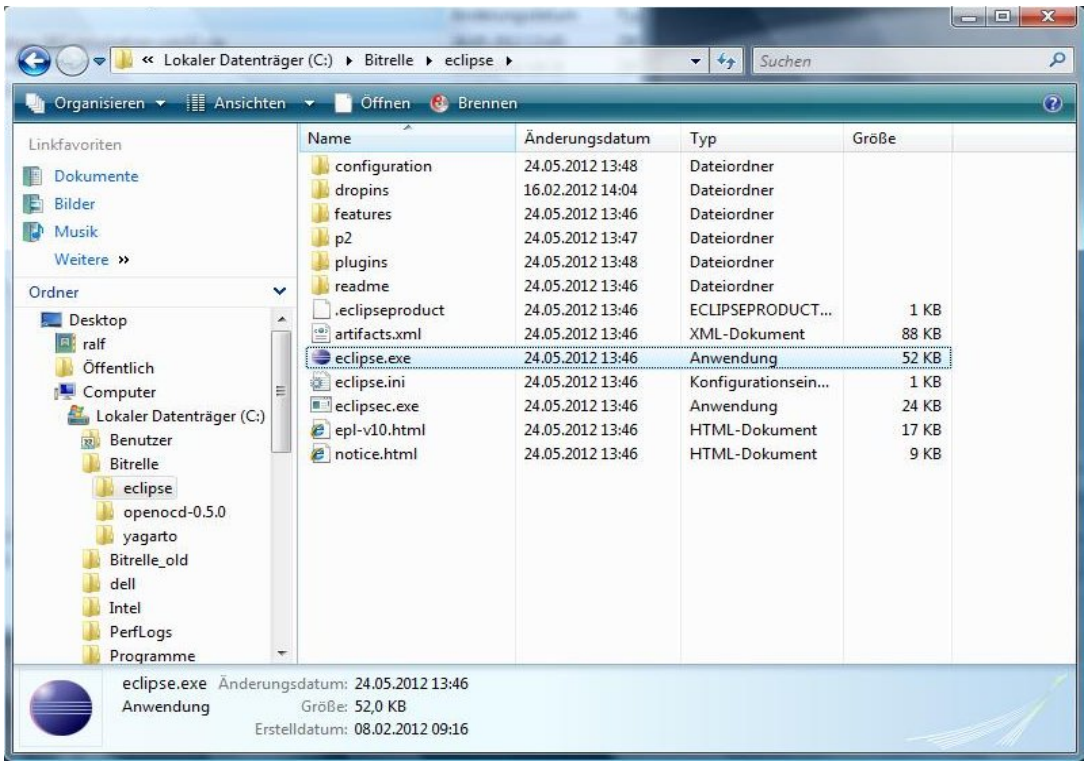
Bitrelle LPC2129 Module
Bitrelle LPC2368 Module

Programming with Eclipse
Manual

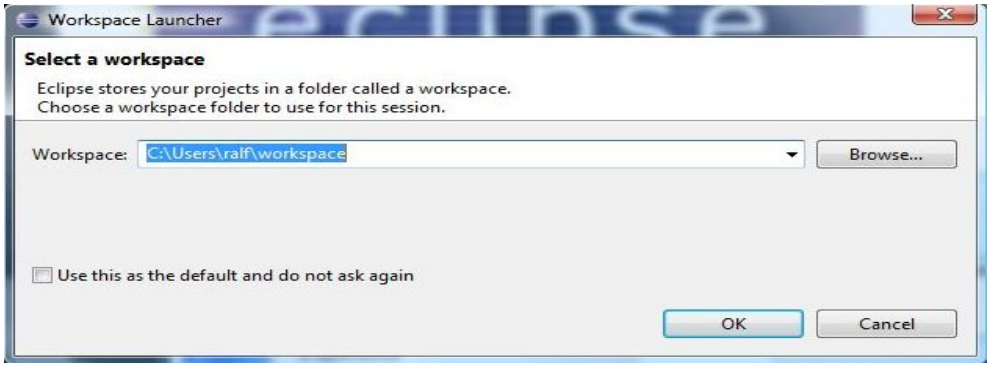
Table of Contents

1 Beginning new Projects

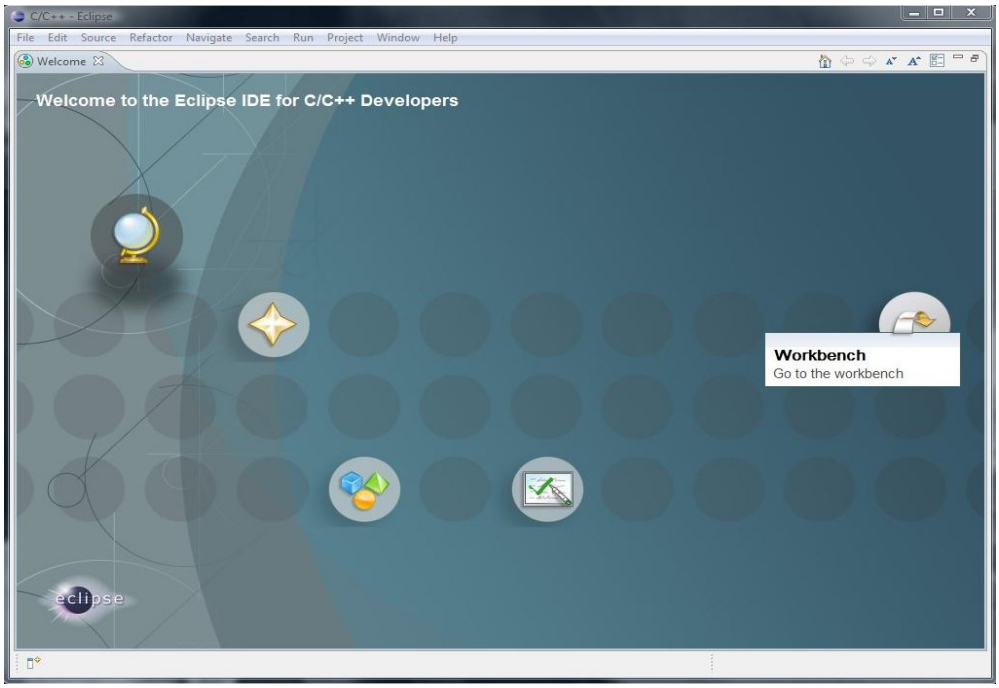
In this chapter we will shortly learn how to begin new projects in Eclipse. First start Eclipse by clicking the shortcut on Your desktop or Start menu or by double clicking the eclipse.exe file in the Eclipse folder:



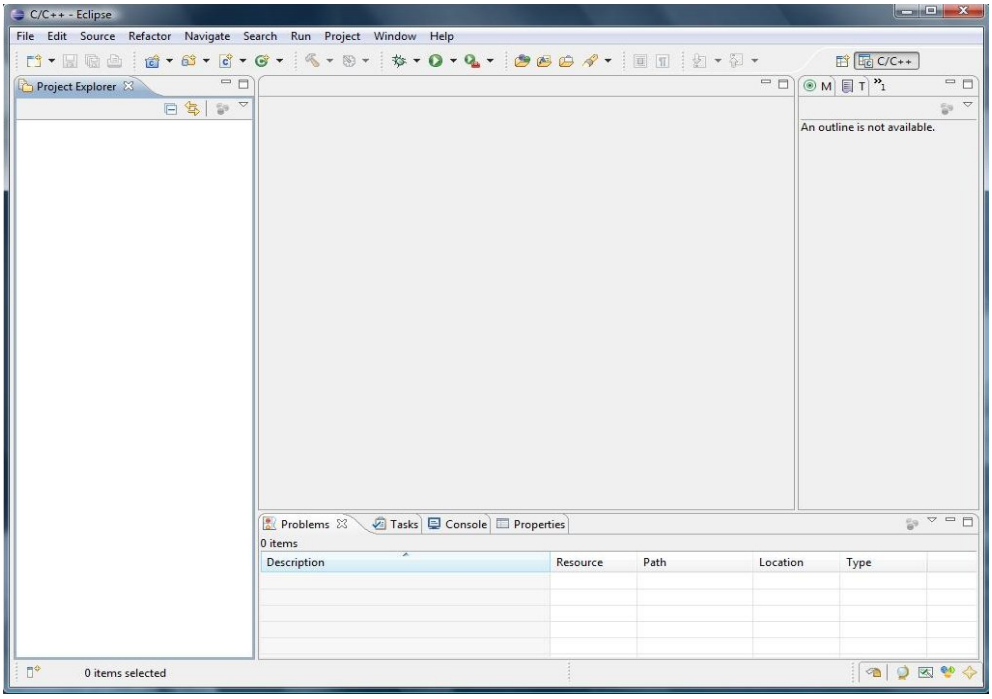
A menu will appear, where You can choose the workspace. It is fully sufficient to use the default workspace at this moment:



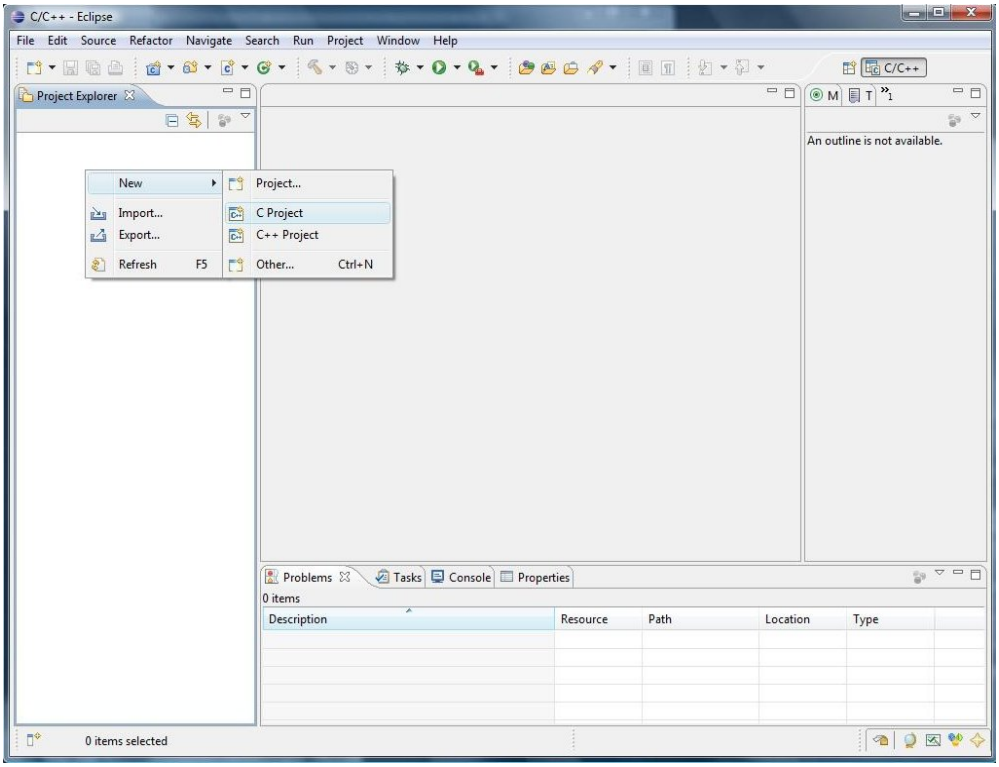
If You start Eclipse for the first time, the welcome screen will appear, click on the Workbench button on the right side:



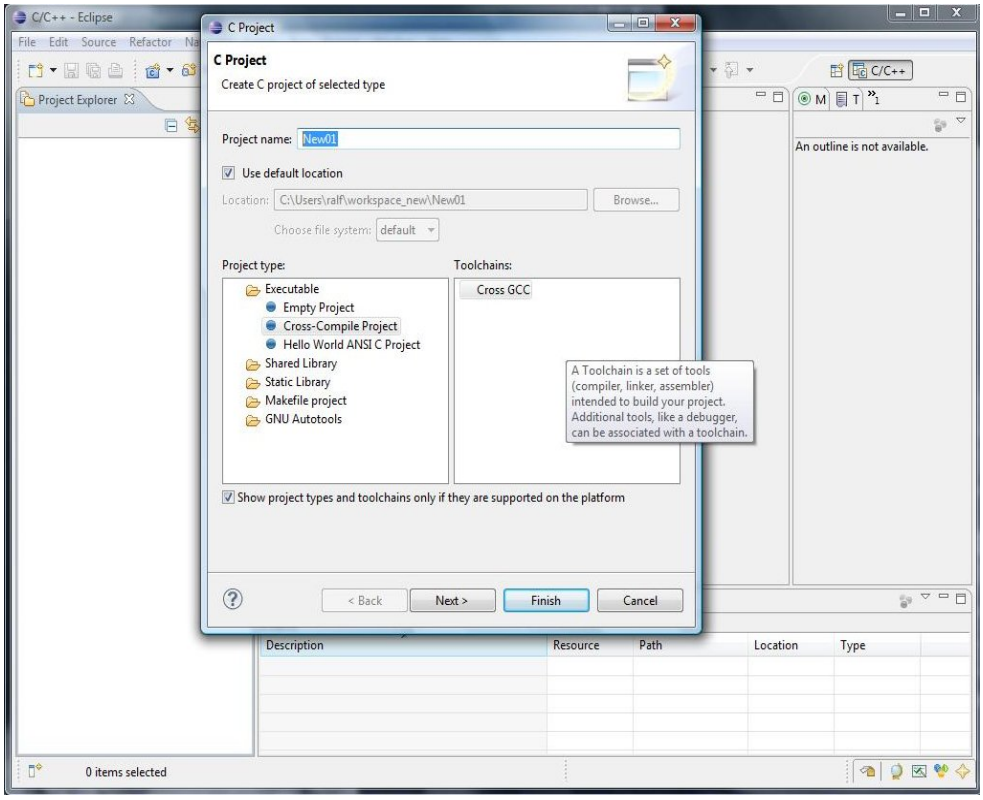
The Eclipse workbench opens, if You didn't begin a project so far You see an empty Project Explorer on the left side:



