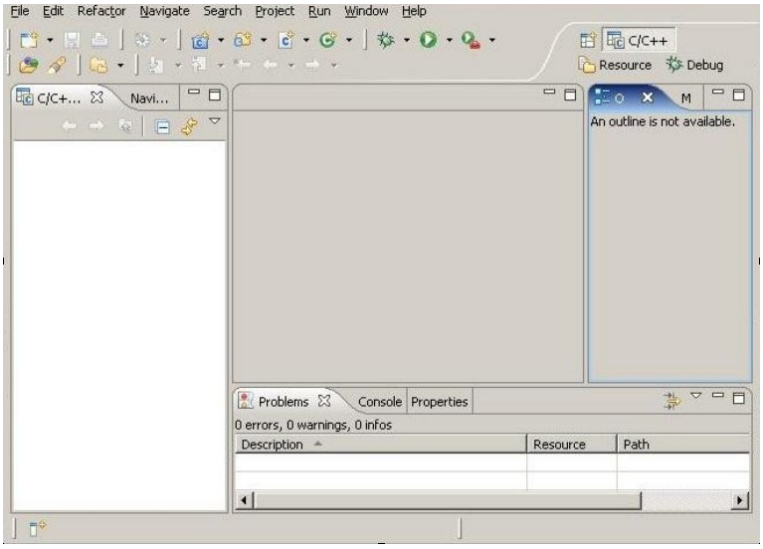
Now it's necessary to switch to the *C/C++ Perspective*: to do it go to the menu *Window->Open Perspective->C/C++*, as you can see in the following picture:



If *C/C++ Perspective* is not in the list, click on *Other*, so a new window with all the perspectives will appear, in which you can choose the C/C++ one, as shown below:

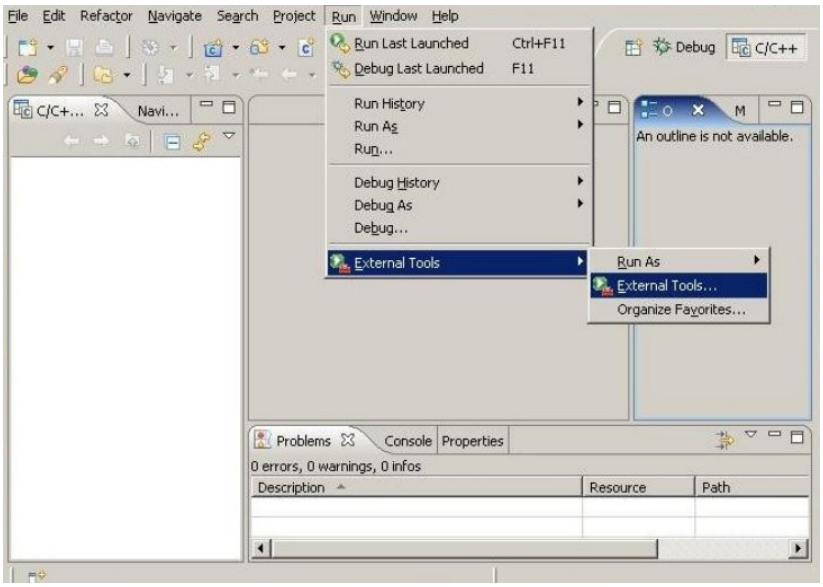


After you have selected C/C++ Perspective, it will appear as shown below:



Inserting OpenOCD in Eclipse external Tools

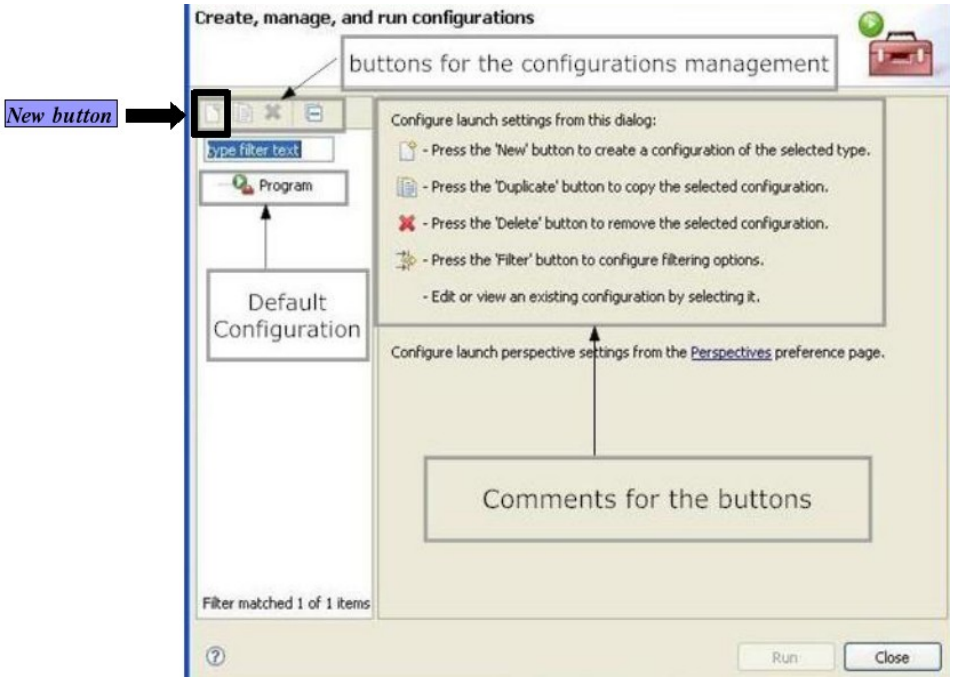
Now let's insert OpenOCD into the Eclipse external tools. Indeed, OpenOCD is necessary for debugging. Moreover you will also use it to erase the microcontroller FLASH memory. Let's see how. Eclipse has a very powerful feature that allows the user to include external programs. To use this feature it's necessary, as shown in next picture, to go to the menu Run->External tools->External tools....



External tools window will appear.

In the following steps we will took as example STR71x microcontroller, but the same steps have to be done for each of the used microcontrollers, so, in our case, we will have in our list *OpenOCD (STR71x)*, *OpenOCD (STR73x)* and *OpenOCD (STR75x)*, each one with its own parameters.

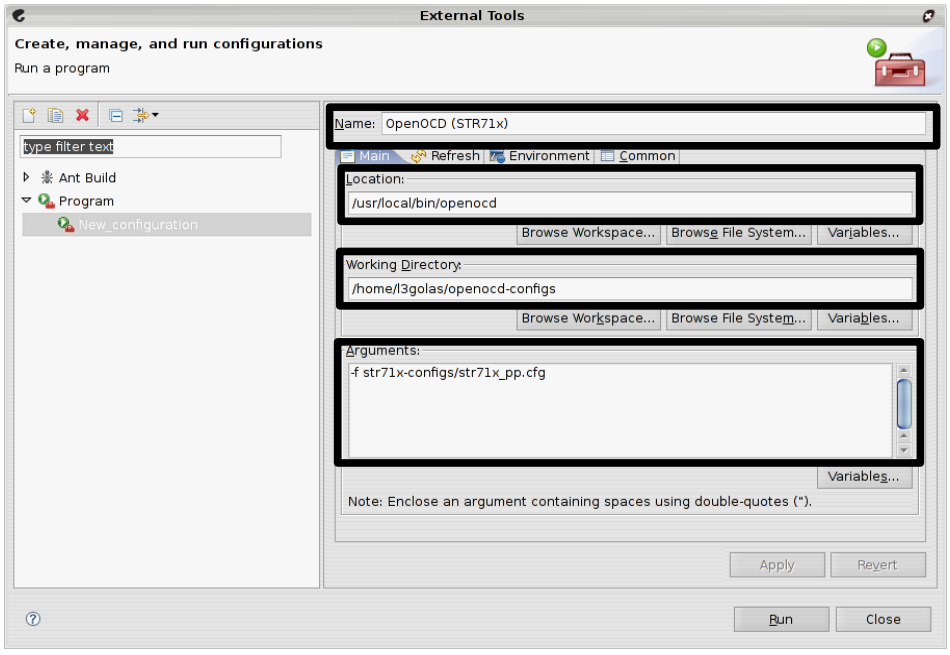
Now click on Program and then on New button to insert an external program in the list, as shown in the following picture:



You have to fill the *External Tools* window this way:

- In the *Name* textbox insert *OpenOCD (STR71x)*.
- In the *Location* textbox insert the path of the OpenOCD executable. Using *Browse File System…* option it's possible to find this file location, that should be */usr/local/bin/openocd*
- In the *Working Directory* textbox insert the path of the folder *openocd-configs* (in our case */home/l3golas/openocd-configs*)
- In the *Arguments* textbox insert the option *–f str71x-configs/str71x_pp.cfg* (to specify OpenOCD configuration file for the chosen microcontroller; we used the parallel port version, but there are also the versions for the alternative connections).

All those settings can be seen in the following picture:

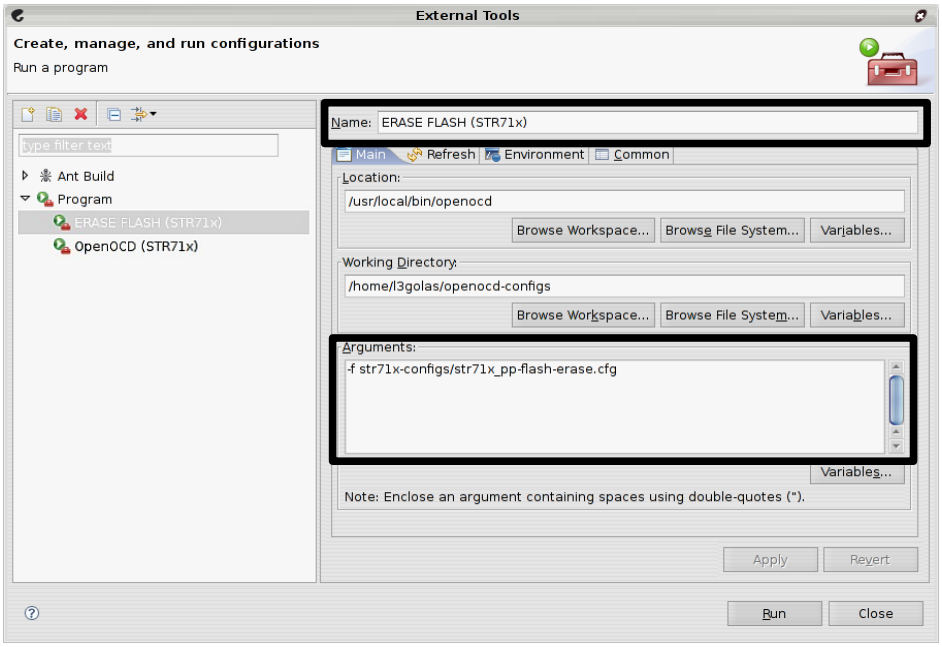


No other changes are necessary in the other tabs, that are *Refresh, Environment* and *Common.*

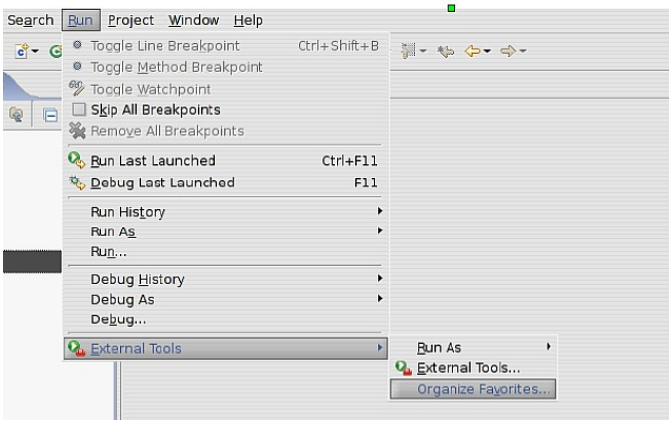
Then, by clicking on Apply, OpenOCD will be registered into Eclipse as an external tool.

Inserting OpenOCD as FLASH eraser among Eclipse External Tools

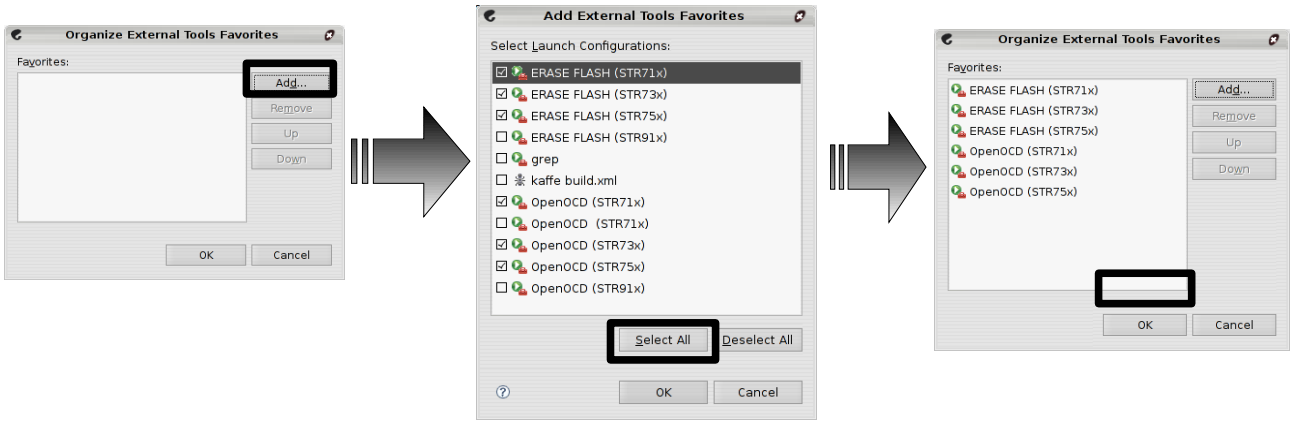
As we have just said, you also have to insert OpenOCD as eraser for the FLASH memory.
To do that, you have to insert a new program with the procedure just seen, inserting the following data.
In the *Name* textbox you have to insert *ERASE FLASH (STR71x)*.
The *Location* and the *Working Directory* textboxes are identical to the ones of *OpenOCD (STR71x)* shown before: in fact in the *Location* textbox you have to insert */usr/local/bin/openocd*, while in the *Working Directory* textbox you have to insert your *openocd-configs* path. In the *Arguments* textbox insert *-f str71x-configs/str71x_pp-flash-erase.cfg*, to specify the file to give to OpenOCD for FLASH erasing.
These settings can be seen in the following picture:



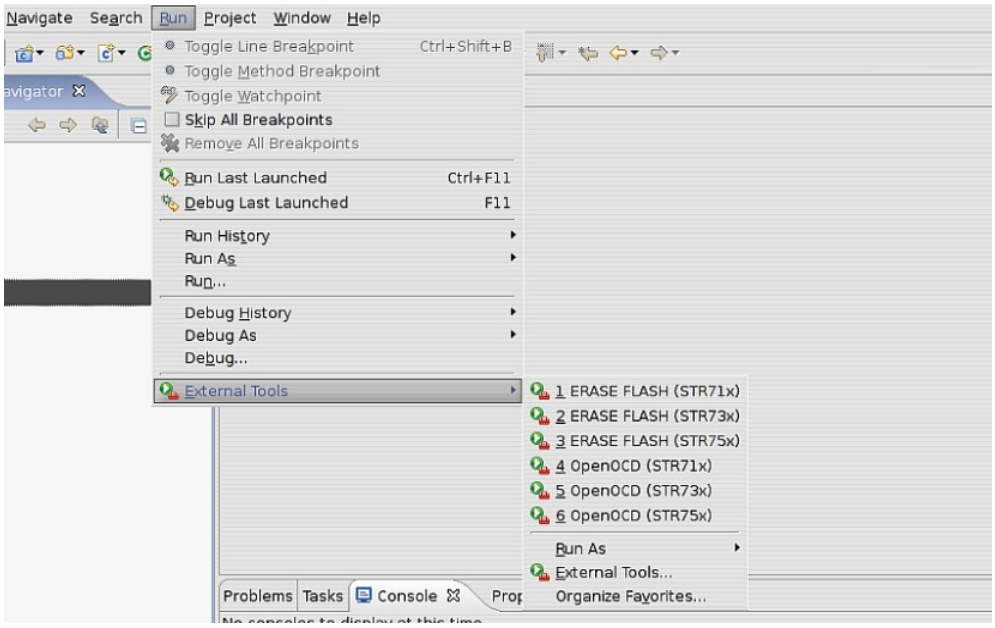
Obviously the same operations have to be executed also for the other microcontrollers.
Finally a little tip: don't you want to organize these “External Tools” to call them more quickly, since their use is frequent?
This is possible: you have to go to the menu *Run->External Tools->Organize Favorites*, as shown in the following picture:



In the window that will appear, click on the *Add* button. A new window will appear where you can select the tools that you want to have among the favourites: when you have done press *OK*.
The three window below show what we have just said:



Now, to call these programs, it's sufficient to go to the *Run->External Tools* menu and select the tool you want to use.

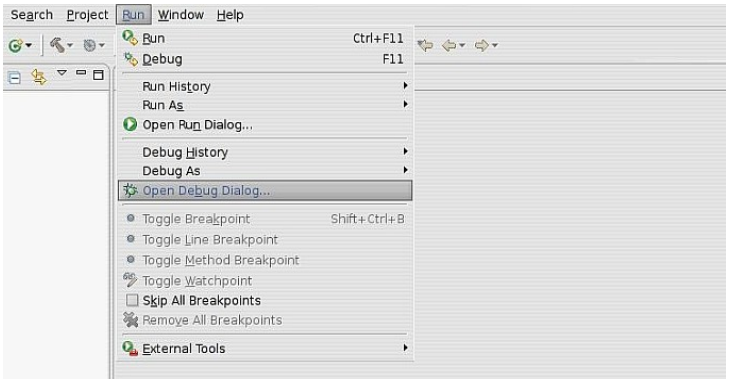


Debug configuration in Eclipse

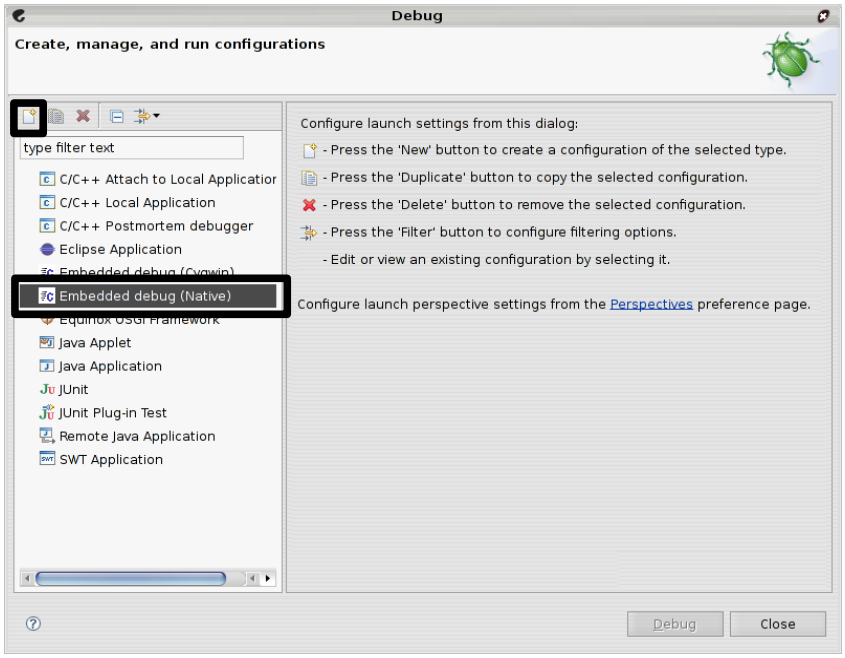
This paragraph explains how to configure Eclipse in order to correctly debug all our examples, it's a maybe annoying procedure, but you have to do it only once.

The objective is to create a configuration model for each used microcontroller, to debug both in RAM and in FLASH so, when you want to debug a project, you will only have to select the correct model and insert in it the project name, because the other parameters will be already present.

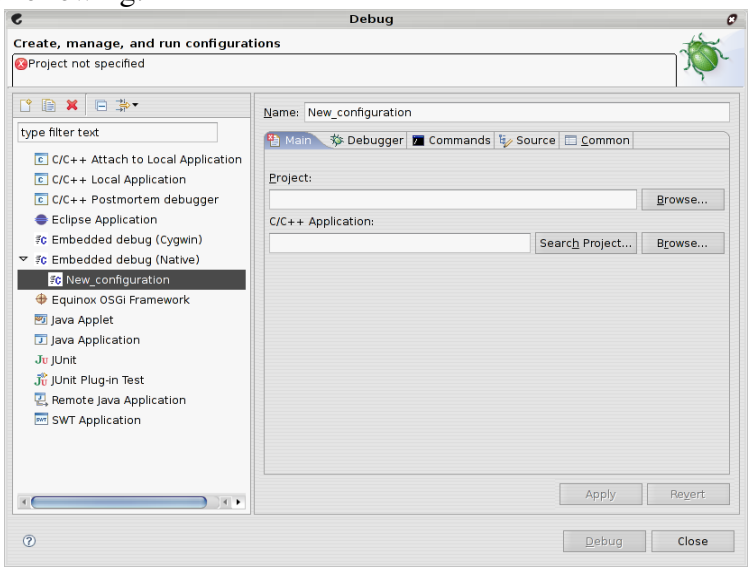
Let's start.
Go to the Run->Open Debug Dialog... menu as shown in the picture:



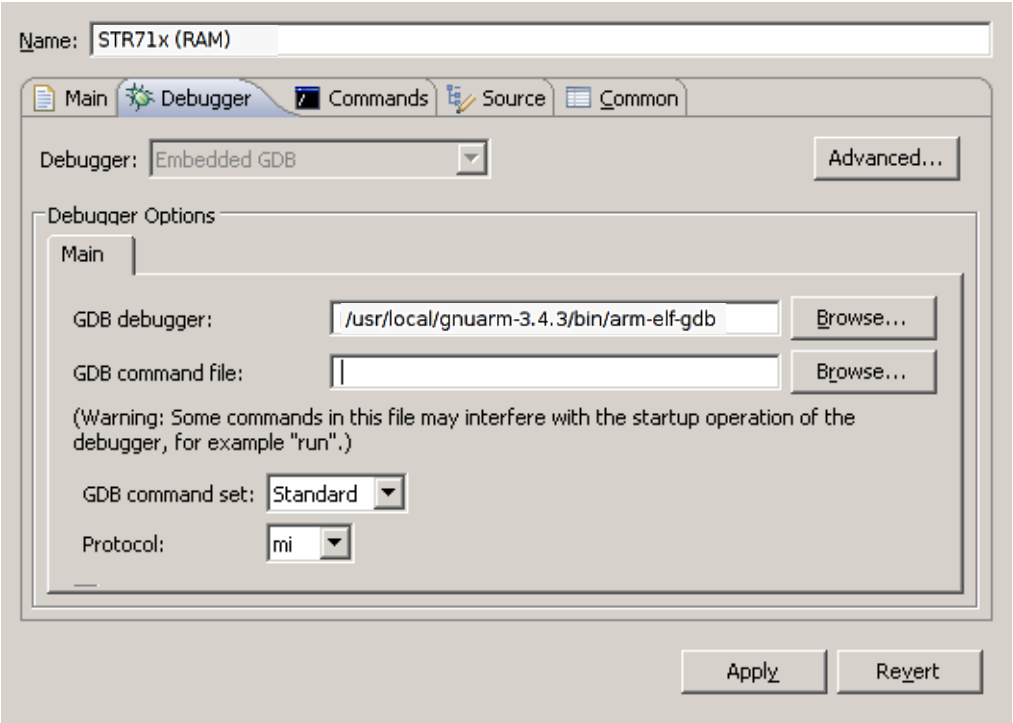
A window will appear, where it's possible to set the debug configurations for the programs in the workspace.
Click on *Embedded Debug (Native)* and then on the *New* button, as shown below:



The result will be the following:



On the right, you have all the properties of your project, let's set them.
First of all, the name: insert *STR71x (RAM)*. Below, in the *Main* section, don't modify anything.
Go to the *Debugger* section and fill it as shown in the following picture:



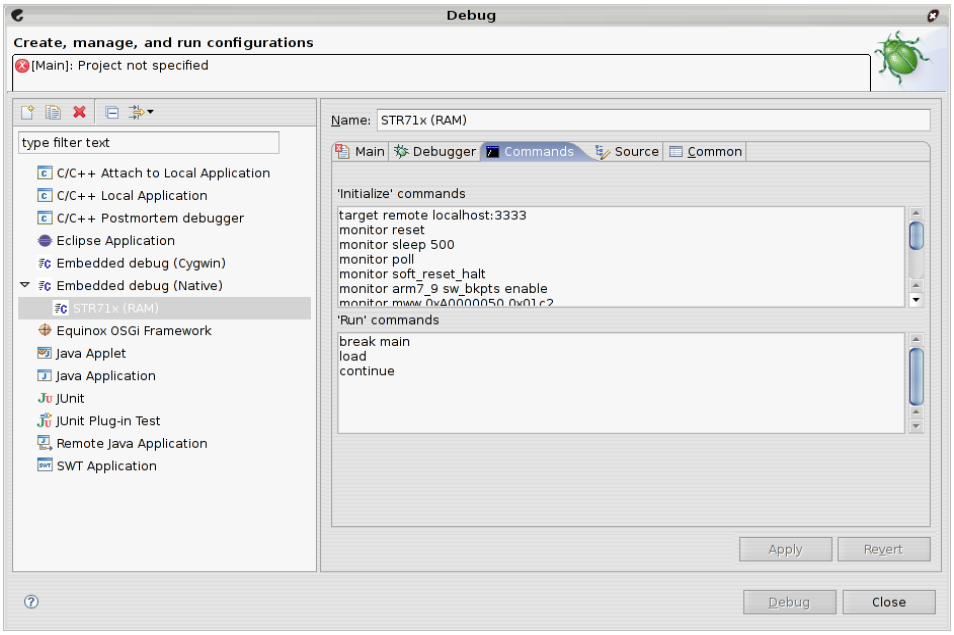
Insert in the section *GDB debugger* the *arm-elf-gdb* path (in our case */usr/local/gnuarm-3.4.3/bin/arm-elf-gdb*) and remove *.gdbinit* from the section *GDB command file*.
You continue with the *Commands* tab where there are two textboxes, *Initialize commands* and *Run commands.*
In the first one it's necessary to insert the contents of the corresponding *gdb* file.
In this case go to the folder *openocd-configs/str71x_configs* and edit the file *str71x_ram.gdb* that has the following content:

```
target remote localhost:3333
monitor reset
monitor sleep 500
monitor poll
monitor soft_reset_halt
monitor arm7_9 sw_bkpts enable
monitor mww 0xA0000050 0x01c2
monitor mdw 0xA0000050
monitor mww 0xE0005000 0x000F
monitor mww 0xE0005004 0x000F
monitor mww 0xE0005008 0x000F
monitor mww 0x6C000004 0x8001
monitor mdw 0x6C000004
```

As we have just said, this content has to be copied in *Initialize Commands* textbox.
In *Run Commands* textbox, instead, you have to insert the following commands:

```
break main
load
continue
```

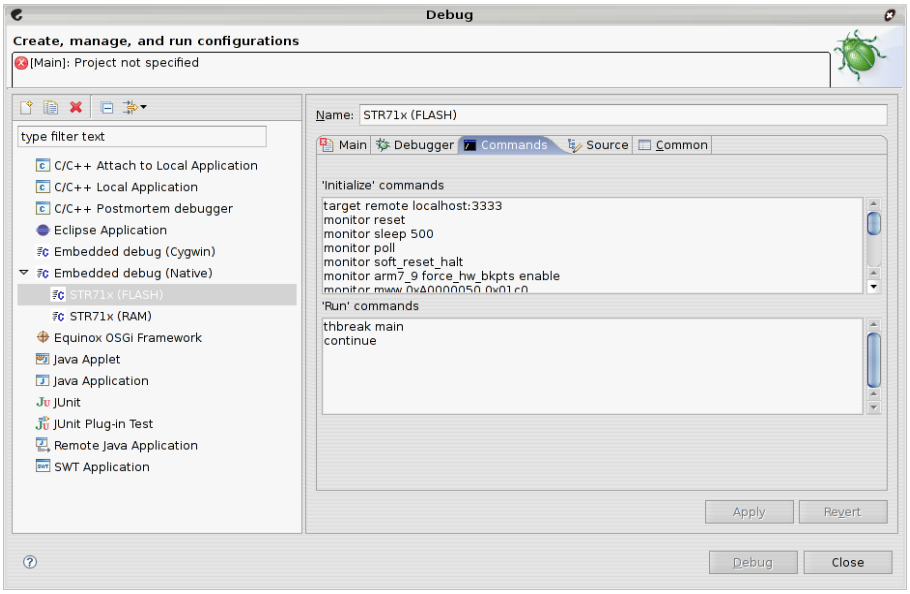
So finally this section will appear as shown below:



Now click on *Apply*.
Now you have to do the same steps to configure this microcontroller for FLASH use.
Here, again, select *Embedded debug (Native)* and press the *New* button. Then insert *STR71x (FLASH)* in the *Name* textbox inside the *Main* section, the *Debugger* textbox has to be modified as shown for the previous example.
About the *Commands* section, in the *Initialize Commands* textbox insert the content of the file *str71x_flash.gdb* that is in the same folder of the previously analysed file, while in the *Run commands* textbox insert the following rows:

```
thbreak main
continue
```

So finally this section will appear as shown below:



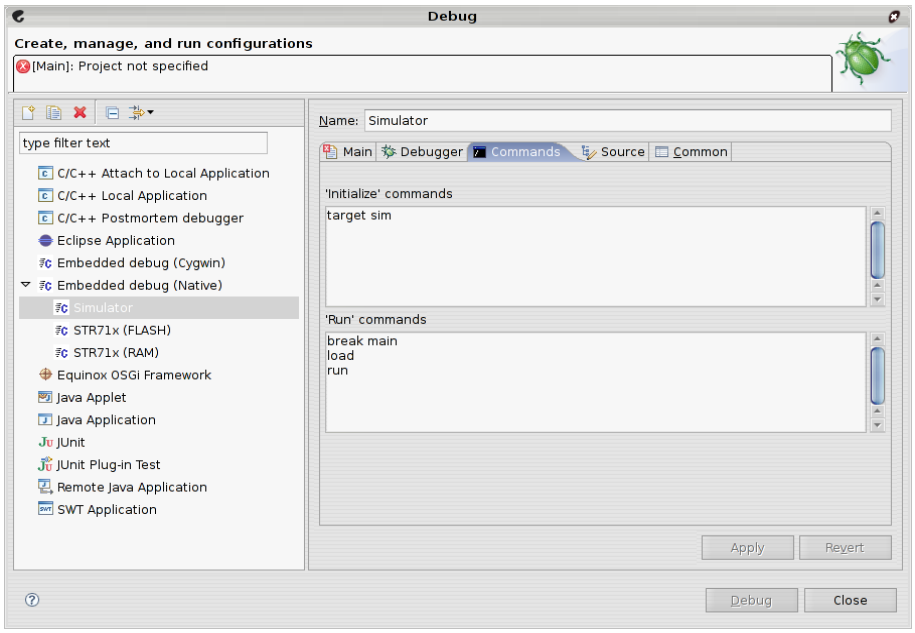
Obviously it's necessary to repeat the above described operations also for the other microcontrollers, configuring both RAM and FLASH and using each time the proper files.
Some words about the **simulator**. What's that? Would you like to have a simulator that allows you to do a preliminary debug of your programs without using your board? There's a way to do that, although it can't be considered a complete simulator.
In fact, in the list of targets you can choose, GDB also offers the option *Simulator*, but it simulates correctly only ARM cores and not the other peripherals. Then you can't watch LEDs switching on and off or try a serial communication, but you can watch registers and variables and the values they assume during program execution and you can also set all the breakpoints you like.
To configure the simulator you have, as usual, to select *Embedded Debug (Native)* in the *Debug* window and to press then the *New* button.
In the *Name* textbox insert *Simulator*, don't modify the *Main* section and modify the *Debugger* section as seen in the previous configuration.
In the *Commands* section, in the *Initialize Commands* textbox, insert the following line:

```
target sim
```

while in the *Run Commands* textbox write:

```
break main
load
run
```

The window will be, finally, similar to the following:



Click on *Apply*. That's all, so close the *Debug* window by clicking on the *Close* button.

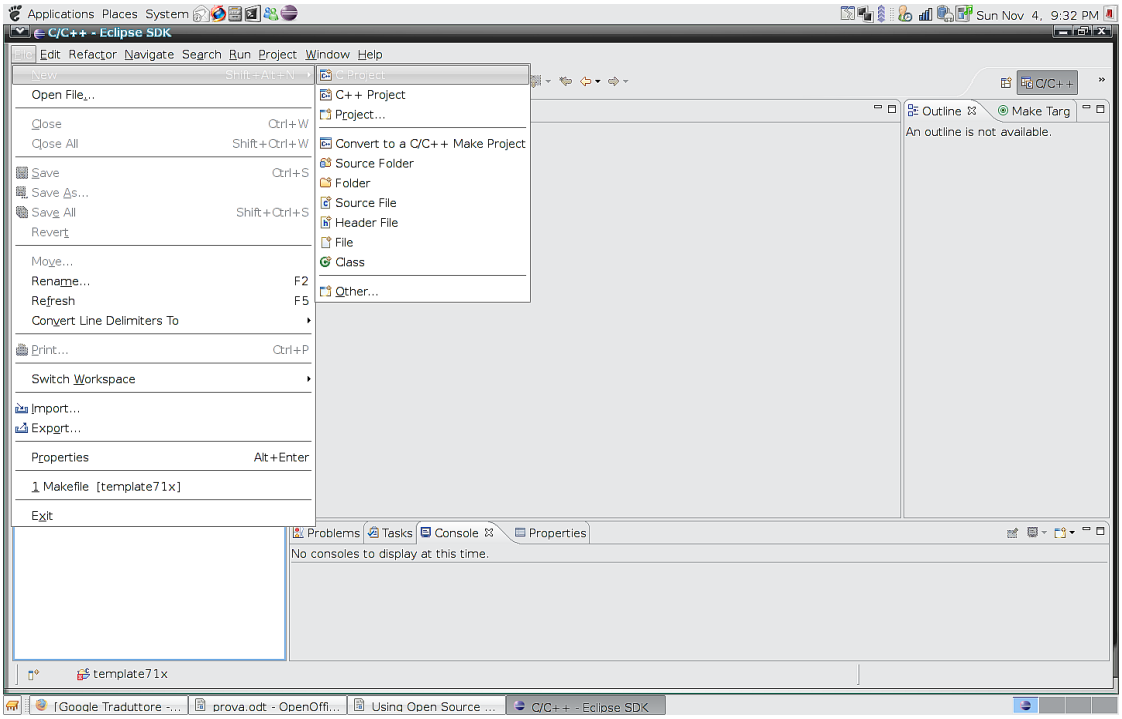
2.2.7 Eclipse: Creation, building and debugging of a project

Now let's see how to create, build and debug a new project in Eclipse.