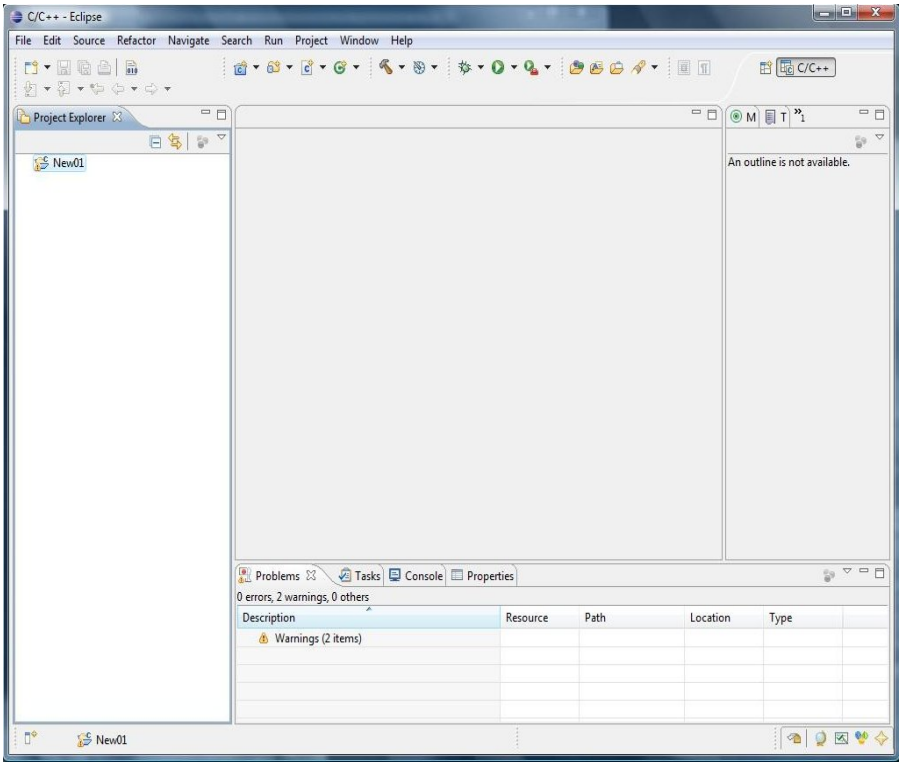
Now open the context menu of the Project Explorer by clicking the right mouse button on the Project Explorer field and select 'New' -> 'C Project':



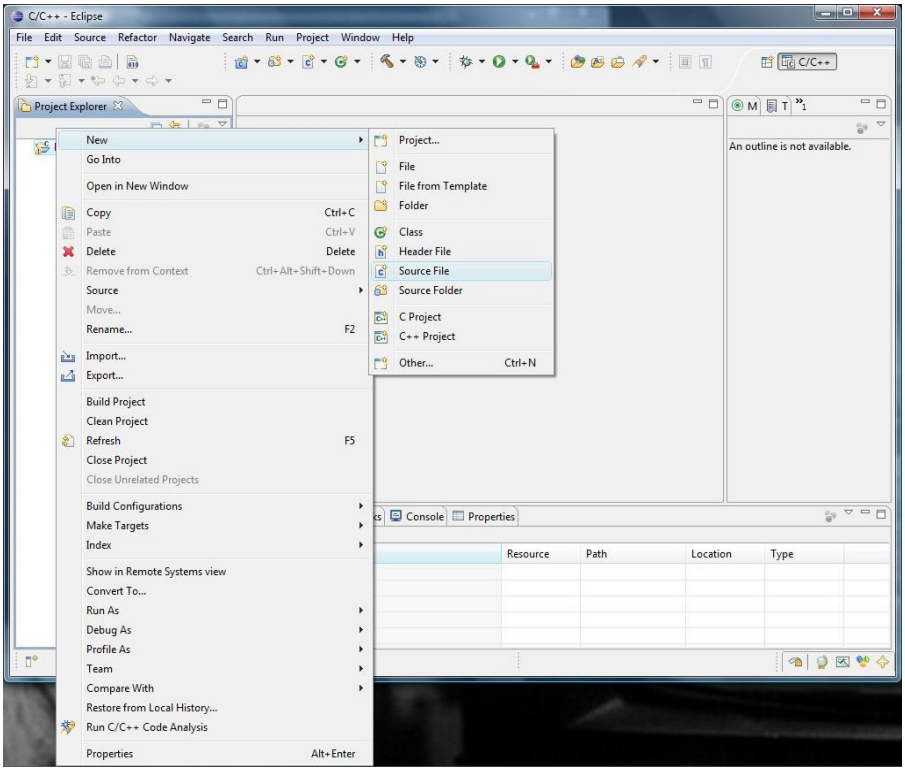
Type a project name, e.g. 'New01' in the C Project dialogue and click the Finish button:



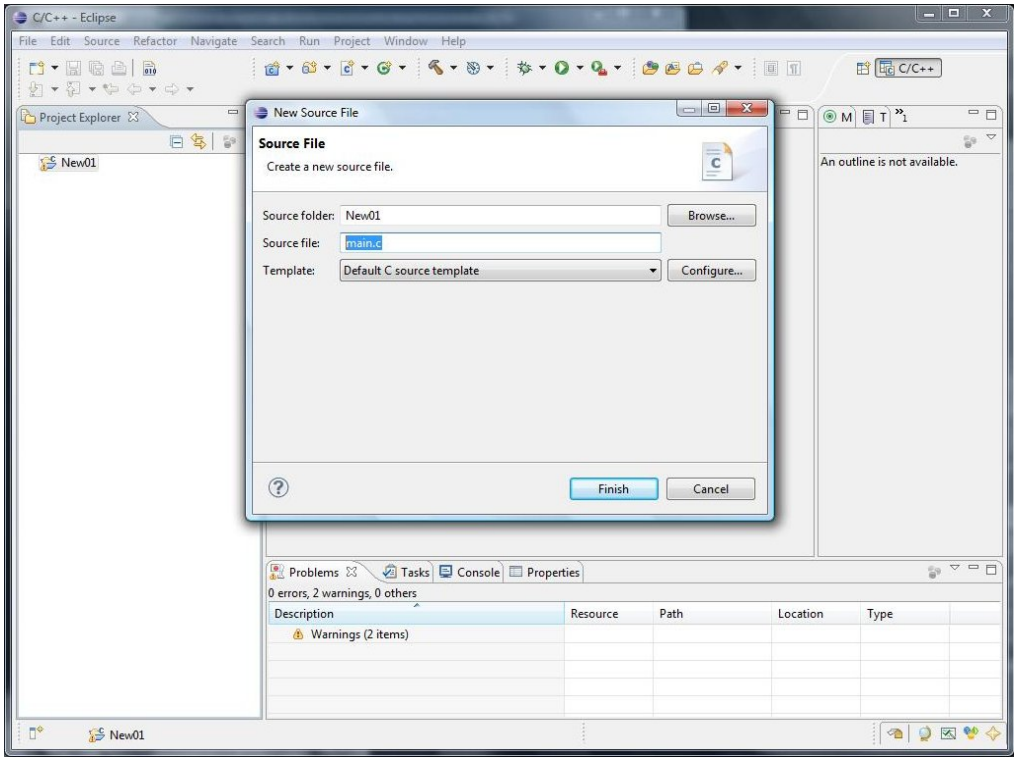
Our new project appears in the project explorer:



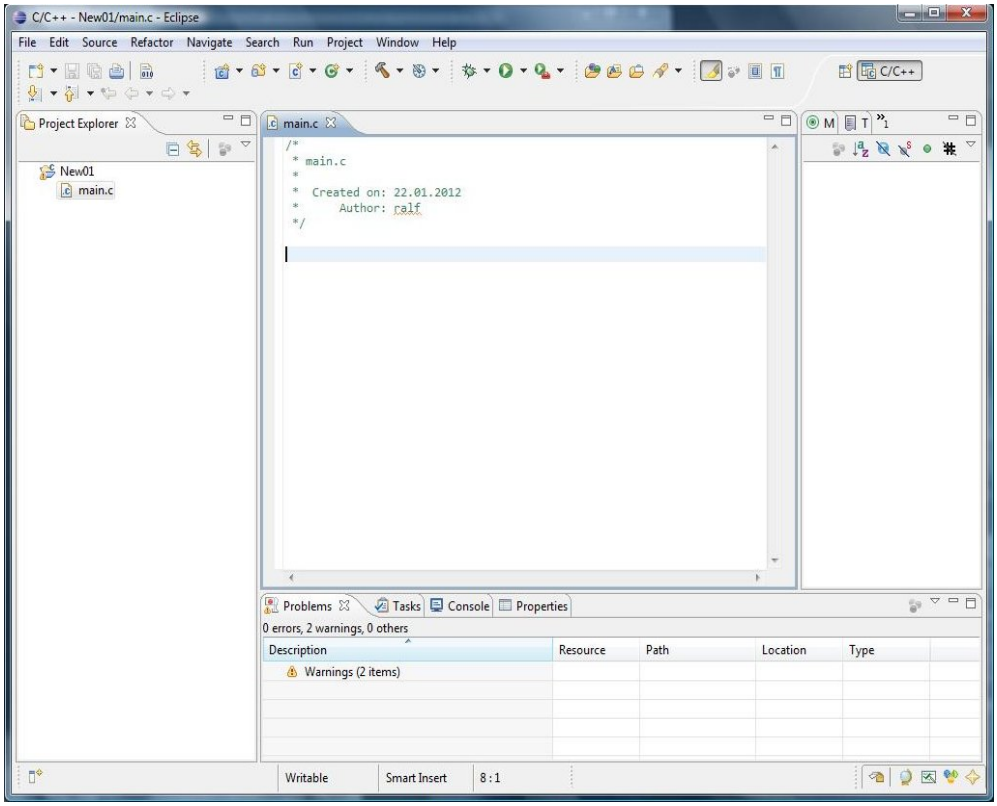
Click with the right mouse button on the new project and select 'New' -> 'Source File' in the context menu:



Add the new source file, in our case this is 'main.c' and click the Finish button:



The new C file appears in our project. You can add as many C files and header files as You need. You can also give Your project a customized directory structure by selecting 'New' -> 'Folder' when clicking the context menu of the project:



Now, You made Your first experiences with Eclipse. At this time You can not do much more, because You need the Harware Abstraction Layer (HAL) for Your board to get access to the microcontroller peripherals, e.g. the LED's, the serial port or the USB bus. You will learn in Chapter 2 how to import our HAL, libraries and tutorials.

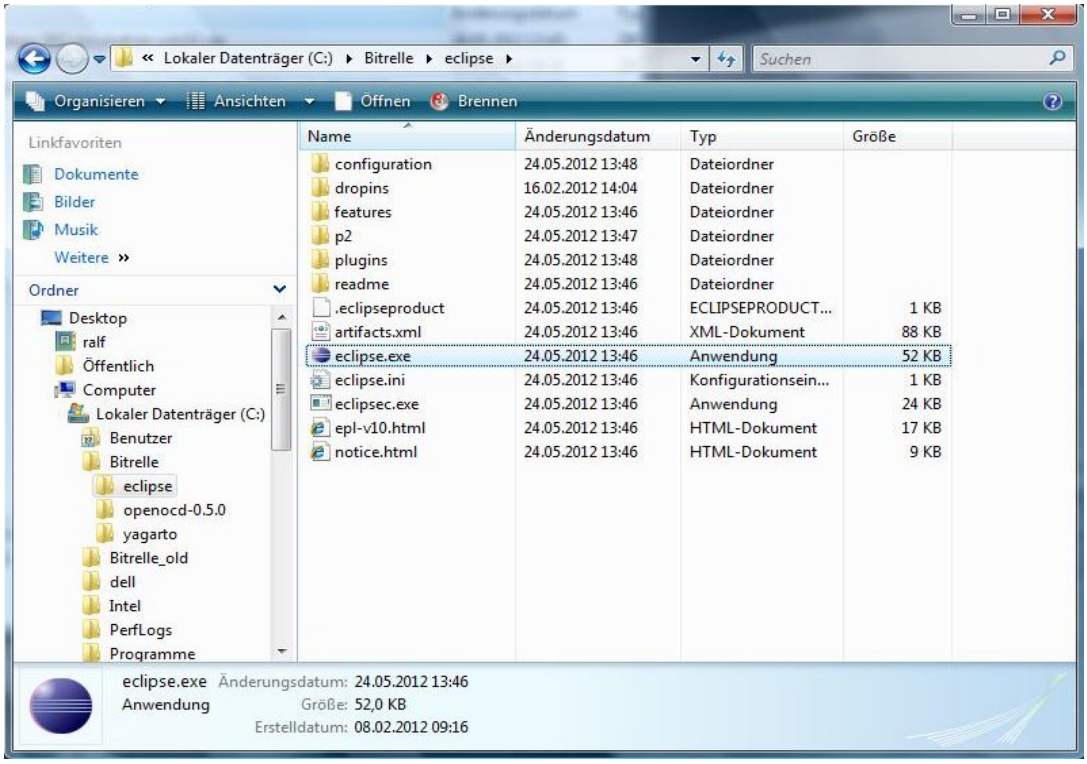
You must also set the correct Project Properties before You can build the new project. How to set the project properties will be explained in Chapter 3 Setting Project Properties.

In Chapter 4 we show You how to build a project and in Chapters 5 to 8 You will learn how to debug Your projects. Chapter 9 is a example how to flash a program into the Flash Memory.

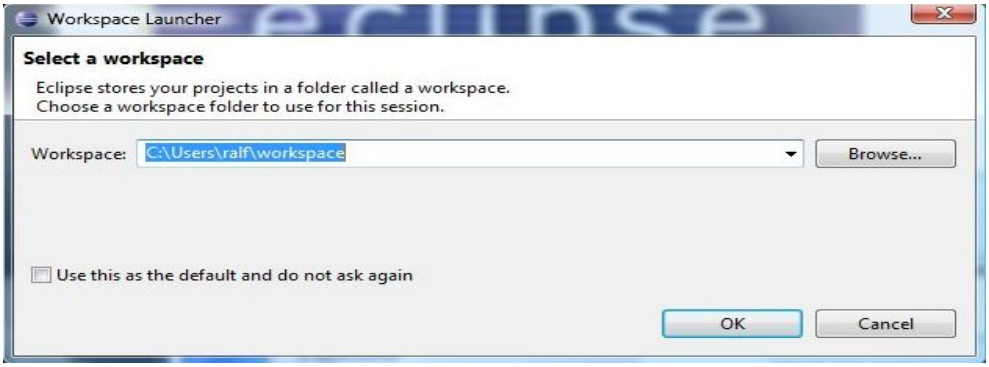
Annex 11 tells You shortly how to delete a project.

2 Importing Tutorials into Workspace

First start Eclipse by clicking the shortcut on Your desktop or Start menu or by double clicking the eclipse.exe file in the Eclipse folder:

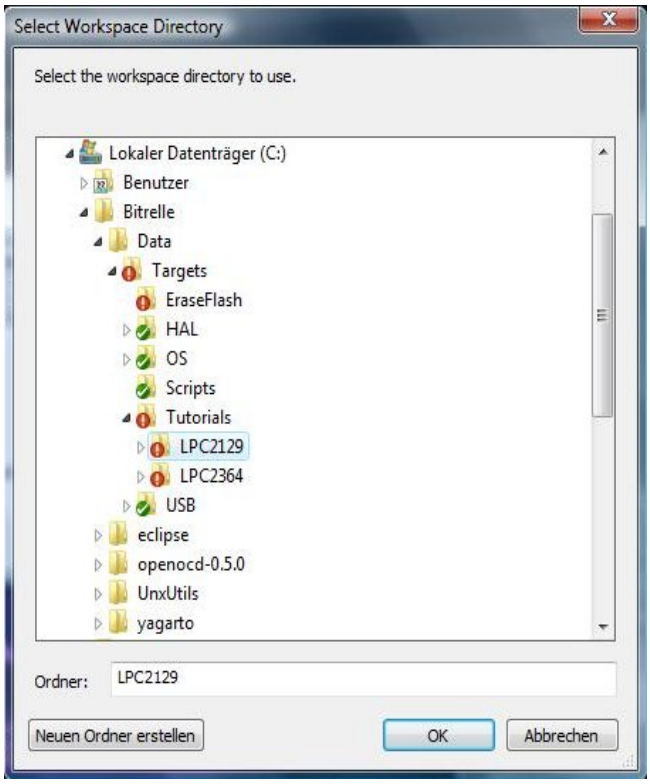


A menu will appear, where You can choose the workspace. Don't use the default workspace now, but go to the browse sub-menu by pressing the browse button, and..



..choose the path to the Turorials fitting to Your microcontroller board.

Read our Install Manual for Windows to install our tutorials and libraries to the correct position in the tree, if You did not, so far.

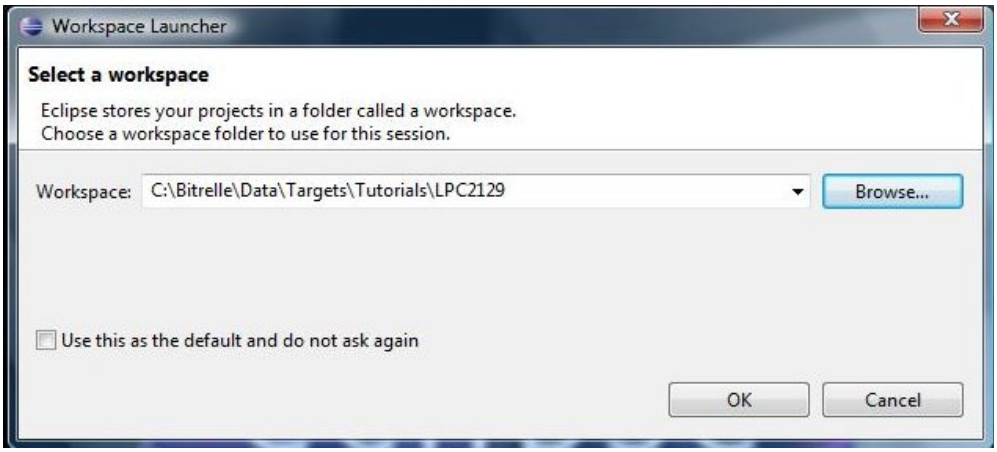


The select workspace menu should now have the following entries.

C:\Bitrelle\Data\Targets\Tutorials\LPC2129

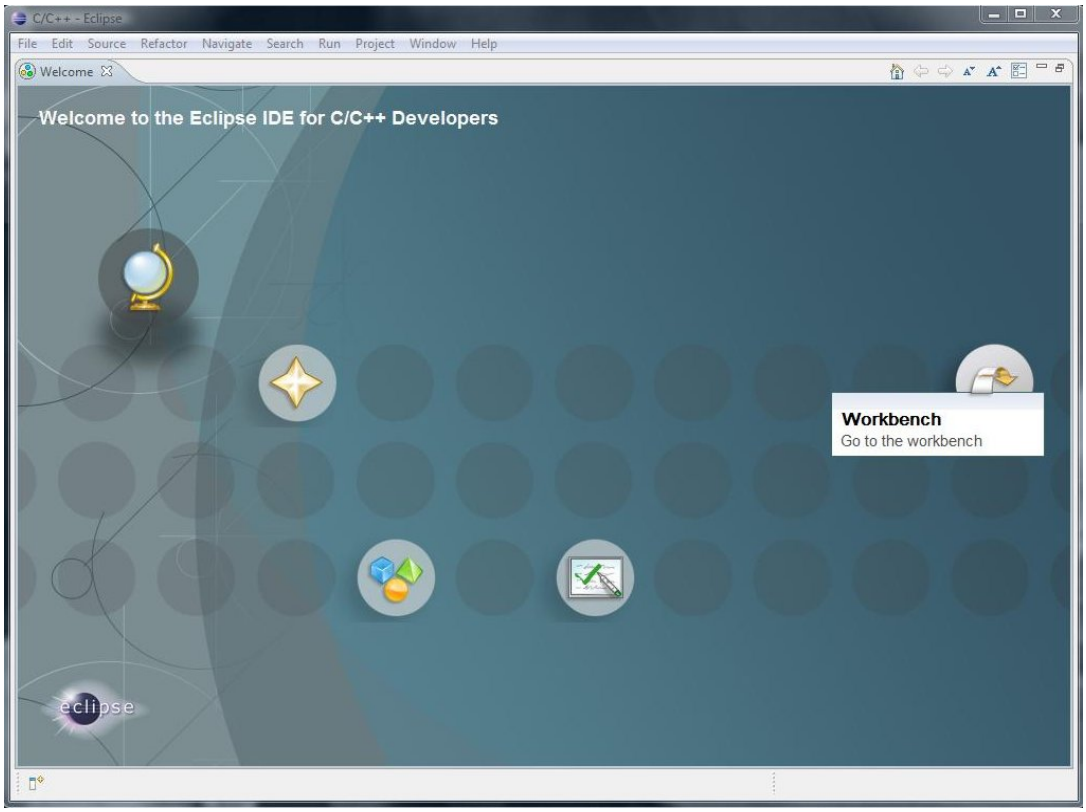
If You use the LPC2368 board it should be

C:\Bitrelle\Data\Targets\Tutorials\LPC2364

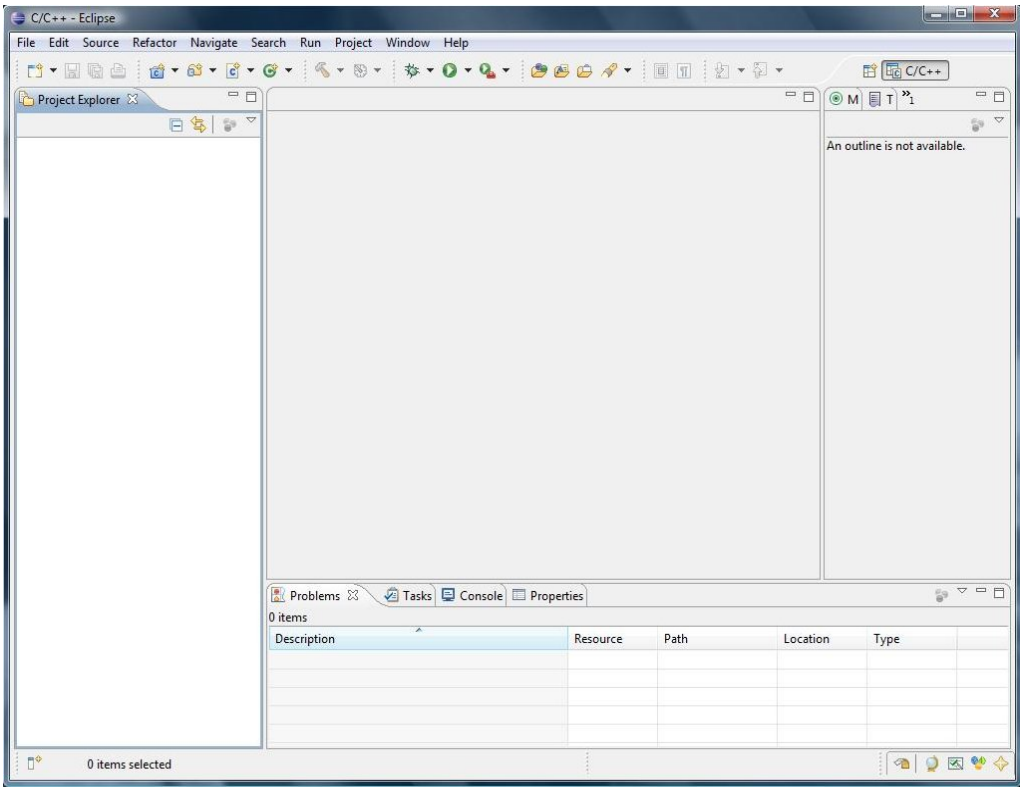


Press OK.

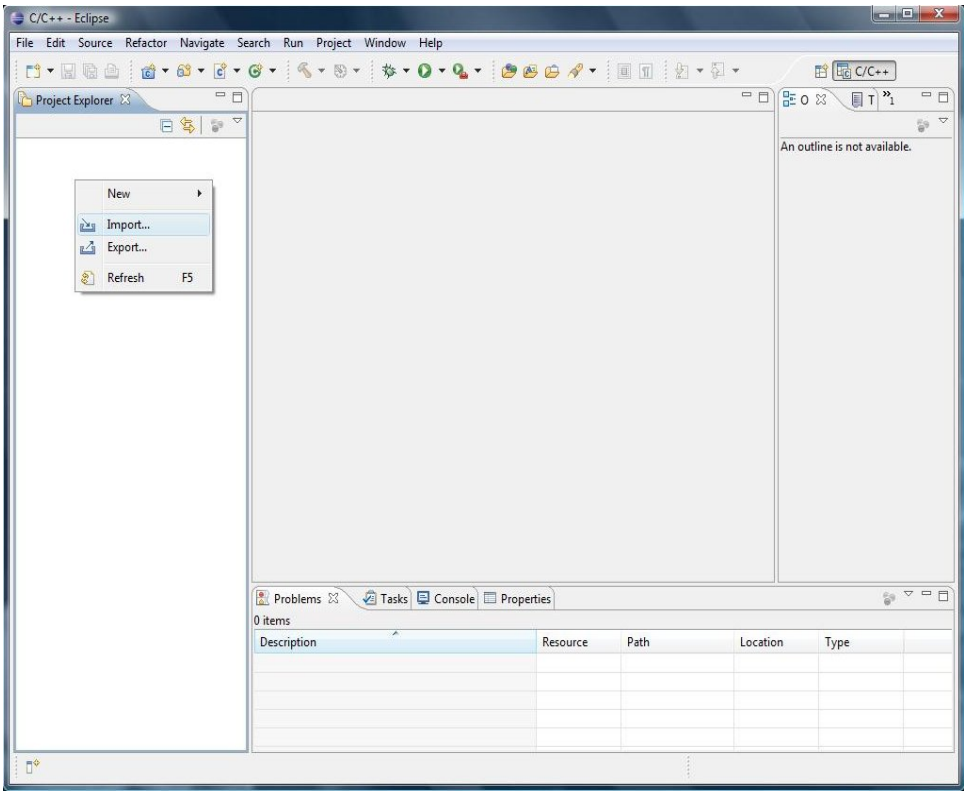
If You start Eclipse for the first time the Welcome Screen appears. Go to the workbench:



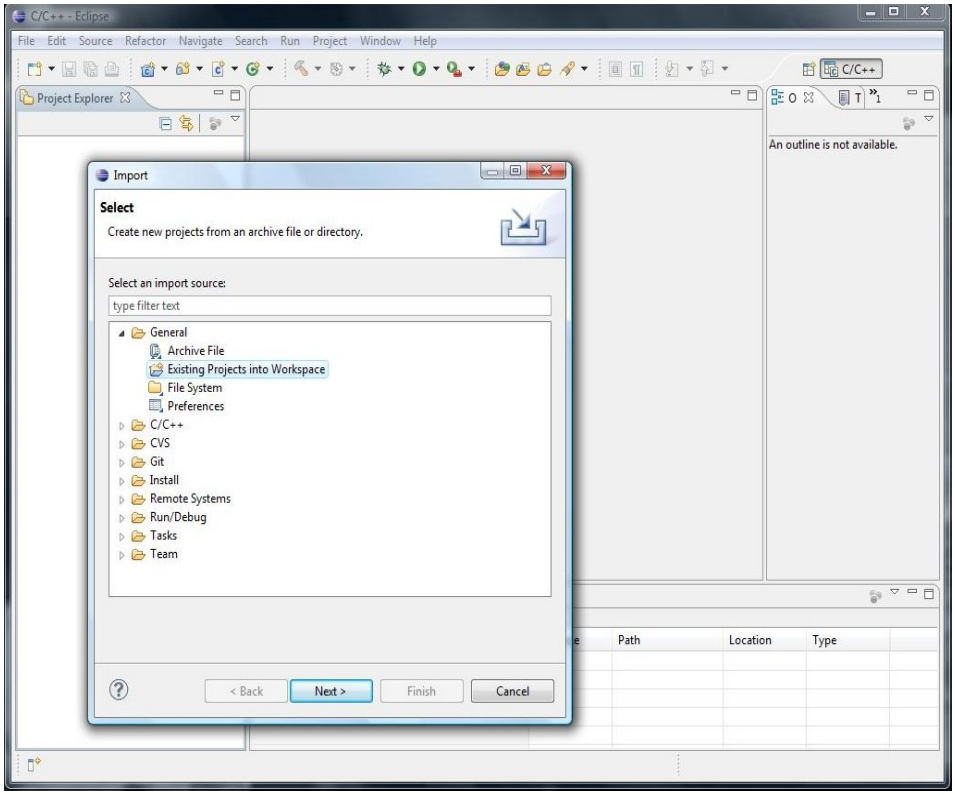
In our example Eclipse is starting with empty Project Explorer:



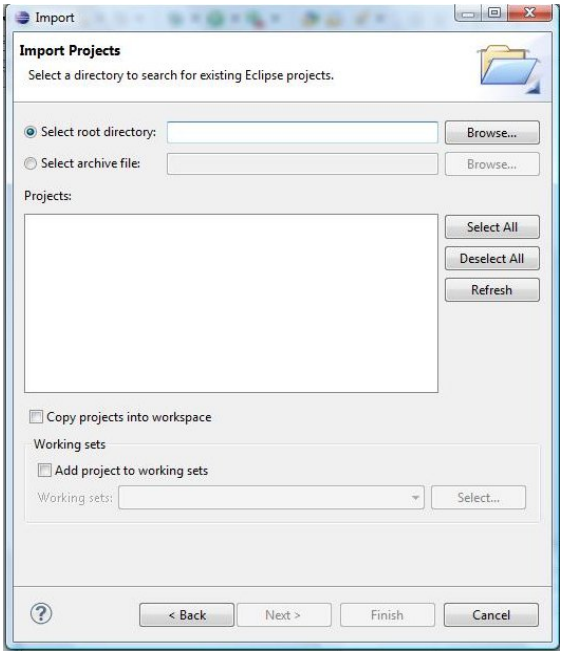
Click into the Project Explorer field with the right mouse button and select 'Import':



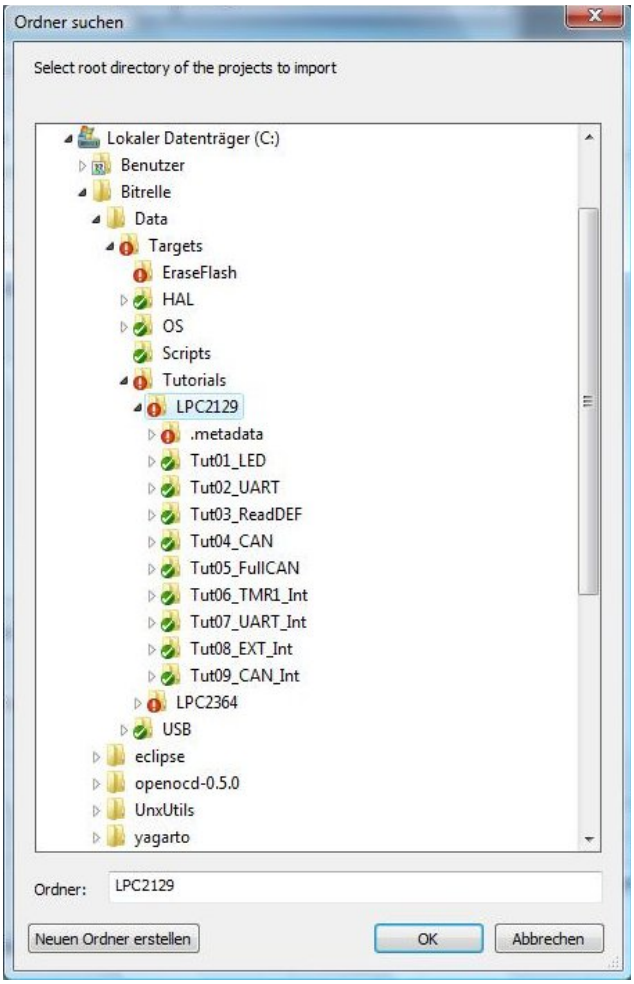
Select 'General' -> 'Existing Projects into Workspace':