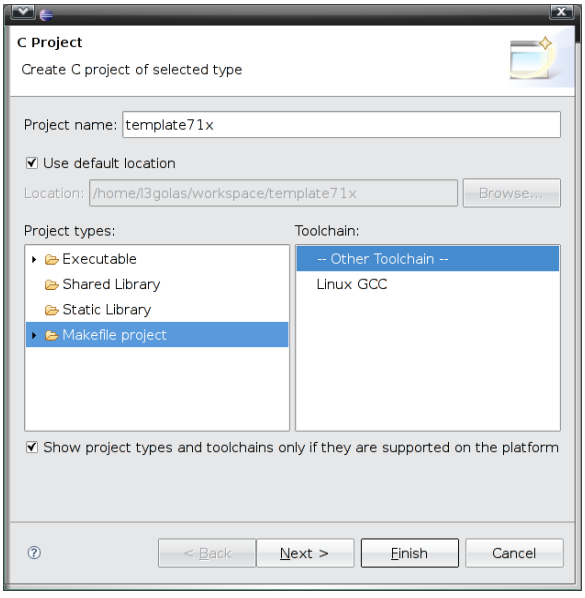
We will choose a model (in this case the STR71x microcontroller), but the operations are the same for the other microcontrollers (except for some modifications, such as file or folder names).

Creation of a new project

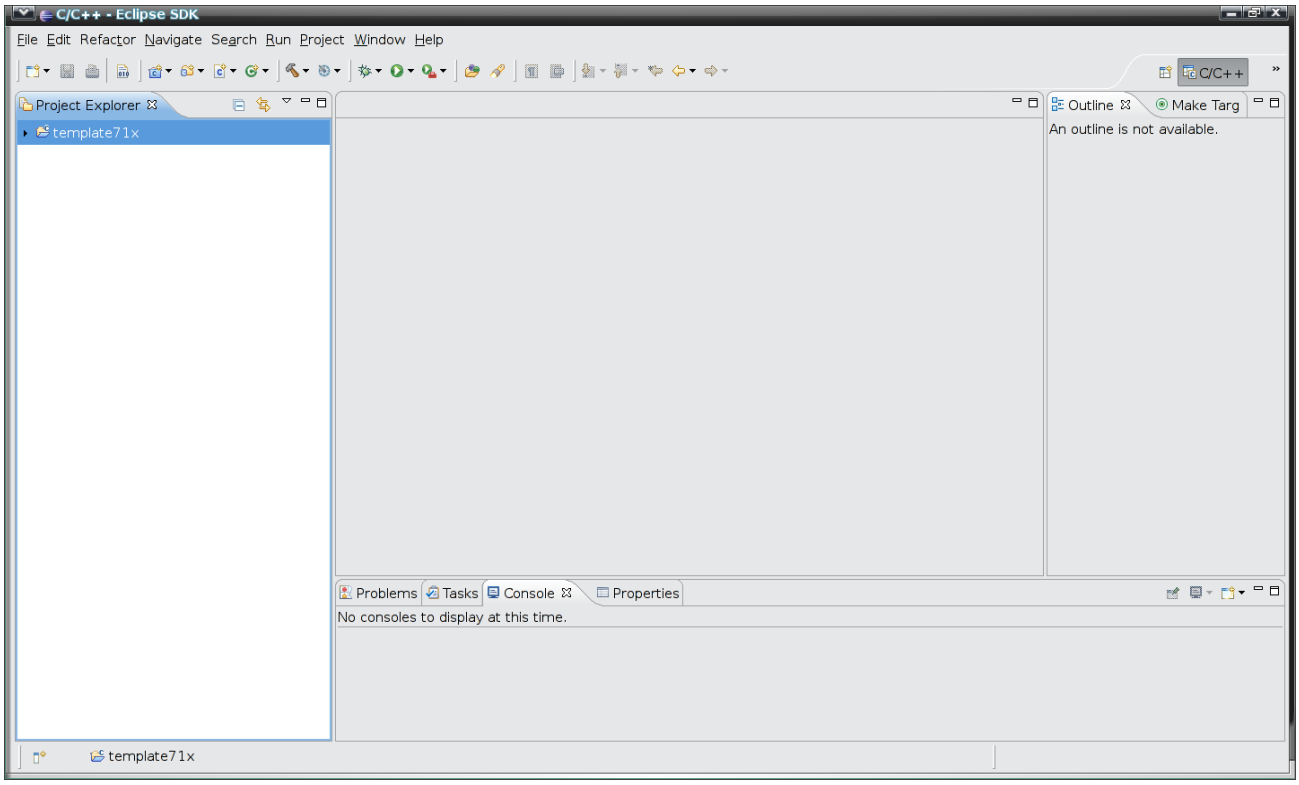
Before creating a new project, you have to open the menu *Project* and deselect the option *Start a build Automatically*, so Eclipse won't try to compile the project automatically, but only when asked.
To create a new project in Eclipse you have to go to the menu *File*, select *New* and then choose *C Project* as shown in the following picture:



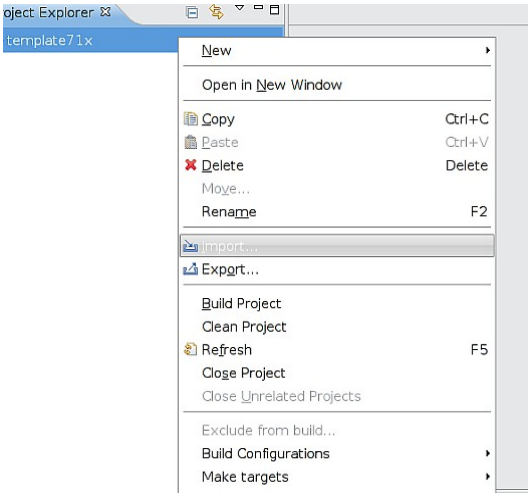
In the window that appears you have to insert the project name (in our case *template71x*) and the location, that is by default your workspace. Click on *Makefile project* in the section *Project types* and on -- *Other Toolchain* -- in the section *Toolchain*. The following picture shows what we have just said:



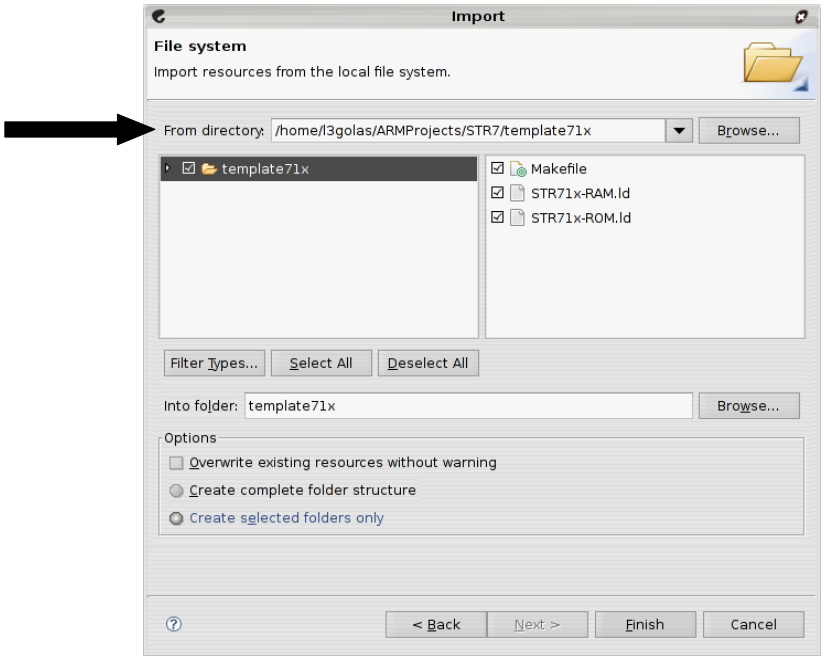
Then press the *Finish* button. In the following picture you can notice how the C/C++ perspective shows the just created project:



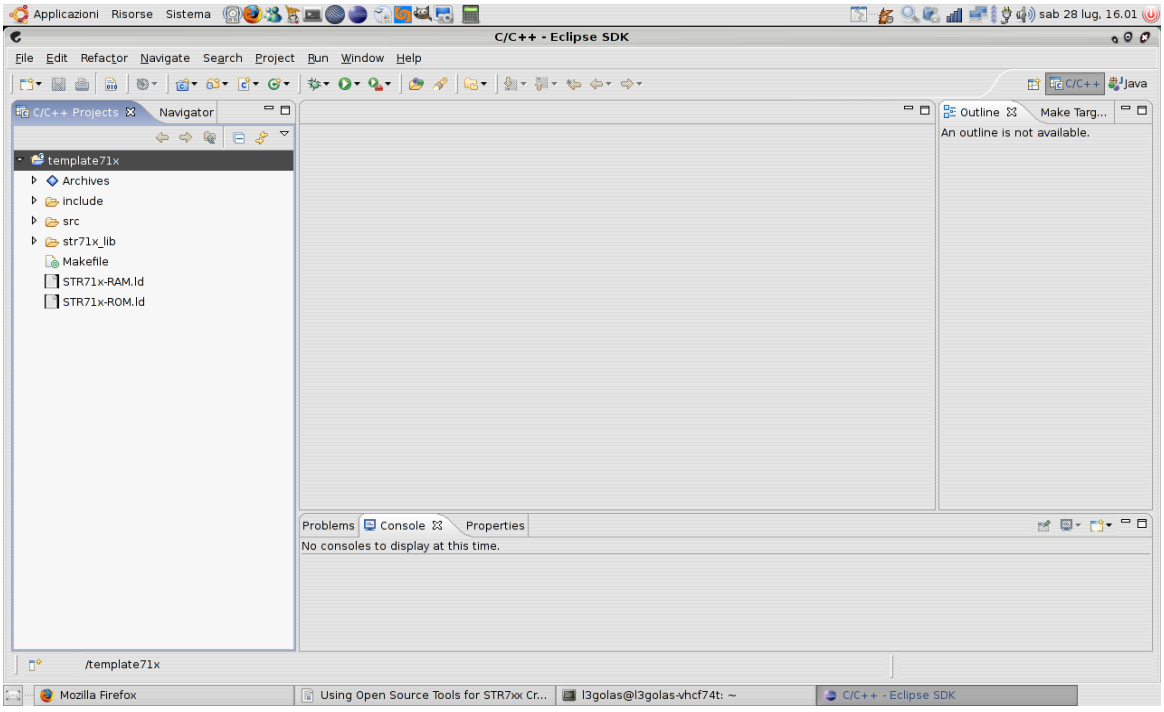
Now go to the menu Project and uncheck *Build automatically*.
To add a source file to the project it's necessary to go to *File->New->Source File*, but in this section we will import an already existing project.
To do that, right-click on the project name and select *Import*, as shown in the following picture:



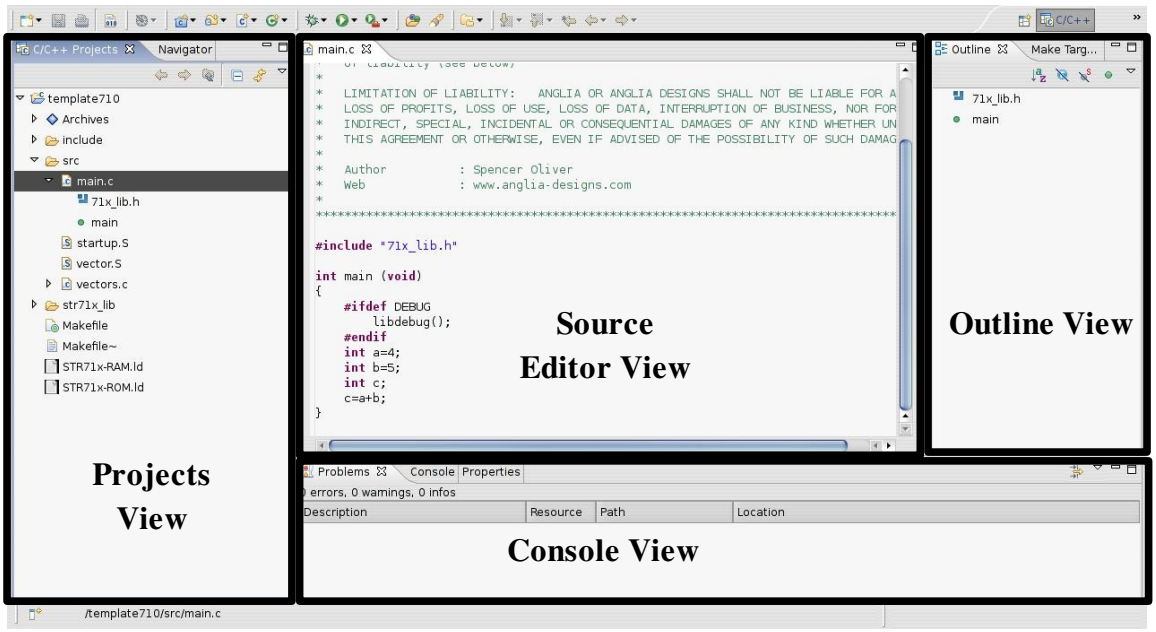
In the window that will appear you have to go to the folder *General*, select *File System* and press *Next*. The following window will appear:



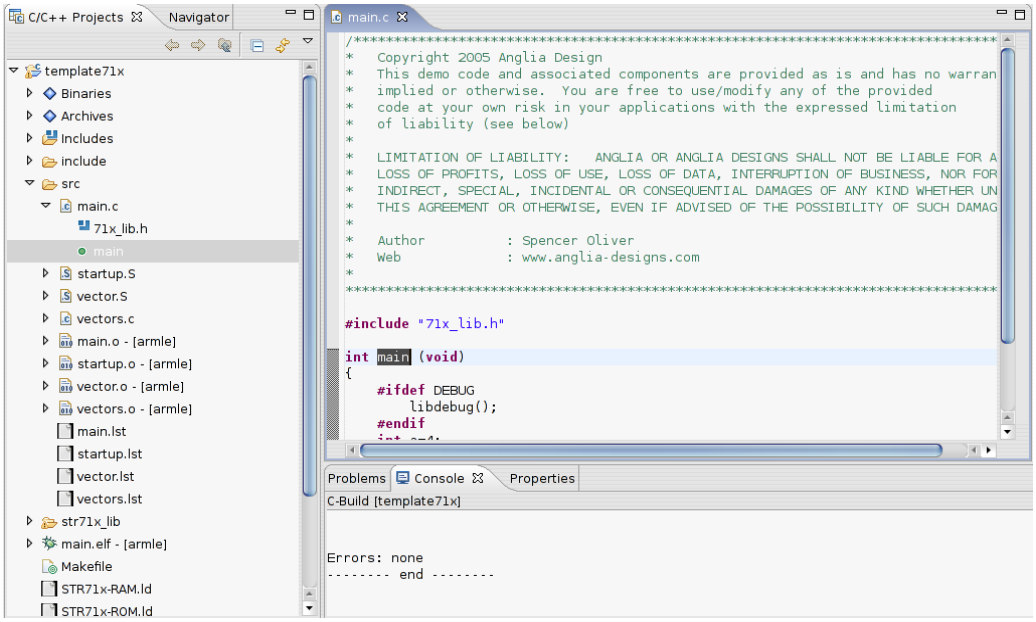
In the textbox *From directory*, as you can see from the picture above, it's necessary to insert the project folder, that you can search using the *Browse* button.
As soon as the folder of *template71x* has been selected (in our case */home/l3golas/ARMProjects/STR7/template71x*), you will be redirected to the window *Import*; in the left column, which is below the project path, check the box near the project folder name and all the files contained in it will be automatically selected. All these files will be shown in the right column.
When you click on *Finish* , you will complete the files import operation.
Expanding the tree of the project *template71x*, you can see all the imported files, as shown in the following picture:



Now, it's possible to see the content of each file by double clicking on it. Source files are in the folder *src*, below you can see the content of the file *main.c*:



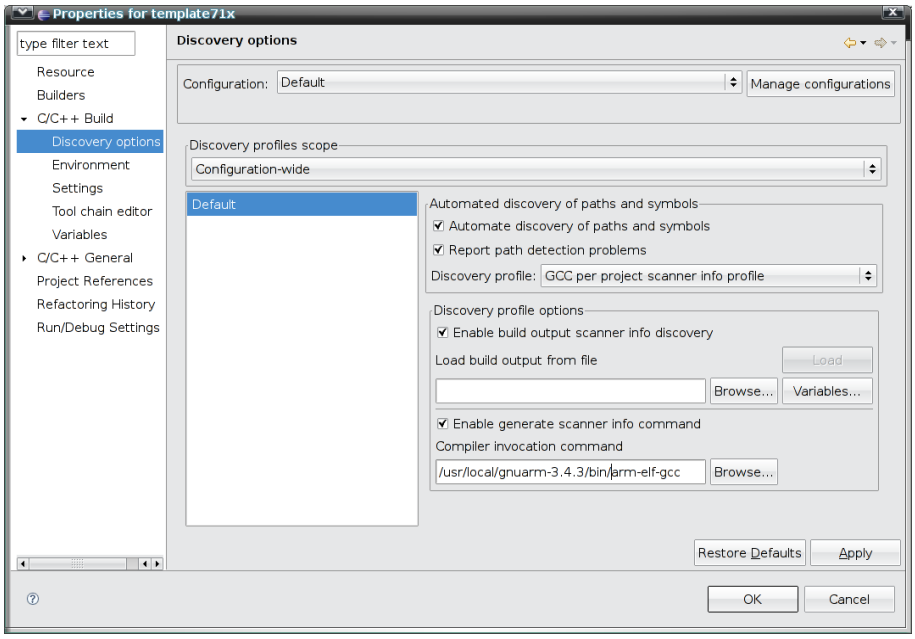
In the *Projects View* you can click on any source file and the *Source File Editor View* will be automatically updated.
Moreover, by seeing source files in the *Projects View* you can also see all the single variables and functions contained inside: by clicking on the symbol on the left of each file (it should be a little "+" or something similar), you will see a list of what is contained in the file (you can see the same list in the *Outline View*); for example, by clicking on the symbol on the left of main, you will automatically see in the *Source Editor View* the function *main*, as shown in the following picture:



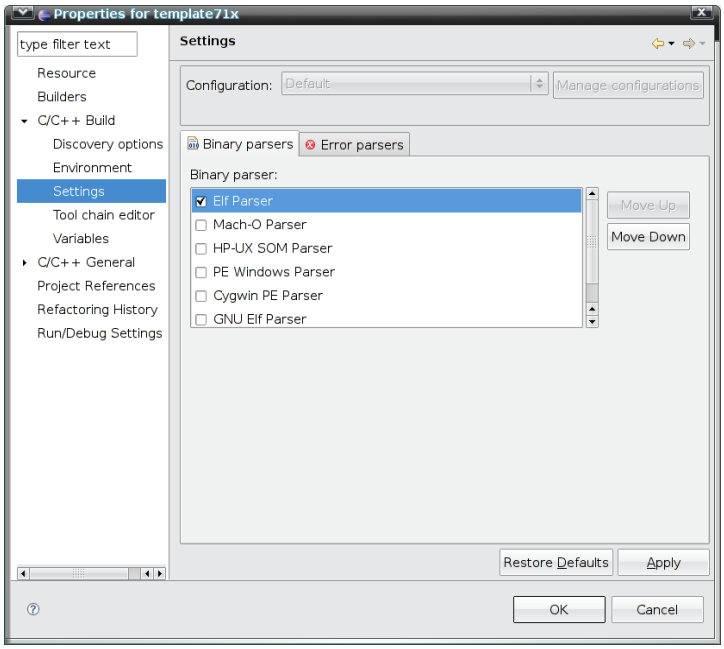
This feature is very useful to avoid scrolling very long and complex source files.

Project building

To build a project you need to tell Eclipse where it can find the right compiler: to do that, right-click on the project, choose *Properties* and, in the window which opens, go to *C/C++ Build* and *Discovery Options*.
Then you have to insert the compiler path in the box *Compiler invocation command* (in our case */usr/local/gnuarm-3.4.3/bin/arm-elf-gcc*), as shown in the following picture:

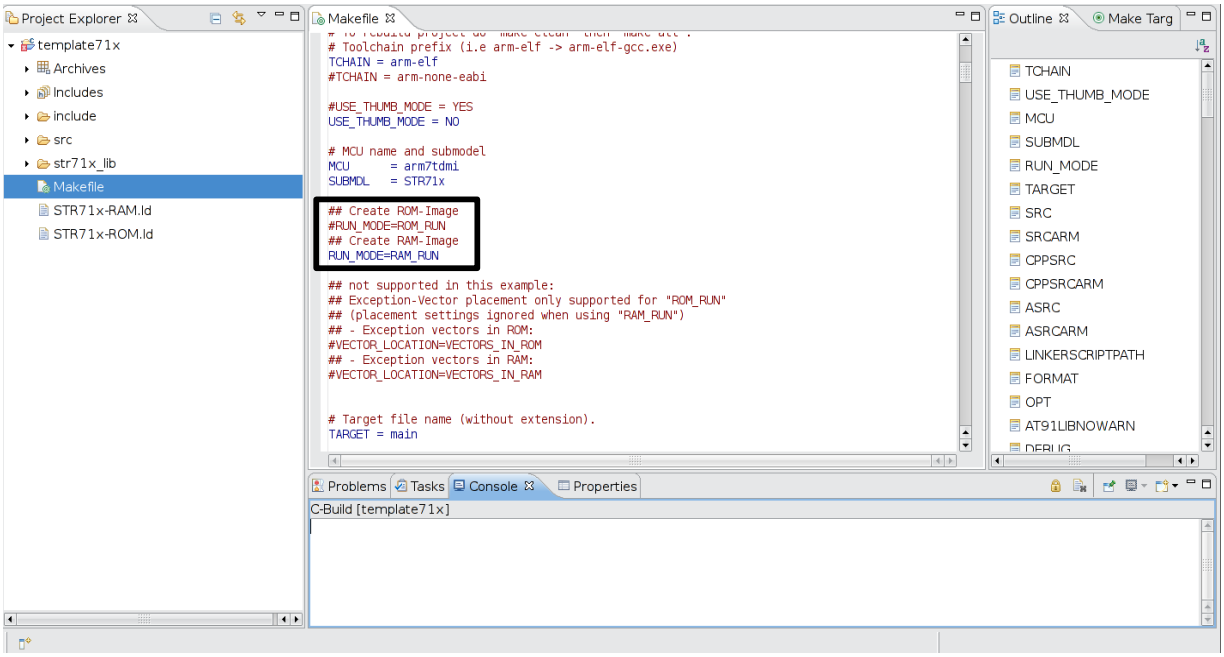


After that, move to *C/C++ Build* and *Settings*. In the window which opens select *Elf Parser*, as shown below:

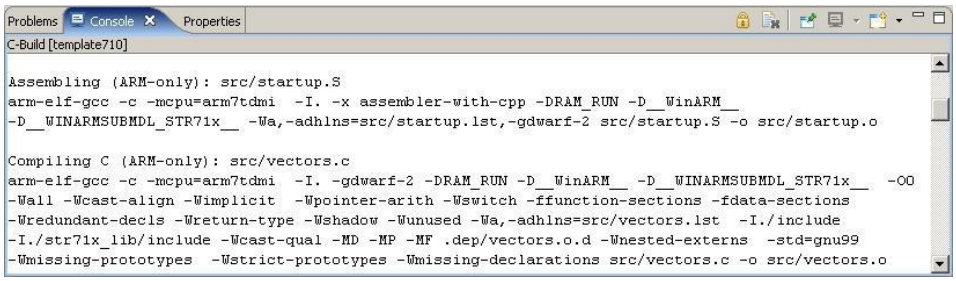


Now clean the project: right-click on the project name and choose *Clean Project*. So the *Console View* will appear (in particular look at the *Console* tab) after the *make clean* execution.

Before building the project, you have to choose if your program will execute in RAM or in FLASH. To do that, you have to modify the right section in *Makefile*, as we did in 2.2.3 paragraph; you can edit the *Makefile* directly from Eclipse through a double click on it.
In the section shown in the picture, you have to uncomment your choice, commenting the other one: for example in this case we chose RAM.

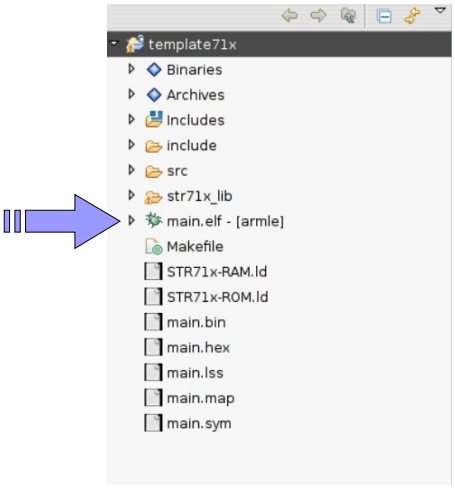


Save the changes in Makefile and then build the project by doing right-click on the project name and choosing *Build Project*.
The *Console* tab of the *Console View* shows the *make* execution, which contains the phases of assembling, compiling and linking of the project.

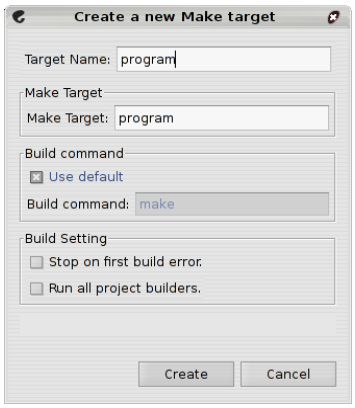


If there are problems, it's possible to see them both in the *Console* tab and in the *Problems* tab which gives more information about what went wrong.
After compilation has been correctly executed, a file *main.elf* is created, as shown in the following

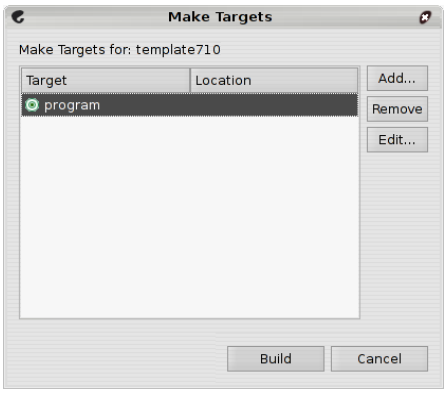
picture:



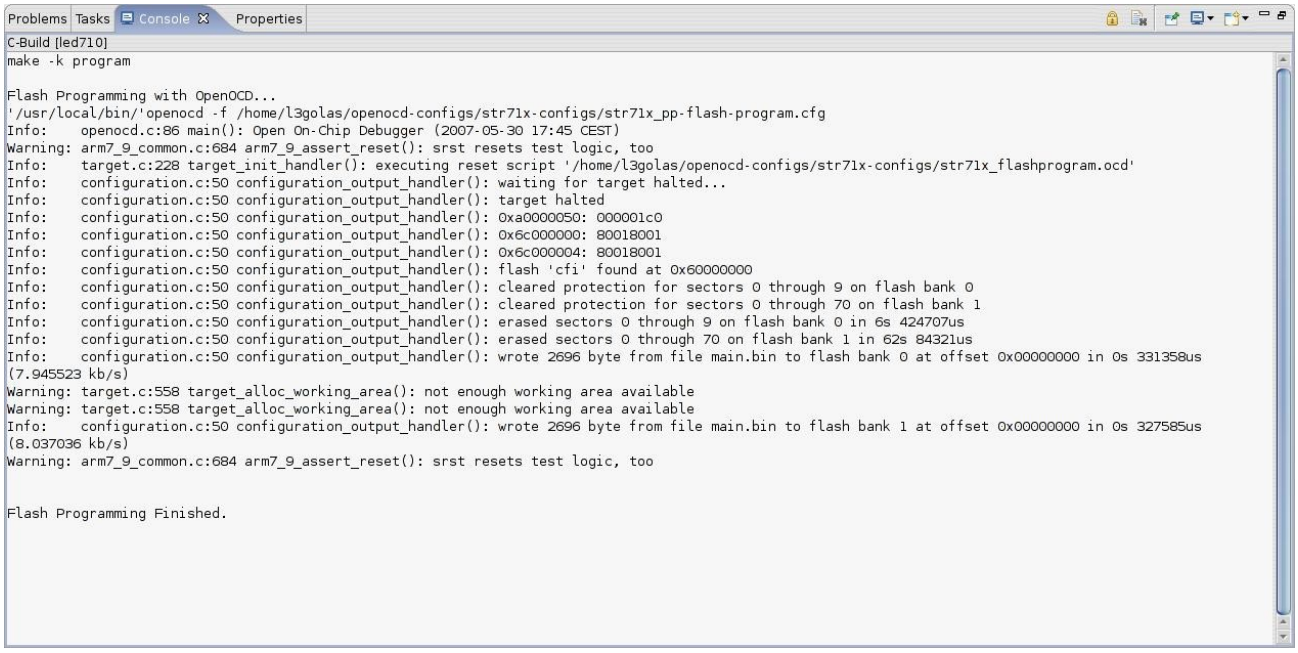
If you chose FLASH in the Makefile, you have to download the program in it (you have to launch *make program*).
To do this operation, you have to create a "*Make target*": this is done by right-clicking on the project name and by choosing then *Make Targets->Create*.



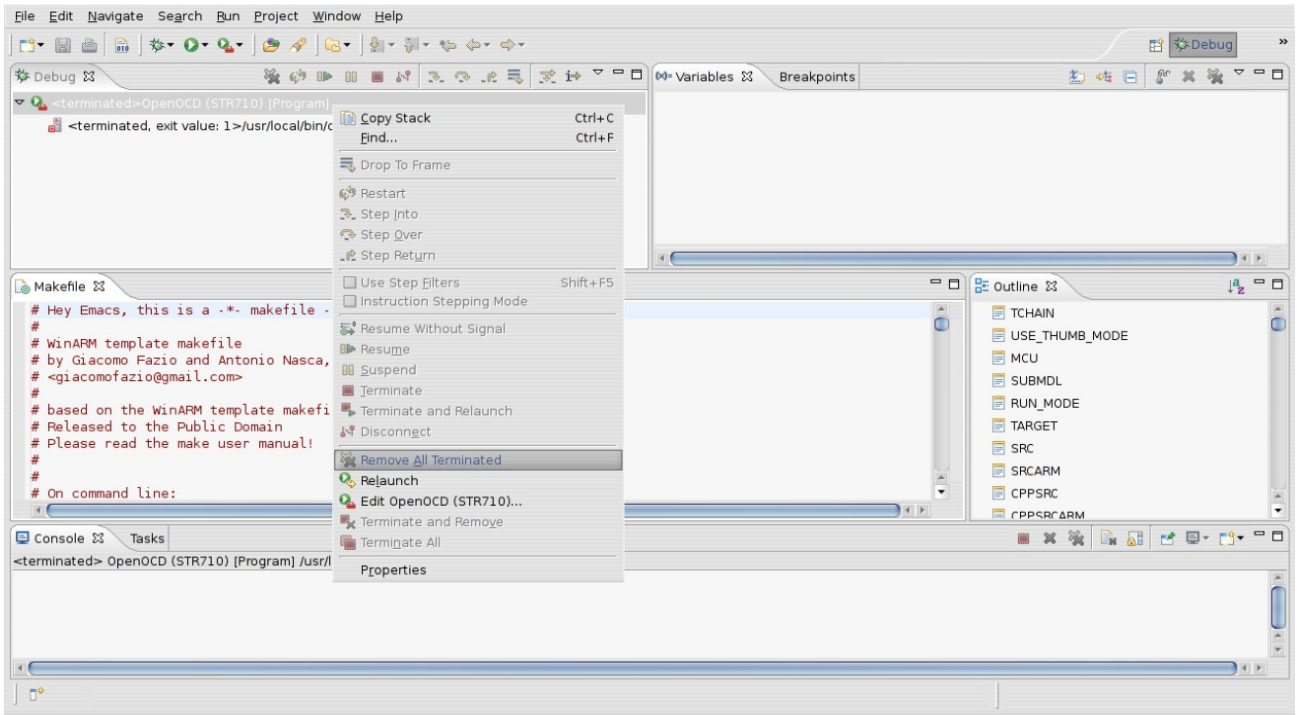
One window will appear, where you have to insert the options shown in the picture on the left.
Don't forget to deselect the checkbox *Run all projects builders*.
Finally press *Create*.
Now, by right-clicking on the project name and by choosing *Make Targets->Build* you will have the window on the right where you have to select your target.



Before pressing the *Build* button, choose *Edit* and make sure all the settings are the same as those you put when you created the target (Eclipse sometimes decides to restore the checkbox *Run all project builders*). Now you can finally click on the *Build* button and in the console you will see the FLASH programming result, similar to the following:

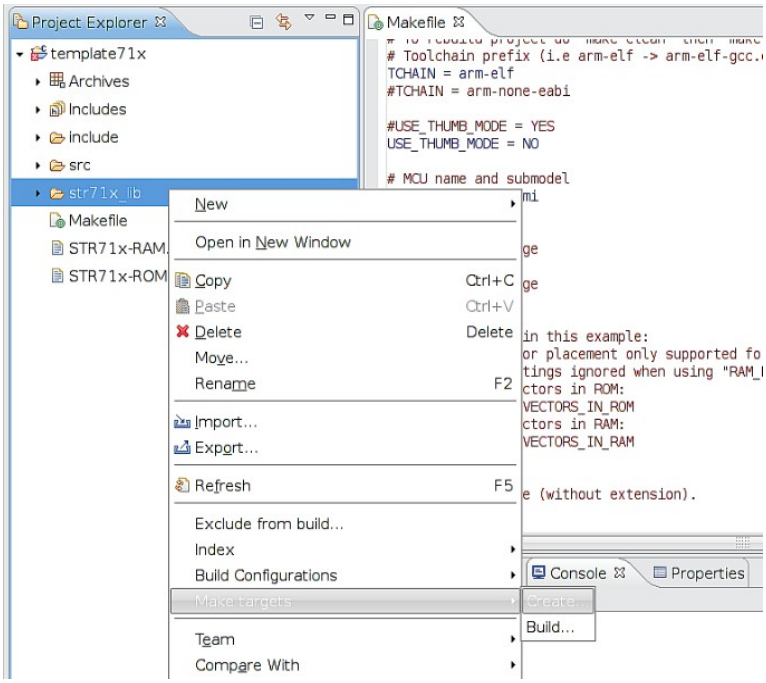


To finish, after having written successfully FLASH memory, switch to *Debug* perspective by going to *Window->Open Perspective->Debug*.
The following window will appear, right-click on "*<terminated>OpenOCD...*" in the *Debug* section and choose *Remove All Terminated*, as shown in the following picture:

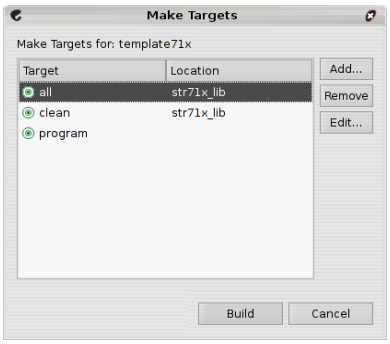


Now come back to the *C/C++* perspective.
Finally last note about software library: probably you will rarely modify and recompile the software library, however it could happen, for example if you modify some settings in *71x_conf.h* file (for more details see chapter 3).
In this case you can create two "Make Targets" on the software library folder, one for the *clean* and the other one for the actual *make*.
To do that, right-click on *str71x_lib* folder and choose *Make Targets->Create*, as shown in the

following picture:



Now insert the two targets *clean* and *all*, following the same steps specified before for the target *program*. So the two new targets will be added to the target *program* already present in the main project. Now when you want to perform a clean and then a recompilation of the software library, it will be sufficient to right-click on the project name and choose *Make Targets->Build*. You will see the following window



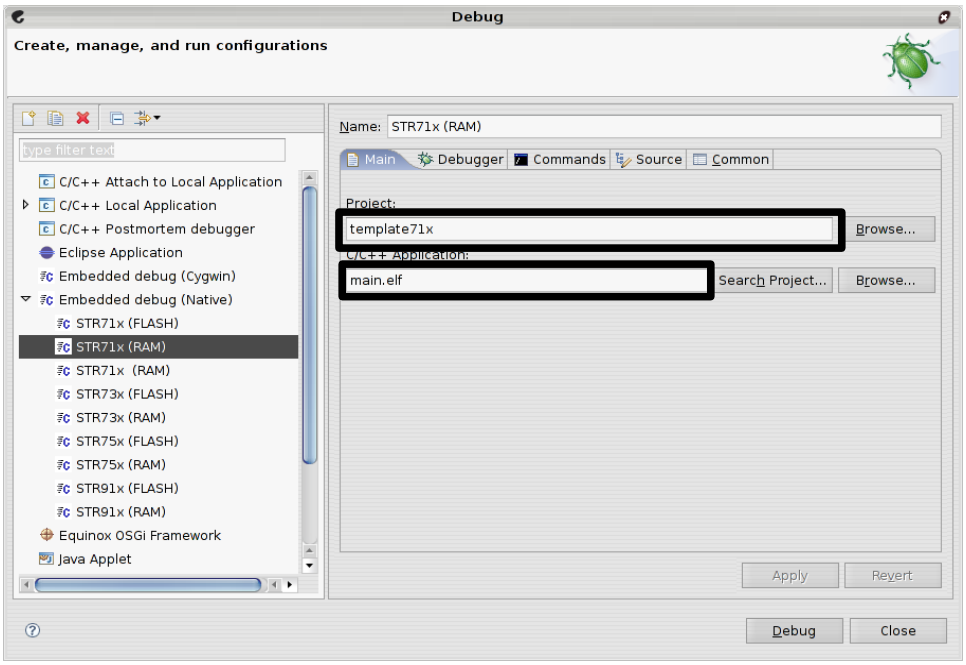
where there will be both the two targets of the software library (you can notice that under *Location* there's *str71x_lib*, which helps you in distinguishing the targets) and the target *program* of the main project.