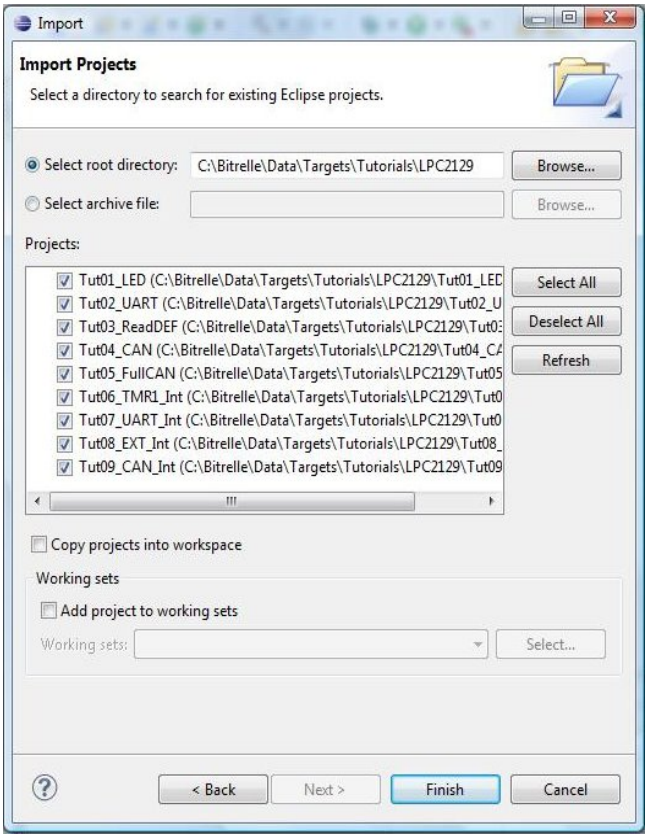
A menu opens where You can choose the projects to import:



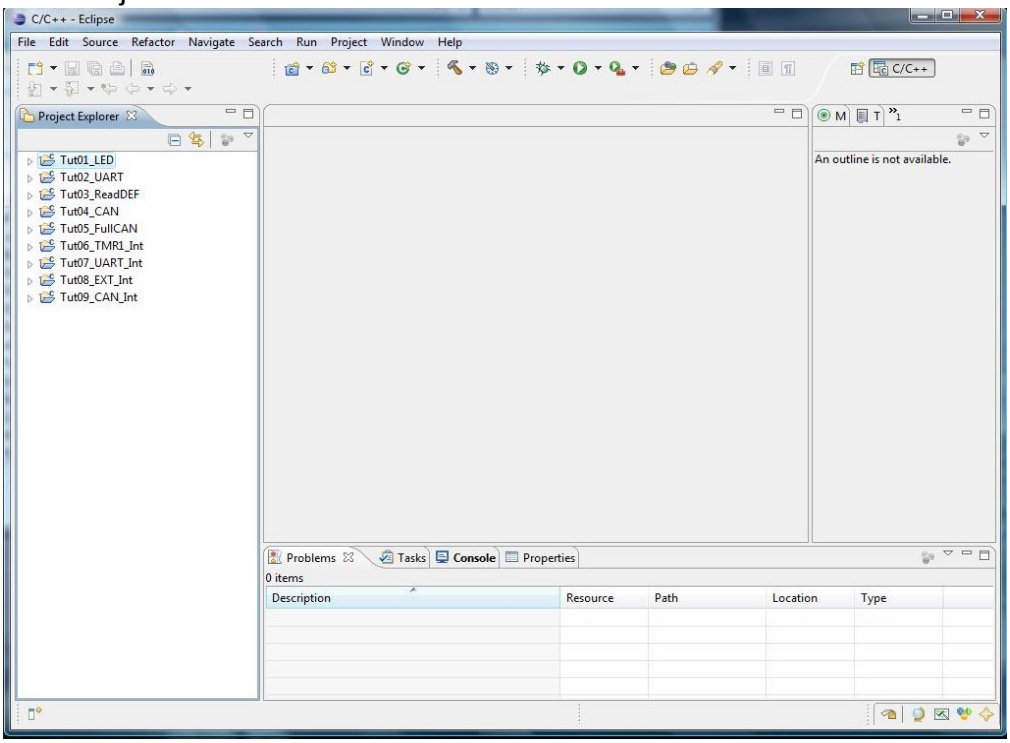
Now, browse the Tutorials folder and press OK:



Finish importing by clicking the Finish button:



Your Projects Sidebar should now look like this:



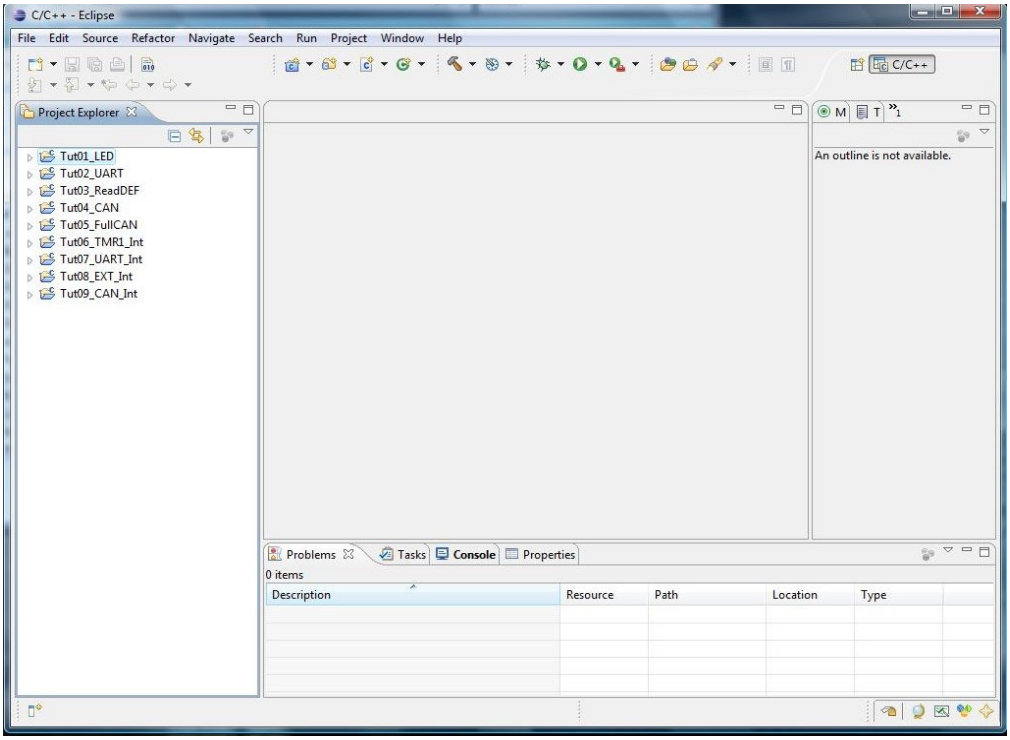
Before You can build the imported projects You must set the correct Project Properties. How to set the Project Properties will be explained in the next chapter.

3 Setting Project Properties

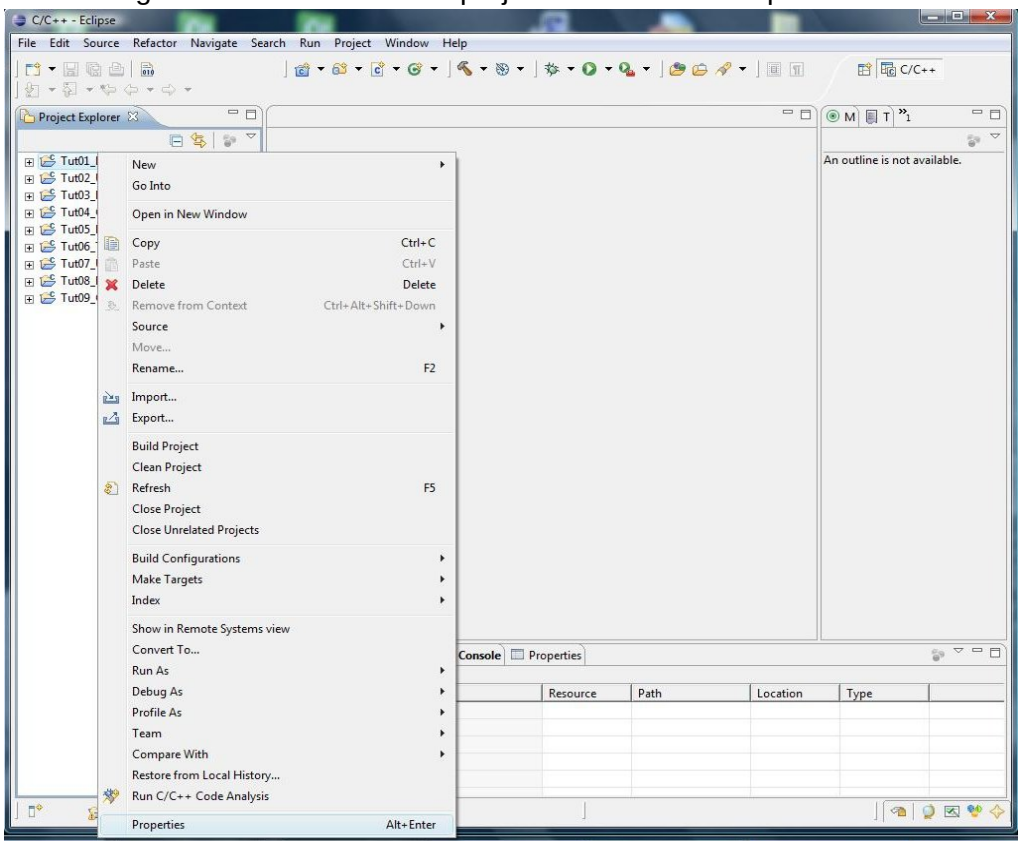
To be able to compile and link source code, You have to do several settings. First You have to choose a builder. This can be the Eclipse internal builder, or - as we use it - an external build tool, e.g. GNU make. Then You have to choose a configuration, binary and error parsers, include paths, source locations, output locations and some more settings.

All these settings are done in the Project Properties dialogue. This chapter will show You how to set up our tutorial 'Test01_LED' project as an example.

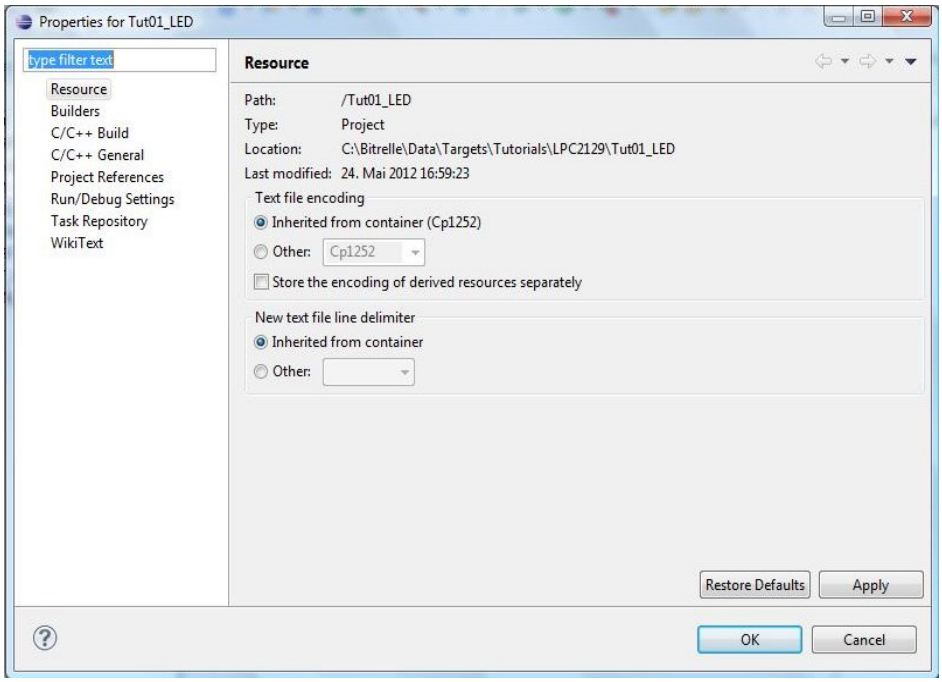
Your Projects Sidebar should now look like this, if not, go back to chapter 2 and learn how to import our tutorial projects:



Click the right mouse button on the project and choose 'Properties':

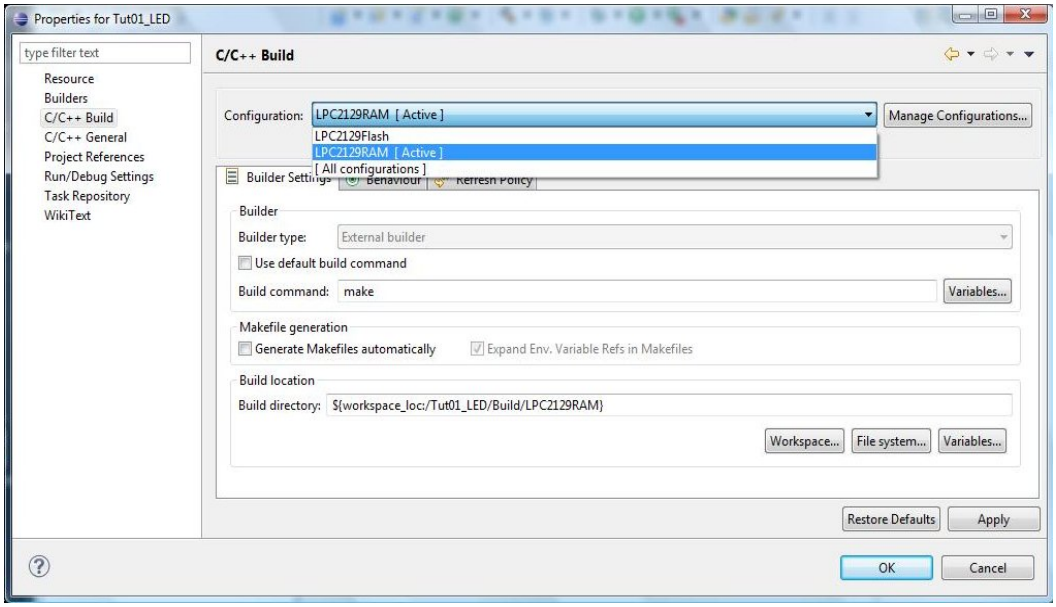


The Project Properties window opens:

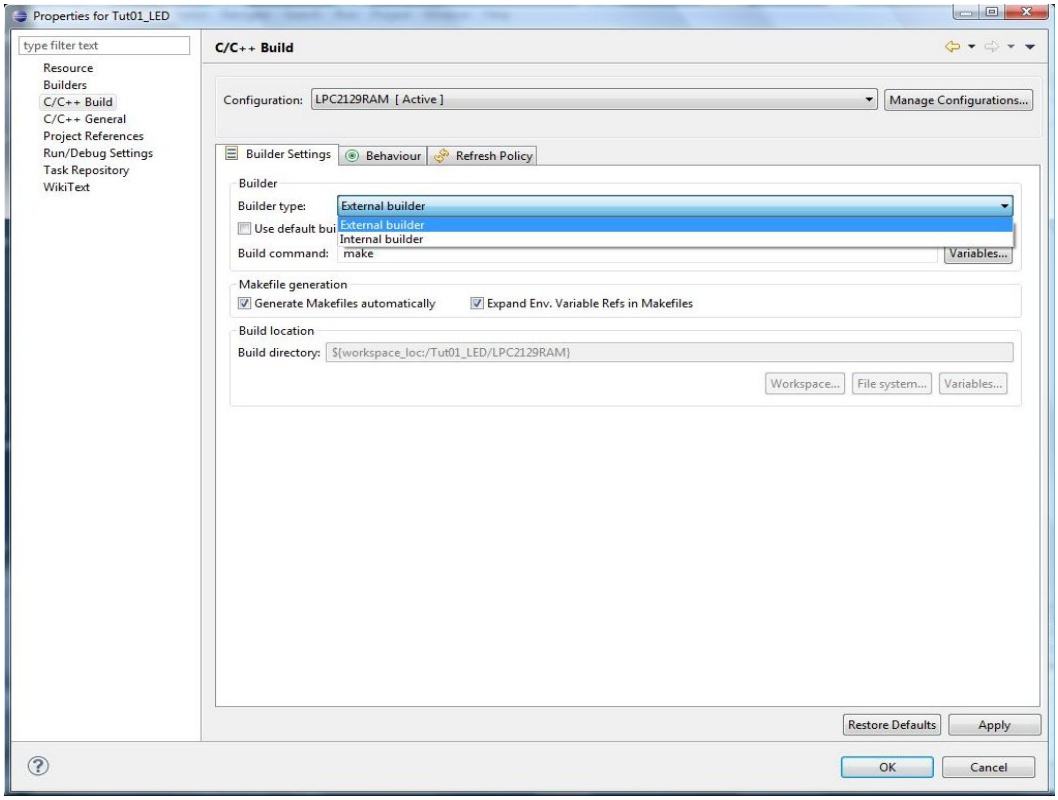


Go to 'C/C++ Build' menu and choose LPC2129RAM configuration. Set the configuration active with the Manage Configuration option, if it is not the active configuration.

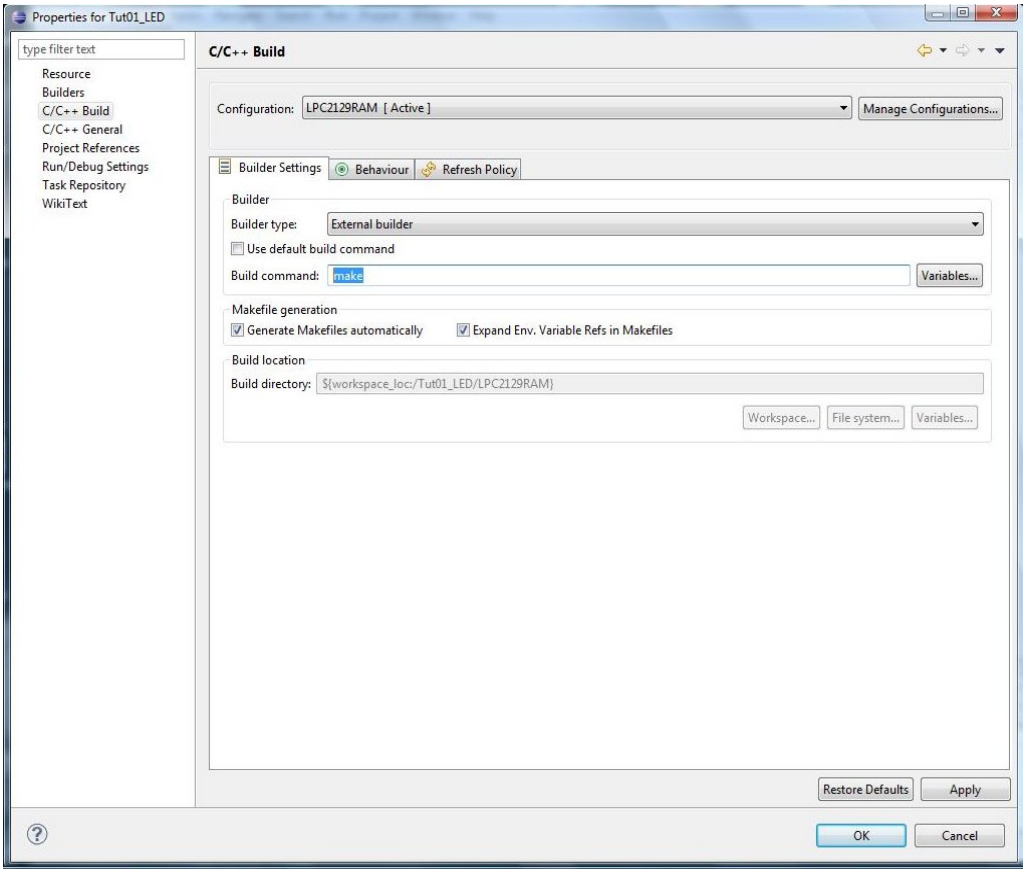
If You don't see the LPC2129RAM configuration, You didn't either install the target source code properly (see Install Manual for Windows) or You didn't import tutorials into workspace properly (see Chapter 2 of this manual):



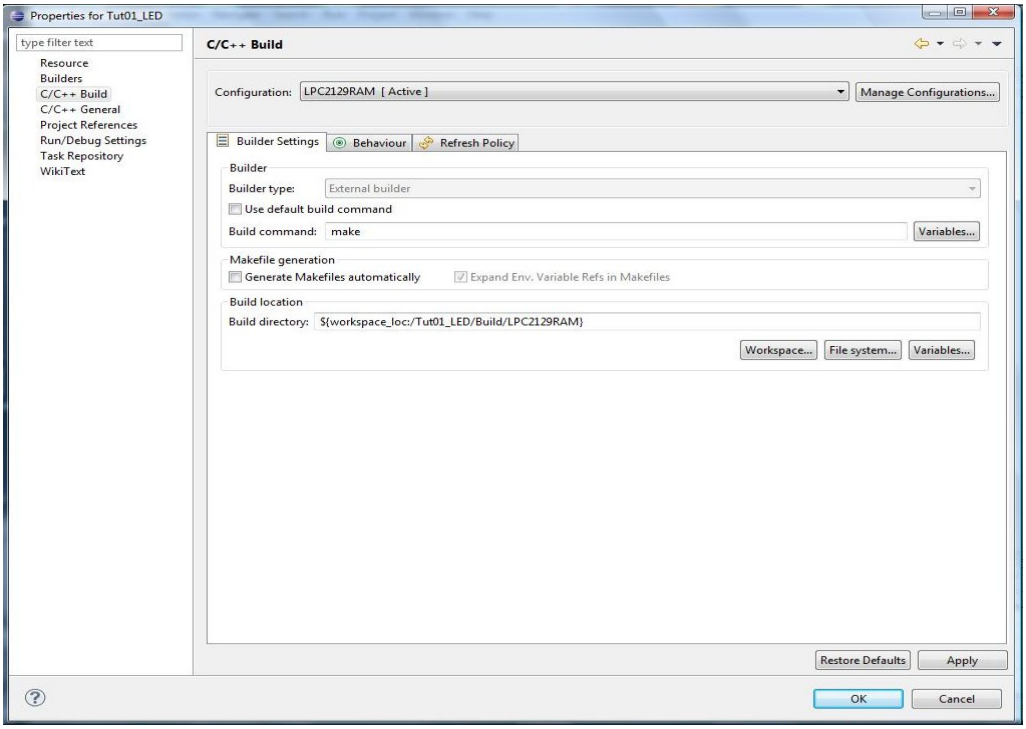
Choose 'External builder', then press Apply:



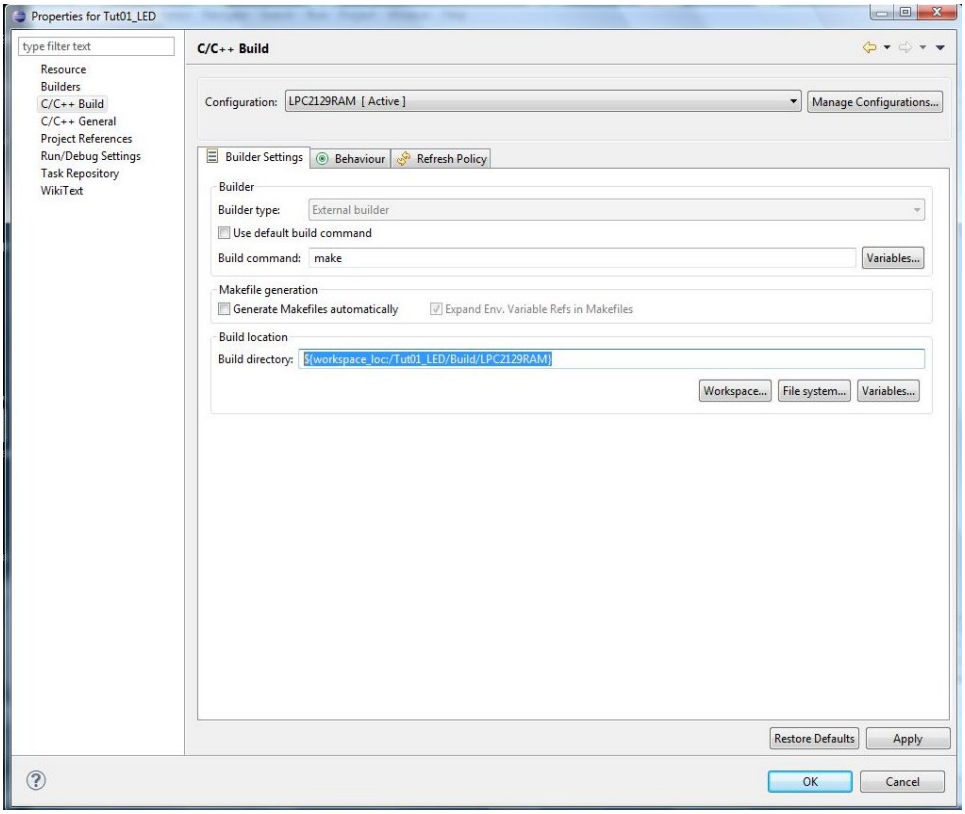
Deactivate the the 'Use default build command' checkbox, and make sure, that the build command is 'make':



Deactivate the 'Generate Makefiles automatically' checkbox:



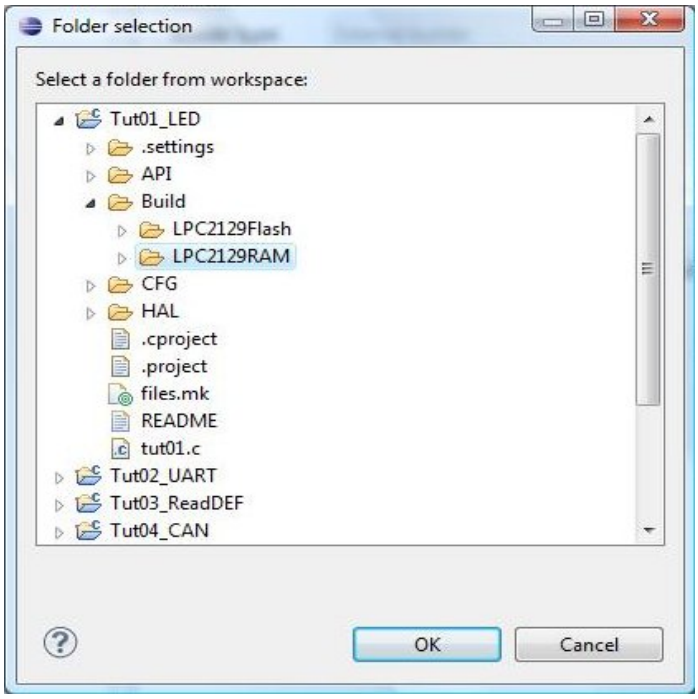
Make sure the build directory path looks like that:



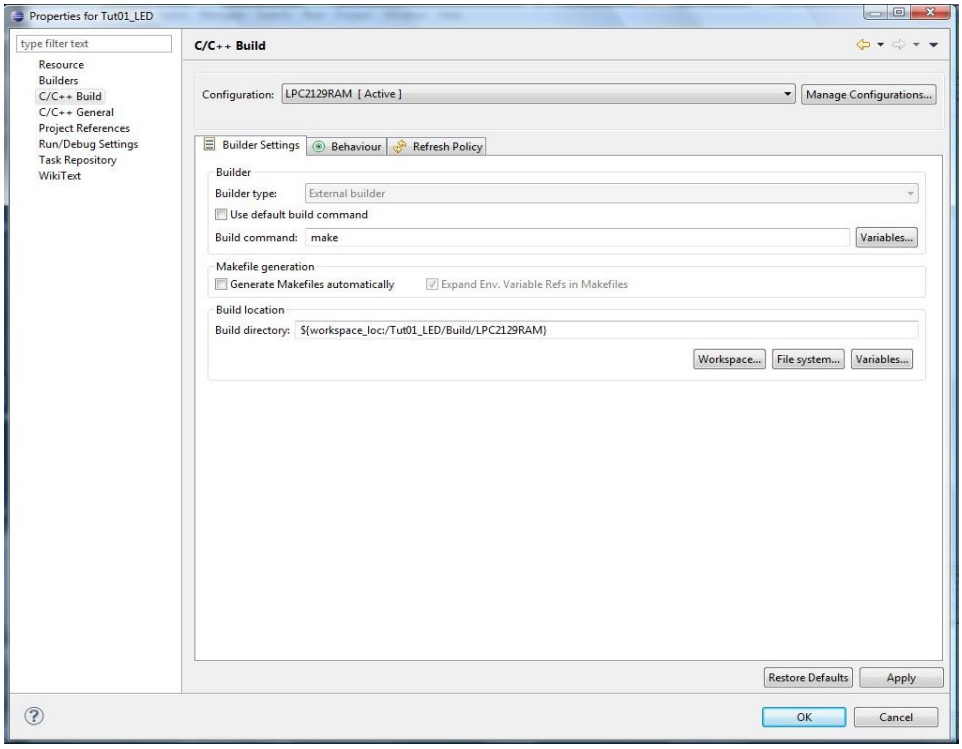
If not, press the 'Workspace' button and browse until the build location is