Project debug

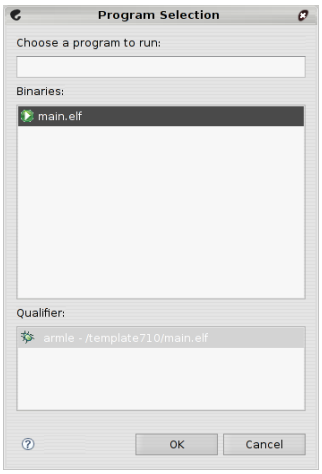
Now let's see how to debug the project you have just compiled (and eventually, already downloaded in FLASH). As usual, we take as example *template71x*, but the operations are the same also for the other projects of the other microcontrollers.
Let's start debugging in RAM: the first thing to do is setting the right section in the Makefile to make the program run in RAM and then recompiling it.

In the debug settings you have the various models of microcontrollers, where you have to add only the project name. Then whenever you have to debug a project, you have to use the correct model for your microcontroller and add the project name, as you will see by reading below.

To debug the program we will use *STR71x (RAM)* model which is already configured with all the parameters, except the project name that you can add immediately going to the *Run->Open Debug Dialog...* and selecting *STR71x (RAM)* from the window that will appear.
On the right side, go to the *Main* section, where you have to fill the two textboxes as shown way below:

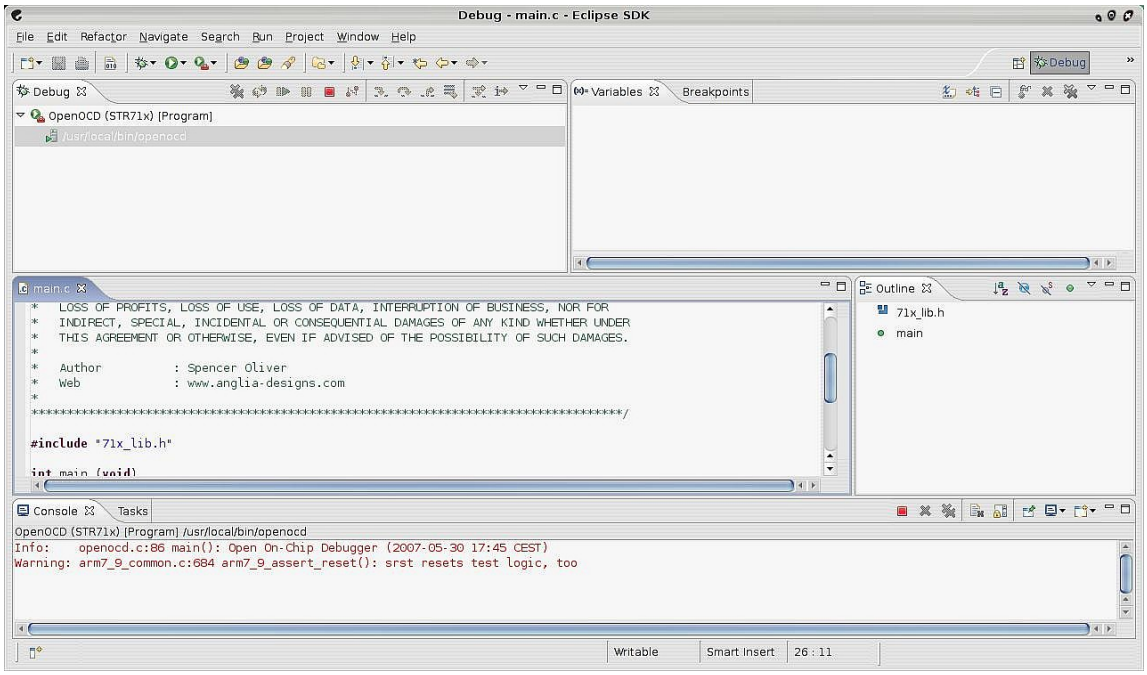


In particular, to fill the first one, click on the *Browse* button located on the textbox's right side and select the project you want to debug; to fill the second one instead, click on the *Search Project* button and select the file *main.elf*, as shown in the following picture:

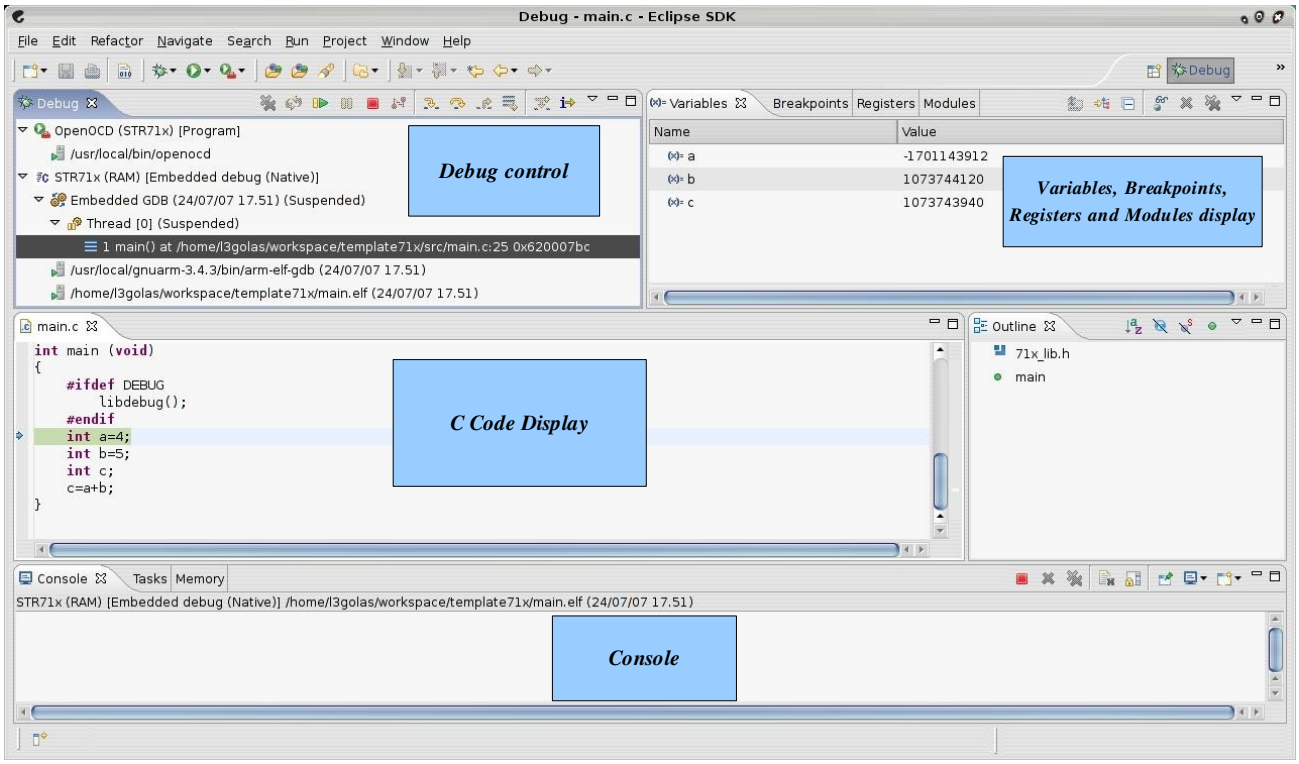


Click on *Apply* and finally on *Close*.
Now you can start debugging. After having powered on the board, switch to *Debug* perspective, go

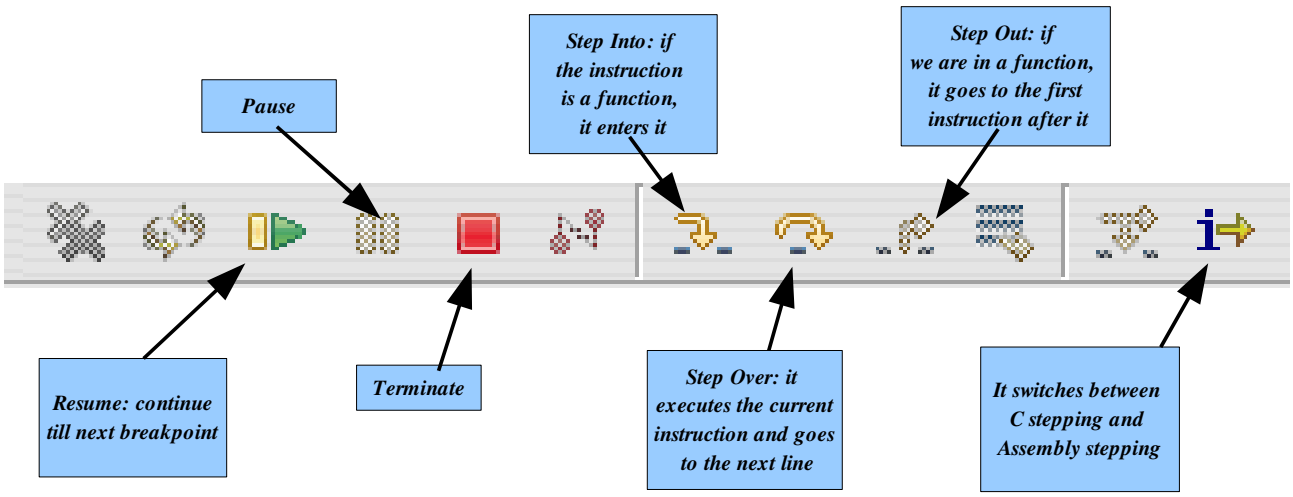
to *Run->External Tools* menu and select *OpenOCD (STR71x)*. If all went fine, in the Console you will see that OpenOCD has blocked waiting for an input, while in Debug section on the upper-left side, OpenOCD executes, as you can see in the following picture:



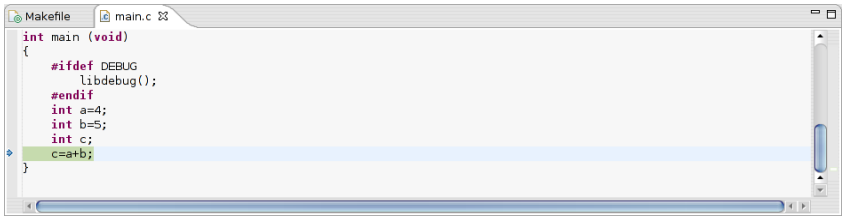
Once that OpenOCD is running, let's start the true actual debug phase, which is based on GDB.
Go to *Run->Open Debug Dialog...*, select *STR71x (RAM)* and click on the *Debug* button. The program should block on the first instruction after the *main*, waiting for any input.
To clarify what explained, look at the following picture, where the fundamental parts of the *Debug* perspective are indicated, to help the debugging operations:



Now you can control the debug through the buttons situated in the Debug section:

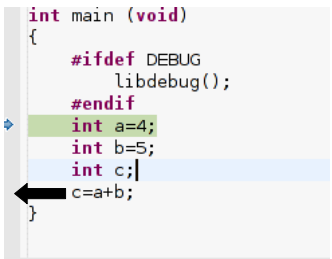


So if you click on the *Step over* button, you will execute only one instruction (in the shown case also *Step Into* will do so because the instructions aren't functions) and the program will block on the following instruction.

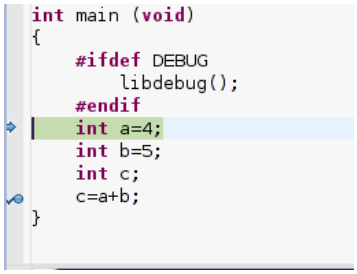


In the source code box, the next instruction to execute is always green-colored with an arrow on the left side.

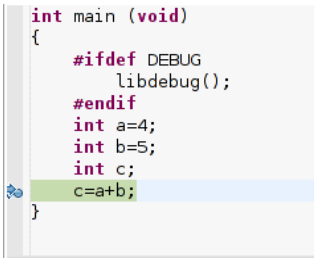
If the code calls a function, the *Step over* button will execute it, without letting you debug the inner code of the function, while the *Step Into* button will enter it, debug its code and return back after completing the execution; if the function is defined in another file, the *Step Into* button will automatically open it. You can also set breakpoints (since you are working in RAM, you use software breakpoints and so there is no limitation in their number) so, if you press the *Resume* button, Eclipse will continue the code execution and will block before to execute the instruction where you put the breakpoint. To set a breakpoint you have to double click on the chosen row's left, as shown in the following picture:



Set the breakpoint where indicated by the arrow, that is the instruction c=a +b

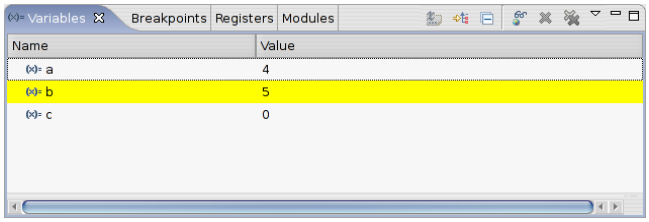


The blue little ball indicates the just set breakpoint. Now press the button Resume



As expected, the execution stopped just before the required instruction. Press Step Over to execute it

If you see the *Variables* tab, shown below, you will find the allocated variables with their value, which is updated in real-time.



To finish the execution, select STR71x (RAM) and press the button *Terminate* (or right-click and choose *Terminate and Remove*) and then do the same with OpenOCD.
At the end you can also right-click and choose *Remove All Terminated*.

Don’t choose *Terminate All* instead of the just described procedure, because it would terminate at the same time both GDB and OpenOCD and this can cause errors or slowdowns in Eclipse; instead you should at first terminate GDB and only then terminate OpenOCD.

Obviously the analysed code is deliberately very simple and it doesn't use any I/O devices, then you will see nothing on the board during execution; however, if this attempt was successfully executed, this means that all the settings are ok and you will be able to compile and debug more complex programs, too.
Now let's try for FLASH the steps you have done for RAM (remember that after you have compiled the code and before starting the debug, you have to download it in FLASH, as explained before).
The difference with RAM debug is obviously that you have to use the STR71x (FLASH) model and that you have to use now hardware (and not anymore software) breakpoints, then you can use a maximum of two breakpoints (or one if you want to use *Step Into* and *Step Over*, since those functions use one breakpoint to stop on the following instruction).

2.2.8 Conclusion

If you arrived here without problems, you have properly installed and tested all the parts of the solution. Now, with the same procedure, you can try to import and debug other and more complex examples that are situated in the subfolder *STR7* of the folder *ARMProjects* provided together with this work: you can find at least one example for each analysed microcontroller.
you can also modify or create new examples using the microcontroller functions, described very well in the Software Library manual of each microcontroller; you can download it from the website http://mcu.st.com/mcu. Make sure also to read the sections about how to develop a program in the chapter 3.

The provided examples are relative to the boards we have used: for example the small program that switches on and off LEDs does that activating bits of one or more GPIO ports that, in our board are connected to those LEDs; however in other boards they could do other things and the LEDs could be connected to different GPIO ports.
So check the code of each example and adapt it to your board, the things to modify shouldn't be so many.

Chapter 3 – Detailed description of the solution

3.1 What you find in the folder "openocd-configs"

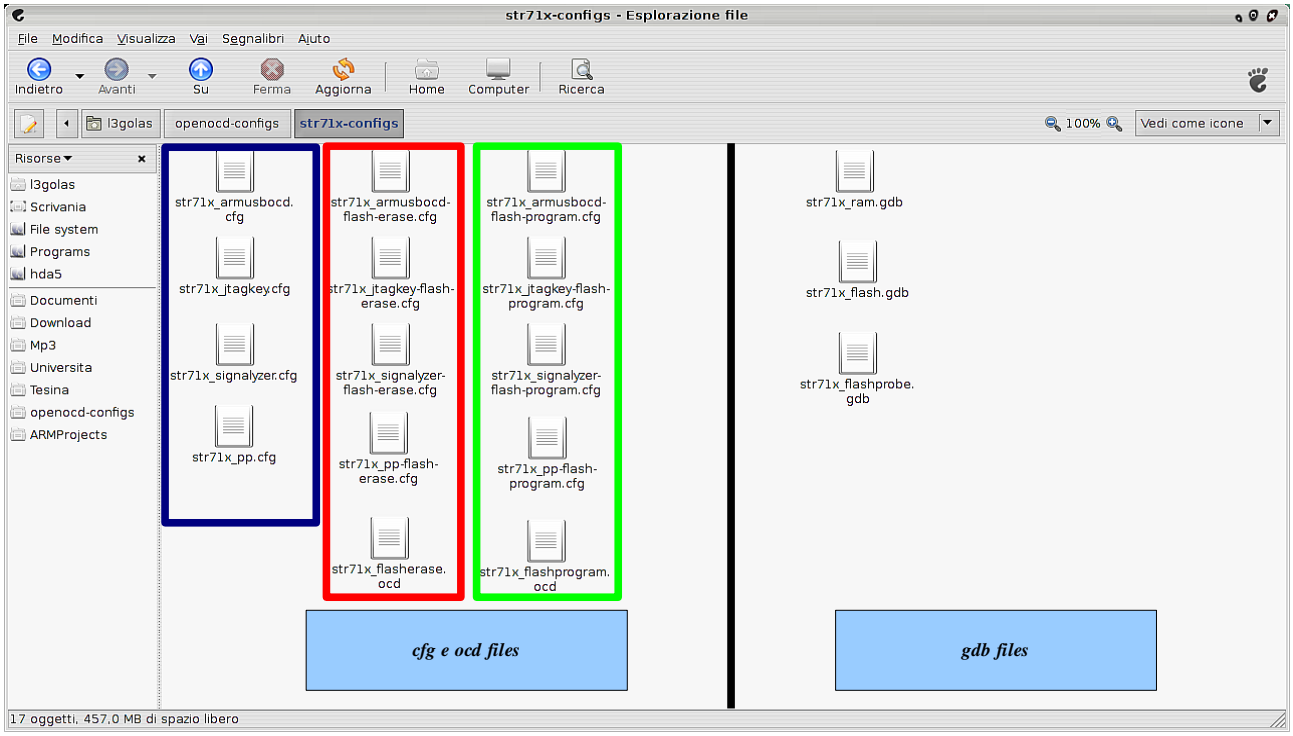
The folder *openocd-configs* that you find together with this work (you find both the Windows and the Linux version) contains all the needed files to debug, program and erase FLASH. It's organized in various subfolders, one for each microcontroller; in each subfolder there are the needed files.

About the STR71x microcontroller, these files are configured to use EMI memory (External Memory Interface), that should be present on the board with this microcontroller.

External memory is also supported by the other microcontrollers of the STR7 family, but we decided to implement it only on the STR71x serie, because it embeds only a small amount of FLASH memory to develop programs.
That's why for the other microcontrollers the files present in the folder *openocd-configs* follow the schema of STR71x without the lines for the EMI configuration.
Anyway, in the future it will be possible to implement the EMI support for the other microcontrollers, too.

Don't forget that the lines for STR71x EMI implementation are relative only to our board, because in the other ones, even if they have the same microcontroller, the implementation of EMI memory could be different.

This information and much more are present in the microcontroller's reference manual, which you can download from the site http://mcu.st.com/mcu.
Why do we explain those lines? This is due to two reasons: the first one is that they told us to take note of everything we did; the second one, for which we invite everyone to read further, is that the logic we used could also be applied to other boards, though there could be differences in registers and variables.
Since the STR71x files are the most complex ones, we will use them as example.
Don't forget that those files use very much OpenOCD options and commands, so if you want more information about all the options and their meaning, visit the website http://openfacts.berlios.de/index-en.phtml?title=Open_On-Chip_Debugger and read the sections there reported (*Configuration*, *commands*, etc.).
Let's see the content of the subfolder *str71x-configs*:



3.1.1 cfg and ocd files

In the paragraph 2.2.1 we dealt with how to install and use OpenOCD, saying that you must launch it specifying a configuration file that contains the description of the target's parameters. You saw the OpenOCD different functions: it's not only a GDB server, but it's also used to program and erase FLASH. You also saw how to configure Eclipse to execute easily OpenOCD for those functions.
So, these files can be divided in three groups:

Files used to debug

They are indicated in the picture with the blue color and they can be distinguished by the interface used for the connection. They must be launched together with OpenOCD in the following way (we talked about this in the chapter 2:

```
openocd -f FILENAME
```

where *FILENAME* must be substituted by the right *cfg* file.

> We made our tests using only parallel port connection, so configuration files that make use of other solutions (taken from examples present especially on the OpenOCD site), are provided as they are, don't complain if they don't work properly.

For example, the file used for the connection by parallel port, is called *str71x_pp.cfg*. Let's read its content:

```
#daemon configuration
telnet_port 4444          <= port on which OpenOCD must wait to receive Telnet connections
gdb_port 3333             <= port on which OpenOCD must wait to receive GDB connections

#interface
```

```
interface parport                    <- used interface
parport_port 0                       <- interface port
parport_cable wiggler                <- used parallel cable
jtag_speed 0                         <- JTAG interface speed, in this case we put maximum speed

#use combined on interfaces or targets that can't set TRST/SRST separately
reset_config trst_and_srst srst_pulls_trst   <- configura i segnali di reset

#jtag scan chain
#format L IRC IRCM IDCODE (Length, IR Capture, IR Capture Mask, IDCODE)
jtag_device 4 0x1 0xf 0xe            <- parameters of the devices that form JTAG chain

#target configuration
daemon_startup reset                 <- this line resets target when the daemon connects to it

#target <type> <startup mode>
#target arm7tdmi <reset mode> <chainpos> <endianness> <variant>
target arm7tdmi little run_and_halt 0 arm7tdmi   <- info on the target to debug and on what to do after reset
run_and_halt_time 0 30               <- how long OpenOCDmust wait after reset to block itself waiting for debug commands

working_area 0 0x2000C000 0x4000 nobackup   <- working area parameters, not fundamental but sometimes useful
#flash bank <driver> <base> <size> <chip_width> <bus_width>
flash bank str7x 0x40000000 0x00040000 0 0 0 STR71x   <- internalFLASH parameters
flash bank cfi   0x60000000 0x00400000 2 2 0          <- external FLASH parameters

# For more information about the configuration files, take a look at:
# http://openfacts.berlios.de/index-en.phtml?title=Open+On-Chip+Debugger
```

The other files follow this schema, too, with minor changes, which depend on both the target type and the interface you use.
For example the parallel port interface is *0* on Linux and *0x378* on Windows, the working area is an area that is part of the RAM, so its parameters (start address and length) vary depending on the used microcontroller; flash banks vary depending on the target and, if present, it's possible to add other banks after the first one, as we did in this case, in which we added the line

```
flash bank cfi   0x60000000 0x00400000 2 2 0
```

Another word about the function *run_and_halt*, that is present in the line of target configuration: it commands to OpenOCD that after the target reset (required in the previous lines) it has to block waiting the debugging commands.
After you have launched OpenOCD with one of those files as input, you are ready to start debugging, using a *gdb* file, that we will describe in the paragraph 3.1.2.

Files used for FLASH programming

These are the files that we indicate in the picture in green, they distinguish from the others by the *flash-program* termination (for the *cfg* files) and *flashprogram* (for the *ocd* file). The *cfg* files distinguish each from the others by the interface used for the connection. They are given as input to OpenOCD in the same way than the previous group and each one calls the *ocd* file, whose task is to write on FLASH a program you have already built (remember this is an electric operation). For example, the *cfg* file to give as input to OpenOCD if you want to use parallel port, is *str71x_pp-flash-program.cfg*:

```
#daemon configuration
telnet_port 4444
gdb_port 3333
```

```
#interface
interface parport
parport_port 0
parport_cable wiggler
jtag_speed 0
jtag_nsrst_delay 200
jtag_ntrst_delay 200

#use combined on interfaces or targets that can't set TRST/SRST separately
reset_config trst_and_srst srst_pulls_trst

#jtag scan chain
#format L IRC IRCM IDCODE (Length, IR Capture, IR Capture Mask, IDCODE)
jtag_device 4 0x1 0xf 0xe

#target configuration
daemon_startup reset

#target <type> <startup mode>
#target arm7tdmi <reset mode> <chainpos> <endianness> <variant>
target arm7tdmi little run_and_init 0 arm7tdmi
run_and_halt_time 0 30

working_area 0 0x2000C000 0x4000 nobackup

#flash bank <driver> <base> <size> <chip_width> <bus_width>
flash bank str7x 0x40000000 0x00040000 0 0 0 STR71x
flash bank cfi   0x60000000 0x00400000 2 2 0

#Script used for FLASH programming
target_script 0 reset /home/l3golas/openocd-configs/str71x-configs/str71x_flashprogram.ocd
```

As you can see, the file is very similar to the one used for the debugging (since the board is the same), let's see the changes:

1) The two lines

```
jtag_nsrst_delay 200
jtag_ntrst_delay 200
```

set a 200 ms delay from when *nSRST* (*nTRST* for the second instruction) becomes inactive and when JTAG operations start.

2) The line

```
target arm7tdmi little run_and_init 0 arm7tdmi
```

contains the option *run_and_init* instead of *run_and_halt*, because this time you haven't to block waiting for debugging commands, instead you have to run *ocd* file.
3) Finally the line

```
target_script 0 reset /home/l3golas/openocd-configs/str71x-configs/str71x_flashprogram.ocd
```

used to call *str71x_flashprogram.ocd* (you have to change the path following this file's location), which we indicate below:

```
wait_halt
mww 0xE0005000 0x000F
mww 0xE0005004 0x000F
mww 0xE0005008 0x000F
mww 0x6C000000 0x8001
mdw 0x6C000000
mww 0x6C000004 0x8001
mdw 0x6C000004
flash probe 1
```

```
sleep 1000
flash protect 0 0 9 off
flash protect 1 0 70 off
flash erase 0 0 9
flash erase 1 0 70
flash write 0 main.bin 0
flash write 1 main.bin 0
reset
shutdown
```

Let's explain each line's meaning. The first one stops the CPU and waits. The lines

```
mww 0xE0005000 0x000F
mww 0xE0005004 0x000F
mww 0xE0005008 0x000F
mww 0x6C000000 0x8001
mdw 0x6C000000
mww 0x6C000004 0x8001
mdw 0x6C000004
```

aren't present in the files for the other microcontrollers, their task is to configure properly the EMI external memory (if you don't have it on your board, it's sufficient to uncomment the lines by preceding them by #).
Let's skip for now the first 3 lines and start explaining the last 4 ones

```
mww 0x6C000000 0x8001
mdw 0x6C000000
mww 0x6C000004 0x8001
mdw 0x6C000004
```

that enable the first two EMI banks, let's see why. This is the map of the external memory supported by the microcontroller:

Table 7. EMI Memory Map

Address Range	Description	Addressable Size (Bytes)	Bus Width (bit)
0x6000 0000 - 0x60FF FFFF	Bank 0 - BOOT (CSn.0)	16M	16 bit access only[1)]
0x6200 0000 - 0x62FF FFFF	Bank 1 (CSn.1)	16M	8/16 SW Selectable
0x6400 0000 - 0x64FF FFFF	Bank 2 (CSn.2)	16M	8/16 SW Selectable
0x6600 0000 - 0x66FF FFFF	Bank 3 (CSn.3)	16M	8/16 SW Selectable
0x6C00 0000 - 0x6C00 0010	Internal Registers	n/a	16 bit access only

How you can see, the microcontroller supports 4 banks, each one has a maximum dimension of 16 MB. On the board we used there are only 2 banks of memory: *bank0* of FLASH and *bank1* of SRAM, both of 4 MB, while *bank2* is used by the LCD display. At the address *0x6C000000* there are 4 registers, that we show in the following table:

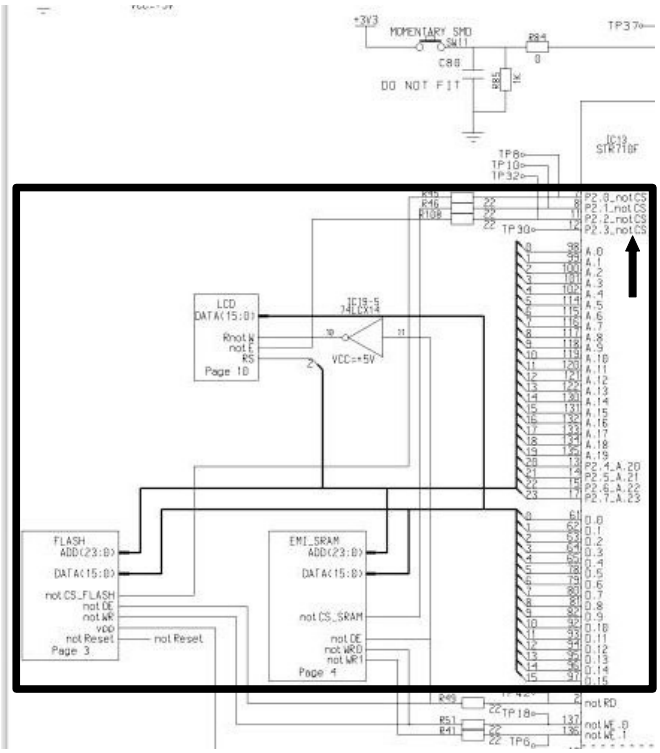
Table 8. EMI Register Map

Addr. Offset	Register Name	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	**BCON0**	BE	reserved									C_LENGTH[3:0]				B_SIZE[1:0]	
4	**BCON1**	BE	reserved									C_LENGTH[3:0]				B_SIZE[1:0]	
8	**BCON2**	BE	reserved									C_LENGTH[3:0]				B_SIZE[1:0]	
C	**BCON3**	BE	reserved									C_LENGTH[3:0]				B_SIZE[1:0]	

Each of them sets one of the 4 banks of memory: BCON0 sets *bank0*, BCON1 sets *bank1* and so on. Being, in this case, interested in *bank0* (FLASH) and *bank1* (SRAM), you have to work on BCON0 and BCON1, located respectively at the addresses *0x6C000000* and *0x6C000004*: you have to set the bit 15 to 1 to enable the bank (by default it's not enabled), and you also have to set the bit 0 to 1 to set the effective dimension of the external bus to 16 bit. That's why we have the two lines *monitor mww 0x6C000000 0x8001* and *monitor mww 0x6C000004 0x8001*, which corresponds to *1000000000000001* in binary.
The lines *monitor mdw 0x6C000000* and *monitor mdw 0x6C000004* are used only to show the two registers value, in order to make sure they have been written properly.
Now let's talk about the 3 lines we skipped before :

```
monitor mww 0xE0005000 0x000F
monitor mww 0xE0005004 0x000F
monitor mww 0xE0005008 0x000F
```

With these lines you set the GPIO2 port, that in our board is necessary to make EMI (so external FLASH, RAM and LCD) work properly, as you can see in the following piece of the datasheet, in which you can notice a microcontroller part.



In the schematics, we have indicated with an arrow the pins from 0 to 3 of the GPIO2 port, whose first three ones make respectively FLASH, RAM and LCD work. The idea is to set the 4 pins as AF (Alternate Function). To do so, for each I/O port there are 3 16-bit registers (PC0, PC1 e PC2) and 1 16-bit data register (PD), shown in the following table:

Table 21. I/O-port Register Map

Addr. Offset	Register Name	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	PC0	C0[15:0]															
4	PC1	C1[15:0]															
8	PC2	C2[15:0]															
C	PD	D[15:0]															

The registers of GPIO2 port are located at the base address *0xE0005000*. You have to work on the three configuration registers, that have to be set together: imagine them as a 3x16 table in which the three registers are the rows while each column is used to set the correspondent bit of the port, which goes from 0 to 15. So, since you need to set the pins from 0 to 3 as AF, you have to set the last 4 columns. But how can you set them as AF? The answer can be found in the following:

Table 20. Port Bit Configuration Table

Port Configuration Registers (bit)	Values							
PC0(n)	0	1	0	1	0	1	0	1
PC1(n)	0	0	1	1	0	0	1	1
PC2(n)	0	0	0	0	1	1	1	1
Configuration	HiZ/AIN	IN	IN	IPUPD	OUT	OUT	AF	AF
Output	TRI	TRI	TRI	WP	OD	PP	OD	PP
Input	AIN	TTL	CMOS	CMOS	N.A.	N.A.	CMOS	CMOS

You have to fill each of the last 4 columns with 1. How can you do it? Using the 3 instructions of the *gdb* file of which we are talking:

```
monitor mww 0xE0005000 0x000F
monitor mww 0xE0005004 0x000F
monitor mww 0xE0005008 0x000F
```

Doing so you set in each of the 3 registers the last 4 bits to 1 (*0x000F* is in binary *1111*).
Now return to the *ocd* file and see the next two lines:

```
flash probe 1
sleep 1000
```

used respectively to make sure bank1 has the same parameters we described for it in the *cfg* file and to stop all for 1 second. The lines

```
flash protect 0 0 9 off
flash protect 1 0 70 off
flash erase 0 0 9
```

```
flash erase 1 0 70
flash write 0 main.bin 0
flash write 1 main.bin 0
reset
shutdown
```

are easy to understand: the first 2 lines deactivate the first two FLASH banks protection (9 e 70 are respectively the number of sectors of the first bank and of the second one, notice how the second one is bigger than the first one); after that you erase both the banks (to erase the second one it can take some minutes) and then you write on it the file *main.bin* (obtained by a previous compilation) starting from the beginning of the bank. At the end let's see the lines

```
reset
shutdown
```

used respectively to reset the target and to exit OpenOCD returning control to the user.