./Tut01_LED/Build/LPC2129RAM

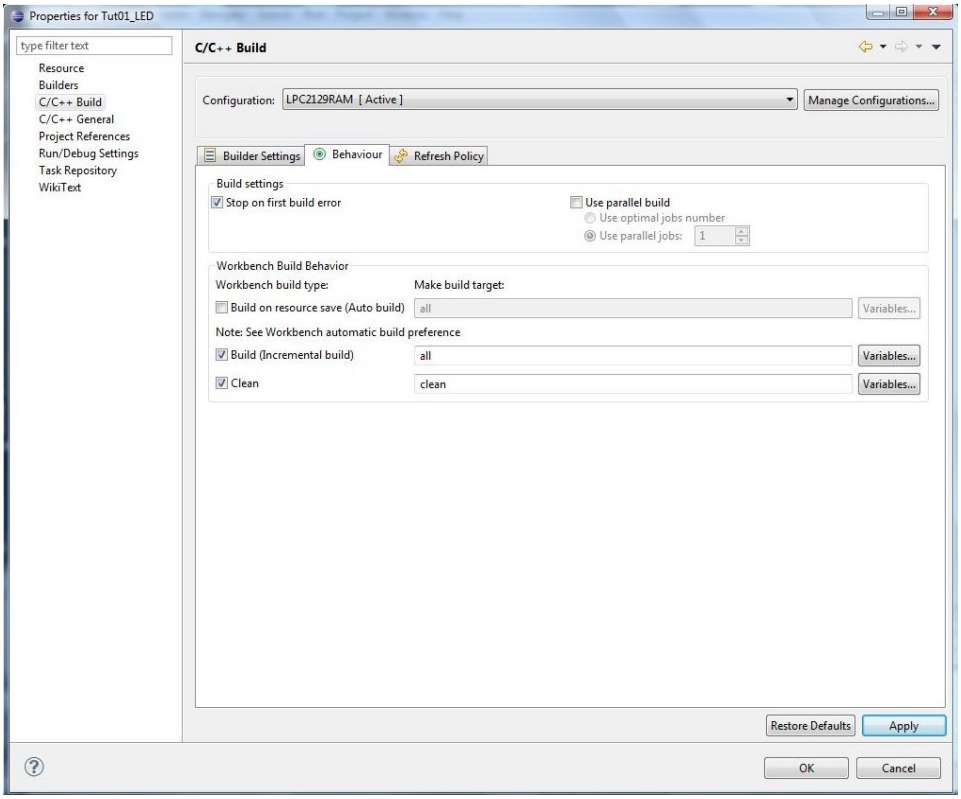
Then press OK:



Be sure the 'C/C++ Build' menu settings are correct and look like this:



Open the 'Behavior' slider and make sure that all entries look like that:



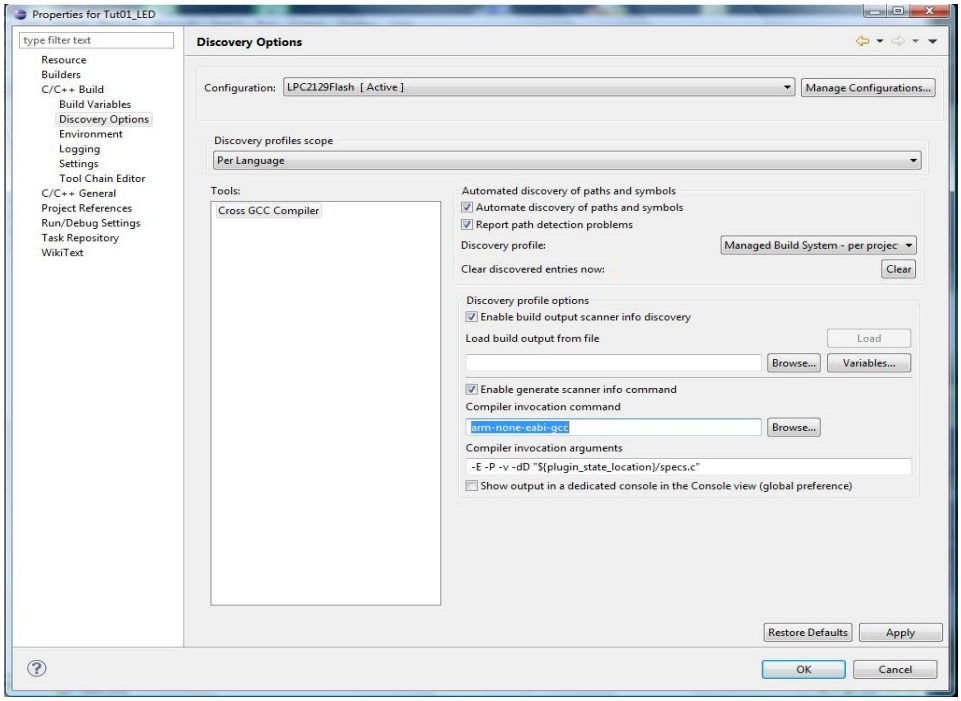
Go to the 'Discovery options' sub-menu. Below the checkbox 'Enable generate scanner info command' add at the compiler invocation command the following prefix:

arm-none-eabi-

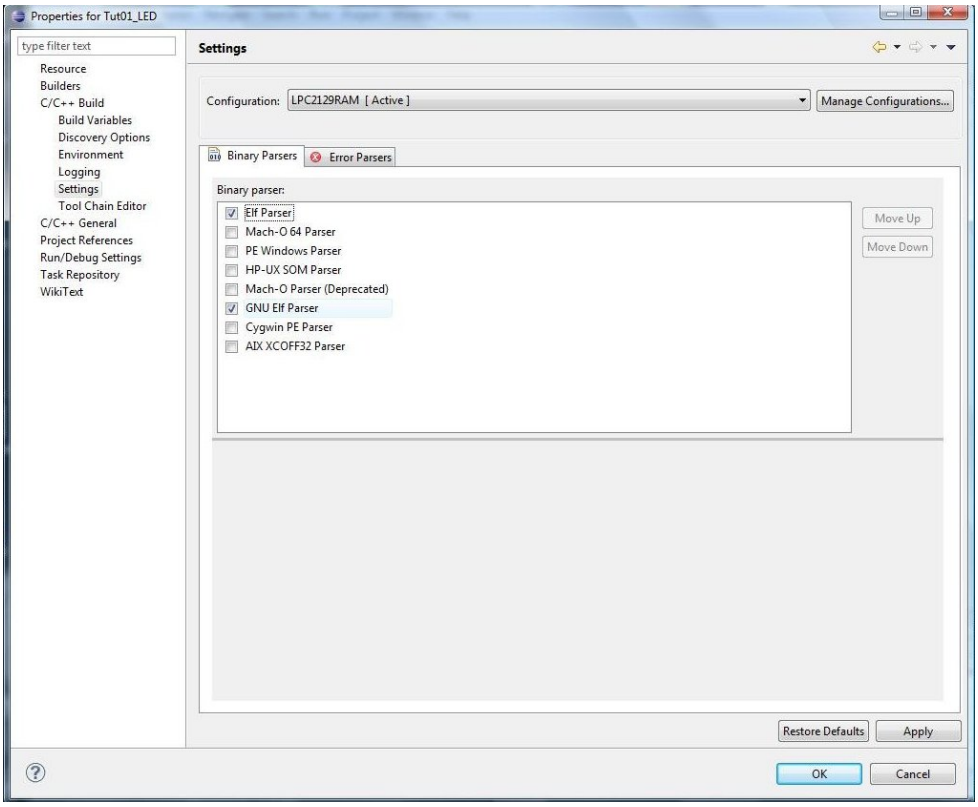
on Windows or otherwise

arm-elf-

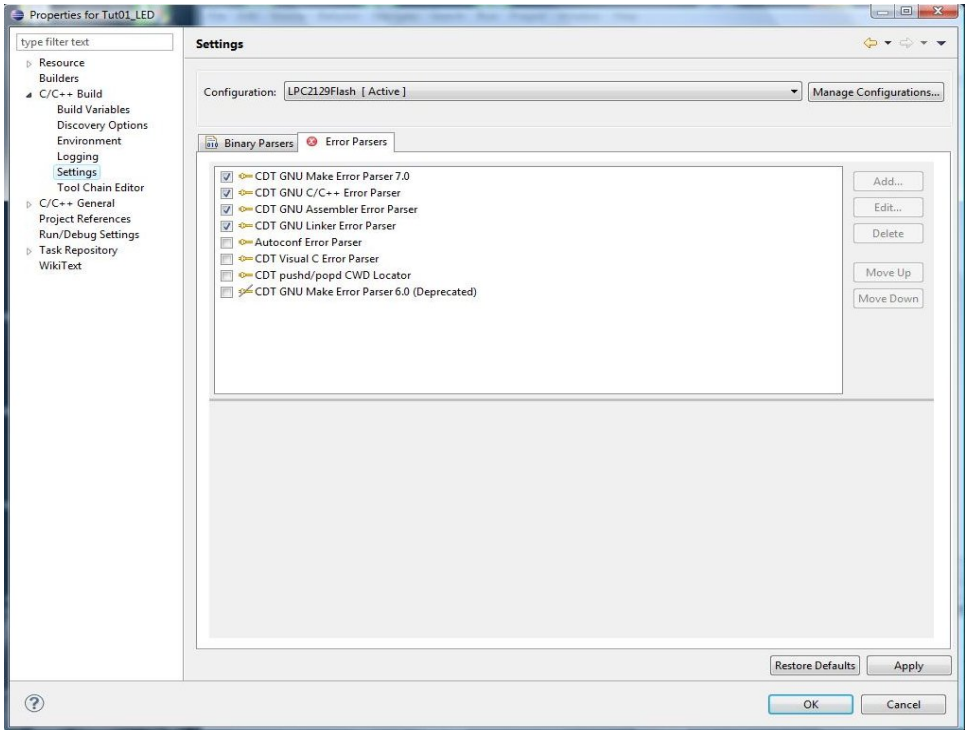
before 'gcc'. Check the Compiler invocation arguments, then click Apply:



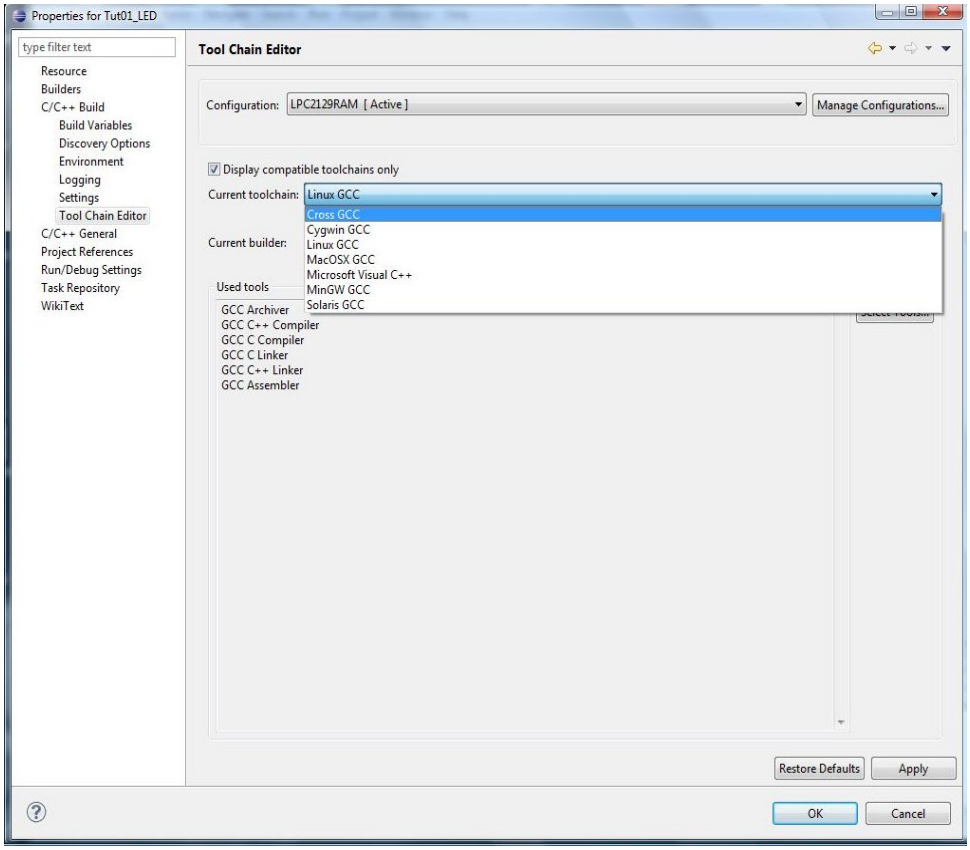
Go to 'Settings' sub-menu and the 'Binary Parsers' slider and check if the 'Elf Parser' and 'GNU Elf Parser' are activated, click Apply:



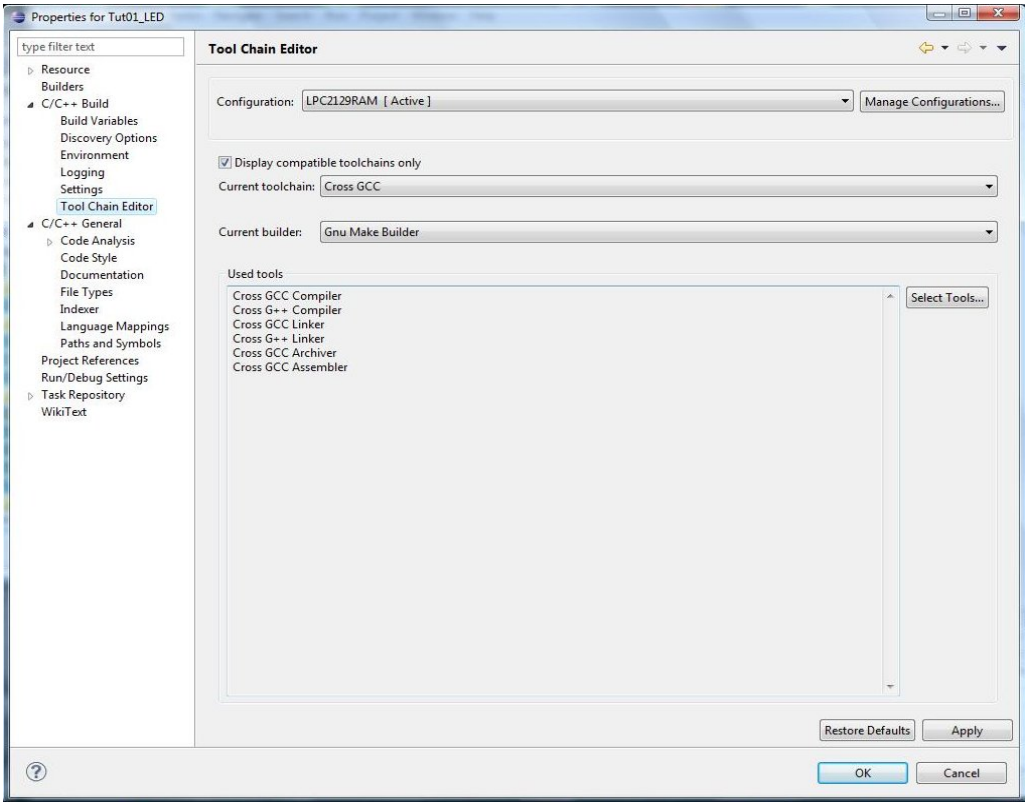
Go to the 'Error Parsers' slider and check if the following parsers are activated: 'C/C++ Error Parser', 'Assembler Error Parser', 'Linker Error Parser' and 'Make Error Parser'. Click OK.



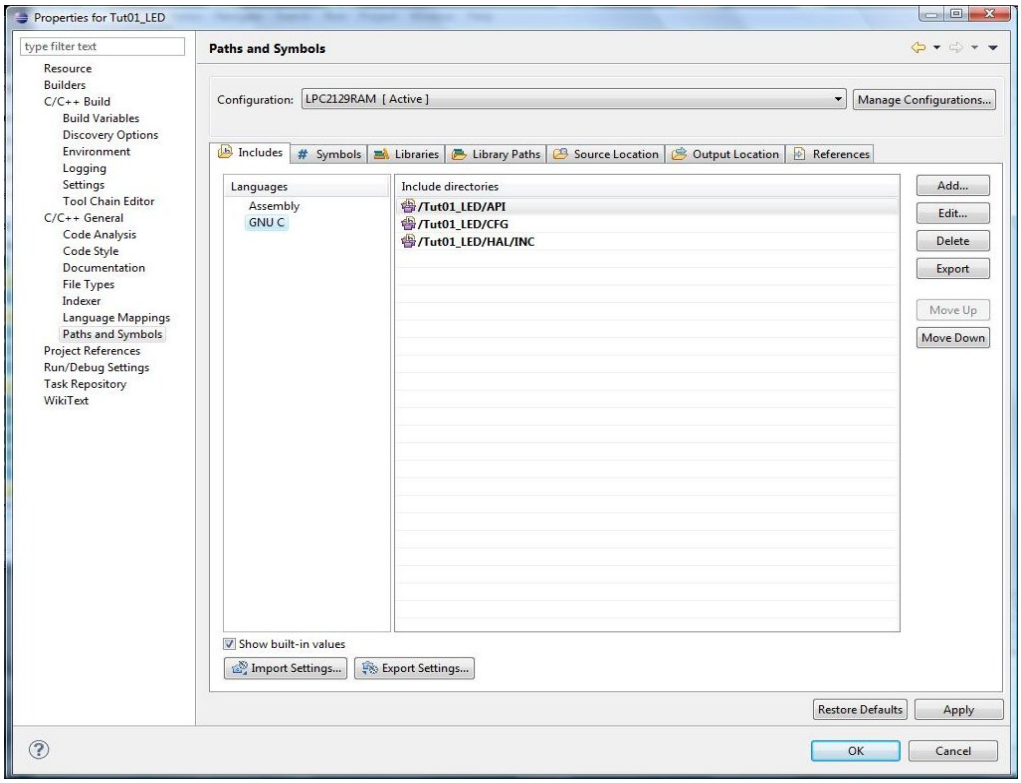
Go to 'Tool Chain Editor' sub-menu and choose Current tool chain 'Cross GCC':



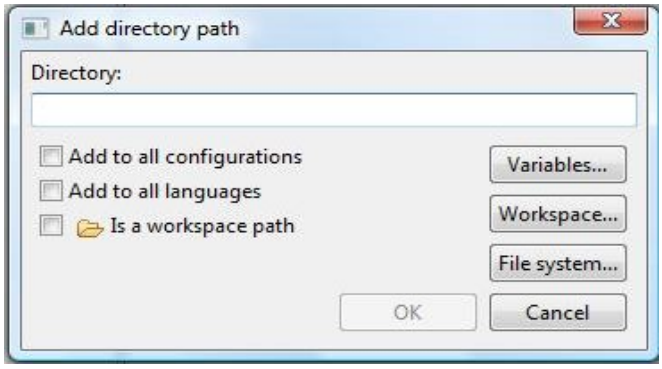
Check if the settings in the sub-menu 'Tool Chain Editor' look like this and click Apply:



Go to 'C/C++ General' menu and 'Path and Symbols' sub-menu to the 'Includes' slider and look if all include directories entries look similar:



If not, press the 'Add' button and the following menu appears:



Press the 'Workspace' button and add the

./Tut01_LED/API

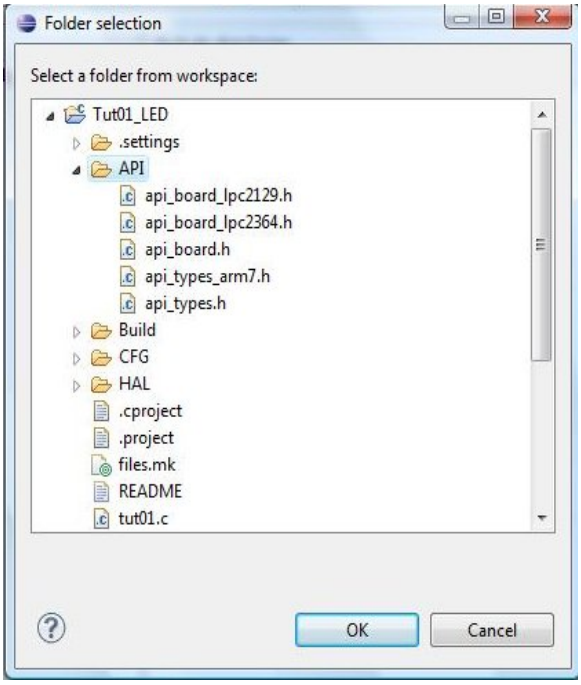
directory. Add similarly the

./Tut01_LED/CFG

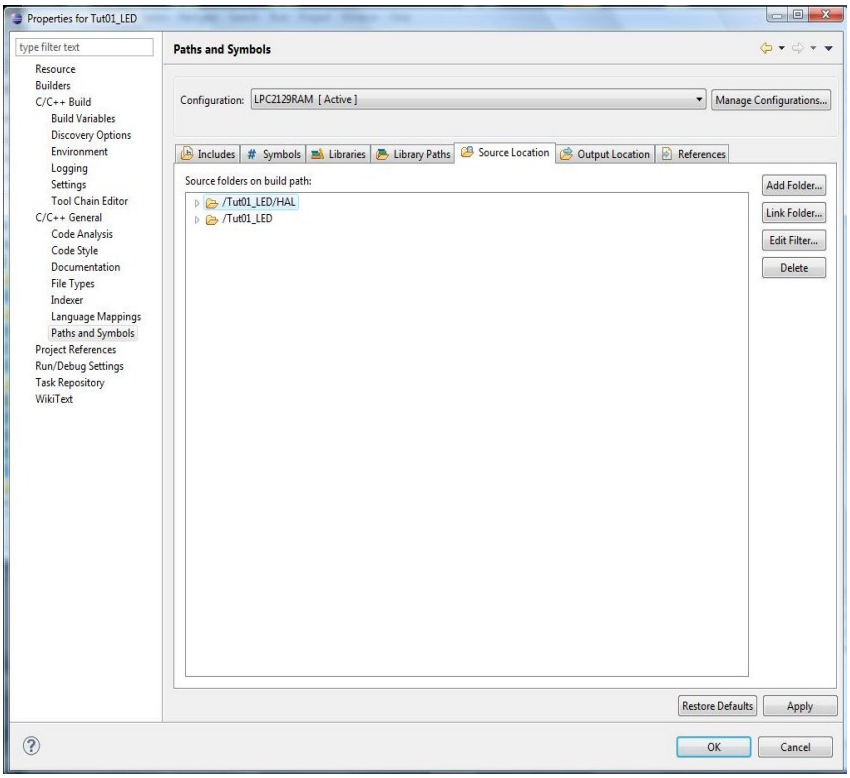
and

./Tut01_LED/HAL/INC

directories. If You can't find the HAL directory at this time, You first have to add the source location to the project directory, that we show You in the next step.

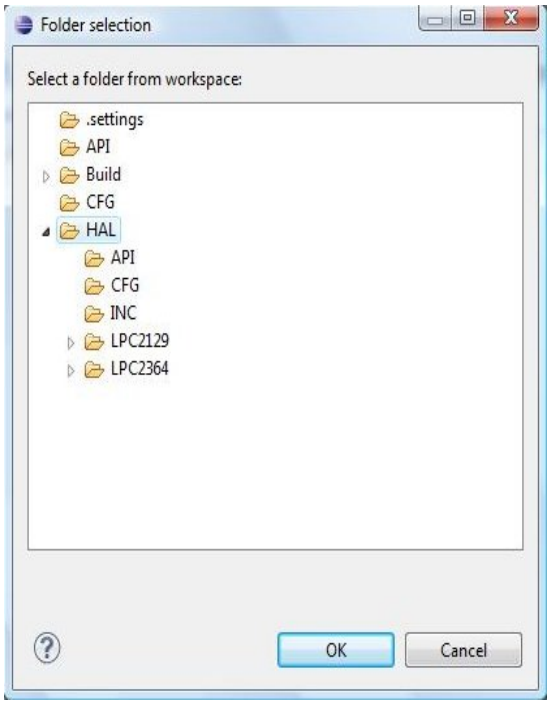


Go to the 'Source Location' slider. See, if it looks like this:

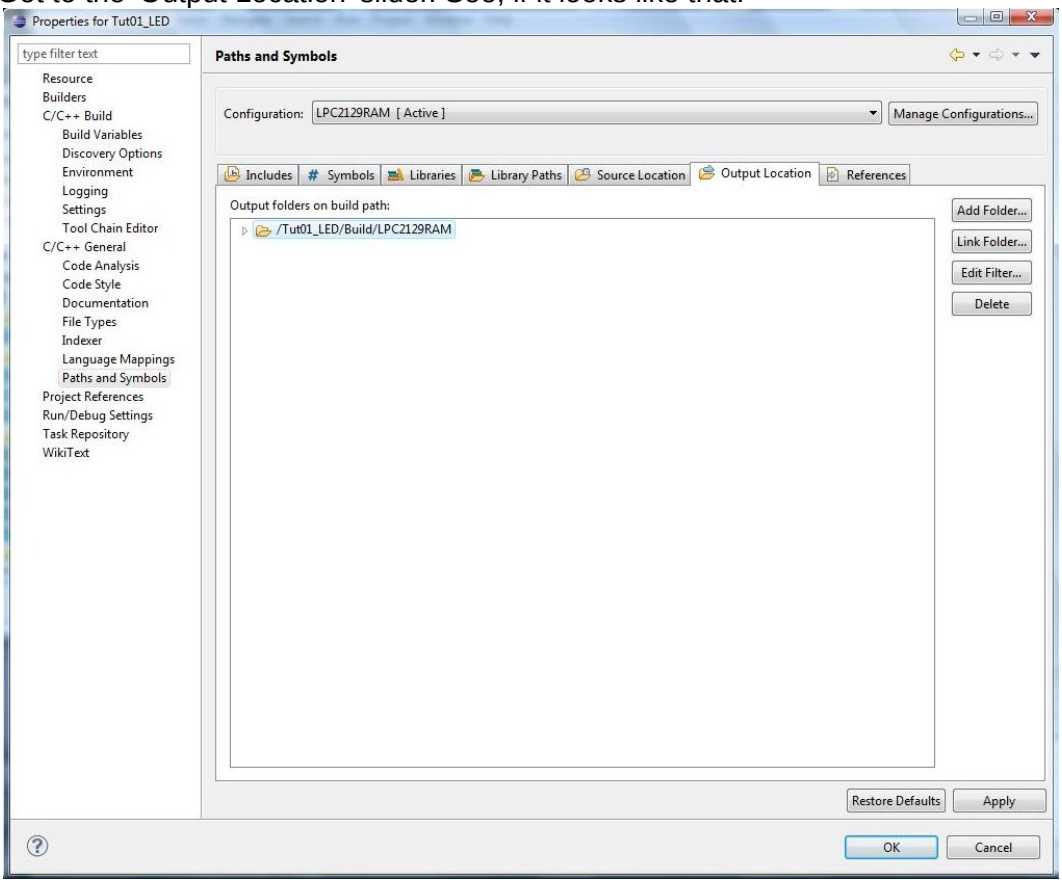


If not, press the 'Add Folder...' button and select the folder You want to get into Eclipse source control from Your workspace.

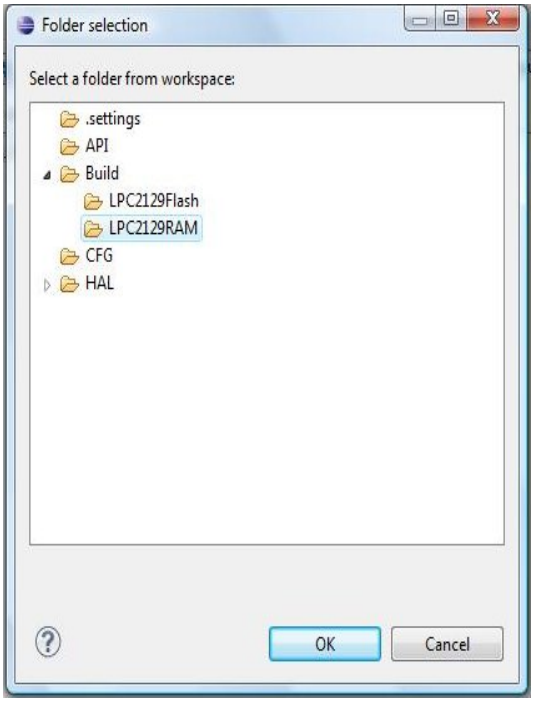
This will only work, if the folder is already part of Your workspace, otherwise You can't select it. If You don't know how to make a folder part of Your workspace see 'Annex Adding Directories to Your Project'.



Got to the 'Output Location' slider. See, if it looks like that:



If not, press the 'Add Folder...' button and select the folder You want from Your workspace. You should tell Eclipse the correct output path to prevent some warnings:



You are now ready for building the project. This is shown in the next chapter.

4 Building Projects

Now You will learn how to build projects. Building includes all steps like pre-compiling makro commands to C commands, translating from C to machine code, translating from ARM Assembler to machine code and linking all machine code files (object files) to one executable.

The build process is controlled by the GNU make builder. So, first You have to make sure, that Your Makefile is correct. Compare it to the Makefile for RAM and Makefile for Flash in the 'Annex Makefiles' and check that You have also a correct 'files.mk' in the project directories, which is a file that is included by Makefile and contains all project files to be compiled.

You must be sure, too, that Your linker scripts 'ram2129.ld' or 'ram2364.ld' are correct. Therefore compare it to the linker scripts in 'Annex Linker Scripts'.

Without correct project settings it is also not possible to build the project. So if You didn't so far, go to chapter 'Setting Project Properties' and do all project settings steps precisely.

You must be sure that You have set the right build configuration, in our example this is 'LPC2129RAM'. All project settings we did in the chapter 'Setting Project Properties' referred to this build configuration.