Files for FLASH erasing

These files are indicated in picture in red and distinguish from the others for the termination *flash-erase* (for the *cfg* files) and *flasherase* (for the *ocd* ones). The *cfg* files generally distinguish from each other by the interface used for the connection. They are given as input to OpenOCD as for the previous group, and they call the *ocd* file, whose task is to erase FLASH (remember it's an electrical operation). For example, the *cfg* file to give as input to OpenOCD is *str71x_pp-flash-erase.cfg*:

```
#daemon configuration
telnet_port 4444
gdb_port 3333

#interface
interface parport
parport_port 0
parport_cable wiggler
jtag_speed 0
jtag_nsrst_delay 200
jtag_ntrst_delay 200

#use combined on interfaces or targets that can't set TRST/SRST separately
reset_config trst_and_srst srst_pulls_trst

#jtag scan chain
#format L IRC IRCM IDCODE (Length, IR Capture, IR Capture Mask, IDCODE)
jtag_device 4 0x1 0xf 0xe

#target configuration
daemon_startup reset

#target <type> <startup mode>
#target arm7tdmi <reset mode> <chainpos> <endianness> <variant>
target arm7tdmi little run_and_init 0 arm7tdmi
run_and_halt_time 0 30

working_area 0 0x2000C000 0x4000 nobackup

#flash bank <driver> <base> <size> <chip_width> <bus_width>
flash bank str7x 0x40000000 0x00040000 0 0 0 STR71x
flash bank cfi   0x60000000 0x00400000 2 2 0

#Script used for FLASH erasing
target_script 0 reset /home/l3golas/openocd-configs/str71x-configs/str71x_flasherase.ocd
```

As you can see, the file is nearly the same of the file used for FLASH programming, except the last

line

```
target_script 0 reset /home/l3golas/openocd-configs/str71x-configs/str71x_flasherase.ocd
```

that you use to call the file *str71x_flasherase.ocd* (you must change the path depending on the file's position), that we indicate right now:

```
wait_halt
mww 0xE0005000 0x000F
mww 0xE0005004 0x000F
mww 0xE0005008 0x000F
mww 0x6C000000 0x8001
mdw 0x6C000000
flash probe 1
sleep 1000
flash protect 0 0 9 off
flash protect 1 0 70 off
flash erase 0 0 9
flash erase 1 0 70
reset
shutdown
```

This file is very similar to the one used for FLASH programming, so please read that section for more information, here we only talk about the differences: this file doesn't use the external SRAM because you don't need it in this moment; moreover, after the FLASH erasing, obviously you don't write anything on it.

3.1.2 GDB startup files

They contain the commands that the GDB based debugger (GDB client) must run before starting to debug the application. Located in the directory *openocd-configs*, they have the *gdb* extension and they are present in RAM and FLASH versions. They are executed when you start Insight and you used their content when you configured the debug in Eclipse (see Chapter 2). In fact, as we already said, Eclipse uses GDB for the debugging, talking with it through the GDB/MI protocol.

The GDB files must be started after you start OpenOCD, because it works as a server, so it knows how to handle them.

Let's see the content of these files:

str71x_flash.gdb

```
target remote localhost:3333
monitor reset
monitor sleep 500
monitor poll
monitor soft_reset_halt
monitor arm7_9 force_hw_bkpts enable
monitor mww 0xA0000050 0x01c0
monitor mdw 0xA0000050
monitor mww 0xE0005000 0x000F
monitor mww 0xE0005004 0x000F
monitor mww 0xE0005008 0x000F
monitor mww 0x6C000000 0x8001
monitor mdw 0x6C000000
monitor mww 0x6C000004 0x8001
monitor mdw 0x6C000004
```

Through the first line, the GDB client is requested to send (through the RSP protocol) the

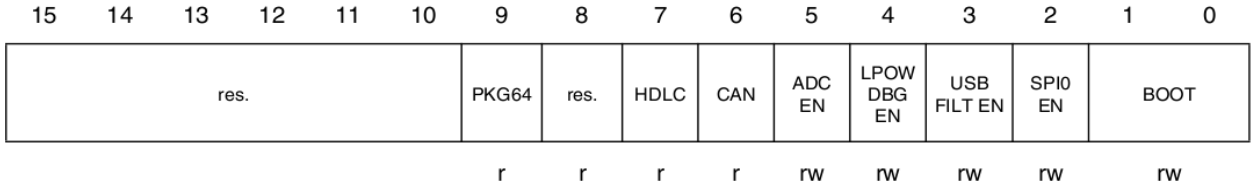
debugging commands to the GDB server (OpenOCD), which listens on the TCP 3333 port.
The lines

```
monitor reset
monitor sleep 500
monitor poll
monitor soft_reset_halt
```

ask OpenOCD (if the command begins with the word "*monitor*", it's not a GDB command, but it's a command for OpenOCD) to reset the target, to wait 500 msec, to do a polling of the target to obtain the current state and to do at the end a *soft reset* followed by a waiting, respectively.
The three following lines

```
monitor arm7_9 force_hw_bkpts enable
monitor mww 0xA0000050 0x01c0
monitor mdw 0xA0000050
```

ask OpenOCD to enable the hardware breakpoints (we use them only for FLASH debugging, in that for the RAM debugging we use the software breakpoints), to write on the register located at the address *0xA0000050* the value *0x01c0* and to read and show you that register value to make sure it has been written properly, respectively. Why is it necessary to write that register? It's a register present (with different names and addresses) in all the STR7 family microcontrollers and it establishes if the boot from RAM, FLASH or external memory: if for example you set that register for the boot from FLASH, FLASH will be mapped at the address *0* too, so the system will boot from it (you can notice this since, if you have written a program in FLASH and you switch on the board, the program runs instantaneously). The board has surely some jumpers that you can use to set the boot in different ways (refer to the board user manual) and the position of those jumpers automatically sets the right value in the register we talked about; however it's possible to override via software the register value by writing directly in the register, that's why we are doing so. All this information is present in the microcontroller reference manual, please refer to it for further details. In this case the register is called *PCU_BOOTCR* and this is its structure:



The bits in position *0* and *1* are the two bits that rule BOOT, and this is the table with the different configurations:

BOOT EN	BOOT1 (B1)	BOOT0 (B0)	Mode	Boot Memory Mapping	Note
0	any	any	USER	FLASH mapped at 0h	• System executes code from Flash
1	0	0			
1	0	1	System-Memory	SystemMemory mapped at 0h	• System executes a factory installed boot loader from SystemMemory (Reserved mode). • Clock FROZEN
1	1	0	RAM	RAM mapped at 0h	• System executes code from internal RAM • For Lab development.
1	1	1	EXTMEM	EXTMEM mapped at 0h	• System executes code from external memory

We are interested in BOOT from FLASH, so it's necessary that the bits in the positions 0 and 1 of the register are set to 0: the exadecimal values *0x01c0* we wrote, in fact, corresponds to *111000000* (the bits 0 and *1* have so the requested values).
After this digression, please return to the *gdb* file. Look at the remaining lines:

```
monitor mww 0xE0005000 0x000F
monitor mww 0xE0005004 0x000F
monitor mww 0xE0005008 0x000F
monitor mww 0x6C000000 0x8001
monitor mdw 0x6C000000
monitor mww 0x6C000004 0x8001
monitor mdw 0x6C000004
```

you saw them in the *ocd* file used to program FLASH and here they have the same meaning (and they vary from board to board), that is to activate the first two banks of external memory (FLASH and RAM) and configure the GPIO2 port that drives the external memory. For more details refer to the previous paragraph.

str71x_ram.gdb

```
target remote localhost:3333
monitor reset
monitor sleep 500
monitor poll
monitor soft_reset_halt
monitor arm7_9 sw_bkpts enable
monitor mww 0xA0000050 0x01c2
monitor mdw 0xA0000050
monitor mww 0xE0005000 0x000F
monitor mww 0xE0005004 0x000F
monitor mww 0xE0005008 0x000F
monitor mww 0x6C000004 0x8001
monitor mdw 0x6C000004
```

These are the lines present in the *gdb* file of the STR71x microcontroller and their function is to set

the boot from RAM. As you can see it's really similar to the FLASH version with few changes: the line *monitor arm7_9 sw_bkpts enable* enables the software breakpoints (in the FLASH version we used the hardware ones); then, through the line *monitor mww 0xA0000050 0x01c2*, you set the registry at address *0xA0000050* with the value *0x01c2* (in binary *111000010*) to set the boot from RAM. The remaining lines configure, as for FLASH, the bits from 0 to 3 of the GPIO2 as AF and then, through the line *monitor mww 0x6C000004 0x8001*, you enable only the SRAM bank in the external memory, in that in this case the FLASH external bank is not necessary.

Remember RAM is a volatile memory. So if you put your program in RAM, when you switch off the board it will be automatically erased; for FLASH it's different, because when you write on it a program, it can be erased only with a proper electrical operation.

str71x_flashprobe.gdb

```
target remote localhost:3333
monitor reset
monitor sleep 500
monitor poll
monitor soft_reset_halt
#monitor flash protect 0 0 9 off
monitor flash banks
monitor flash probe 0
monitor flash info 0
monitor flash probe 1
monitor flash info 1
monitor reset run
```

We decided to insert this file because it gives information about the FLASH banks that are present, in this case two. We didn't insert it in Eclipse, so you have to run it when you start Insight and after you start OpenOCD, as you do for the other *gdb* files (see paragraph 2.2.3, the part dealing with debugging through Insight). The file is entirely made of commands directed to OpenOCD (they start all with *monitor*). After the initial commands you saw in the other *gdb* file, too, you can see the line showing the banks set in the *cfg* file which you put as input to OpenOCD. Then there are the two lines

```
monitor flash probe 0
monitor flash info 0
```

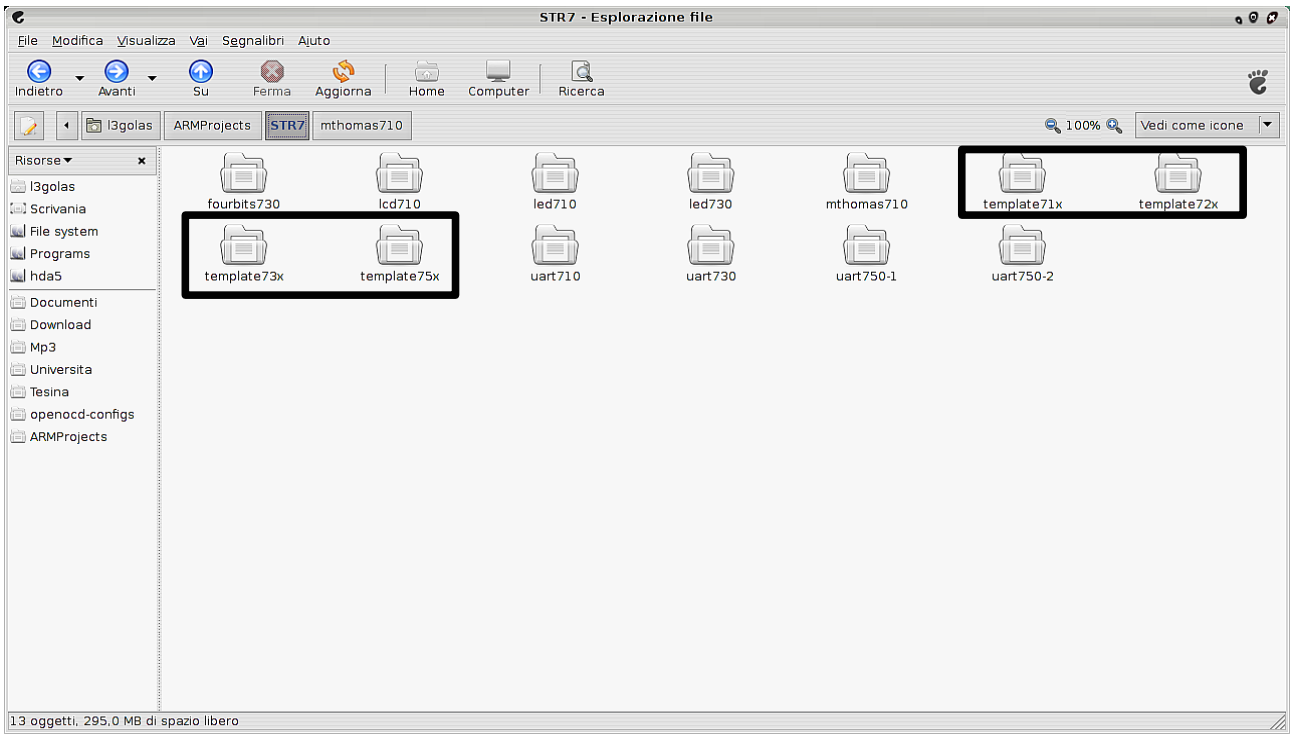
that respectively make sure the information you inserted in the *cfg* file about bank 0 are more or less correct and show information on this bank; the same thing is done for the other bank through the successive two lines. The last line

```
monitor reset run
```

resets the target.

3.2 Folder “ARMProjects” content

This folder contains all the templates and the examples you have created, divided by microcontroller family (in fact you can find the folders STR7 and STR9). For example the following is the folder STR7 content:



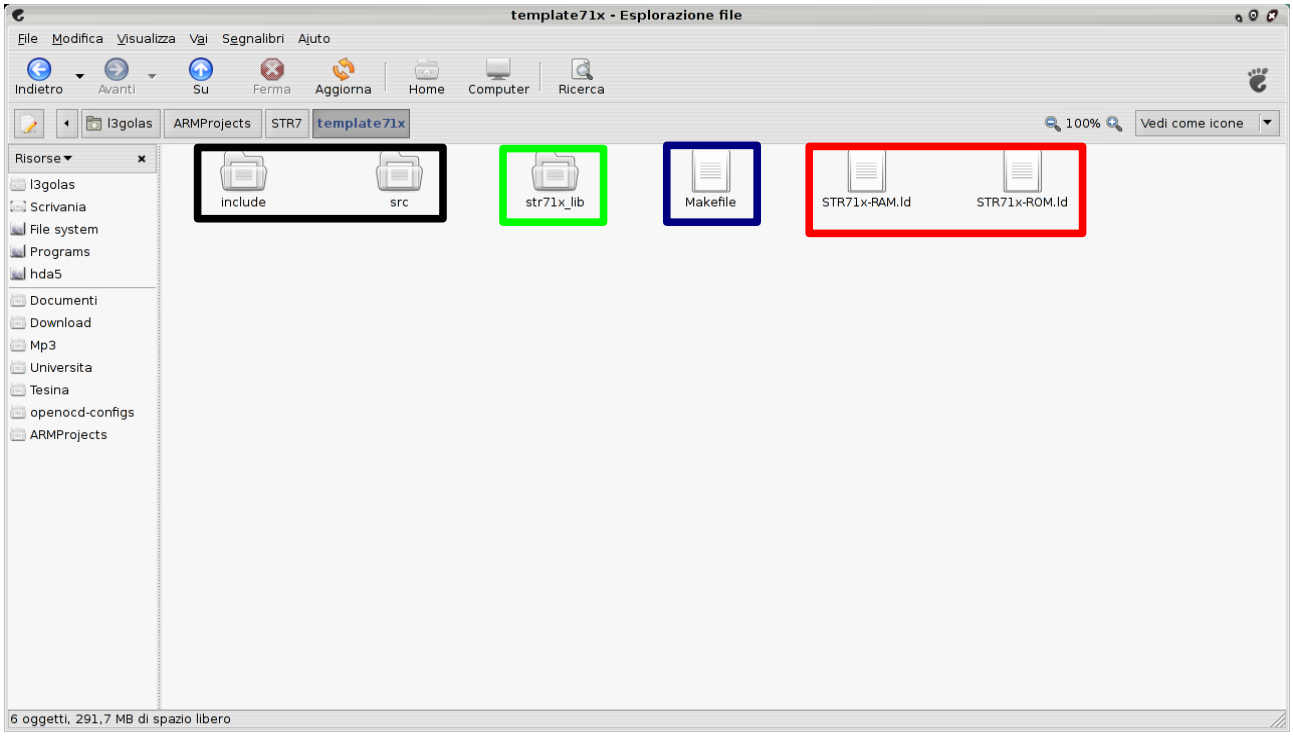
We circled in black the STR71x, STR72x, STR73x and STR75x templates.
For each microcontroller we developed some simple examples, by using the software library functions. In addition, for the STR71x microcontroller, we have added the example *mthomas710*, directly derived from the program *STR71x Demo 1*, created by Martin Thomas and available at the address
http://gandalf.arubi.uni-kl.de/avr_projects/arm_projects/index_str.html#str7_demo1.

As already specified, the examples we provide are relative to the boards we have used: for example the program which switches on and off the LEDs, does that by setting the bits of one or more GPIO ports that, in our board, are linked to those LEDs, while in other boards could serve to something else and the LEDs could be linked to different GPIO ports. So please check each example code and adapt it to your board, the things you have to modify shouldn't be so many.

Now we deal with the template and its parts, and we cover the parts about how we created it and how to create a program from it.

3.2.1 The template structure

You can find a template for each microcontroller and you have to start from it when you want to create a program. Let's see its parts, considering as example the STR71x template:



From the picture you can see how the template is done by many components, that must be used together to obtain, as final result, the compiled program; in doing that, you have to follow some rules, described in the *Makefile* (blue square). Now let's see in detail each of the template components:

Software library

It's the folder represented in the picture in the green square. We already dealt with it in the tutorial chapter, so you already know that it contains the ST software library files, that is all the functions and variables that allow you to use completely the microcontroller functionalities. In particular this version of the software library was slightly modified by Anglia, and contains some facilities if compared to the original version: for more information please refer to the 3.2.2 paragraph. The folder also contains a *Makefile*, used to compile the software library files in a library (*.a* file), which you can include in your programs. It's strongly based on the software library *Makefile* provided by Martin Thomas in the example we talked about. This is the code:

```
# efsl library Makefile for STR71x by Giacomo Fazio and Antonio Nasca
# (based on efsl library makefile for AT91SAM7S by Martin Thomas)

MCU    = arm7tdmi
#THUMB = -mthumb -mthumb-interwork
THUMB =

LIBNAME = libSTR71x_lib.a

COPT= -mcpu=$(MCU) $(THUMB) -gdwarf-2 -Wall -Os
INCLUDEDIRS=-Iinclude
CFLAGS=$(COPT) $(INCLUDEDIRS)
# gcc4 unused code-removal:
CFLAGS += -ffunction-sections -fdata-sections

CC=arm-elf-gcc
AR=arm-elf-ar
OBJCOPY=arm-elf-objcopy
```

```
OBJ=src/71x_lib.o   src/adc12.o     src/apb.o      src/bspi.o
OBJ+=src/can.o      src/eic.o       src/emi.o      src/flash.o    src/gpio.o
OBJ+=src/i2c.o          src/pcu.o           src/rccu.o        src/rtc.o          src/tim.o
src/uart.o
OBJ+=src/wdg.o      src/xti.o


all: lib

libandclean: lib srcclean

lib: $(OBJ)
       $(AR) rcs $(LIBNAME) $(OBJ)

srcclean :
       rm -f $(OBJ)

clean :
       rm -f $(OBJ) $(LIBNAME)
```

As you can see, you have to specify in it the files to compile, that is all the files present in the software library, that after the compilation are linked in the file *libSTR71x_lib.a*. We activated the space optimization, through the option *-Os* present in the definition of the *COPT* macro, if this option cause any problems you can modify the optimization level or disable optimization by inserting the option *-O0* instead of *-Os*; moreover we disabled the *THUMB mode*, because with it the examples didn't always work properly, to activate it you have just to comment the macro *THUMB* (empty) and uncomment the macro *THUMB = -mthumb -mthumb-interwork*. For more details on optimizations and *THUMB mode* please refer to the B appendix.

For each program we also provided the file *.a*, but it could be necessary or preferable to recreate it on your system. In that case, enter in the folder *str71x_lib* and type the following commands:

```
make clean
make all
```

The first one performs a clean of the files obtained by the first compilation (included the *.a* file), while the second one commands the compilation, at the end of which we should have the *.a* file.

Since ST periodically releases new versions of the software library, Anglia releases new patches of it, so you should check in Internet for new versions: you can easily update the software library in your template by substituting the files with the new ones and by re-compiling it as you saw before.

Linker scripts

They are the files with extension *.ld*, represented in the picture in the red square. Their role is crucial, in that they indicate memory location of the components of the program code and data and that are emitted from the compiler and from the assembler. You find two files: one is used if you want to put and run the code in RAM (in that case both code and data are put there), while the other one is used if instead you want to put and run the code in FLASH (in that case the code is put in FLASH, but data are put in RAM).
You can distinguish the two files by the name (RAM or ROM termination) and they are provided from Spencer Oliver of Anglia, who did a really good job with them.
We want to remember that we use both internal and external memory for the STR71x microcontroller, while for the other microcontrollers we only use the internal memory.

We show now the file *STR71x-RAM.ld* code, about the loading of data in RAM:

```
/* Stack Sizes */

       _STACKSIZE = 1024;
       _STACKSIZE_IRQ = 256;
       _STACKSIZE_FIQ = 256;
       _STACKSIZE_SVC = 0;
       _STACKSIZE_ABT = 0;
       _STACKSIZE_UND = 0;
       _HEAPSIZE = 1024;

/* Memory Definitions */

MEMORY
{
       DATA (rw) : ORIGIN = 0x20000000, LENGTH = 0x00010000
       XDATA (rw) : ORIGIN = 0x62000000, LENGTH = 0x00400000
}

/* Section Definitions */

SECTIONS
{
       /* first section is .text which is used for code */

       .text :
       {
              KEEP(*(.vectrom))
              KEEP(*(.init))
              *(.text .text.*)
              *(.gnu.linkonce.t.*)
              *(.glue_7t .glue_7)
              KEEP(*(.fini))
              *(.gcc_except_table)
       } >XDATA =0
       . = ALIGN(4);

       /* .ctors .dtors are used for c++ constructors/destructors */

       .ctors :
       {
              PROVIDE(__ctors_start__ = .);
              KEEP(*(SORT(.ctors.*)))
              KEEP(*(.ctors))
              PROVIDE(__ctors_end__ = .);
       } >XDATA

       .dtors :
       {
              PROVIDE(__dtors_start__ = .);
              KEEP(*(SORT(.dtors.*)))
              KEEP(*(.dtors))
              PROVIDE(__dtors_end__ = .);
       } >XDATA

       /* .rodata section which is used for read-only data (constants) */

       .rodata :
       {
              *(.rodata .rodata.*)
              *(.gnu.linkonce.r.*)
       } >XDATA
       . = ALIGN(4);

       _vectext = .;
       PROVIDE (vtext = .);

       .vect : AT (_vectext)
       {
              _vecstart = .;
              KEEP(*(.vectram))
              _vecend = .;
       } >DATA

       _etext = _vectext + SIZEOF(.vect);
       PROVIDE (etext = .);

       /* .data section which is used for initialized data */

       .data : AT (_etext)
       {
              *(.data .data.*)
              *(.gnu.linkonce.d.*)
              SORT(CONSTRUCTORS)
       } >XDATA
       . = ALIGN(4);

       __data_start = .;
```

```
_edata = .;
PROVIDE (edata = .);

/* .bss section which is used for uninitialized data */

.bss :
{
        __bss_start = .;
        __bss_start__ = .;
        *(.bss .bss.*)
        *(.gnu.linkonce.b.*)
        *(COMMON)
        . = ALIGN(4);
} >XDATA
. = ALIGN(4);
__bss_end__ = .;

_end = .;
PROVIDE(end = .);

/* .heap section which is used for memory allocation */

.heap (NOLOAD) :
{
        __heap_start__ = .;
        *(.heap)
        . = MAX(__heap_start__ + _HEAPSIZE , .);
} >DATA
__heap_end__ = __heap_start__ + SIZEOF(.heap);

/* .stack section - user mode stack */

.stack (__heap_end__ + 3) / 4 * 4 (NOLOAD) :
{
        __stack_start__ = .;
        *(.stack)
        . = MAX(__stack_start__ + _STACKSIZE , .);
} >DATA
__stack_end__ = __stack_start__ + SIZEOF(.stack);

/* .stack_irq section */

.stack_irq (__stack_end__ + 3) / 4 * 4 (NOLOAD) :
{
        __stack_irq_start__ = .;
        *(.stack_irq)
        . = MAX(__stack_irq_start__ + _STACKSIZE_IRQ , .);
} >DATA
__stack_irq_end__ = __stack_irq_start__ + SIZEOF(.stack_irq);

/* .stack_fiq section */

.stack_fiq (__stack_irq_end__ + 3) / 4 * 4 (NOLOAD) :
{
        __stack_fiq_start__ = .;
        *(.stack_fiq)
    . = MAX(__stack_fiq_start__ + _STACKSIZE_FIQ , .);
} >DATA
__stack_fiq_end__ = __stack_fiq_start__ + SIZEOF(.stack_fiq);

/* .stack_svc section */

.stack_svc (__stack_fiq_end__ + 3) / 4 * 4 (NOLOAD) :
{
        __stack_svc_start__ = .;
        *(.stack_svc)
        . = MAX(__stack_svc_start__ + _STACKSIZE_SVC , .);
} >DATA
__stack_svc_end__ = __stack_svc_start__ + SIZEOF(.stack_svc);

/* .stack_abt section */

.stack_abt (__stack_svc_end__ + 3) / 4 * 4 (NOLOAD) :
{
        __stack_abt_start__ = .;
        *(.stack_abt)
        . = MAX(__stack_abt_start__ + _STACKSIZE_ABT , .);
} >DATA
__stack_abt_end__ = __stack_abt_start__ + SIZEOF(.stack_abt);

/* .stack_und section */

.stack_und (__stack_abt_end__ + 3) / 4 * 4 (NOLOAD) :
{
        __stack_und_start__ = .;
        *(.stack_und)
. = MAX(__stack_und_start__ + _STACKSIZE_UND , .);
} >DATA
__stack_und_end__ = __stack_und_start__ + SIZEOF(.stack_und);
```

```
        /* Stabs debugging sections.  */
        .stab          0 : { *(.stab) }
        .stabstr       0 : { *(.stabstr) }
        .stab.excl     0 : { *(.stab.excl) }
        .stab.exclstr  0 : { *(.stab.exclstr) }
        .stab.index    0 : { *(.stab.index) }
        .stab.indexstr 0 : { *(.stab.indexstr) }
        .comment       0 : { *(.comment) }
        /* DWARF debug sections.
                Symbols in the DWARF debugging sections are relative to the beginning
                of the section so we begin them at 0.  */
        /* DWARF 1 */
        .debug          0 : { *(.debug) }
        .line           0 : { *(.line) }
        /* GNU DWARF 1 extensions */
        .debug_srcinfo  0 : { *(.debug_srcinfo) }
        .debug_sfnames  0 : { *(.debug_sfnames) }
        /* DWARF 1.1 and DWARF 2 */
        .debug_aranges  0 : { *(.debug_aranges) }
        .debug_pubnames 0 : { *(.debug_pubnames) }
        /* DWARF 2 */
        .debug_info     0 : { *(.debug_info .gnu.linkonce.wi.*) }
        .debug_abbrev   0 : { *(.debug_abbrev) }
        .debug_line     0 : { *(.debug_line) }
        .debug_frame    0 : { *(.debug_frame) }
        .debug_str      0 : { *(.debug_str) }
        .debug_loc      0 : { *(.debug_loc) }
        .debug_macinfo  0 : { *(.debug_macinfo) }
        /* SGI/MIPS DWARF 2 extensions */
        .debug_weaknames 0 : { *(.debug_weaknames) }
        .debug_funcnames 0 : { *(.debug_funcnames) }
        .debug_typenames 0 : { *(.debug_typenames) }
        .debug_varnames  0 : { *(.debug_varnames) }
}
```

We can divide each of these files in three principal parts: configuration of the stack, memory and sections. The first one defines the dimension of each stack part; the second one defines the start address and the length of *DATA* (internal RAM) and *XDATA* (external RAM); at the end we have the definition of the sections, located either in *XDATA* (*.text*, *.ctors*, *.dtors, .rodata*, *.data*, *.bss*), or in *DATA* (*.vect*, *.heap*, *.stack*, *.stack_irq*, *.stack_fiq*, *.stack_svc*, *.stack_abt*, *.stack_und*).
Let's see now the code of the corresponding file for FLASH (*STR71x-ROM.ld*):

```
/* Stack Sizes */

        _STACKSIZE = 1024;
        _STACKSIZE_IRQ = 256;
        _STACKSIZE_FIQ = 256;
        _STACKSIZE_SVC = 0;
        _STACKSIZE_ABT = 0;
        _STACKSIZE_UND = 0;
        _HEAPSIZE = 1024;

/* Memory Definitions */

MEMORY
{
        XCODE (rx) : ORIGIN = 0x60000000, LENGTH = 0x00400000
        CODE (rx) : ORIGIN = 0x40000000, LENGTH = 0x00040000
        XDATA (rw) : ORIGIN = 0x62000000, LENGTH = 0x00400000
        DATA (rw) : ORIGIN = 0x20000000, LENGTH = 0x00010000
}

/* Section Definitions */

SECTIONS
{
        /* first section is .text which is used for code */

        .xtext :
        {
                *(.xtext .xtext.*)
                /**lfunc.o (.text*)*/
        } >XCODE =0
        . = ALIGN(4);

        _xtext = . + SIZEOF(.xtext);
        PROVIDE (xtext = .);

        .xdata : AT (_xtext)
        {
                *(.xdata .xdata.*)
                /**lfunc.o (.data*)*/
        } >XDATA
        . = ALIGN(4);
```

```
.xbss :
{
        *(.xbss .xbss.*)
        /**lfunc.o (.bss COMMON)*/
        . = ALIGN(4);
} >XDATA
. = ALIGN(4);

.text :
{
        KEEP(*(.vectrom))
        KEEP(*(.init))
        *(.text .text.*)
        *(.gnu.linkonce.t.*)
        *(.glue_7t .glue_7)
        KEEP(*(.fini))
        *(.gcc_except_table)
} >CODE =0
. = ALIGN(4);

/* .ctors .dtors are used for c++ constructors/destructors */

.ctors :
{
        PROVIDE(__ctors_start__ = .);
        KEEP(*(SORT(.ctors.*)))
        KEEP(*(.ctors))
        PROVIDE(__ctors_end__ = .);
} >CODE

.dtors :
{
        PROVIDE(__dtors_start__ = .);
        KEEP(*(SORT(.dtors.*)))
        KEEP(*(.dtors))
        PROVIDE(__dtors_end__ = .);
} >CODE

/* .rodata section which is used for read-only data (constants) */

.rodata :
{
        *(.rodata .rodata.*)
        *(.gnu.linkonce.r.*)
} >CODE
. = ALIGN(4);

_vectext = .;
PROVIDE (vtext = .);

.vect : AT (_vectext)
{
        _vecstart = .;
        KEEP(*(.vectram))
        _vecend = .;
} >DATA

_etext = _vectext + SIZEOF(.vect);
PROVIDE (etext = .);

/* .data section which is used for initialized data */

.data : AT (_etext)
{
        __data_start = .;
        *(.data .data.*)
        *(.gnu.linkonce.d.*)
        SORT(CONSTRUCTORS)
        . = ALIGN(4);
        *(.fastrun .fastrun.*)
} >XDATA
. = ALIGN(4);

_edata = .;
PROVIDE (edata = .);

/* .bss section which is used for uninitialized data */

.bss :
{
        __bss_start = .;
        __bss_start__ = .;
        *(.bss .bss.*)
        *(.gnu.linkonce.b.*)
        *(COMMON)
        . = ALIGN(4);
} >XDATA
. = ALIGN(4);
__bss_end__ = .;
```

```
_end = .;
PROVIDE(end = .);

/* .heap section which is used for memory allocation */

.heap (NOLOAD) :
{
        __heap_start__ = .;
        *(.heap)
        . = MAX(__heap_start__ + _HEAPSIZE , .);
} >DATA
__heap_end__ = __heap_start__ + SIZEOF(.heap);

/* .stack section - user mode stack */

.stack (__heap_end__ + 3) / 4 * 4 (NOLOAD) :
{
        __stack_start__ = .;
        *(.stack)
        . = MAX(__stack_start__ + _STACKSIZE , .);
} >DATA
__stack_end__ = __stack_start__ + SIZEOF(.stack);

/* .stack_irq section */

.stack_irq (__stack_end__ + 3) / 4 * 4 (NOLOAD) :
{
        __stack_irq_start__ = .;
        *(.stack_irq)
        . = MAX(__stack_irq_start__ + _STACKSIZE_IRQ , .);
} >DATA
__stack_irq_end__ = __stack_irq_start__ + SIZEOF(.stack_irq);

/* .stack_fiq section */

.stack_fiq (__stack_irq_end__ + 3) / 4 * 4 (NOLOAD) :
{
        __stack_fiq_start__ = .;
        *(.stack_fiq)
    . = MAX(__stack_fiq_start__ + _STACKSIZE_FIQ , .);
} >DATA
__stack_fiq_end__ = __stack_fiq_start__ + SIZEOF(.stack_fiq);

/* .stack_svc section */

.stack_svc (__stack_fiq_end__ + 3) / 4 * 4 (NOLOAD) :
{
        __stack_svc_start__ = .;
        *(.stack_svc)
        . = MAX(__stack_svc_start__ + _STACKSIZE_SVC , .);
} >DATA
__stack_svc_end__ = __stack_svc_start__ + SIZEOF(.stack_svc);

/* .stack_abt section */

.stack_abt (__stack_svc_end__ + 3) / 4 * 4 (NOLOAD) :
{
        __stack_abt_start__ = .;
        *(.stack_abt)
        . = MAX(__stack_abt_start__ + _STACKSIZE_ABT , .);
} >DATA
__stack_abt_end__ = __stack_abt_start__ + SIZEOF(.stack_abt);

/* .stack_und section */

.stack_und (__stack_abt_end__ + 3) / 4 * 4 (NOLOAD) :
{
        __stack_und_start__ = .;
        *(.stack_und)
. = MAX(__stack_und_start__ + _STACKSIZE_UND , .);
} >DATA
__stack_und_end__ = __stack_und_start__ + SIZEOF(.stack_und);

/* Stabs debugging sections.  */
.stab          0 : { *(.stab) }
.stabstr       0 : { *(.stabstr) }
.stab.excl     0 : { *(.stab.excl) }
.stab.exclstr  0 : { *(.stab.exclstr) }
.stab.index    0 : { *(.stab.index) }
.stab.indexstr 0 : { *(.stab.indexstr) }
.comment       0 : { *(.comment) }
/* DWARF debug sections.
        Symbols in the DWARF debugging sections are relative to the beginning
        of the section so we begin them at 0.  */
/* DWARF 1 */
.debug          0 : { *(.debug) }
.line           0 : { *(.line) }
/* GNU DWARF 1 extensions */
.debug_srcinfo  0 : { *(.debug_srcinfo) }
```

```
        .debug_sfnames  0 : { *(.debug_sfnames) }
        /* DWARF 1.1 and DWARF 2 */
        .debug_aranges  0 : { *(.debug_aranges) }
        .debug_pubnames 0 : { *(.debug_pubnames) }
        /* DWARF 2 */
        .debug_info     0 : { *(.debug_info .gnu.linkonce.wi.*) }
        .debug_abbrev   0 : { *(.debug_abbrev) }
        .debug_line     0 : { *(.debug_line) }
        .debug_frame    0 : { *(.debug_frame) }
        .debug_str      0 : { *(.debug_str) }
        .debug_loc      0 : { *(.debug_loc) }
        .debug_macinfo  0 : { *(.debug_macinfo) }
        /* SGI/MIPS DWARF 2 extensions */
        .debug_weaknames 0 : { *(.debug_weaknames) }
        .debug_funcnames 0 : { *(.debug_funcnames) }
        .debug_typenames 0 : { *(.debug_typenames) }
        .debug_varnames  0 : { *(.debug_varnames) }
}
```

The schema followed is the same: at first there is the section which configures the stack areas dimensions; then there is the memory dimension and this time you can notice the presence, beyond to *DATA* and *XDATA*, of *CODE* (internal FLASH) and *XCODE* (external FLASH), since data have to be put in RAM but the code has to be put in FLASH; at the end the sections configuration, some of which go in *CODE* (*.text*, *.ctors*, *.dtors*, *.rodata*), one goes in *XCODE* (*.xtext*), other ones in *DATA* (*.vect*, *.heap*, *.stack*, *.stack_irq*, *.stack_fiq*, *.stack_svc*, *.stack_abt*, *.stack_und*) and the remaining ones in *XDATA* (*.xdata*, *.xbss*, *.data*, *.bss*).

We did a little change from the Spencer Oliver version: *.data* and *.bss* sections are now located in XDATA and not in DATA, because if we leave them in DATA the code doesn't work properly.

The two files have *.S* extension and not *.s*, so pay attention especially because Windows, if you copy these files, can change the extension in *.s*, since for this operating system the file name is not case sensitive.

src and *include* subfolders

They are showed in the picture in a black square.
The *include* subfolder contains the *.h* files that are included by the *.c* files present in the subfolder *src*: you can always find the file *vectors.h*, strictly tied to the software library (we will talk about it in the paragraph 3.2.2).
The *src* subfolder contains the files with extension *.c* and *.s*: among the first ones you can find the file *main.c* (which contains the *main* and the code of your program), the file *vectors.c* (also tied to the software library and with which we will deal in the paragraph 3.2.2) and eventually other files that have to be linked together with *main.c*; the *.S* files (*startup.S* and *vector.S*) are written in Assembly and called “startup files”, they come from Spencer Oliver of Anglia, too and they are compiled through the assembler of the toolchain. Since they are really important, we show now their code, explaining the most important parts. Here you have *startup.S* code, that must be run after the reset:

```
/**** Startup Code (executed after Reset) ****/

/* Frequency values */
/* set to suit target hardware */

        .equ    FOSC,                   16000000
        .equ    FRTC,                   32768

/* Standard definitions of Mode bits and Interrupt (I & F) flags in PSRs */

        .equ    Mode_USR,       0x10
        .equ    Mode_FIQ,       0x11
        .equ    Mode_IRQ,       0x12
        .equ    Mode_SVC,       0x13
        .equ    Mode_ABT,       0x17
```

```
        .equ    Mode_UND,       0x1B
        .equ    Mode_SYS,       0x1F                            /* available on ARM Arch 4 and later */

        .equ    I_Bit,          0x80                            /* when I bit is set, IRQ is disabled */
        .equ    F_Bit,          0x40                            /* when F bit is set, FIQ is disabled */

/* --- System memory locations */

        .equ    EIC_Base_addr,          0xFFFFF800              /* EIC base address */
        .equ    ICR_off_addr,           0x00                    /* Interrupt Control register offset */
        .equ    CIPR_off_addr,          0x08                    /* Current Interrupt Priority Register offset */
        .equ    IVR_off_addr,           0x18                    /* Interrupt Vector Register offset */
        .equ    FIR_off_addr,           0x1C                    /* Fast Interrupt Register offset */
        .equ    IER_off_addr,           0x20                    /* Interrupt Enable Register offset */
        .equ    IPR_off_addr,           0x40                    /* Interrupt Pending Bit Register offset */
        .equ    SIR0_off_addr,          0x60                    /* Source Interrupt Register 0 */

        .equ    EMI_Base_addr,          0x6C000000              /* EMI base address */
        .equ    BCON0_off_addr,         0x00                    /* Bank 0 configuration register offset */
        .equ    BCON1_off_addr,         0x04                    /* Bank 1 configuration register offset */
        .equ    BCON2_off_addr,         0x08                    /* Bank 2 configuration register offset */
        .equ    BCON3_off_addr,         0x0C                    /* Bank 3 configuration register offset */

        .equ    GPIO2_Base_addr,        0xE0005000              /* GPIO2 base address */
        .equ    PC0_off_addr,           0x00                    /* Port Configuration Register 0 offset */
        .equ    PC1_off_addr,           0x04                    /* Port Configuration Register 1 offset */
        .equ    PC2_off_addr,           0x08                    /* Port Configuration Register 2 offset */
        .equ    PD_off_addr,            0x0C                    /* Port Data Register offset */

        .equ    CPM_Base_addr,          0xA0000040              /* CPM Base Address */
        .equ    BOOTCONF_off_addr,      0x10                    /* CPM - Boot Configuration Register */
        .equ    FLASH_mask,                     0x0000                  /* to remap FLASH at 0x0 */
        .equ    RAM_mask,                       0x0002                  /* to remap RAM at 0x0 */
        .equ    EXTMEM_mask,            0x0003                  /* to remap EXTMEM at 0x0 */

/*      define remapping, if using ram based vectors or rom based
        REMAP will need to be define in vector.s aswell
        Valid Options:
        REMAP 0 - default config, no remapping, vectors in rom
        REMAP 1 - vectors in rom
        REMAP 2 - vectors in ram
        REMAP 3 - vectors in external ram
*/

.ifndef REMAP
        .equ    REMAP,  0
.endif

/*      if we are debugging in ram then we need
        the vectors to point to the correct location
        only if BOOT0/BOOT1 have been set for Flash @ zero.
*/

.ifndef DBGRAM
        .equ    DBGRAM, 0
.endif

/* Startup Code must be linked first at Address at which it expects to run. */

        .text
        .arm
        .section .init, "ax"

        .global _start
        .global RCCU_Main_Osc
        .global RCCU_RTC_Osc

.if REMAP

/* Exception Vector (before Remap) */

/* Reset Handler */
/* On reset, an aliased copy of ROM is at 0x0. */
/* Continue execution from 'real' ROM rather than aliased copy */

                LDR     pc, =HardReset
HardReset:

/*****************************************************************************
REMAPPING
Description  : Remapping  memory whether RAM,FLASH or External memory
               at Address 0x0 after the application has started executing.
               Remapping is generally done to allow RAM to replace FLASH
               or EXTMEM at 0x0.
               the remapping of RAM allow copying of vector table into RAM
*****************************************************************************/

.if REMAP == 1
                MOV     r4, #FLASH_mask
.endif
```

```
.if REMAP == 2
                MOV     r4, #RAM_mask
.endif

.if REMAP == 3
                /* copying the program into SRAM */
                LDR     r0, =GPIO2_Base_addr                    /* Configure P2.0 -> 3 in AF_PP mode */
                LDR     r1, =0x0000000F
                STR     r1, [r0, #PC0_off_addr]
                STR     r1, [r0, #PC1_off_addr]
                STR     r1, [r0, #PC2_off_addr]

                LDR     r0, =EMI_Base_addr                              /* Enable the EMI */
                LDR     r1, =0x00008001
                STR     r1, [r0, #BCON0_off_addr]               /*  Enable bank 0 16-bit 0 wait state */

                MOV     r0, #EXTMEM_mask
                LDR     r1, =CPM_Base_addr
                LDRH    r2, [r1, #BOOTCONF_off_addr]    /* Read BOOTCONF Register */
                LDR     r4, =0x3
                AND     r2, r2, r4
                CMP     r2, r0
                BEQ     _start
                MOV     r4, #EXTMEM_mask
.endif

remap:
                LDR     r1, =_vecstart                                  /* r1 = start address from which to copy */
                LDR     r3, =_vecend
                 SUB            r3, r3, r1                                              /* r3 = number of bytes to
copy */
                LDR     r0, =_vectext                                   /* r0 = start address where to copy */
copy_ram:
                LDR     r2, [r0], #4                                    /* Read a word from the source */
                STR     r2, [r1], #4                                    /* copy the word to destination */
                 SUBS    r3, r3, #4                                             /* Decrement number of words to
copy */
                BNE     copy_ram

                LDR     r1, =CPM_Base_addr
                LDRH    r2, [r1, #BOOTCONF_off_addr]    /* Read BOOTCONF Register */
                  BIC     r2, r2, #0x03                                         /* Reset the two LSB bits of BOOTCONF
Register */
                  ORR     r2, r2, r4                                                    /* change the two LSB bits of
BOOTCONF Register */
                STRH    r2, [r1, #BOOTCONF_off_addr]    /* Write BOOTCONF Register */
.endif

/* After remap this will be our reset handler */

_start:
                LDR     pc, =NextInst
NextInst:

                NOP             /* Wait for OSC stabilization */
                NOP
                NOP
                NOP
                NOP
                NOP
                NOP
                NOP
                NOP

.if DBGRAM
                MOV     r0, #RAM_mask
                LDR     r1, =CPM_Base_addr
                LDRH    r2, [r1, #BOOTCONF_off_addr]    /* Read BOOTCONF Register */
                  BIC     r2, r2, #0x03                                         /* Reset the two LSB bits of BOOTCONF
Register */
                  ORR     r2, r2, r0                                                    /* change the two LSB bits of
BOOTCONF Register */
                STRH    r2, [r1, #BOOTCONF_off_addr]    /* Write BOOTCONF Register */
.endif

/* Setup Stack for each mode */

/* Enter Abort Mode and set its Stack Pointer */
                MSR     cpsr_c, #Mode_ABT|I_Bit|F_Bit
                LDR     sp, =__stack_abt_end__

/* Enter Undefined Instruction Mode and set its Stack Pointer */
                MSR     cpsr_c, #Mode_UND|I_Bit|F_Bit
                LDR     sp, =__stack_und_end__

/* Enter Supervisor Mode and set its Stack Pointer */
                MSR     cpsr_c, #Mode_SVC|I_Bit|F_Bit
                LDR     sp, =__stack_svc_end__
```

```
/* Enter FIQ Mode and set its Stack Pointer */
                MSR     cpsr_c, #Mode_FIQ|I_Bit|F_Bit
                LDR     sp, =__stack_fiq_end__

/* Enter IRQ Mode and set its Stack Pointer */
                MSR     cpsr_c, #Mode_IRQ|I_Bit|F_Bit
                LDR     sp, =__stack_irq_end__

/* Enter User Mode and set its Stack Pointer */
                MSR     cpsr_c, #Mode_USR
                LDR     sp, =__stack_end__

/* Setup a default Stack Limit (when compiled with "-mapcs-stack-check") */
                LDR     sl, =__bss_end__

/*****************************************************************************
EIC initialization
Description  : Initialize the EIC as following :
               - IRQ disabled
               - FIQ disabled
               - IVR contain the load PC opcode (0xF59FF00)
               - Current priority level equal to 0
               - All channels are disabled
               - All channels priority equal to 0
               - All SIR registers contain offset to the related IRQ table entry
*****************************************************************************/

EIC_INIT:
                LDR     r3, =EIC_Base_addr
                LDR     r4, =0x00000000
                STR     r4, [r3, #ICR_off_addr]         /* Disable FIQ and IRQ */
                STR     r4, [r3, #IER_off_addr]         /* Disable all channels interrupts */
                LDR     r4, =0xFFFFFFFF
                STR     r4, [r3, #IPR_off_addr]         /* Clear all IRQ pending bits */
                LDR     r4, =0x0C
                STR     r4, [r3, #FIR_off_addr]         /* Disable FIQ channels and clear FIQ pending bits */
                LDR     r4, =0x00000000
                STR     r4, [r3, #CIPR_off_addr]        /* Reset the current priority register */
                LDR     r4, =0xE59F0000
                    STR      r4, [r3, #IVR_off_addr]           /* Write the LDR pc,pc,#offset instruction code in
IVR[31:16] */
                LDR     r2, =32                                         /* 32 Channel to initialize */
                LDR     r0, =T0TIMI_Addr                        /* Read the address of the IRQs address table */
                LDR     r1, =0x00000FFF
                AND     r0, r0, r1
                LDR     r5, =SIR0_off_addr                      /* Read SIR0 address */
                SUB     r4, r0, #8                                      /* subtract 8 for prefetch */
                  LDR      r1, =0xF7E8                                          /* add the offset to the 0x00000000
address(IVR address + 7E8 = 0x00000000) */
                                                                                /* 0xF7E8 used to complete
the LDR pc,pc,#offset opcode */
                ADD     r1, r4, r1                                      /* compute the jump offset */
EIC_INI:
                MOV     r4, r1, LSL #16                         /* Left shift the result */
                STR     r4, [r3, r5]                            /* Store the result in SIRx register */
                ADD     r1, r1, #4                                      /* Next IRQ address */
                ADD     r5, r5, #4                                      /* Next SIR */
                 SUBS    r2, r2, #1                                      /* Decrement the number of SIR registers to
initialize */
                BNE     EIC_INI                                         /* If more then continue */

/* Change to System mode */
                MSR     cpsr_c, #Mode_SYS

/* Relocate .data section (Copy from ROM to RAM) */
                LDR     r1, =_etext
                LDR     r2, =__data_start
                LDR     r3, =_edata
LoopRel:
                CMP     r2, r3
                LDRLO   r0, [r1], #4
                STRLO   r0, [r2], #4
                BLO     LoopRel

/* Clear .bss section (Zero init) */
                MOV     r0, #0
                LDR     r1, =__bss_start__
                LDR     r2, =__bss_end__
LoopZI:
                CMP     r1, r2
                STRLO   r0, [r1], #4
                BLO     LoopZI

/* Call C++ constructors */
                LDR     r0, =__ctors_start__
                LDR     r1, =__ctors_end__
ctor_loop:
                CMP     r0, r1
                BEQ     ctor_end
                LDR     r2, [r0], #4
```

```
                STMFD   sp!, {r0-r1}
                MOV     lr, pc
                MOV     pc, r2
                LDMFD   sp!, {r0-r1}
                B               ctor_loop
ctor_end:

/* Need to set up standard file handles */
/* Only used under simulator, normally overide syscall.c */
#               BL              initialise_monitor_handles

/* if we use debug version of str7lib this will call the init function */

                BL              libdebug
libdebug:

/* Enter the C code, use B instruction so as to never return */
/* use BL main if you want to use c++ destructors below */
                B               main

/* Call destructors */
#               LDR             r0, =__dtors_start__
#               LDR             r1, =__dtors_end__
dtor_loop:
#               CMP             r0, r1
#               BEQ             dtor_end
#               LDR             r2, [r0], #4
#               STMFD   sp!, {r0-r1}
#               MOV             lr, pc
#               MOV             pc, r2
#               LDMFD   sp!, {r0-r1}
#               B               dtor_loop
dtor_end:

/* Return from main, loop forever. */
exit_loop:
#               B               exit_loop

/* Fosc values, used by libstr7 */

RCCU_Main_Osc: .long   FOSC
RCCU_RTC_Osc:  .long   FRTC

        .weak libdebug

        .end
```

Now let's see *vector.S*, which contains the interrupts vector table, that must be inserted in memory at the address *.text*:

```
        .equ    EIC_base_addr,  0xFFFFF800      /* EIC base address. */
        .equ    CICR_off_addr,  0x04            /* Current Interrupt Channel Register. */
        .equ    IVR_off_addr,   0x18            /* Interrupt Vector Register. */
        .equ    IPR_off_addr,   0x40            /* Interrupt Pending Register. */

        /*      set HIRAM 1 for Interrupt Handlers reside in ram Vectors in ram uses 192 bytes
                Vectors/Handlers uses 1452 bytes, normally 0 */

.ifndef HIRAM
        .equ    HIRAM,  0
.endif

.ifndef REMAP
        .equ    REMAP,  0
.endif

        .text
        .arm

.if REMAP > 1
        .section .vectram, "ax"
.else
        .section .vectrom, "ax"
.endif

        .global T0TIMI_Addr
        .global Reset_Vec

/* Note: LDR PC instructions are used here, though branch (B) instructions */
/* could also be used, unless the ROM is at an address >32MB. */

/*******************************************************************************
                        Exception vectors
*******************************************************************************/

Reset_Vec:      LDR     pc, Reset_Addr          /* Reset Handler */
Undef_Vec:      LDR     pc, Undefined_Addr
```

```
SWI_Vec:       LDR     pc, SWI_Addr
PAbt_Vec:      LDR     pc, Prefetch_Addr
DAbt_Vec:      LDR     pc, Abort_Addr
               NOP                                                    /* Reserved vector */
IRQ_Vec:       LDR     pc, IRQ_Addr
FIQ_Vec:       LDR     pc, FIQ_Addr

/*************************************************************************
            Exception handlers address table
*************************************************************************/

Reset_Addr:            .word   _start
Undefined_Addr:        .word   UndefinedHandler
SWI_Addr:              .word   SWIHandler
Prefetch_Addr: .word   PrefetchHandler
Abort_Addr:            .word   AbortHandler
                       .word   0                                  /* reserved */
IRQ_Addr:              .word   IRQHandler
FIQ_Addr:              .word   FIQHandler

/*************************************************************************
          Peripherals IRQ handlers address table
*************************************************************************/

T0TIMI_Addr:   .word   T0TIMIIRQHandler
FLASH_Addr:            .word   FLASHIRQHandler
RCCU_Addr:             .word   RCCUIRQHandler
RTC_Addr:              .word   RTCIRQHandler
WDG_Addr:              .word   WDGIRQHandler
XTI_Addr:              .word   XTIIRQHandler
USBHP_Addr:            .word   USBHPIRQHandler
I2C0ITERR_Addr:        .word   I2C0ITERRIRQHandler
I2C1ITERR_ADDR:        .word   I2C1ITERRIRQHandler
UART0_Addr:            .word   UART0IRQHandler
UART1_Addr:            .word   UART1IRQHandler
UART2_ADDR:            .word   UART2IRQHandler
UART3_ADDR:            .word   UART3IRQHandler
BSPI0_ADDR:            .word   BSPI0IRQHandler
BSPI1_Addr:            .word   BSPI1IRQHandler
I2C0_Addr:             .word   I2C0IRQHandler
I2C1_Addr:             .word   I2C1IRQHandler
CAN_Addr:              .word   CANIRQHandler
ADC12_Addr:            .word   ADC12IRQHandler
T1TIMI_Addr:   .word   T1TIMIIRQHandler
T2TIMI_Addr:   .word   T2TIMIIRQHandler
T3TIMI_Addr:   .word   T3TIMIIRQHandler
                       .word   0                                  /* reserved */
                       .word   0                                  /* reserved */
                       .word   0                                  /* reserved */
HDLC_Addr:             .word   HDLCIRQHandler
USBLP_Addr:            .word   USBLPIRQHandler
                       .word   0                                  /* reserved */
                       .word   0                                  /* reserved */
T0TOI_Addr:            .word   T0TOIIRQHandler
T0OC1_Addr:            .word   T0OC1IRQHandler
T0OC2_Addr:            .word   T0OC2IRQHandler

/*************************************************************************
                        Exception Handlers
*************************************************************************/

.ifeq HIRAM
       .section .text, "ax"
.endif

/*************************************************************************
* macro for long jump from ram-rom if handlers are
* located in ram and are thumb mode
*************************************************************************/

.macro mBLX brAddr
       .ifeq HIRAM
              BL               \brAddr
       .else
              LDR              r0,=\brAddr
              MOV              lr, pc
              BX               r0
       .endif
.endm

/*************************************************************************
* Macro Name     : SaveContext
* Description    : This macro used to save the context before entering
                   an exception handler.
* Input          : The range of registers to store.
* Output         : none
*************************************************************************/

.macro SaveContext reg1 reg2
              STMFD    sp!,{\reg1-\reg2,lr}     /* Save The workspace plus the current return */
```

```
                                                                        /* address lr_ mode into the stack
*/
                MRS             r1, spsr                                /* Save the spsr_mode into r1 */
                STMFD   sp!, {r1}                               /* Save spsr */
.endm

/*****************************************************************************
* Macro Name     : RestoreContext
* Description    : This macro used to restore the context to return from
                   an exception handler and continue the program execution.
* Input          : The range of registers to restore.
* Output         : none
*****************************************************************************/

.macro RestoreContext reg1 reg2
                LDMFD   sp!, {r1}                               /* Restore the saved spsr_mode into r1 */
                MSR             spsr_cxsf, r1                   /* Restore spsr_mode */
                LDMFD   sp!, {\reg1-\reg2,pc}^  /* Return to the instruction following */
                                                                        /* the exception interrupt */
.endm

/*****************************************************************************
* Function Name  : IRQHandler
* Description    : This function called when IRQ exception is entered.
* Input          : none
* Output         : none
*****************************************************************************/

IRQHandler:
                SUB             lr, lr, #4                              /* Update the link register */
                SaveContext r0, r12
                LDR             lr, =ReturnAddress              /* Read the return address. */
                LDR             r0, =EIC_base_addr
                LDR             r1, =IVR_off_addr
                ADD             pc, r0, r1                              /* Branch to the IRQ handler. */
ReturnAddress:
                                                                        /* Clear pending bit in EIC (using
the proper IPRx) */
                LDR             r0, =EIC_base_addr
                LDR             r2, [r0, #CICR_off_addr]/* Get the IRQ channel number. */
                MOV             r3, #1
                MOV             r3, r3, LSL r2
                STR             r3, [r0, #IPR_off_addr] /* Clear the corresponding IPR bit. */
                RestoreContext r0, r12

/*****************************************************************************
* Function Name  : SWIHandler
* Description    : This function called when SWI instruction executed.
* Input          : none
* Output         : none
*****************************************************************************/

SWIHandler:
                SaveContext r0, r12                     /* r0 holds swi number */
                MOV     r1, sp                          /* load regs */
                mBLX    SWI_Handler
                RestoreContext r0, r12

/*****************************************************************************
* Function Name  : UndefinedHandler
* Description    : This function called when undefined instruction
                   exception is entered.
* Input          : none
* Output         : none
*****************************************************************************/

UndefinedHandler:
                SaveContext r0, r12
                mBLX    Undefined_Handler
                RestoreContext r0, r12

/*****************************************************************************
* Function Name  : PrefetchAbortHandler
* Description    : This function called when Prefetch Abort
                   exception is entered.
* Input          : none
* Output         : none
*****************************************************************************/

PrefetchHandler:
                SUB             lr, lr, #4                      /* Update the link register. */
                SaveContext r0, r12
                mBLX    Prefetch_Handler
                RestoreContext r0, r12

/*****************************************************************************
* Function Name  : DataAbortHandler
* Description    : This function is called when Data Abort
                   exception is entered.
* Input          : none
```

```
* Output         : none
*******************************************************************************/

AbortHandler:
               SUB             lr, lr, #8                      /* Update the link register. */
               SaveContext r0, r12
               mBLX    Abort_Handler
               RestoreContext r0, r12

/*******************************************************************************
* Function Name  : FIQHandler
* Description    : This function is called when FIQ
                   exception is entered.
* Input          : none
* Output         : none
*******************************************************************************/

FIQHandler:
               SUB             lr, lr, #4                      /* Update the link register. */
               SaveContext r0, r7
               mBLX    FIQ_Handler
               RestoreContext r0, r7

/*******************************************************************************
* Macro Name     : IRQ_to_SYS
* Description    : This macro used to switch form IRQ mode to SYS mode
* Input          : none.
* Output         : none
*******************************************************************************/

       .macro  IRQ_to_SYS
               MSR             cpsr_c, #0x1F   /* Switch to SYS mode */
               STMFD   sp!, {lr}               /* Save the link register. */
       .endm

/*******************************************************************************
* Macro Name     : SYS_to_IRQ
* Description    : This macro used to switch from SYS mode to IRQ mode
                   then to return to IRQHnadler routine.
* Input          : none.
* Output         : none.
*******************************************************************************/

       .macro  SYS_to_IRQ
               LDMFD   sp!, {lr}               /* Restore the link register. */
               MSR             cpsr_c, #0xD2   /* Switch to IRQ mode. */
               MOV             pc, lr                  /* Return to IRQHandler routine to clear the */
                                                       /* pending bit. */
       .endm

/*******************************************************************************
* Function Name  : T0TIMIIRQHandler
* Description    : This function used to switch to SYS mode before entering
                   the T0TIMI_IRQHandler function
                   Then to return to IRQ mode after the
                   T0TIMI_IRQHandler function termination.
* Input          : none.
* Output         : none.
*******************************************************************************/

T0TIMIIRQHandler:
               IRQ_to_SYS
               mBLX    T0TIMI_IRQHandler
               SYS_to_IRQ

/*******************************************************************************
* Function Name  : FLASHIRQHandler
* Description    : This function used to switch to SYS mode before entering
                   the FLASH_IRQHandler function
                   Then to return to IRQ mode after the
                   FLASH_IRQHandler function termination.
* Input          : none
* Output         : none
*******************************************************************************/

FLASHIRQHandler:
               IRQ_to_SYS
               mBLX    FLASH_IRQHandler
               SYS_to_IRQ

/*******************************************************************************
* Function Name  : RCCUIRQHandler
* Description    : This function used to switch to SYS mode before entering
                   the RCCU_IRQHandler function
                   Then to return to IRQ mode after the
                   RCCU_IRQHandler function termination.
* Input          : none
* Output         : none
*******************************************************************************/
```

```
RCCUIRQHandler:
               IRQ_to_SYS
               mBLX    RCCU_IRQHandler
               SYS_to_IRQ

/*******************************************************************************
* Function Name  : RTCIRQHandler
* Description    : This function used to switch to SYS mode before entering
                   the RTC_IRQHandler function
                   Then to return to IRQ mode after the
                   RTC_IRQHandler function termination.
* Input          : none
* Output         : none
*******************************************************************************/

RTCIRQHandler:
               IRQ_to_SYS
               mBLX    RTC_IRQHandler
               SYS_to_IRQ

/*******************************************************************************
* Function Name  : WDGIRQHandler
* Description    : This function used to switch to SYS mode before entering
                   the WDG_IRQHandler function
                   Then to return to IRQ mode after the
                   WDG_IRQHandler function termination.
* Input          : none
* Output         : none
*******************************************************************************/

WDGIRQHandler:
               IRQ_to_SYS
               mBLX    WDG_IRQHandler
               SYS_to_IRQ

/*******************************************************************************
* Function Name  : XTIIRQHandler
* Description    : This function used to switch to SYS mode before entering
                   the XTI_IRQHandler function
                   Then to return to IRQ mode after the
                   XTI_IRQHandler function termination.
* Input          : none
* Output         : none
*******************************************************************************/

XTIIRQHandler:
               IRQ_to_SYS
               mBLX    XTI_IRQHandler
               SYS_to_IRQ

/*******************************************************************************
* Function Name  : USBHPIRQHandler
* Description    : This function used to switch to SYS mode before entering
                   the USBHP_IRQHandler function
                   Then to return to IRQ mode after the
                   USBHP_IRQHandler function termination.
* Input          : none
* Output         : none
*******************************************************************************/

USBHPIRQHandler:
               IRQ_to_SYS
               mBLX    USBHP_IRQHandler
               SYS_to_IRQ

/*******************************************************************************
* Function Name  : I2C0ITERRIRQHandler
* Description    : This function used to switch to SYS mode before entering
                   the I2C0ITERR_IRQHandler function
                   Then to return to IRQ mode after the
                   I2C0ITERR_IRQHandler function termination.
* Input          : none
* Output         : none
*******************************************************************************/

I2C0ITERRIRQHandler:
               IRQ_to_SYS
               mBLX    I2C0ITERR_IRQHandler
               SYS_to_IRQ

/*******************************************************************************
* Function Name  : I2C1ITERRIRQHandler
* Description    : This function used to switch to SYS mode before entering
                   the I2C1ITERR_IRQHandler function
                   Then to return to IRQ mode after the
                   I2C1ITERR_IRQHandler function termination.
* Input          : none
* Output         : none
*******************************************************************************/
```

```
I2C1ITERRIRQHandler:
                IRQ_to_SYS
                mBLX    I2C1ITERR_IRQHandler
                SYS_to_IRQ

/*******************************************************************************
* Function Name  : UART0IRQHandler
* Description    : This function used to switch to SYS mode before entering
                   the UART0_IRQHandler function
                   Then to return to IRQ mode after the
                   UART0_IRQHandler function termination.
* Input          : none
* Output         : none
*******************************************************************************/

UART0IRQHandler:
                IRQ_to_SYS
                mBLX    UART0_IRQHandler
                SYS_to_IRQ

/*******************************************************************************
* Function Name  : UART1IRQHandler
* Description    : This function used to switch to SYS mode before entering
                   the UART1_IRQHandler function
                   Then to return to IRQ mode after the
                   UART1_IRQHandler function termination.
* Input          : none
* Output         : none
*******************************************************************************/

UART1IRQHandler:
                IRQ_to_SYS
                mBLX    UART1_IRQHandler
                SYS_to_IRQ

/*******************************************************************************
* Function Name  : UART2IRQHandler
* Description    : This function used to switch to SYS mode before entering
                   the UART2_IRQHandler function
                   Then to return to IRQ mode after the
                   UART2_IRQHandler function termination.
* Input          : none
* Output         : none
*******************************************************************************/

UART2IRQHandler:
                IRQ_to_SYS
                mBLX    UART2_IRQHandler
                SYS_to_IRQ

/*******************************************************************************
* Function Name  : UART3IRQHandler
* Description    : This function used to switch to SYS mode before entering
                   the UART3_IRQHandler function
                   Then to return to IRQ mode after the
                   UART3_IRQHandler function termination.
* Input          : none
* Output         : none
*******************************************************************************/

UART3IRQHandler:
                IRQ_to_SYS
                mBLX    UART3_IRQHandler
                SYS_to_IRQ

/*******************************************************************************
* Function Name  : BSPI0IRQHandler
* Description    : This function used to switch to SYS mode before entering
                   the BSPI0_IRQHandler function
                   Then to return to IRQ mode after the
                   BSPI0_IRQHandler function termination.
* Input          : none
* Output         : none
*******************************************************************************/

BSPI0IRQHandler:
                IRQ_to_SYS
                mBLX    BSPI0_IRQHandler
                SYS_to_IRQ

/*******************************************************************************
* Function Name  : BSPI1IRQHandler
* Description    : This function used to switch to SYS mode before entering
                   the BSPI1_IRQHandler function
                   Then to return to IRQ mode after the
                   BSPI1_IRQHandler function termination.
* Input          : none
* Output         : none
*******************************************************************************/
```

```
BSPI1IRQHandler:
               IRQ_to_SYS
               mBLX    BSPI1_IRQHandler
               SYS_to_IRQ

/*******************************************************************************
* Function Name  : I2C0IRQHandler
* Description    : This function used to switch to SYS mode before entering
                   the I2C0_IRQHandler function
                   Then to return to IRQ mode after the
                   I2C0_IRQHandler function termination.
* Input          : none
* Output         : none
*******************************************************************************/

I2C0IRQHandler:
               IRQ_to_SYS
               mBLX    I2C0_IRQHandler
               SYS_to_IRQ

/*******************************************************************************
* Function Name  : I2C1IRQHandler
* Description    : This function used to switch to SYS mode before entering
                   the I2C1_IRQHandler function
                   Then to return to IRQ mode after the
                   I2C1_IRQHandler function termination.
* Input          : none
* Output         : none
*******************************************************************************/

I2C1IRQHandler:
               IRQ_to_SYS
               mBLX    I2C1_IRQHandler
               SYS_to_IRQ

/*******************************************************************************
* Function Name  : CANIRQHandler
* Description    : This function used to switch to SYS mode before entering
                   the CAN_IRQHandler function
                   Then to return to IRQ mode after the
                   CAN_IRQHandler function termination.
* Input          : none
* Output         : none
*******************************************************************************/

CANIRQHandler:
               IRQ_to_SYS
               mBLX    CAN_IRQHandler
               SYS_to_IRQ

/*******************************************************************************
* Function Name  : ADC12IRQHandler
* Description    : This function used to switch to SYS mode before entering
                   the ADC12_IRQHandler function
                   Then to return to IRQ mode after the
                   ADC12_IRQHandler function termination.
* Input          : none
* Output         : none
*******************************************************************************/

ADC12IRQHandler:
               IRQ_to_SYS
               mBLX    ADC12_IRQHandler
               SYS_to_IRQ

/*******************************************************************************
* Function Name  : T1TIMIIRQHandler
* Description    : This function used to switch to SYS mode before entering
                   the T1TIMI_IRQHandler function
                   Then to return to IRQ mode after the
                   T1TIMI_IRQHandler function termination.
* Input          : none
* Output         : none
*******************************************************************************/

T1TIMIIRQHandler:
               IRQ_to_SYS
               mBLX    T1TIMI_IRQHandler
               SYS_to_IRQ

/*******************************************************************************
* Function Name  : T2TIMIIRQHandler
* Description    : This function used to switch to SYS mode before entering
                   the T2TIMI_IRQHandler function
                   Then to return to IRQ mode after the
                   T2TIMI_IRQHandler function termination.
* Input          : none
* Output         : none
*******************************************************************************/
```

```
T2TIMIIRQHandler:
                IRQ_to_SYS
                mBLX    T2TIMI_IRQHandler
                SYS_to_IRQ

/*******************************************************************************
* Function Name  : T3TIMIIRQHandler
* Description    : This function used to switch to SYS mode before entering
                   the T3TIMI_IRQHandler function
                   Then to return to IRQ mode after the
                   T3TIMI_IRQHandler function termination.
* Input          : none
* Output         : none
*******************************************************************************/

T3TIMIIRQHandler:
                IRQ_to_SYS
                mBLX    T3TIMI_IRQHandler
                SYS_to_IRQ

/*******************************************************************************
* Function Name  : HDLCIRQHandler
* Description    : This function used to switch to SYS mode before entering
                   the HDLC_IRQHandler function
                   Then to return to IRQ mode after the
                   HDLC_IRQHandler function termination.
* Input          : none
* Output         : none
*******************************************************************************/

HDLCIRQHandler:
                IRQ_to_SYS
                mBLX    HDLC_IRQHandler
                SYS_to_IRQ

/*******************************************************************************
* Function Name  : USBLPIRQHandler
* Description    : This function used to switch to SYS mode before entering
                   the USBLP_IRQHandler function
                   Then to return to IRQ mode after the
                   USBLP_IRQHandler function termination.
* Input          : none
* Output         : none
*******************************************************************************/

USBLPIRQHandler:
                IRQ_to_SYS
                mBLX    USBLP_IRQHandler
                SYS_to_IRQ

/*******************************************************************************
* Function Name  : T0TOIIRQHandler
* Description    : This function used to switch to SYS mode before entering
                   the T0TOI_IRQHandler function
                   Then to return to IRQ mode after the
                   T0TOI_IRQHandler function termination.
* Input          : none
* Output         : none
*******************************************************************************/

T0TOIIRQHandler:
                IRQ_to_SYS
                mBLX    T0TOI_IRQHandler
                SYS_to_IRQ

/*******************************************************************************
* Function Name  : T0OC1IRQHandler
* Description    : This function used to switch to SYS mode before entering
                   the T0OC1_IRQHandler function
                   Then to return to IRQ mode after the
                   T0OC1_IRQHandler function termination.
* Input          : none
* Output         : none
*******************************************************************************/

T0OC1IRQHandler:
                IRQ_to_SYS
                mBLX    T0OC1_IRQHandler
                SYS_to_IRQ

/*******************************************************************************
* Function Name  : T0OC2IRQHandler
* Description    : This function used to switch to SYS mode before entering
                   the T0OC2_IRQHandler function
                   Then to return to IRQ mode after the
                   T0OC2_IRQHandler function termination.
* Input          : none
* Output         : none
*******************************************************************************/
```

```
T0OC2IRQHandler:
                IRQ_to_SYS
                mBLX    T0OC2_IRQHandler
                SYS_to_IRQ

        .end
```

Makefile

It's represented in the picture with a blue square and it's the ring of conjunction among all the parts seen till now, because it contains all the rules to obtain at the end a unique file. It's strongly based on the *Makefile* of the known example *STR71x Demo 1* of Martin Thomas, with some modifications to adapt it to our solution and to make it work properly on Linux. Here you have the code:

```
# Hey Emacs, this is a -*- makefile -*-
#
# WinARM template makefile
# by Giacomo Fazio and Antonio Nasca, Catania, Italy
# <giacomofazio@gmail.com>
# <antodani.nasca@hotmail.it>
#
# based on the WinARM template makefile written by Martin Thomas
# Released to the Public Domain
# Please read the make user manual!
#
#
# On command line:
#
# make all = Make software.
#
# make clean = Clean out built project files.
#
# make program = Download the hex file to the device
#
# (TODO: make filename.s = Just compile filename.c into the assembler code only)
#
# To rebuild project do "make clean" then "make all".
# Toolchain prefix (i.e arm-elf -> arm-elf-gcc.exe)
TCHAIN = arm-elf
#TCHAIN = arm-none-eabi

#USE_THUMB_MODE = YES
USE_THUMB_MODE = NO

# MCU name and submodel
MCU      = arm7tdmi
SUBMDL   = STR71x

## Create ROM-Image
#RUN_MODE=ROM_RUN
## Create RAM-Image
RUN_MODE=RAM_RUN

## not supported in this example:
## Exception-Vector placement only supported for "ROM_RUN"
## (placement settings ignored when using "RAM_RUN")
## - Exception vectors in ROM:
#VECTOR_LOCATION=VECTORS_IN_ROM
## - Exception vectors in RAM:
#VECTOR_LOCATION=VECTORS_IN_RAM


# Target file name (without extension).
TARGET = main

# List C source files here. (C dependencies are automatically generated.)
# use file-extension c for "c-only"-files
SRC  = src/$(TARGET).c

# List C source files here which must be compiled in ARM-Mode.
# use file-extension c for "c-only"-files
SRCARM  = src/vectors.c
# thumb is possible too for vectors.c - keep ARM, TODO: profile

# List C++ source files here.
# use file-extension cpp for C++-files (use extension .cpp)
CPPSRC =

# List C++ source files here which must be compiled in ARM-Mode.
# use file-extension cpp for C++-files (use extension .cpp)
#CPPSRCARM = $(TARGET).cpp
CPPSRCARM =

# List Assembler source files here.
# Make them always end in a capital .S.  Files ending in a lowercase .s
```

```
# will not be considered source files but generated files (assembler
# output from the compiler), and will be deleted upon "make clean"!
# Even though the DOS/Win* filesystem matches both .s and .S the same,
# it will preserve the spelling of the filenames, and gcc itself does
# care about how the name is spelled on its command-line.
ASRC =

# List Assembler source files here which must be assembled in ARM-Mode..
ASRCARM  = src/vector.S src/startup.S

# Path to Linker-Scripts
LINKERSCRIPTPATH = .

## Output format. (can be ihex or binary or both)
## (binary i.e. for openocd and SAM-BA, hex i.e. for lpc21isp and uVision)
#FORMAT = ihex
#FORMAT = binary
FORMAT = both

# Optimization level, can be [0, 1, 2, 3, s].
# 0 = turn off optimization. s = optimize for size.
# (Note: 3 is not always the best optimization level. See avr-libc FAQ.)
#OPT = s
OPT = 0

## Using the Atmel AT91_lib produces warning with
## the default warning-levels.
## yes - disable these warnings; no - keep default settings
#AT91LIBNOWARN = yes
AT91LIBNOWARN = no

# Debugging format.
# Native formats for AVR-GCC's -g are stabs [default], or dwarf-2.
# AVR (extended) COFF requires stabs, plus an avr-objcopy run.
#DEBUG = stabs
DEBUG = dwarf-2

# List any extra directories to look for include files here.
#     Each directory must be seperated by a space.
EXTRAINCDIRS = ./include ./str71x_lib/include

# List any extra directories to look for library files here.
#     Each directory must be seperated by a space.
#EXTRA_LIBDIRS = ../arm7_efsl_0_2_4
EXTRA_LIBDIRS = ./str71x_lib


# Compiler flag to set the C Standard level.
# c89   - "ANSI" C
# gnu89 - c89 plus GCC extensions
# c99   - ISO C99 standard (not yet fully implemented)
# gnu99 - c99 plus GCC extensions
CSTANDARD = -std=gnu99

# Place -D or -U options for C here
CDEFS =  -D$(RUN_MODE)

# Place -I options here
CINCS =

# Place -D or -U options for ASM here
ADEFS =  -D$(RUN_MODE)

ifdef VECTOR_LOCATION
CDEFS += -D$(VECTOR_LOCATION)
ADEFS += -D$(VECTOR_LOCATION)
endif

CDEFS += -D__WinARM__ -D__WINARMSUBMDL_$(SUBMDL)__
ADEFS += -D__WinARM__ -D__WINARMSUBMDL_$(SUBMDL)__

# Compiler flags.

ifeq ($(USE_THUMB_MODE),YES)
THUMB    = -mthumb
THUMB_IW = -mthumb-interwork
else
THUMB    =
THUMB_IW =
endif

#  -g*:          generate debugging information
#  -O*:          optimization level
#  -f...:        tuning, see GCC manual and avr-libc documentation
#  -Wall...:     warning level
#  -Wa,...:      tell GCC to pass this to the assembler.
#    -adhlns...: create assembler listing
#
# Flags for C and C++ (arm-elf-gcc/arm-elf-g++)
CFLAGS = -g$(DEBUG)
```

```
CFLAGS += $(CDEFS) $(CINCS)
CFLAGS += -O$(OPT)
CFLAGS += -Wall -Wcast-align -Wimplicit
CFLAGS += -Wpointer-arith -Wswitch
CFLAGS += -ffunction-sections -fdata-sections
CFLAGS += -Wredundant-decls -Wreturn-type -Wshadow -Wunused
CFLAGS += -Wa,-adhlns=$(subst $(suffix $<),.lst,$<)
CFLAGS += $(patsubst %,-I%,$(EXTRAINCDIRS))

# flags only for C
CONLYFLAGS += -Wnested-externs
CONLYFLAGS += $(CSTANDARD)

ifneq ($(AT91LIBNOWARN),yes)
#AT91-lib warnings with:
CFLAGS += -Wcast-qual
CONLYFLAGS += -Wmissing-prototypes
CONLYFLAGS += -Wstrict-prototypes
CONLYFLAGS += -Wmissing-declarations
endif

# flags only for C++ (arm-elf-g++)
# CPPFLAGS = -fno-rtti -fno-exceptions
CPPFLAGS =

# Assembler flags.
#  -Wa,...:    tell GCC to pass this to the assembler.
#  -ahlns:     create listing
#  -g$(DEBUG): have the assembler create line number information
ASFLAGS = $(ADEFS) -Wa,-adhlns=$(<:.S=.lst), -g$(DEBUG)


#Additional libraries.

# Extra libraries
#    Each library-name must be seperated by a space.
#    To add libxyz.a, libabc.a and libefsl.a:
#    EXTRA_LIBS = xyz abc efsl
#EXTRA_LIBS = efsl
EXTRA_LIBS = STR71x_lib

#Support for newlibc-lpc (file: libnewlibc-lpc.a)
#NEWLIBLPC = -lnewlib-lpc

MATH_LIB = -lm

# CPLUSPLUS_LIB = -lstdc++


# Linker flags.
#  -Wl,...:     tell GCC to pass this to linker.
#    -Map:      create map file
#    --cref:    add cross reference to  map file
LDFLAGS = -nostartfiles -Wl,-Map=$(TARGET).map,--cref,--gc-sections
LDFLAGS += -lc
LDFLAGS += $(NEWLIBLPC) $(MATH_LIB)
LDFLAGS += -lc -lgcc
LDFLAGS += $(CPLUSPLUS_LIB)
LDFLAGS += $(patsubst %,-L%,$(EXTRA_LIBDIRS))
LDFLAGS += $(patsubst %,-l%,$(EXTRA_LIBS))

# Set Linker-Script Depending On Selected Memory and Controller
ifeq ($(RUN_MODE),RAM_RUN)
LDFLAGS +=-T$(LINKERSCRIPTPATH)/$(SUBMDL)-RAM.ld
else
LDFLAGS +=-T$(LINKERSCRIPTPATH)/$(SUBMDL)-ROM.ld
endif


# Define directories, if needed.
## DIRARM = c:/WinARM/
## DIRARMBIN = $(DIRAVR)/bin/
## DIRAVRUTILS = $(DIRAVR)/utils/bin/

# Define programs and commands.
SHELL = sh
CC = $(TCHAIN)-gcc
CPP = $(TCHAIN)-g++
AR = $(TCHAIN)-ar
OBJCOPY = $(TCHAIN)-objcopy
OBJDUMP = $(TCHAIN)-objdump
SIZE = $(TCHAIN)-size
NM = $(TCHAIN)-nm
REMOVE = rm -f
REMOVEDIR = rm -f -r
COPY = cp

# Define Messages
# English
MSG_ERRORS_NONE = Errors: none
```

```
MSG_BEGIN = "-------- begin (mode: $(RUN_MODE)) --------"
MSG_END = --------  end  --------
MSG_SIZE_BEFORE = Size before:
MSG_SIZE_AFTER = Size after:
MSG_FLASH = Creating load file for Flash:
MSG_EXTENDED_LISTING = Creating Extended Listing:
MSG_SYMBOL_TABLE = Creating Symbol Table:
MSG_LINKING = Linking:
MSG_COMPILING = Compiling C:
MSG_COMPILING_ARM = "Compiling C (ARM-only):"
MSG_COMPILINGCPP = Compiling C++:
MSG_COMPILINGCPP_ARM = "Compiling C++ (ARM-only):"
MSG_ASSEMBLING = Assembling:
MSG_ASSEMBLING_ARM = "Assembling (ARM-only):"
MSG_CLEANING = Cleaning project:
MSG_FORMATERROR = Can not handle output-format
MSG_LPC21_RESETREMINDER = You may have to bring the target in bootloader-mode now.

# Define all object files.
COBJ      = $(SRC:.c=.o)
AOBJ      = $(ASRC:.S=.o)
COBJARM   = $(SRCARM:.c=.o)
AOBJARM   = $(ASRCARM:.S=.o)
CPPOBJ    = $(CPPSRC:.cpp=.o)
CPPOBJARM = $(CPPSRCARM:.cpp=.o)

# Define all listing files.
LST = $(ASRC:.S=.lst) $(ASRCARM:.S=.lst) $(SRC:.c=.lst) $(SRCARM:.c=.lst)
LST += $(CPPSRC:.cpp=.lst) $(CPPSRCARM:.cpp=.lst)

# Compiler flags to generate dependency files.
### GENDEPFLAGS = -Wp,-M,-MP,-MT,$(*F).o,-MF,.dep/$(@F).d
GENDEPFLAGS = -MD -MP -MF .dep/$(@F).d

# Combine all necessary flags and optional flags.
# Add target processor to flags.
ALL_CFLAGS  = -mcpu=$(MCU) $(THUMB_IW) -I. $(CFLAGS) $(GENDEPFLAGS)
ALL_ASFLAGS = -mcpu=$(MCU) $(THUMB_IW) -I. -x assembler-with-cpp $(ASFLAGS)


# Default target.
all: begin gccversion sizebefore build sizeafter finished end

ifeq ($(FORMAT),ihex)
build: elf hex lss sym
hex: $(TARGET).hex
IMGEXT=hex
else
ifeq ($(FORMAT),binary)
build: elf bin lss sym
bin: $(TARGET).bin
IMGEXT=bin
else
ifeq ($(FORMAT),both)
build: elf hex bin lss sym
hex: $(TARGET).hex
bin: $(TARGET).bin
else
$(error "$(MSG_FORMATERROR) $(FORMAT)")
endif
endif
endif

elf: $(TARGET).elf
lss: $(TARGET).lss
sym: $(TARGET).sym

# Eye candy.
begin:
	@echo
	@echo $(MSG_BEGIN)

finished:
	@echo $(MSG_ERRORS_NONE)

end:
	@echo $(MSG_END)
	@echo


# Display size of file.
HEXSIZE = $(SIZE) --target=$(FORMAT) $(TARGET).hex
ELFSIZE = $(SIZE) -A $(TARGET).elf
sizebefore:
	@if [ -f $(TARGET).elf ]; then echo; echo $(MSG_SIZE_BEFORE); $(ELFSIZE); echo; fi

sizeafter:
	@if [ -f $(TARGET).elf ]; then echo; echo $(MSG_SIZE_AFTER); $(ELFSIZE); echo; fi
```

```
# Display compiler version information.
gccversion :
	@$(CC) --version

# FLASH Programming with OPENOCD

# specify the directory where openocd executable resides (openocd-ftd2xx.exe or openocd-pp.exe)
# Note: you may have to adjust this if a newer version of YAGARTO has been downloaded
OPENOCD_DIR = '/usr/local/bin/'

# specify OpenOCD executable (pp is for the wiggler, ftd2xx is for the USB debugger)
OPENOCD = $(OPENOCD_DIR)openocd
#OPENOCD = $(OPENOCD_DIR)openocd-ftd2xx.exe

# specify OpenOCD configuration file (pick the one for your device)
#OPENOCD_CFG = /home/l3golas/openocd-configs/str71x-configs/str71x_signalyzer-flash-program.cfg
#OPENOCD_CFG = /home/l3golas/openocd-configs/str71x-configs/str71x_jtagkey-flash-program.cfg
#OPENOCD_CFG = /home/l3golas/openocd-configs/str71x-configs/str71x_armusbocd-flash-program.cfg
OPENOCD_CFG = /home/l3golas/openocd-configs/str71x-configs/str71x_pp-flash-program.cfg

program:
	@echo
	@echo "Flash Programming with OpenOCD..."
	$(OPENOCD) -f $(OPENOCD_CFG)
	@echo
	@echo
	@echo "Flash Programming Finished."


# Create final output file (.hex) from ELF output file.
%.hex: %.elf
	@echo
	@echo $(MSG_FLASH) $@
	$(OBJCOPY) -O ihex $< $@

# Create final output file (.bin) from ELF output file.
%.bin: %.elf
	@echo
	@echo $(MSG_FLASH) $@
	$(OBJCOPY) -O binary $< $@


# Create extended listing file from ELF output file.
# testing: option -C
%.lss: %.elf
	@echo
	@echo $(MSG_EXTENDED_LISTING) $@
	$(OBJDUMP) -h -S -C $< > $@


# Create a symbol table from ELF output file.
%.sym: %.elf
	@echo
	@echo $(MSG_SYMBOL_TABLE) $@
	$(NM) -n $< > $@


# Link: create ELF output file from object files.
.SECONDARY : $(TARGET).elf
.PRECIOUS : $(AOBJARM) $(AOBJ) $(COBJARM) $(COBJ) $(CPPOBJ) $(CPPOBJARM)
%.elf:  $(AOBJARM) $(AOBJ) $(COBJARM) $(COBJ) $(CPPOBJ) $(CPPOBJARM)
	@echo
	@echo $(MSG_LINKING) $@
	  $(CC) $(THUMB) $(ALL_CFLAGS) $(AOBJARM) $(AOBJ) $(COBJARM) $(COBJ) $(CPPOBJ) $(CPPOBJARM) --output $@ $
(LDFLAGS)
#	$(CPP) $(THUMB) $(ALL_CFLAGS) $(AOBJARM) $(AOBJ) $(COBJARM) $(COBJ) $(CPPOBJ) $(CPPOBJARM) --output $@ $
(LDFLAGS)

# Compile: create object files from C source files. ARM/Thumb
$(COBJ) : %.o : %.c
	@echo
	@echo $(MSG_COMPILING) $<
	$(CC) -c $(THUMB) $(ALL_CFLAGS) $(CONLYFLAGS) $< -o $@

# Compile: create object files from C source files. ARM-only
$(COBJARM) : %.o : %.c
	@echo
	@echo $(MSG_COMPILING_ARM) $<
	$(CC) -c $(ALL_CFLAGS) $(CONLYFLAGS) $< -o $@

# Compile: create object files from C++ source files. ARM/Thumb
$(CPPOBJ) : %.o : %.cpp
	@echo
	@echo $(MSG_COMPILINGCPP) $<
	$(CPP) -c $(THUMB) $(ALL_CFLAGS) $(CPPFLAGS) $< -o $@

# Compile: create object files from C++ source files. ARM-only
$(CPPOBJARM) : %.o : %.cpp
	@echo
	@echo $(MSG_COMPILINGCPP_ARM) $<
```

```
	$(CPP) -c $(ALL_CFLAGS) $(CPPFLAGS) $< -o $@


# Compile: create assembler files from C source files. ARM/Thumb
## does not work - TODO - hints welcome
##$(COBJ) : %.s : %.c
##	$(CC) $(THUMB) -S $(ALL_CFLAGS) $< -o $@


# Assemble: create object files from assembler source files. ARM/Thumb
$(AOBJ) : %.o : %.S
	@echo
	@echo $(MSG_ASSEMBLING) $<
	$(CC) -c $(THUMB) $(ALL_ASFLAGS) $< -o $@


# Assemble: create object files from assembler source files. ARM-only
$(AOBJARM) : %.o : %.S
	@echo
	@echo $(MSG_ASSEMBLING_ARM) $<
	$(CC) -c $(ALL_ASFLAGS) $< -o $@


# Target: clean project.
clean: begin clean_list finished end


clean_list :
	@echo
	@echo $(MSG_CLEANING)
	$(REMOVE) $(TARGET).hex
	$(REMOVE) $(TARGET).bin
	$(REMOVE) $(TARGET).obj
	$(REMOVE) $(TARGET).elf
	$(REMOVE) $(TARGET).map
	$(REMOVE) $(TARGET).obj
	$(REMOVE) $(TARGET).a90
	$(REMOVE) $(TARGET).sym
	$(REMOVE) $(TARGET).lnk
	$(REMOVE) $(TARGET).lss
	$(REMOVE) $(COBJ)
	$(REMOVE) $(CPPOBJ)
	$(REMOVE) $(AOBJ)
	$(REMOVE) $(COBJARM)
	$(REMOVE) $(CPPOBJARM)
	$(REMOVE) $(AOBJARM)
	$(REMOVE) $(LST)
	$(REMOVE) $(SRC:.c=.s)
	$(REMOVE) $(SRC:.c=.d)
	$(REMOVE) $(SRCARM:.c=.s)
	$(REMOVE) $(SRCARM:.c=.d)
	$(REMOVE) $(CPPSRC:.cpp=.s)
	$(REMOVE) $(CPPSRC:.cpp=.d)
	$(REMOVE) $(CPPSRCARM:.cpp=.s)
	$(REMOVE) $(CPPSRCARM:.cpp=.d)
	$(REMOVEDIR) .dep | exit 0


# Include the dependency files.
-include $(shell mkdir .dep 2>/dev/null) $(wildcard .dep/*)


# Listing of phony targets.
.PHONY : all begin finish end sizebefore sizeafter gccversion \
build elf hex bin lss sym clean clean_list program
```

The *Makefile* is really well commented (Martin Thomas did a really good job), we left some options present in the original file and that we don't need in our case, however they have been commented. The lines

```
#USE_THUMB_MODE = YES
USE_THUMB_MODE = NO
```

let you choose if to use or not the THUMB mode, which in this case we don't use. A very important section is the one relative to the execution of the program in RAM or in ROM (FLASH).

```
## Create ROM-Image
#RUN_MODE=ROM_RUN
## Create RAM-Image
RUN_MODE=RAM_RUN
```

You have to edit the *Makefile*, search the section showed in the picture and uncomment the line that corresponds to the chosen execution, leaving the other one commented: in this case we chose to execute the code in RAM.
The line

```
TARGET = main
```

indicates the name (without extension), of the file originated by the process, in this case *main*, don't change it.
In the line

```
SRC  = src/$(TARGET).c
```

you can already find *main.c* and you have to add the other *.c* files contained in the *src* folder.
The lines

```
## Output format. (can be ihex or binary or both)
## (binary i.e. for openocd and SAM-BA, hex i.e. for lpc21isp and uVision)
#FORMAT = ihex
#FORMAT = binary
FORMAT = both
```

allow you to choose the output format, as specified by the comment.
The lines

```
# Optimization level, can be [0, 1, 2, 3, s].
# 0 = turn off optimization. s = optimize for size.
# (Note: 3 is not always the best optimization level. See avr-libc FAQ.)
#OPT = s
OPT = 0
```

indicate the optimization level, as specified in the comments (in this case no optimizations). For more information about the optimizations and the *Thumb mode* refer to the B appendix.
The lines

```
# Debugging format.
# Native formats for AVR-GCC's -g are stabs [default], or dwarf-2.
# AVR (extended) COFF requires stabs, plus an avr-objcopy run.
#DEBUG = stabs
DEBUG = dwarf-2
```

allow you to select the debugging format. Our advice is to select *dwarf-2* in that the debugging information inserted in the final file are more useful (with *stabs* during the debugging you would see only assembly code).
The line

```
EXTRA_LIBDIRS = ./str71x_lib
```

indicates the directory of the *.a* file generated from the compilation of the software library.
At the line `ASFLAGS = $(ADEFS) -Wa,-adhlns=$(<:.S=.lst), -g$(DEBUG)` we added a space before `-g$(DEBUG)` that wasn't present in the original file, because we noticed compilation problems on Linux without that space.
At the end we have the section relative to the FLASH programming, that can be done after the compilation of the program, simply through the command *make program*. In fact the *Makefile* is already prepared to do the operation in a simple way, being only necessary to edit it and modify few

parameters. Search the section

```
# FLASH Programming with OPENOCD

# specify the directory where openocd executable resides (openocd-ftd2xx.exe or openocd-pp.exe)
# Note: you may have to adjust this if a newer version of YAGARTO has been downloaded
OPENOCD_DIR = '/usr/local/bin/'

# specify OpenOCD executable (pp is for the wiggler, ftd2xx is for the USB debugger)
OPENOCD = $(OPENOCD_DIR)openocd
#OPENOCD = $(OPENOCD_DIR)openocd-ftd2xx.exe

# specify OpenOCD configuration file (pick the one for your device)
#OPENOCD_CFG = /home/l3golas/openocd-configs/str71x-configs/str71x_signalyzer-flash-program.cfg
#OPENOCD_CFG = /home/l3golas/openocd-configs/str71x-configs/str71x_jtagkey-flash-program.cfg
#OPENOCD_CFG = /home/l3golas/openocd-configs/str71x-configs/str71x_armusbocd-flash-program.cfg
OPENOCD_CFG = /home/l3golas/openocd-configs/str71x-configs/str71x_pp-flash-program.cfg

program:
	@echo
	@echo "Flash Programming with OpenOCD..."
	$(OPENOCD) -f $(OPENOCD_CFG)
	@echo
	@echo "Flash Programming Finished."
```

The changes you have to do are relative to the OpenOCD path (*OPENOCD_DIR* macro), the executable name (*OPENOCD* macro) and the path and name of the configuration file to use with OpenOCD (*OPENOCD_CFG* macro), depending on the interface you are using (as usual we used the parallel port). Probably you have to modify the last line in the indicated file, too, that is the *ocd* file path, in which there are the commands to write the FLASH memory.
So how does *Makefile* work? By launching the command *make all* you call several sections there present and identified by a label, that performs all the operations: the *.c* files, thanks to the compiler, become object files and the same is for *.S* startup files, which become object files thanks to the assembler; now it's the turn of the linker, whose task is to link the object files, to add the part relative to the addresses (still not present) and also to add the libraries and the linker files we saw above. From this “linking” process you obtain the file *main.elf*, that is your program plus all the debugging information and the file *main.map*, which can help you to determine modules length and their location in memory. The *main.elf* file is converted by the GNU utility *Objcopy* in *.hex* and/or *.bin* (depending on what you chose in the *Makefile*), used to write FLASH. Another file produced is the file *main.lss*, generated from the GNU utility *Objdump*, which shows C code with the corresponding Assembly code and their memory map. At the end the file *main.sym*, that is the symbol tables, thanks to the GNU utility *nm*.

3.2.2 Anglia's modifications to the ST software library and what they mean for program creation

Who wants a “true” program for one of the analysed microcontrollers and therefore wants to use the peripherals controlled by it, must use the software library, consulting the proper manual, that you can download as usual from the site http://mcu.st.com/mcu.
In the previous paragraph we said we chose to use a software library version slightly modified by Anglia, so who wants to develop using our solution, can still consult the original ST software library manual, but has to consider the modifications. Let's see what they are and how we can consider them:

1) the file whose name is similar to *7xx_it.c* (which is present in the original version) has been renamed in *vectors.c* and differs from the original one only because the functions body, rather than being empty, contains the command *while(1)*. You must modify the file *vectors.c* as specified in the software library for *7xx_it.c*, that is by inserting your own code in place of

while(1).

2) there is a new file called *vectors.h*

3) in the original version you had to include the file with the name similar to *7xx_conf.h*, after having uncommented in it the used peripherals and commented the others. Now it's not anymore necessary, in that this file is already present in the software library and the peripherals are all used by default, because, as you know, the software library is already compiled in a *.a* file that has to be included in your program following the rules written in the *Makefile*. Do you think it's a waste to include all the peripherals even if you don't use all of them and that the final file will take more space? **It's not so, because the GCC compiler is clever enough to understand by itself what are the peripherals really used by the program and to include only the corresponding files.** So it's not anymore necessary to select manually the peripherals, because the compiler does it for you! But the file *7xx_conf.h* was used also to set the *Main Oscillator* and *RTC Oscillator* frequencies... Now you don't have anymore to edit the definitions *#define Main_Oscillator* and *#define RC_oscillator*, in that the two values have to be set in the file *startup.S*, respectively at the lines "*.equ FOSC*" and "*.equ FRTC*", substituting them to the already present values.

4) in the original library version it was necessary to copy and define in the *Makefile* the *.c* and *.h* files of the peripherals to use, now it's not anymore necessary, in that these files are already inserted in the library that was already compiled in the *.a* file and included. This point depends obviously on what specified in the previous point. So in the *Makefile*, in the macro `SRC = src/$(TARGET).c` you have to add only the *.c* files which are in the subfolder *src*, except *main.c* (already inserted) and *vectors.c*.

So what you need to create a program? At first you must start from the proper template for your microcontroller and copy its content in a folder whose name is the name you want to give to your program. In the case of Eclipse, you can create a new project of type C standard and import files from the proper template. Now you have to modify the code of the files (for simple programs it's often sufficient to modify only the file *main.c*), following ST software library (taking into account the modifications specified in this paragraph) and your board datasheet (to know where your peripherals are connected and/or for further information). At the end, in the *Makefile*, add *.c* files which are in the folder *src* (except *main.c* and *vectors.c*) in the macro `SRC = src/$(TARGET`. It's simpler to do than explain!

3.2.3 How to adapt a program created for another solution to our model

This paragraph deals with how to adapt a program for our microcontrollers but created following another model to our model (template). There can be several situations, it depends on the structure of the program you want to adapt:

Program created using the software library modified by Anglia

You understand you are in this situation if the file *vectors.c* is present instead of a file with a name similar to *7xx_it.c*. In this case:

1) create a directory named with the chosen name (for example *prova*) and copy in it all the content of the directory of the used board template (for example *template71x*).

2) enter the directory *src* of *prova*. Copy in it the *.c* files of the program (not the software library files), overwriting the already existent ones.

3) edit the file *main.c* and make sure the debug section is present and is like this:

```
#ifdef DEBUG
      libdebug();
#endif
```
for a STR71x board and:
```
#ifdef LIBDEBUG
      libdebug();
#endif
```
for a board with one of the other microcontrollers

4) enter the subfolder *include* of *prova*. Copy in it the *.h* files of the program (not the software library files), overwriting the already existent ones.

5) now you have to modify the *Makefile* to let it know the modifications you did. Return in the directory *prova*, edit the *Makefile* and search the line `SRC = src/$(TARGET).c` inserting at the end of it the other *.c* files of the program (except *vectors.c*) and separating each one from the next one with a space.

Program created using the original ST software library

You understand we are in this situation if a file with a name similar to *7xx_it.c* is present, instead of the file *vectors.c*. In this case:

1) create a directory named with the chosen name (for example *prova*) and copy in it all the content of the directory of the used board template (for example *template71x*).

2) enter the directory *src* of *prova*. Copy in it the *.c* files of the program (not the software library files), overwriting the already existent ones.

3) edit the file *main.c* and make sure the debug section is present and is like this:
```
#ifdef DEBUG
      libdebug();
#endif
```
for a STR71x board and:
```
#ifdef LIBDEBUG
      libdebug();
#endif
```
for a board with one of the other microcontrollers

4) now you have to modify *vectors.c* following the content of the file whose name is similar to *7xx_it.c*. If you edit both these two files, you will note that both define the functions to run for each potential interrupt: by default, in *vectors.c* the function body contains the line *while(1)*, while in *7xx_it.c* the function body is empty. To modify *vectors.c*, at first you need to check if in *7xx_it.c* there are variables defined at the beginning of the program, (before the body of the functions) and if there are, copy them in *vectors.c*; after that check in *7xx_it.c* if there are functions whose body contains code and if so, copy it in the body of the corresponding function in *vectors.c*. Finally delete *7xx_it.c*

5) enter the subfolder *include* of *prova*. Copy in it the *.h* files of the program (not the software library files), overwriting the already existent ones.

6) delete the file with the name similar to *7xx_conf.h*, in that we don't need it in your model

7) now you have to modify the *Makefile* to let it know the modifications we did. Return in the directory *prova*, edit the *Makefile* and search the line `SRC = src/$(TARGET).c` inserting at the end of it the other *.c* files of the program (except *vectors.c*) and separating between each other with a space.

3.2.4 How to convert a program created with our model to a project for IAR environment.

We are writing such a paragraph in that the board with microcontroller STR730 we used is produced by IAR, that sells it together with its own IDE. It was useful for our tests trying to convert our template to a project for that IDE, so we decided to write here how we did it. Obviously it would be possible to do such a paragraph for the conversion in projects of other IDEs, too, we hope someone can spend some time doing that, it would be fine. Let's start with the procedure:

1) create a directory named with the chosen name (for example *prova*) and copy in it all the content of the directory of the used board template (for example *template71x*). From now we will use *prova* and *template71x* respectively as name of the project to create and of the used template, but obviously you have to substitute to these names what you prefer much.

2) rename the 3 files whose name is *template71x* to *prova*, leaving unchanged the extension

3) edit each of the 3 modified files, substituting in each of them all the occurrences of *template71x* with *prova*

4) copy to *prova* the files contained in the subfolders *src* and *include* of the GNU source project, except *vectors.h* because you don't need it

5) edit the file *main.c* and make sure the debug section is present and it's like this:

```
#ifdef DEBUG
      debug();
#endif
```

6) now you have to modify *7xx_it.c* following the content of the file whose name is similar to *vectors.c*. If you edit both these two files, you will note that both define the functions to run for each eventually occurring interrupt: in *vectors.c* by default the function body contains the line *while(1)*, while in *7xx_it.c* the function body is empty. To modify *7xx_it.c*, at first you have to check if in *vectors.c* there are variables defined at the beginning of the program, (before the body of the functions) and if there are, copy them in *7xx_it.c*; after that you have to check if in *vectors.c* there are functions whose body has code different from *while(1)* and if so copy it in the body of the corresponding function in *7xx_it.c*. Finally delete *vectors.c*.

7) now you need to know which files of the software library you have to link with the *main* of your project. To do so, open the source project in Eclipse and select the C/C++ perspective. Build the project. On the left there's a column in which you can see the project name preceded by a +. Click on it, then on the + that precedes *Binaries* and then on the + that precedes *main.elf*. You will see a set of files, take note of the ones with extension *.c* that are shown as *src/nomefile.c*.

8) edit the file whose name is similar to *7xx_conf.h* and enable the *#define* of the peripherals corresponding to the files of which you took note at the previous point.

9) return to the project directory. Double click on the file with extension *.eww* to start IAR IDE.

10) in the column on the right, where there is the project, right click on the directory *source* and select *Add -> Add files*; in the window that will appear enter the subdirectory *source* of the project and insert the files of which you took note at point 7). Then right click on the project name and choose *Add -> Add files*, selecting this time the *.c* files eventually present in the main directory of the project.

11) Now you have finished. Right click on the project name and choose *Clean* and then *Make*. If there are errors, probably you have to enable some other peripherals in the file *7xx_conf.h* (the error message will make you understand which peripheral you have to enable).

3.2.5 Programs used for the tests

To do the tests on the boards and make sure all worked properly, we created some programs which use the software library, so they use the peripherals the microcontroller supports. They are simple programs we wrote for all the boards we have used, for example to switch on and off the LEDs or to do a serial communication. Some of them were already created by ST and included in the software library, we simply converted them to our model, following the steps explained in the paragraph 3.2.3.

> We want to remind you the code of the provided programs is influenced by the board configuration: for example, if in our board the GPIO0 port is connected to LEDs, in another one maybe there are other peripherals connected to that port, so the code of the programs must be properly changed.

Each program we created (except for the programs which work with LEDs because they are really simple) contains a file *readme.txt* that explains its purpose and how it works.
As an example, we show now the code of the program *led730*, whose task is to switch on a group of LEDs in a certain order and then to switch them off in the reverse order. In order to create this program we followed what we said at the end of the paragraph 3.2.2.

```
#include "71x_lib.h"

void Delay(u32 Xtime);

int main(void)
{
#ifdef DEBUG
    libdebug();
#endif
u16 var;
GPIO_Config( GPIO1, 0xFFFF, GPIO_OUT_PP );
GPIO_BitWrite(GPIO1, 15, 0x00);
GPIO_BitWrite(GPIO1, 6, 0x00);
GPIO_BitWrite(GPIO1, 5, 0x00);
GPIO_BitWrite(GPIO1, 4, 0x00);
  // infinite loop
  while (1)
  {
       GPIO_BitWrite(GPIO1, 6, 0x01);
       Delay(0x2FFFF);
       GPIO_BitWrite(GPIO1, 5, 0x01);
       Delay(0x2FFFF);
       GPIO_BitWrite(GPIO1, 4, 0x01);
       Delay(0x2FFFF);
       GPIO_BitWrite(GPIO1, 4, 0x00);
       Delay(0x2FFFF);
       GPIO_BitWrite(GPIO1, 5, 0x00);
       Delay(0x2FFFF);
       GPIO_BitWrite(GPIO1, 6, 0x00);
       Delay(0x2FFFF);
  }
}

void Delay(u32 Xtime)
{
  u32 j;

  for(j=Xtime;j!=0;j--);
}
```

The lines

```
#ifdef DEBUG
    libdebug();
#endif
```

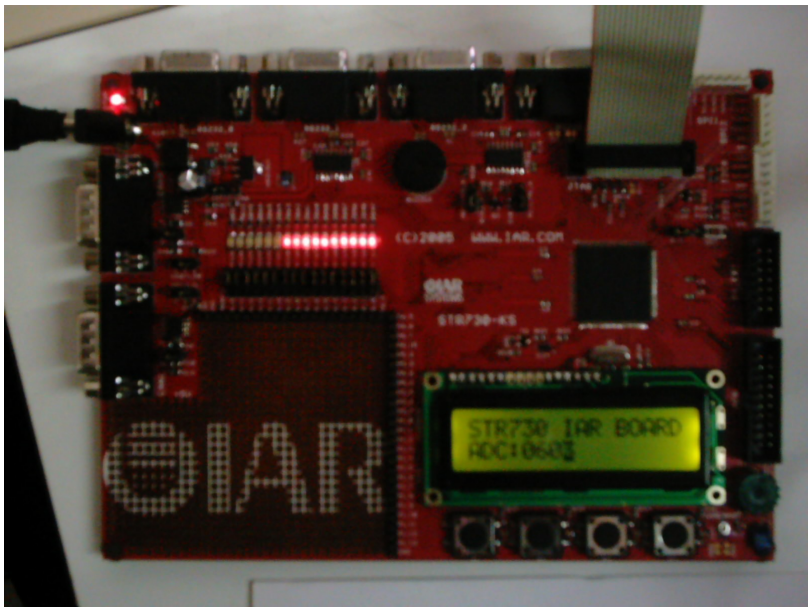
are present more or less in this form for all the analysed microcontrollers and they are used to help debugging. In fact, they indicate that if the *DEBUG* constant (for some microcontrollers it's called *LIBDEBUG*, please pay attention) is defined, the program executes the function *libdebug()*. This function is defined in the file *73x_lib.c* which is located in the subfolder *src* of the software library (folder *str73x_lib*), it's executed when the program starts executing and its task is to create pointers to the peripherals registers: so you can watch the value they assume, for example through the Eclipse view *Expressions* (see Appendix A.2 for more information). This is really useful also because the registers, normally watched through the Eclipse view *Registers* don't have the names with which they are called by ST, instead they have generic names such as *r0*, *r1*, etc., so this way it's possible to have variables that point to those registers and with more familiar names.
Therefore it's a useful function especially when debugging, but it must be used only if you really need it, in that it makes the code slower and bigger. For that reason ST created the constant *DEBUG*, present in the file *73x_conf.h* (which is located in the subfolder *include* of the software library), that only if activated makes the *main* calls the function *libdebug()* (by default this constant is commented).

A bit more complicated program is located in the folder *fourbits730* and it has been derived from a program written for the IAR IDE, we converted it for our solution following what we wrote in paragraph 3.2.3. To see how this program works, let's see a part of the file *readme.txt* contained in it.

```
This example allows displaying "STR730 IAR BOARD" message and the ADC End Of
Conversion interrupt result on the LCD display.

It is made of 6 steps :
1) Analog input configuration (in the GPIOx_InitStructure )
         + Configure the used analog input to High impedance Analog input (Channel0)


2) EIC configuration
   - ADC channel priority configuration
   - Enable ADC channel interrupt
   - Enable the EIC

3) Clock configuration
         + Select the desired ADC clock input: MCLK = 4Mhz


4) ADC configuration  (in the ADC_InitStructure )
  - Initialize the converter
  - Configure the prescaler register: Sampling prescaler = 0x2 and Conversion prescaler = 0x4
  - Configure the conversion mode: Scan mode conversion
  - Select the channel to be converted: only channel0
         + First channel = ADC_CHANNEL0
        + Channel number = 1

  - Enable EOC interrupt  (using the ADC_ITConfig function )
  - Enable the ADC peripheral (using the ADC_Cmd function )

5) LCD configuration
  - Initialize the LCD
  - Clear Line 1 & 2
  - Set cursor position:Line 1, column 1
  - Display "STR730 IAR BOARD" message on LCD: Line 1
  - Set cursor position:Line 2, column 1
  - Display "ADC:" message on LCD: Line 2
```

```
6) Get the conversion results and display it in the ADC_IRQHandler
  - Start the conversion (using the ADC_ConversionCmd function )
  - Read the conversion result (using the ADC_GetConversionValue function )
  - Display the conversion result on LCD display.
  - Clear EOC flag
```

Let's see also a photo of the board while running this program from FLASH:



Notice the display, that has been switched on and shows the string "*STR730 IAR BOARD*" and the ADC value. If you rotate the trimmer that is present at the right side of the LCD, the ADC value will change and the number of switched on LEDs will reflect this change.
To make the display work properly, we had to change something in the software library (we had to comment the line *#define _PRCCU* in the file *73x_conf.h*). It seems to work, even if we didn't understand why, probably it's a bug present in the LCD management functions (made by IAR) or in the software library.

Appendix A – Short tutorial on how to use Insight and Eclipse

You can consider this appendix as an integration to what we dealt with Insight and Eclipse in Chapter 2 (Tutorial), so it's necessary to read and understand it before reading further.

A.1 Insight

A.1.1 How to control code

Let's see how you can control code execution.

STEP (Step Into)

With this button you can control the program execution step by step, that is you can execute the current instruction and stop before running the next one. If the next instruction is a function, Insight enters it and executes its code step by step, so you can control the execution of really every instruction.

NEXT (Step Over)

By pressing this button, Insight executes the current instruction and stops at the next one. The difference with STEP is that NEXT, if the next instruction is a function, doesn't enter it but executes it in one step, so control always stays in the main code.

FINISH

This button allows you to run the code block in which you are till the end. For example it can be useful if you are into a function: suppose you entered it through STEP to do debug but now you think it's not necessary to execute it step by step because you know its behaviour doesn't give any problem; so with this button you can execute the remaining part in only one step, returning the control to the user at the instruction after it.

CONTINUE

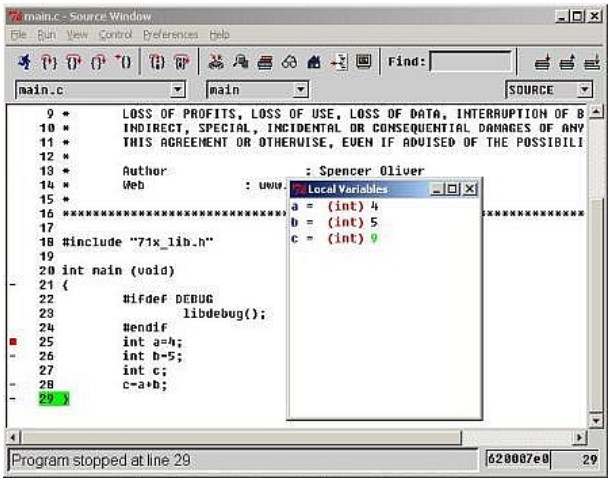
This button allows you to run the whole program from the current point till next breakpoint. If you didn't set any breakpoint, the program will be executed till its end.

However, it's also possible to use all these controls through the menu *Control* of Insight, how you can see in next picture:

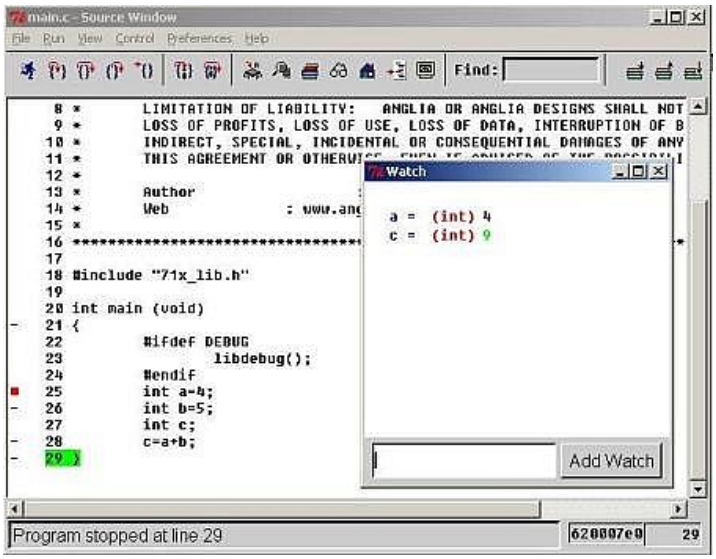
A.1.2 Insight views

Insight doesn't give only tools for controlling execution flow, but it also offers many other facilities which help programmers in code debugging. In fact, during code execution, it's possible to control the values of local variables or to specify a watch-list, that is a list of variables you want to monitor.

Local Variables ( )
This view, as you can see in the following picture, allows you to monitor all the local variables of the program.

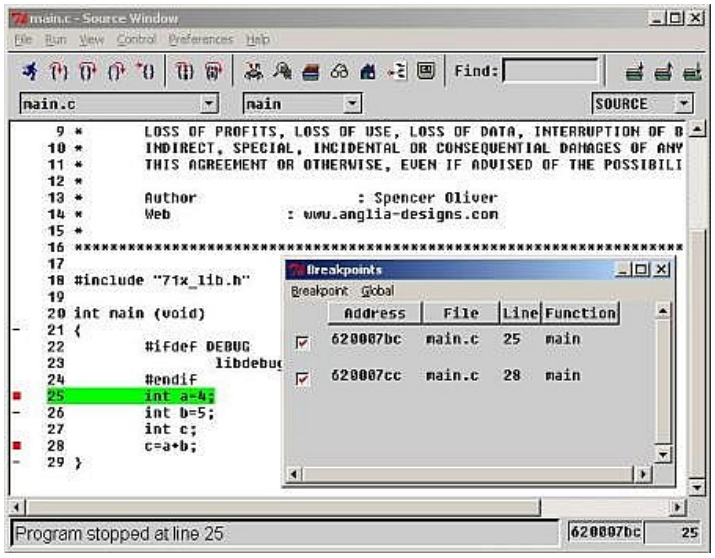


Watch Expressions ( )
Through this view you can create a list of variables of which you want to monitor values: in the next picture, for example, you can see we have inserted in the watch-list the variables *a* and *c*; so this view is useful especially for programs with many variables, if you are interested only to a subset of them.



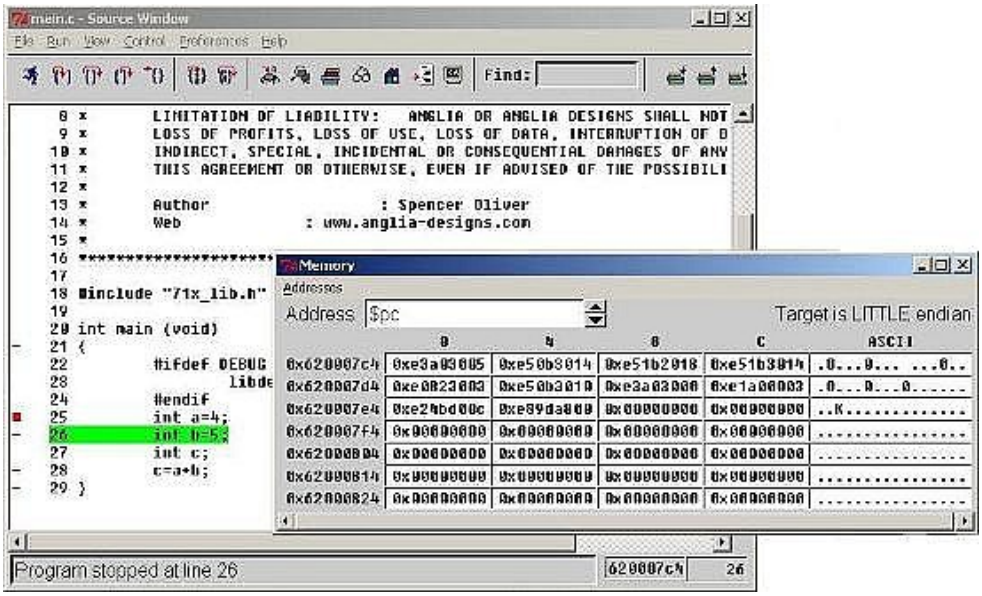
Breakpoints ( )

This view allows us, as you can notice in the next picture, to know all the features of the breakpoints you set in the program.

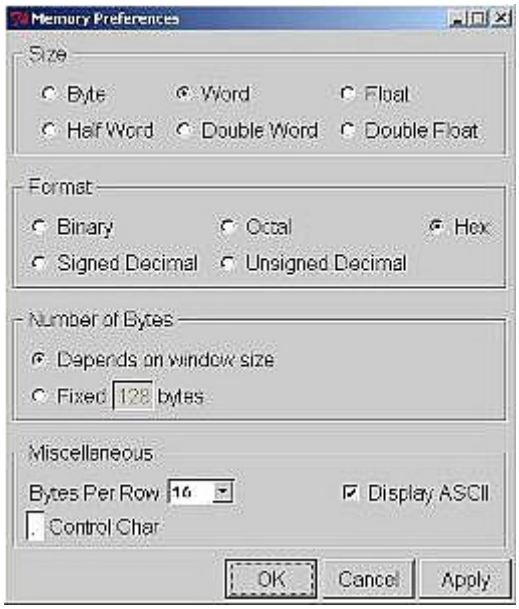


Memory ( )

Through the view *Memory*, it's possible to see the hexadecimal content of the memory, as you can see in the following picture:

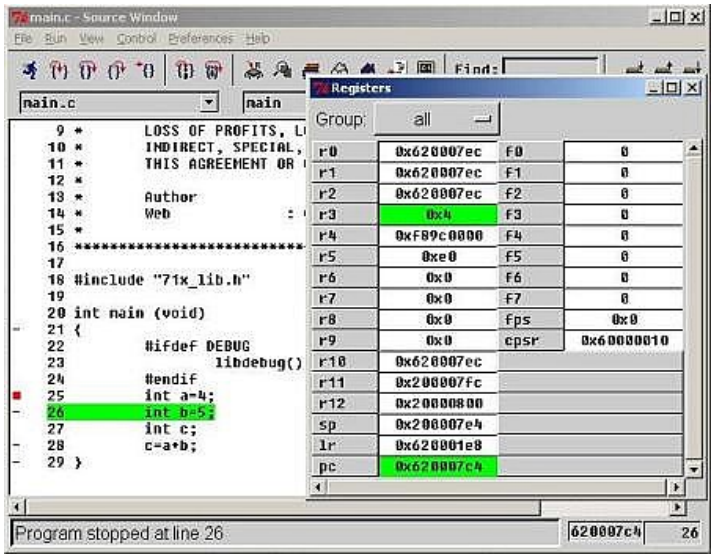


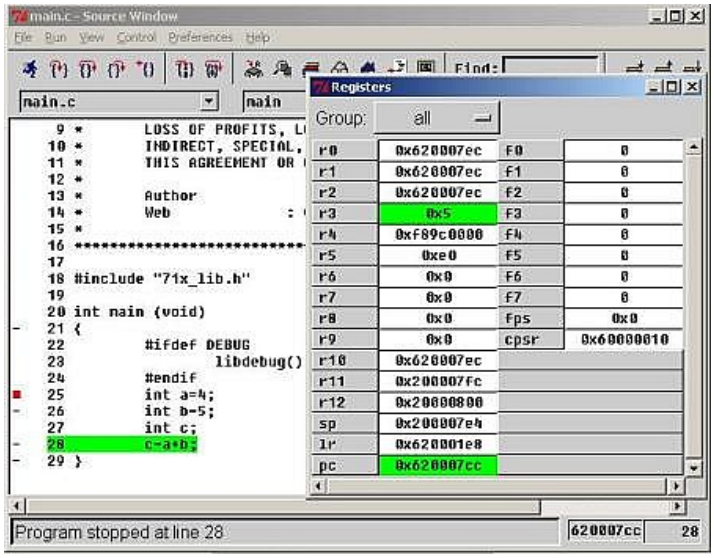
As you can notice, both the addresses and the words content are expressed in hexadecimal; it's possible to change some parameters about the visualization of memory data: to do that it's sufficient to go in the view *Memory*, in the menu *Addresses* and then choose *Preferences* to make the following window appear:



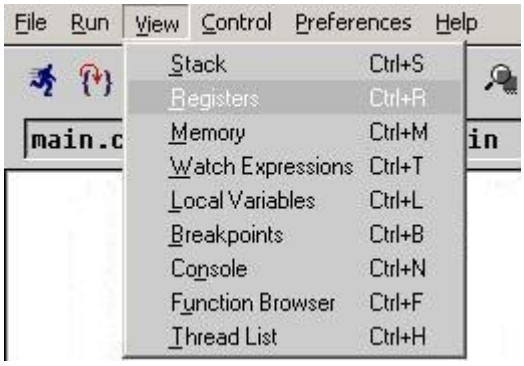
Registers (**)**

It's also possible to monitor the registers of your microcontroller, by selecting the proper view among the others, as you can see in the following pictures:





However, it's possible to choose each of these views by the menu *View* of Insight, as showed in the next picture:



A.2 Eclipse

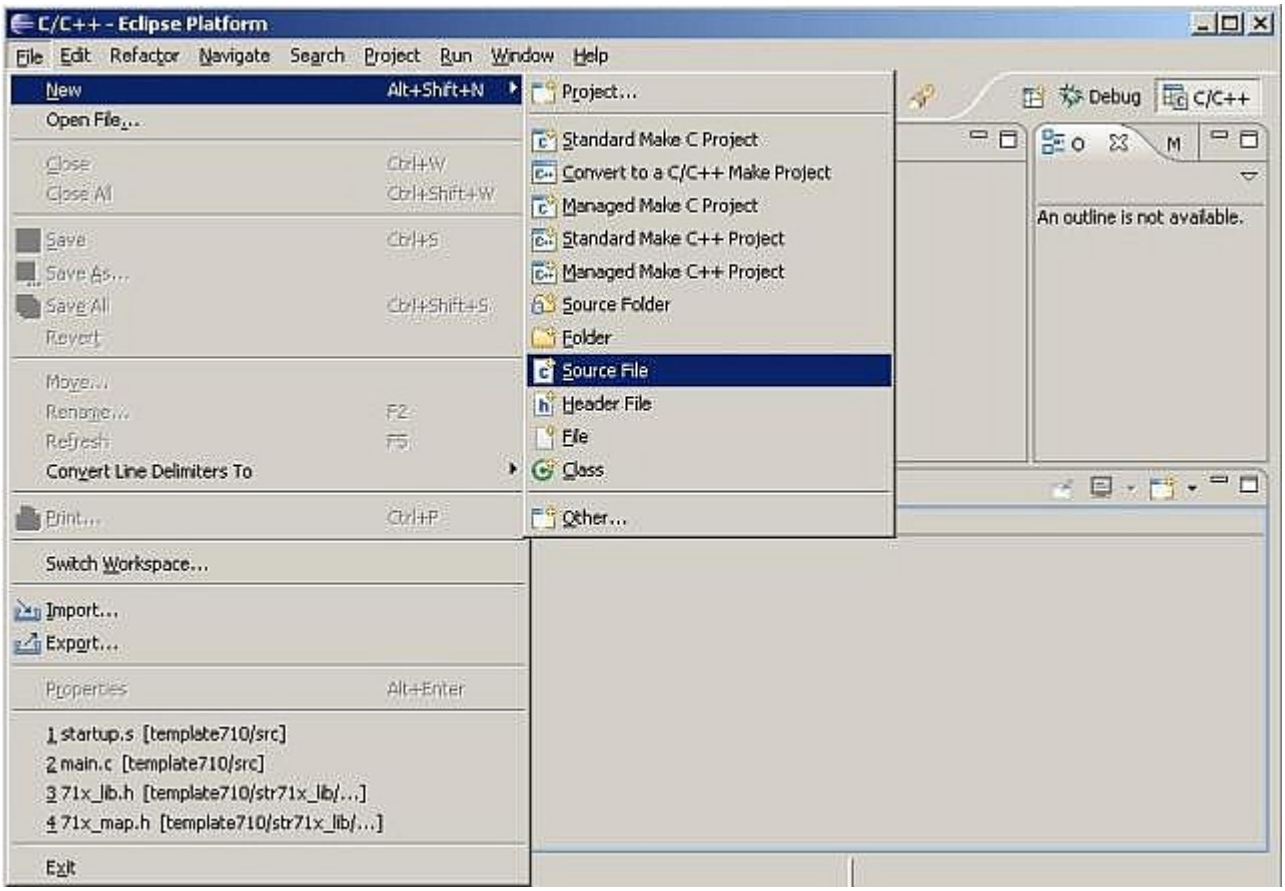
This paragraph gives some indications on some functionalities offered by Eclipse. In particular we will analyse:

- Creation of a new source file
- Typical editing operations
- Saving functions
- Parenthesis checking
- Searching functions
- Assembly debugging
- Views

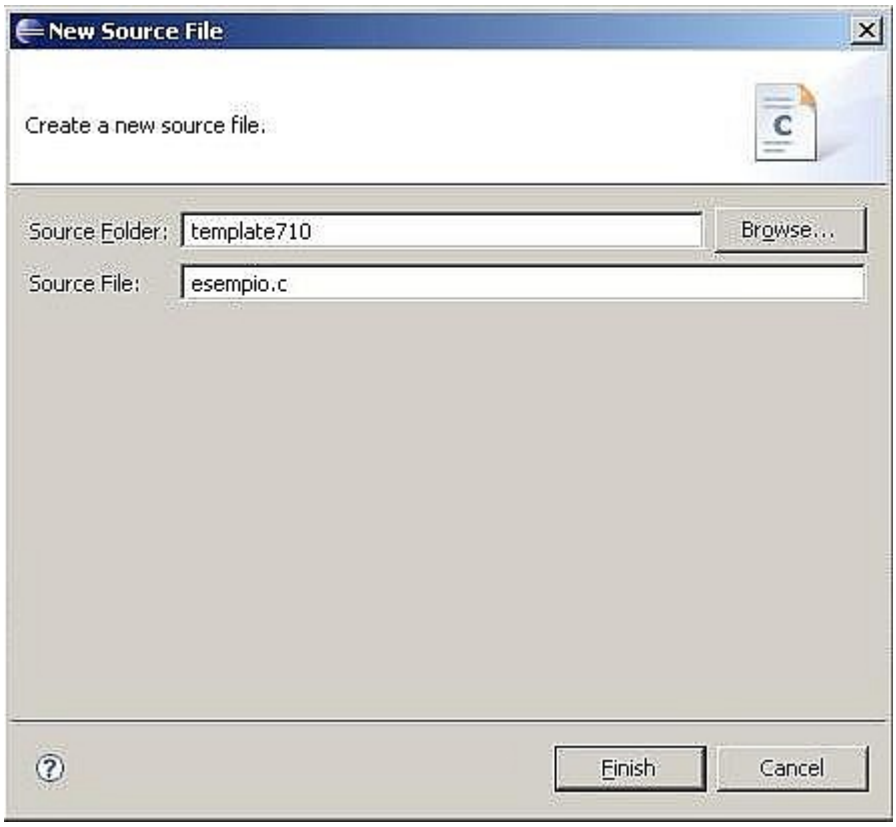
You can find a more detailed section in James P. Lynch's tutorial and in several other tutorials.

A.2.1 Creation of a new source file

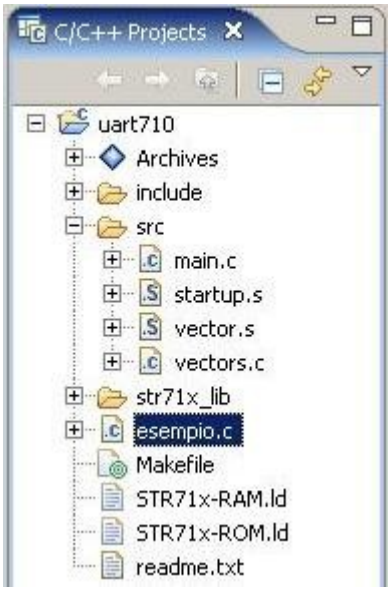
To create a new source file it's sufficient, as showed in the following pictures, to click on the menu *File->New->Source File.*



After that you must insert in the window that will appear the name of the file to create and the folder in which to put it (that is one of the already present projects). If no folder is present, you have to create one before to create the source file.



After having clicked on *Finish*, the new created file will be automatically inserted into *C/C++ Projects Window*, as showed in the following picture, and it will be possible to start writing code in it.

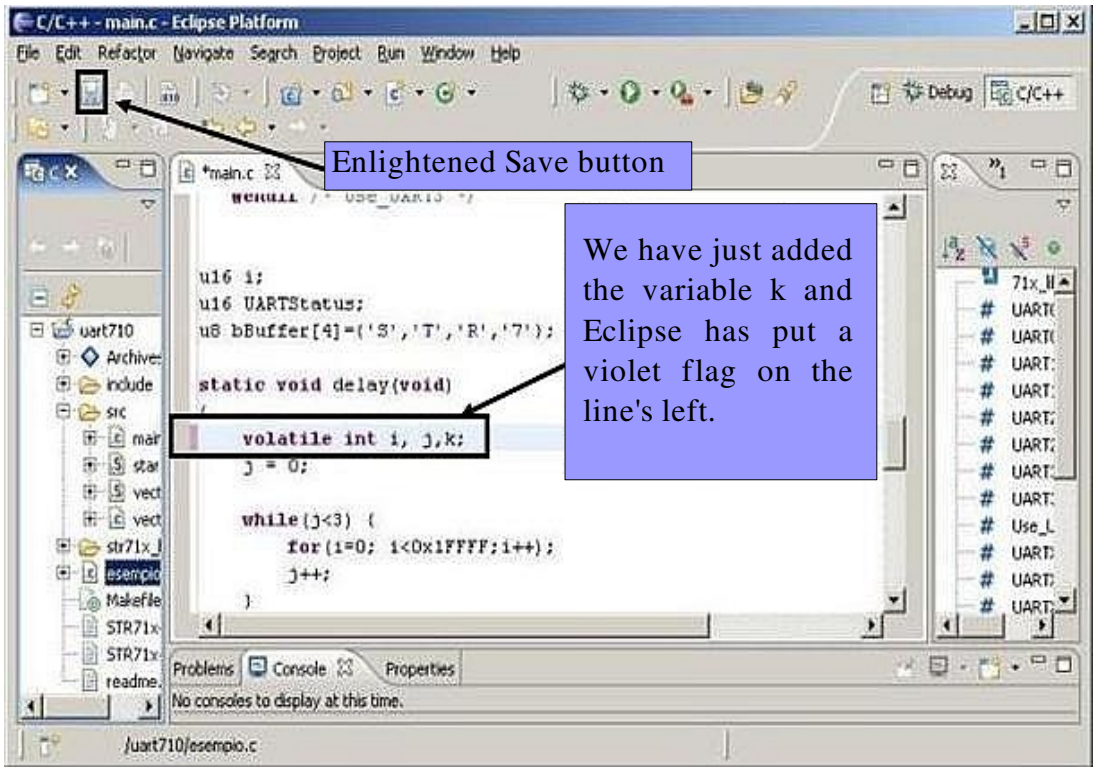


A.2.2 Typical editing operations

Eclipse, like many other IDEs, offers the classical editing operations: *undo, redo, cut, copy* and *paste*; they are located, as we could expect, in the menu *Edit*.

A.2.3 Saving options

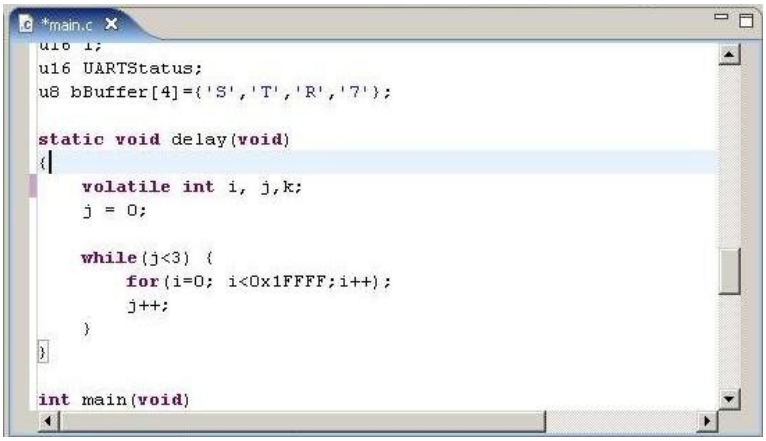
Eclipse offers the user some facilities to save data. In fact, when the user modifies a code line, Eclipse, as you can see in the next picture, marks the just modified line with a little violet flag and enlightens the button *Save* in the bar. Obviously, by clicking on it, you will save the changes you did in the file.



Moreover it's possible to set a timer, after which Eclipse will automatically save the updates and build the project; to do so it's sufficient to go in the menu *Windows->Preferences->General->Workspace*.

A.2.4 Parenthesis checking

Eclipse allows you to automatically find parenthesis, both curly and round; so the editor, once you put the cursor after the opening parenthesis, will automatically find the closing one and shows it. Moreover Eclipse also does the reverse operation: if you put the cursor after the closing parenthesis, it will show the opening one.

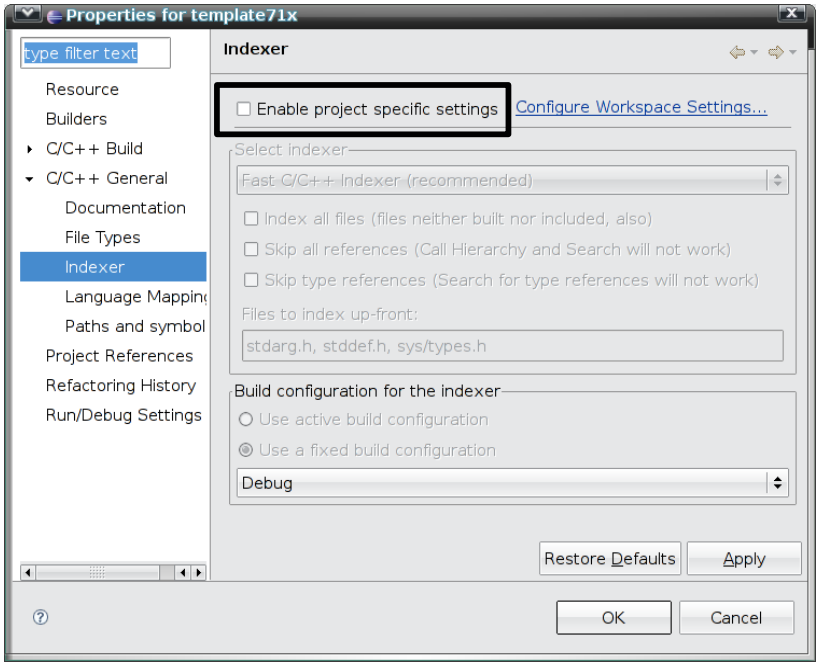


A.2.5 Searching functions

Main Eclipse search functions are:

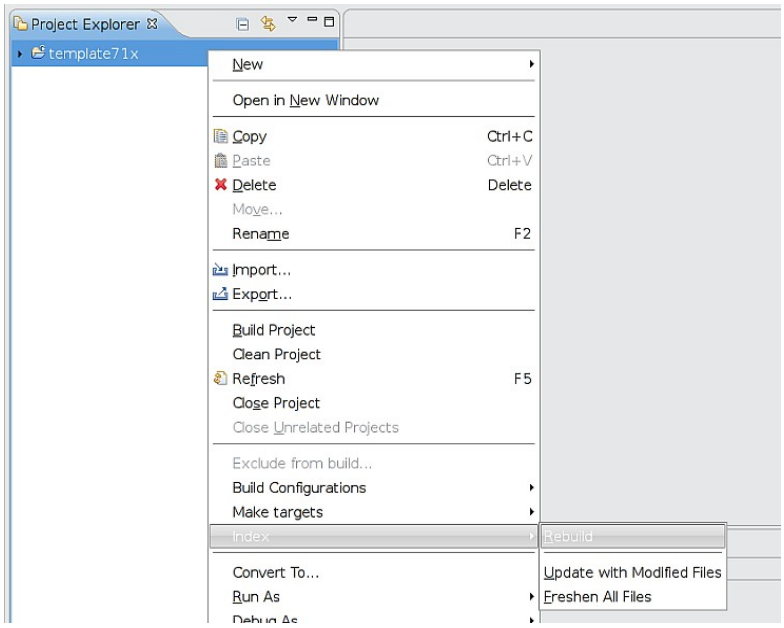
- Showing a selected variable definition
- Showing all the occurrences in the project of a selected variable

To use this function it's at first necessary to make sure the Eclipse *Indexer* is enabled. To do so, you have to go in the menu *Project->Properties->C/C++ General->Indexer*: so you make a window appear on the left, as showed in next picture:



Check the checkbox shown in the black square (*Enable project specific settings*) and then choose from the list below *Fast C/C++ Indexer* or *Full C/C++ Indexer*. For big projects the second one might be the better choice.

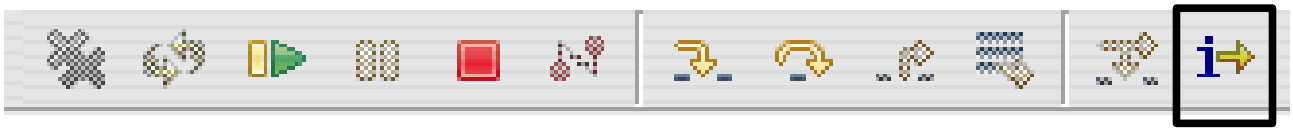
But before starting any search in the project, it's necessary to rebuild the Indexer; to do so you have to select the project, right click on it and choose *Index -> Rebuild*.



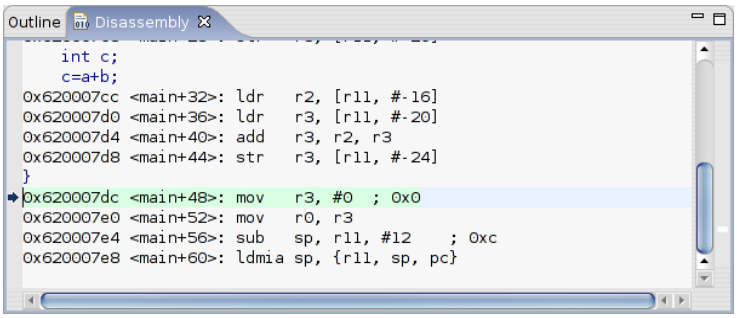
For example, if you select a variable in a source file and you press *F3*, Eclipse will show its definition.

A.2.6 Assembly debugging

In the *Debug* perspective you have the possibility to debug each Assembly instruction: it's sufficient to click on the button showed in the following picture to enable the *Disassembly* window.



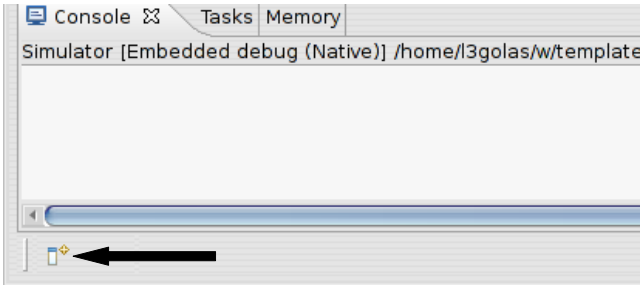
The *Disassembly* window shows the corresponding Assembly code to your program code:



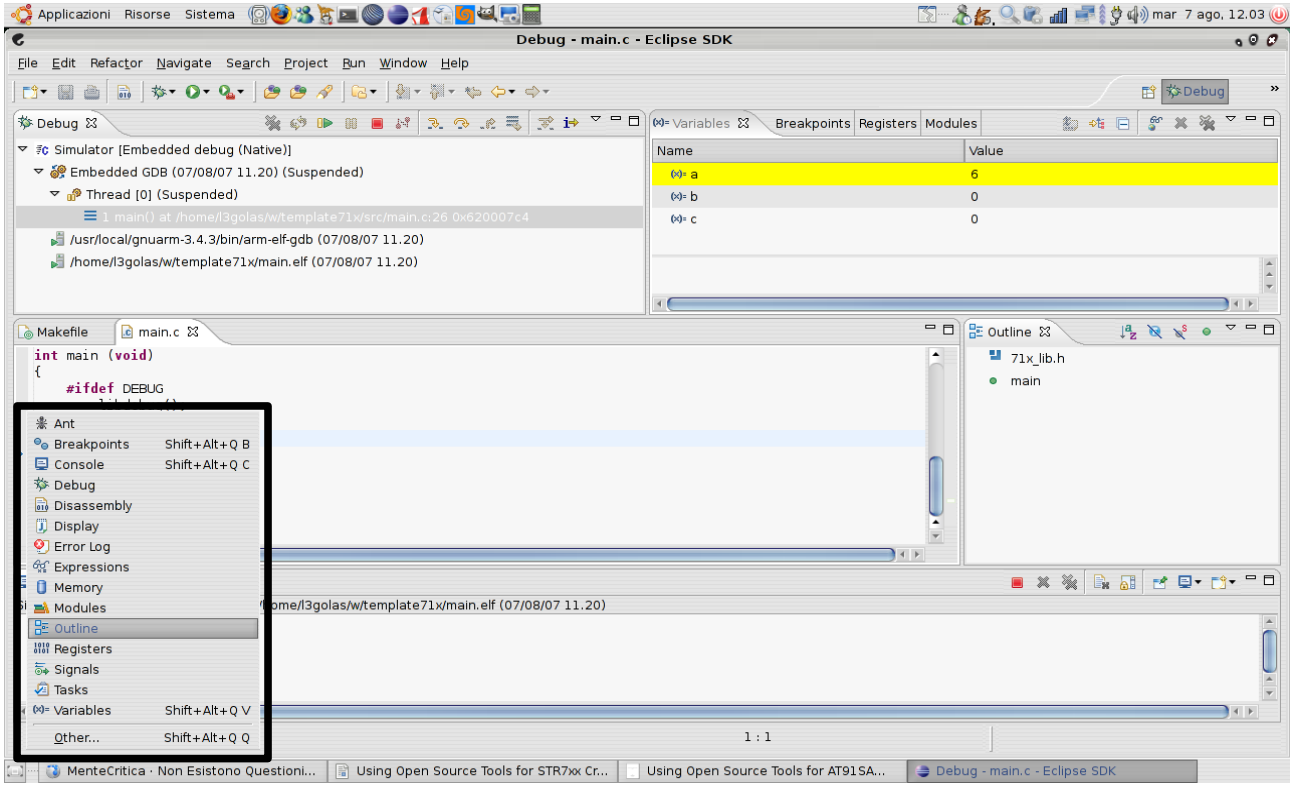
Now the buttons *Step Into*, *Step Over* and *Step Out* don't control anymore C, but Assembly code.

A.2.7 Views

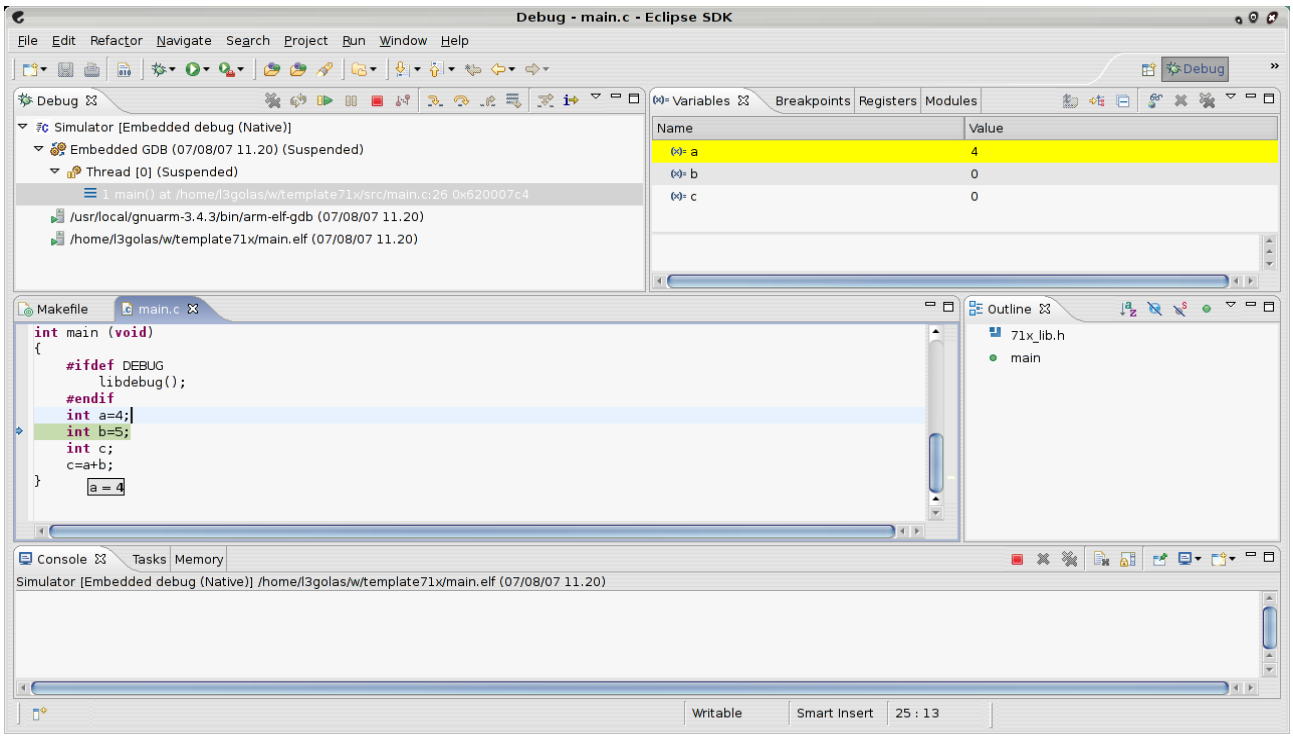
Like Insight, Eclipse has features which help the programmer in debugging his code. In fact, during code execution, it's possible to monitor for example the registers and the memory of the microcontroller, or to control the variables values or to specify a watch-list of variables you want to monitor. The views are easily callable by clicking on the button indicated in the next picture:



A list will appear, in which you can choose the view you want to use:



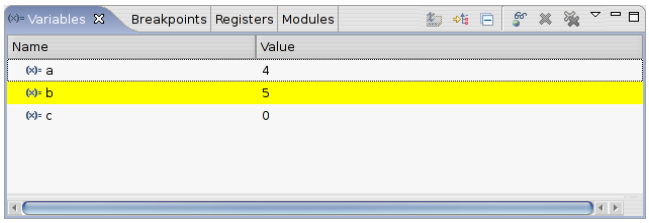
Every perspective has its own views, so the views shown are only the ones of the current perspective, but you can also select the other views by selecting "*Other...*", which will show another window with all the views.
The selected view will be opened in another window, but it can be inserted among the others already present in the upper right (*Variables*, *Breakpoints*, etc.), it's sufficient to select the window title (for example *Registers*) and carry it where there are the other views. Before dealing with the views and the differences among them, it's necessary to say that it's possible to see the value of every variable simply by positioning the cursor on its name in the code, as you can see in the following screenshot:



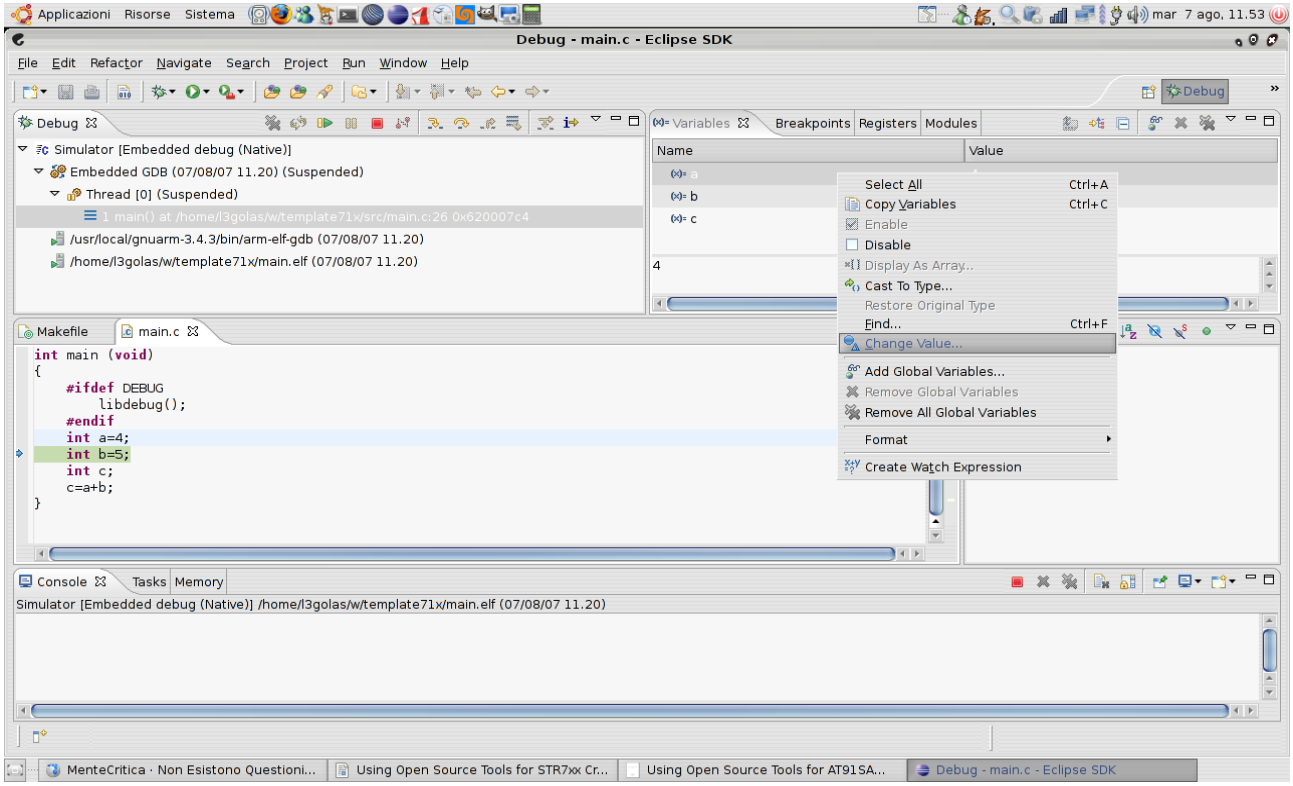
Now let's see the most important views of the Debug perspective, because they're the ones you will probably use more. You can easily try them while debugging one of your projects.

Local Variables

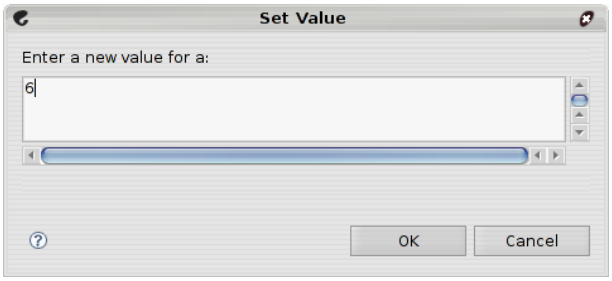
This view allows you to monitor all the local variables of the program, which are updated in real time.



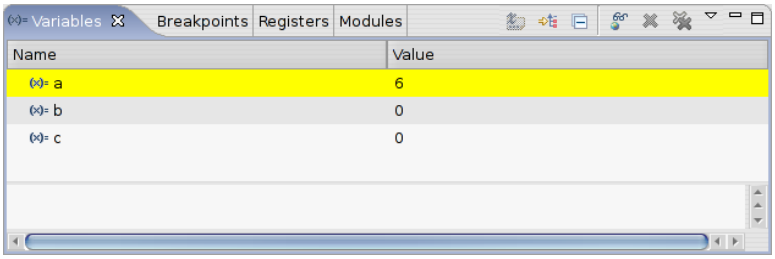
If the variables are more complex, such as structures or arrays, a “+” before the variable name will appear, on which you can click to see the internal elements.
It's also possible to modify a variable value, it's sufficient to right click on its name and choose *Change Value*, as you can see in the following picture:



A window will appear, in which you have to specify the new name:

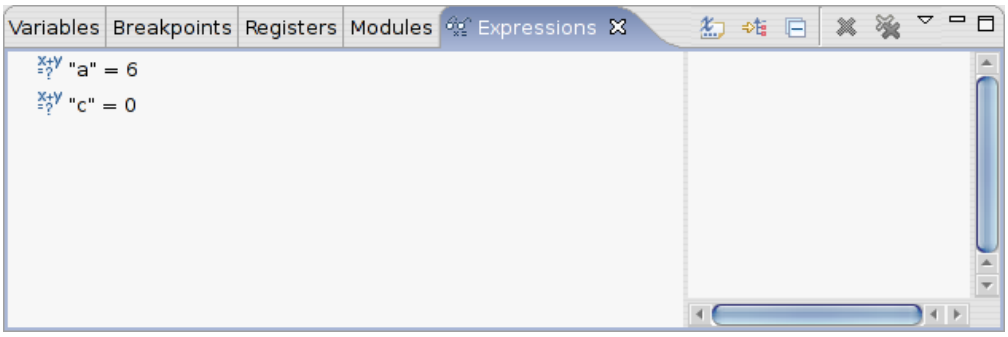


After having pressed OK, the variable name will be updated:



Watch Expressions

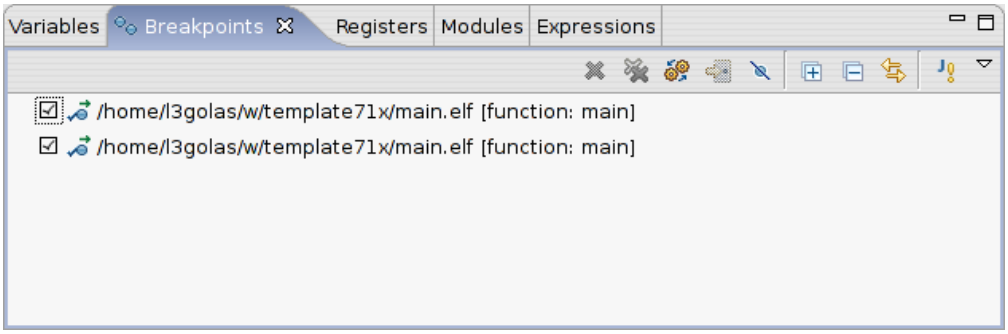
Through this view you can create a list of variables you want to monitor, so it's really useful for programs with many variables, especially if the programmer is interested only in a subset of them. You have to enable the view as we explained before, then right click in it and select *Add Watch Expression*. In the window that will appear insert the value of the variable to control, for example we chose to monitor the variables *a* and *c*.



You can also write complex expressions, given by more variables combination, for example *(a+b)/c*. Anyway, this view is really useful if you want to monitor complex structures elements, such as an array of structures element, or a variable pointed by double or triple pointers. For those elements in fact other views are inadequate or uncomfortable when used, in that they make you see all the elements of the structure and search among all them the element that interests us, while this view allows you to directly write the expression to control and to see only it in a simple and quick way.

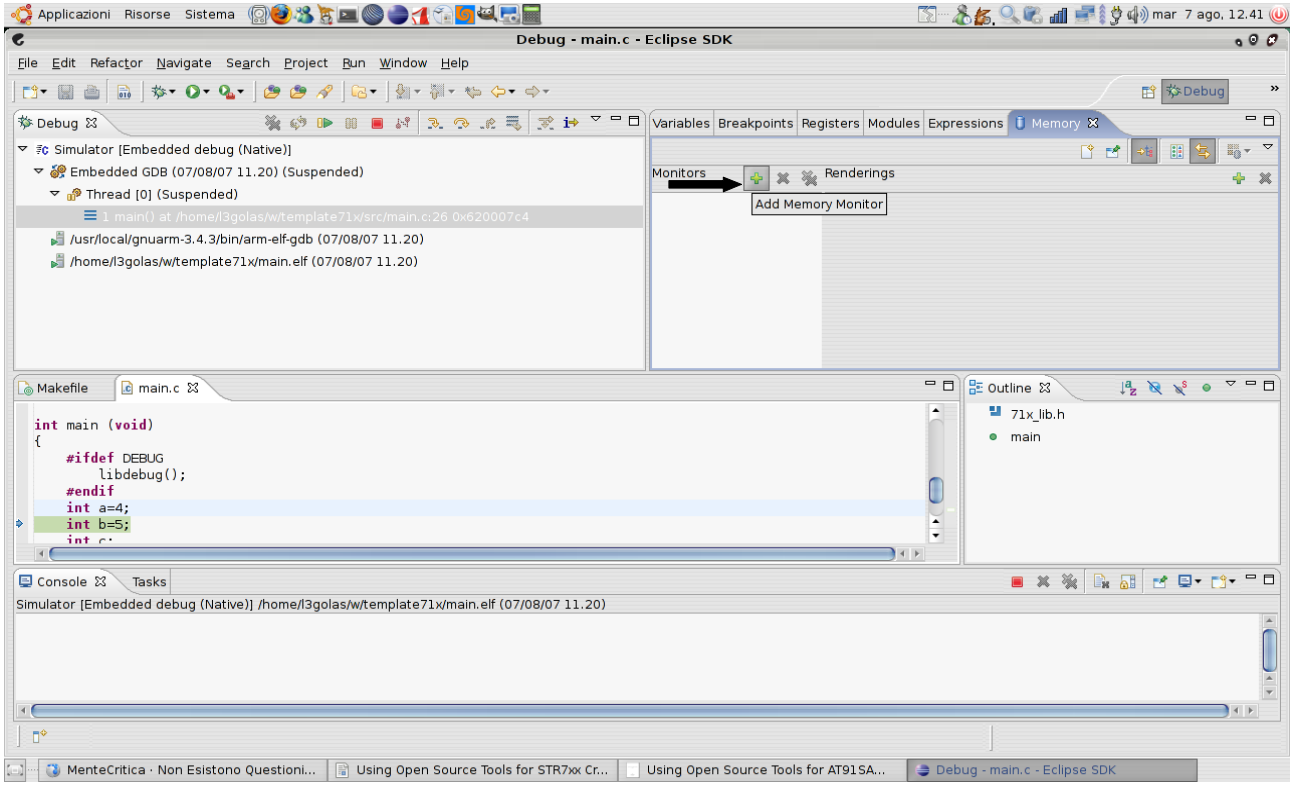
Breakpoints

Through this view you can see all the features of the breakpoints you set into the program. It's also possible to deactivate some of them temporarily.

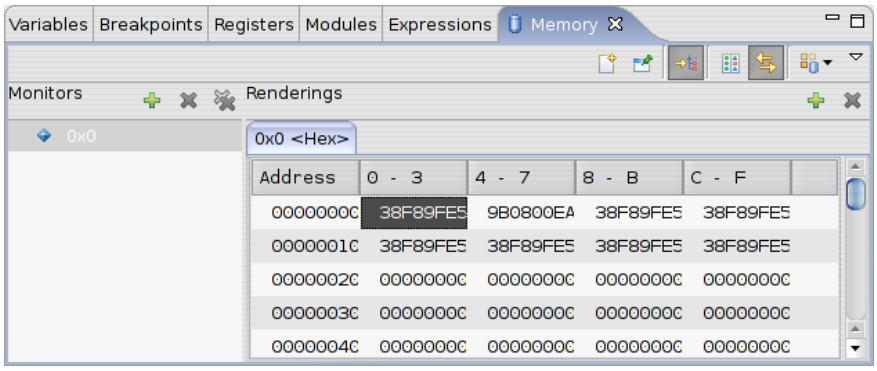


Memory

It's possible, using the view *Memory*, to see the memory content, expressed in hexadecimal. Open the view and in it click on the symbol + indicated in the following picture by an arrow.



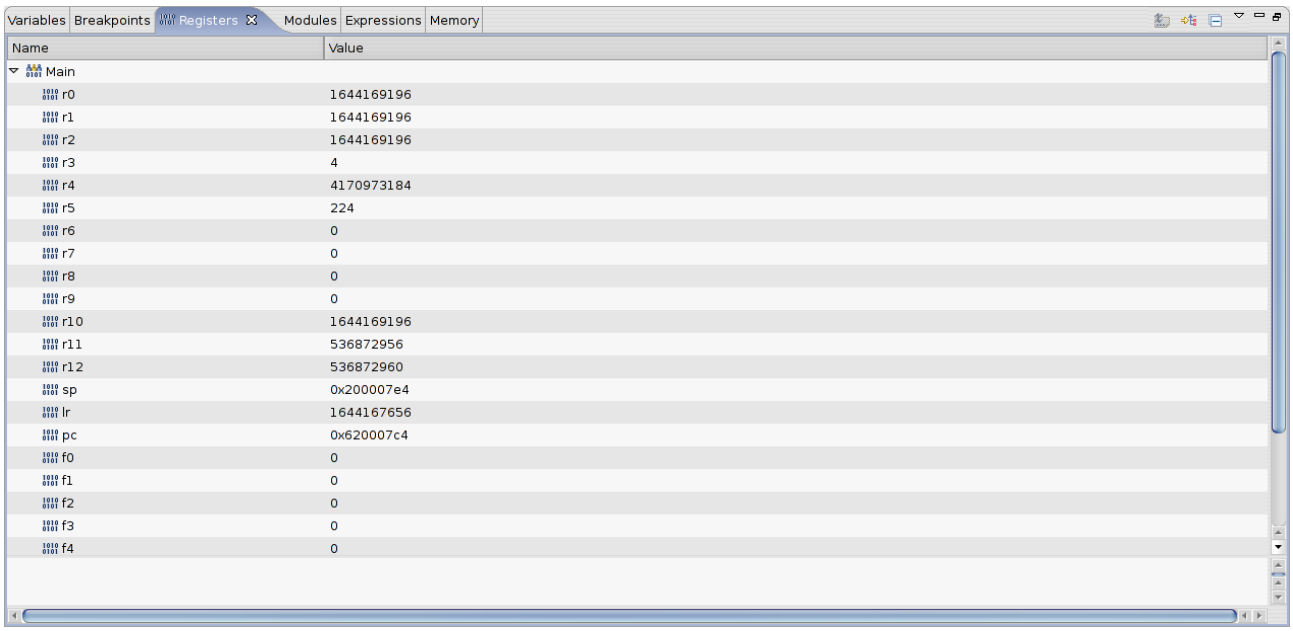
In the window that will appear, insert the memory address to monitor, for example *0x0*. The result is this:



As you can see, both the addresses and the words contents are expressed in hexadecimal; you can also choose another format to see them: it's sufficient to click on the + present on the right (*Add Rendering*).

Registers

It's also possible to monitor the registers of your microcontroller by selecting the proper view. In it the registers with their values will appear, you can also show those values in a different format and modify them:


Variables
Breakpoints
Registers
Modules
Expressions
Memory
Name
Value
Main
r0 1644169196
r1 1644169196
r2 1644169196
r3 4
r4 4170973184
r5 224
r6 0
r7 0
r8 0
r9 0
r10 1644169196
r11 536872956
r12 536872960
sp 0x200007e4
lr 1644167656
pc 0x620007c4
f0 0
f1 0
f2 0
f3 0
f4 0

Appendix B – Some GCC features: optimizations, thumb mode

B.1 Optimizations

The optimizations are one of the most advanced GCC features and they have been introduced to limit compilation time, or memory usage, or to reach a trade off between execution speed and space occupied for the executable generated by the compilation (in fact optimizations usually improve an aspect by worsening another one, so it's necessary to reach a trade off, depending on what you care more).
There are different levels of optimization, that are obtained by the option **-O*n*** with 0<n<3 or **-Os**. Let's see them in detail:

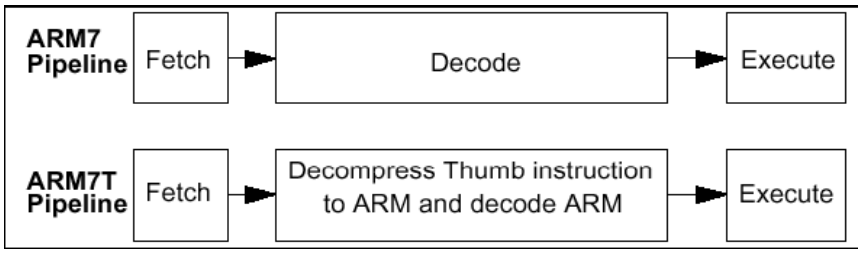
- **-O0** is the default choose, therefore it's automatically enabled if you specify none of these options. It indicated "no optimizations", so each code line will be converted in the corresponding machine instructions without modifying or moving anything. Surely it's the best choice for debugging.
- **-O1** or **-O** enables the most common optimizations, that, since they are still "light" optimizations, are able to improve both speed and space better. So the executables will be both quicker and smaller of the ones generated with the previous option.
- **-O2** enables the optimizations of the previous option, to which it adds the *instruction scheduling*, without worsening the executable final dimensions. The price to pay is a slower compilation process, however it's the best method to develop a software, in fact it's applied to the GNU packages.
- **-O3** enables more optimizations (such as the "*function inlining*") that make the executable be quicker but also bigger in dimensions.
- **-Os** enables space optimizations, even if this worsens speed.

Thus, **-O0** is the best option for debugging a program, in fact it's the one we use in the templates and in the examples we created. However, when you know the program works correctly and debug is not anymore necessary, you can think about changing option, for example using **-O2** (which is a good trade off among the optimizations) or ***-Os***, also because in embedded systems it's very important to respect space specifications (sometimes it happens it's not possible to run a program in RAM because it doesn't fit it and you have so to download and run it in FLASH, in that case **-Os** could be the right option and it may sometimes solve the problem).

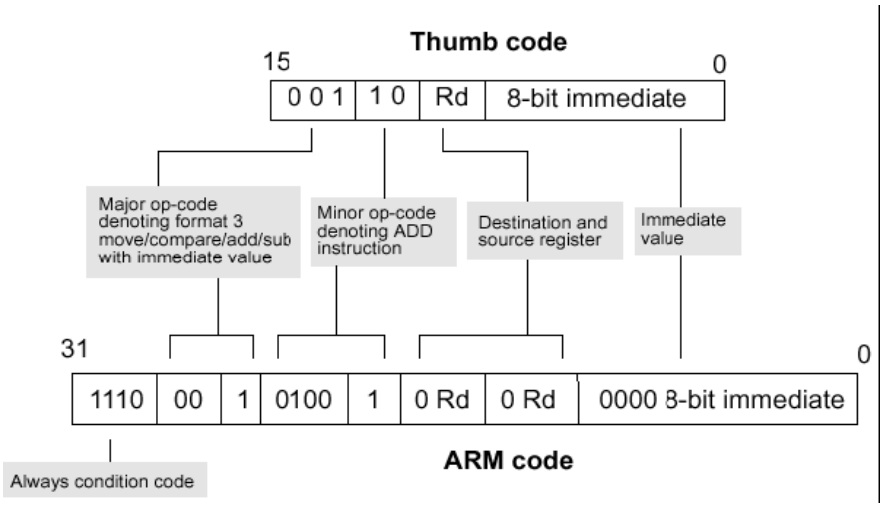
B.2 Thumb mode

The compile time optimizations surely improve some aspects (for example the option *-Os* is useful to get code dimensions smaller), but it's not always possible to apply them and often the improvements are limited. Since embedded systems care code dimensions very much, they looked elsewhere to get their bout. For example another way to get code dimensions smaller is the so called "*thumb mode*", which allows the "code compression": in few words this method introduces an alternative instruction set which does the same operations than the standard ARM one, but it's a 16-bit instruction set instead of the original 32-bit one. But how we can make an instruction smaller? We can get that by making a Thumb instruction use only the first 8 GPRs (General Purpose

Registers) instead of 15 (only few instructions, such as MOV o CMP, can use all the registers). In the following picture you can see the classical *fetch-decode-execute* pipeline and the one used by the Thumb mode:



To be used properly, a THUMB instruction must be converted from the CPU at runtime in the correspondent 32-bit ARM instruction, so it's necessary the CPU is "*Thumb aware*", that is it has a Thumb decompressor in the pipeline.
You can see how a ADD instruction is converted from Thumb to ARM mode:



So what are the advantages in using Thumb instructions? You can save at least 35-40% of memory: if you use *Thumb mode* you have to use more instructions than the ARM mode, but each one occupies half! The consequence is very simple: it's sufficient less memory to contain the same number of instructions, or equivalently we can say more instructions can be contained in the same memory.
The downside is that before executing an instruction, it's necessary to convert it in the correspondent ARM one, but the performance loss is not significant if compared to the space saved.
Anyway, it's possible to use the mode called "*interworking*", through that you can have together Thumb and ARM code: that way you can decide for each subroutine if it's more important for you to save space or to have best performance: for example, for device drivers or exception management (which require best performance), you could use ARM mode (renouncing of course at saving space), while most code could use Thumb mode.
For example, to use interworking mode in GCC, you have to set it by adding the option *-mthumb-interwork*.
But how does the CPU know if a function must be executed in ARM or Thumb mode? When you call a function, a "*branch*" takes place, that is an instruction of the type *BX Rx* (Branch and eXchange) is executed, where *Rx* is the address where the code to execute starts. Now the CPU checks the bit 0 of *Rx*: if it has been setted to 1 it uses Thumb mode, otherwise it uses ARM mode.

Appendix C – Test results with a STR9 board

As we said in the introduction, we decided to extend our work by starting working on the STR9 microcontroller family, based on the core ARM966E-S. We decided this way because on the one hand we liked the work we did on the STR7 serie, on the other we were curious to know if this solution could work properly for STR9 serie, too. So we started working with a board based on the microcontroller STR912, we created a template that uses the same model than the ones for the STR7 serie, we downloaded the proper configuration files for OpenOCD and we created the GDB startup files for debugging. The results are positive, we managed to compile and debug a program both in RAM and in FLASH, so the FLASH programming works properly, too. The program we tested was the classical code which switches on and off some LEDs, but we think more complicated programs should work, too. So we put, among the other files, also the files about the STR9 serie: in the folder *openocd-configs* you can find the OpenOCD configuration files and the *gdb* files for the various JTAG hardware interfaces; in the folder *ARMProjects* there's the subfolder *STR9* which follows the same model as the one of the subfolder STR7 with which we already dealt in Chapter 3.
However, we want to remember you that work on STR9 is only at a preliminary stage and so other tests are required to make it work really properly.
Now we go more in detail on the various aspects of the solution we used for STR9.

C.1 Board

We used the board *STR910-EVAL*, that has a microcontroller STR912F. These are the main features:

- Three 5V power supply options: jack, USB connection or a daughter board
- RTC with tamper detection
- Audio play and record
- 3 RS232 connectors with support of full modem control on one connector
- Infrared Data Access (IrDA)
- USB 2.0 compliant with full-speed (12 Mb/s) data transmission
- CAN 2.0B connection
- Inductor Motor Control connector with 6 PWM output, Emergency Stop and Tachometer input
- IEEE-802.3-2002 compliant Ethernet connection
- 38-pin ETM connector for optional connection to STR9 trace module
- Dot-matrix LCD module
- Joystick with 4-direction control and selector
- Extension connectors for daughter board or wrapping board

C.2 Template

The template follows the same model than the ones already seen for the serie STR7 (both for the

Makefile and for the startup and linker files), so you can completely follow what is there specified. However you have to pay attention to one thing, that is in the *Makefile*: we added the option *--no-warn-mismach* to the linker options (line 214). This option solves a problem at compile time, due to the fact that the library *libgcc.a* (required for linking) and our program didn't use the same type of FP (floating point) instructions. We didn't study more this problem, probably there are better solutions, but doing this way the problem seems not to be anymore present.

C.3 cfg and ocd files

The *cfg* file we used with OpenOCD to debug our programs has been downloaded directly from the site http://openfacts.berlios.de/index-en.phtml?title=Open_On-Chip_Debugger, at the section *OpenOCD scripts*. As you know, you have different versions depending on the different JTAG hardware interfaces you could use, but we tested only the parallel port version. Then there are the other *cfg* files, used to call the *ocd* scripts to write and erase FLASH. Here you have the content of the file *str91x_flashprogram.ocd* used to write FLASH:

```
wait_halt
str9x flash_config 4 2 0 0x80000
flash protect 0 0 10 off
flash erase 0 0 10
flash write 0 main.bin 0
reset
shutdown
```

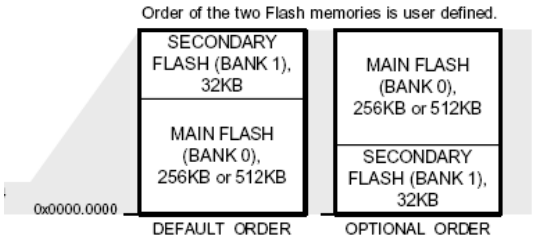
It's the model also used by the other microcontrollers, that is protection deactivation, FLASH erasing and then programming. The only addiction is the line *str9x flash_config 4 2 0 0x80000* whose task is to properly set FLASH before doing the other operations.

C.4 GDB startup files

This is the content of the file *str91x_ram.gdb*:

```
target remote localhost:3333
monitor reset
monitor sleep 500
monitor poll
monitor soft_reset_halt
monitor arm7_9 sw_bkpts enable
```

The FLASH version is different only for the hardware breakpoints instead of the software ones. Anyway, you can notice the absence of lines that set registers for the boot or that enable memory banks, while those lines were present in the STR7 microcontrollers. This is due to the fact that this architecture is different from the STR7 one. Looking in the reference manual you will find out there are 2 banks of FLASH (*Bank0* and *Bank1*), situated one after the other at address 0; you have to define which one is situated before and which one is after, because two different configurations are possible, as you can see in the following picture:
The first one (default configuration) is *Bank0* at address 0 and after it *Bank1*, the other one is to have *Bank1* at first. To choose between the two configurations, you have to properly set the FMI



registers. What does this configuration mean? The system boots from address 0, so setting the right order allows you to boot from one bank or from the other one. However we didn't study more this aspect so we did the boot only from *Bank0*, that in our case is 512 KB (while *Bank1* is only 32KB), so we didn't have to set anything, because it's the default configuration.

To debug properly the STR9 microcontroller, the latest version of GDB/Insight, that is the version 6.6, seems to be necessary. In fact, with previous versions, a bug is present when you try to debug from address 0, that is where FLASH in STR9 microcontrollers is located. If you have the latest YAGARTO version on Windows or the latest GNUARM version on Linux, you should already have the latest version of GDB/Insight, so no problem. If you use Linux and you don't have the latest version of GNUARM, you can download only Insight 6.6 from www.gnuarm.org, compile and install it on your GNUARM version, as explained in the violet box in paragraph 2.2.2.

About the authors

Giacomo Antonino Fazio is 24 years old, he got his 1st level degree in Computer Engineering at University of Catania in October 2005 and now he's studying to get the 2nd level degree. He defines himself as a curious person, in fact he has many hobbies: at first computers, then music, art, literature, philosophy and much other. On-line he's often known as “l3golas” and his e-mail address is giacomofazio@gmail.com

Antonio Daniele Nasca is 26 years old, he got his 1st level degree in Computer Engineering at University of Catania in July 2005 and now he's studying to get the 2nd level degree. He has many hobbies: volleyball, bodybuilding, computers and much other. His e-mail address is antodani.nasca@hotmail.it

Acknowledgements

A big thank you to professor Paolo Arena for the opportunity he gave us.
Many thanks to Eng. Davide Lombardo for the support he offered us during all this time (about 5 months) and for his precious help for the English translation.
Many thanks to ST Microelectronics for lending us the boards on which we tested our configurations.
We also want to thank Spencer Oliver of Anglia, Jim Lynch for his excellent tutorial from which we took inspiration, Martin Thomas for his good work on Makefiles and on one example we used, Michael Fischer for YAGARTO, Dominic Rath for having created OpenOCD and for the support he gave us on the OpenOCD forum (present at http://forum.sparkfun.com) and all the people we forgot to quote, we hope they don't mind about it.