Additionally You have to set the define 'HAL_START_VECTOR_RAM' in the projects SRC/CFG/cfg_global.h file to '1' and to set the define 'HAL_START_VECTOR_FLASH' to '0'. This will set the NXP LPC2129 or LPC2364 memory mapping registers to the correct value.

WITH WRONG MAKEFILES, LINKER SCRIPT, WRONG BUILD CONFIGURATION, UNCORRECT PROJECT SETTINGS OR WRONG MEMORY MAPPING FLAG YOUR SOFTWARE WILL NOT WORK CORRECTLY.

Make always sure that in the case You use our LPC2129 board Your

- build configuration is 'LPC2129RAM'
- 'Makefile' and 'files.mk' are correct
- linker script is 'ram2129.ld' and is correct
- project settings for 'LPC2129RAM' are correct
- memory mapping flag 'HAL_START_VECTOR_RAM' is set to '1'
- memory mapping flag 'HAL_START_VECTOR_FLASH' is set to '0'

In the case of our LPC2368 board it is the same but 2129 is 2364. Now, You are ready to proceed.

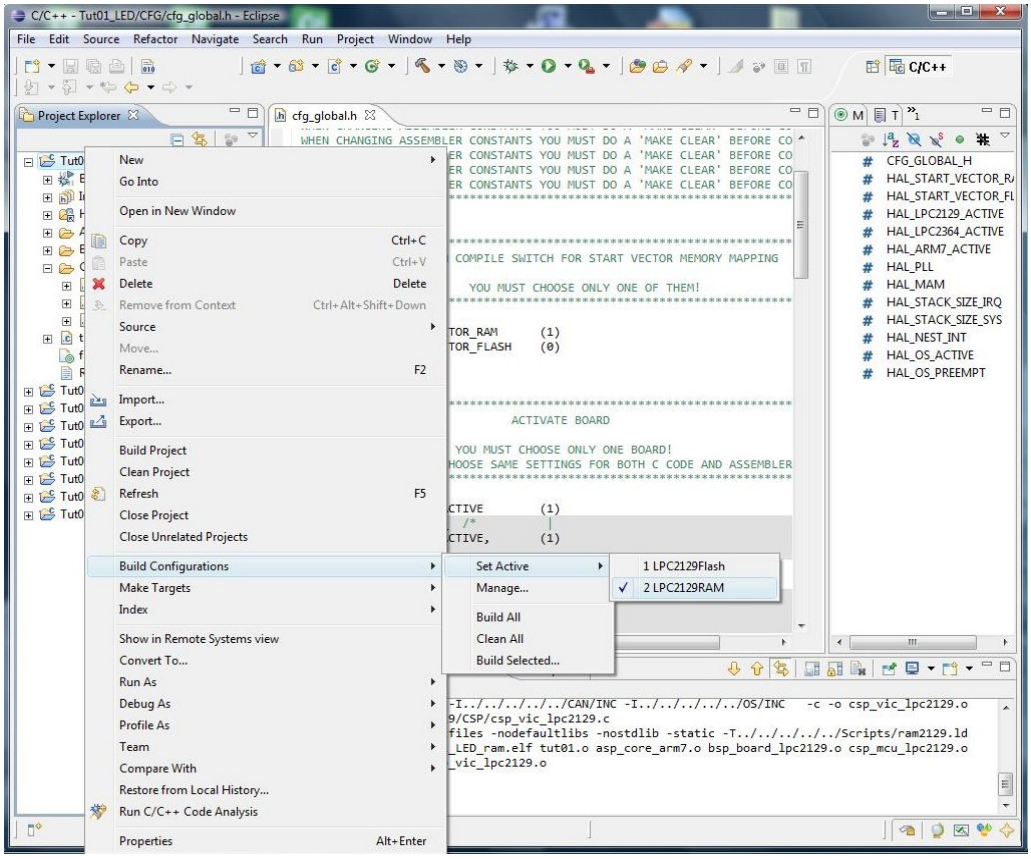
At the building process You should get a message in the 'Console' output slider of Eclipse IDE like:

```
**** Build of configuration LPC2129RAM for project Tut01_LED ****
make all
…
**** Build Finished ****
```

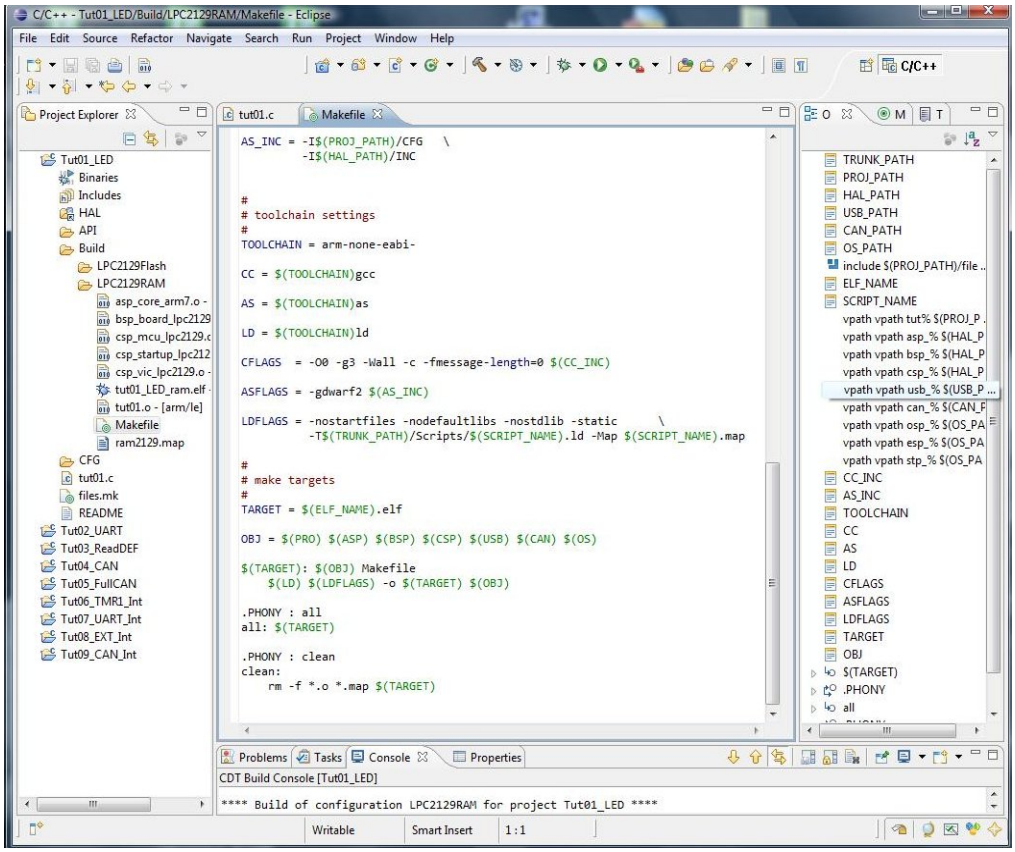
Or, if You just did a successful build before, something like:

```
...
make: Nothing to be done for 'all'
**** Build Finished ****
```

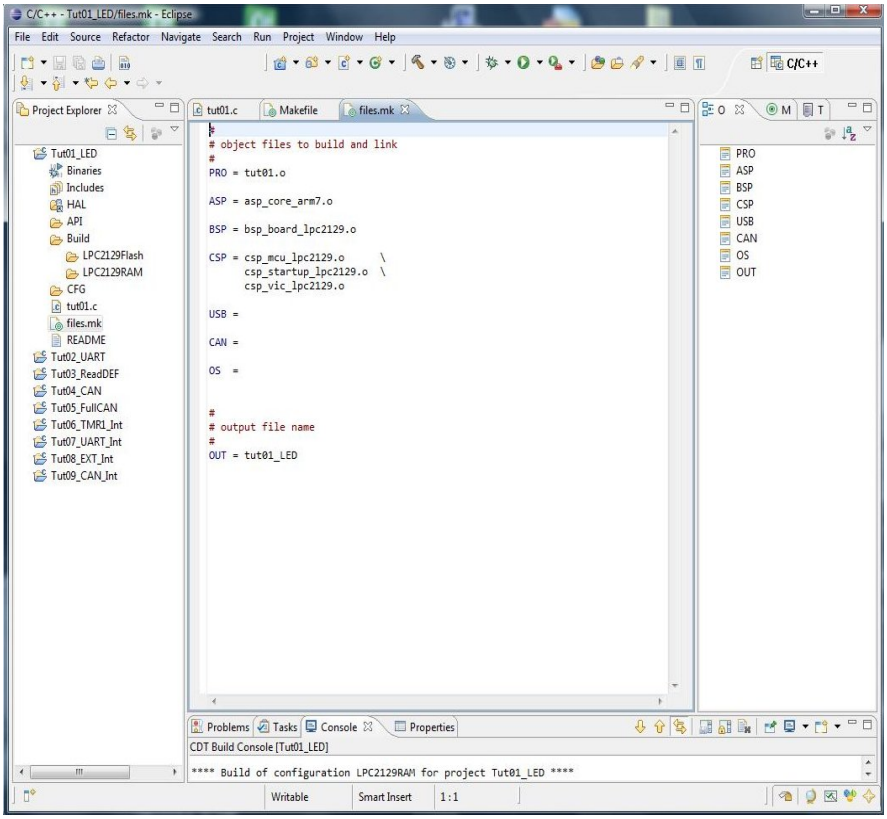
Set the build configuration to 'Build Configuration' -> 'Set Active' -> 'LPC2129RAM':



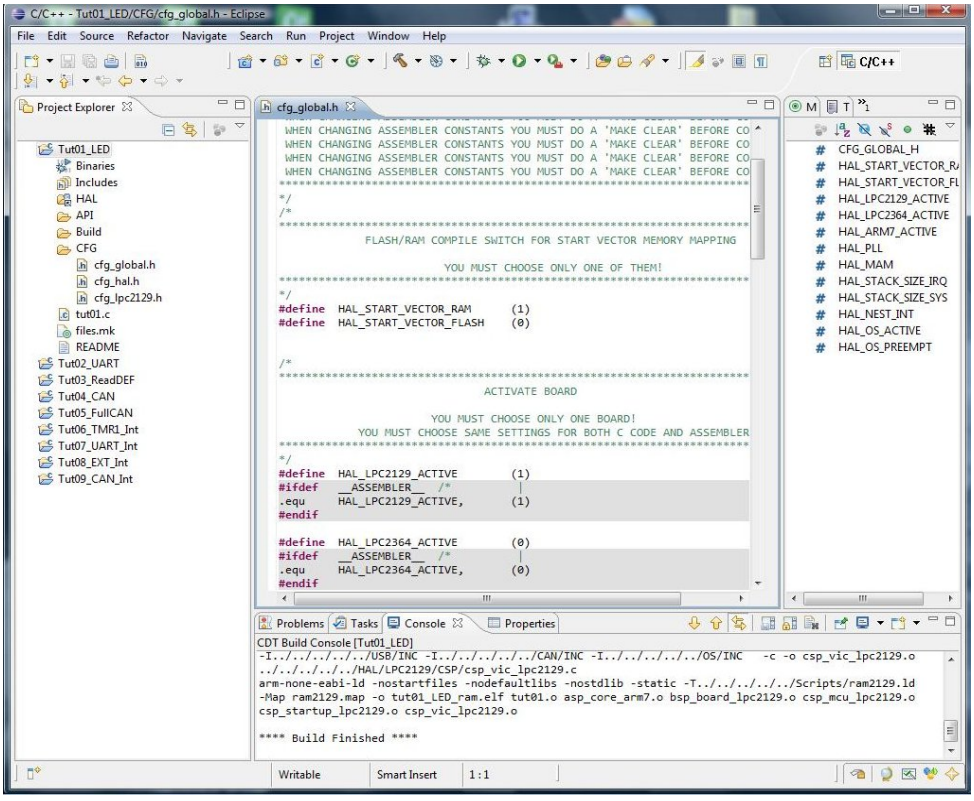
Check if the Makefile is the correct RAM makefile and that it contains 'ram2129.ld':



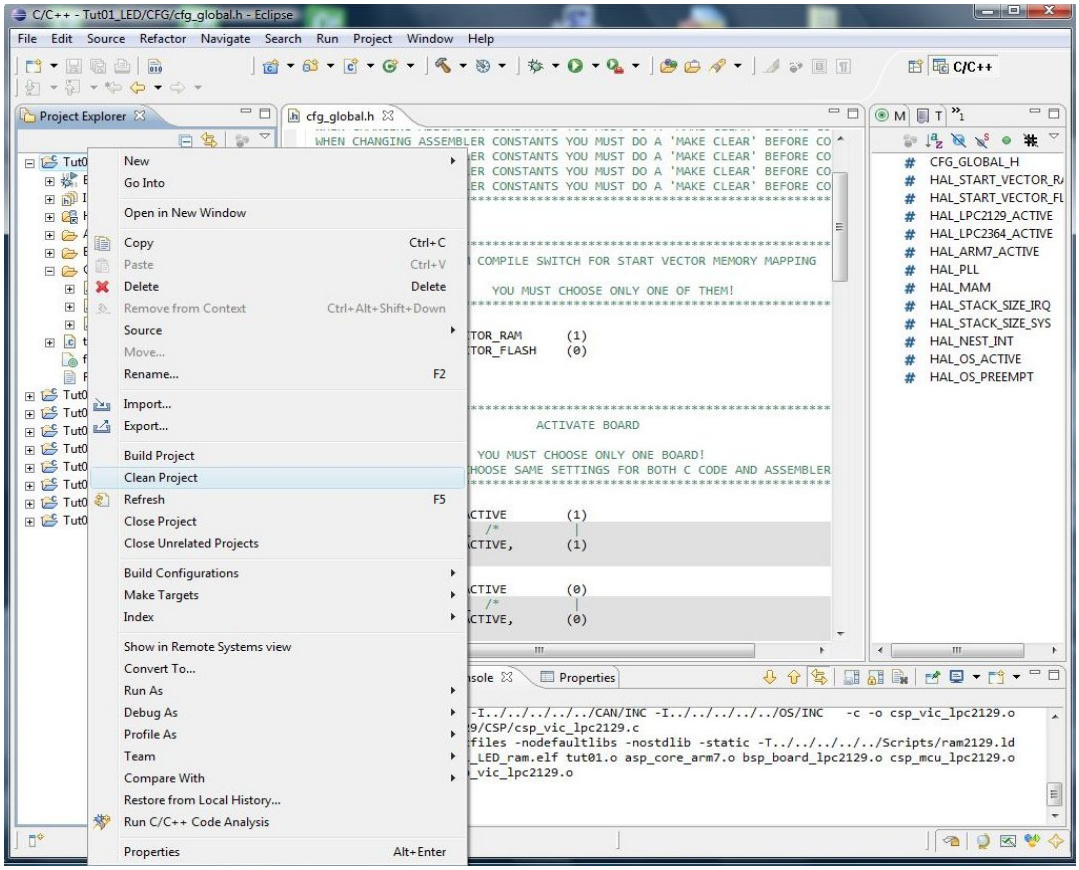
Check if all files that are needed for compilation are noted in the files.mk file:



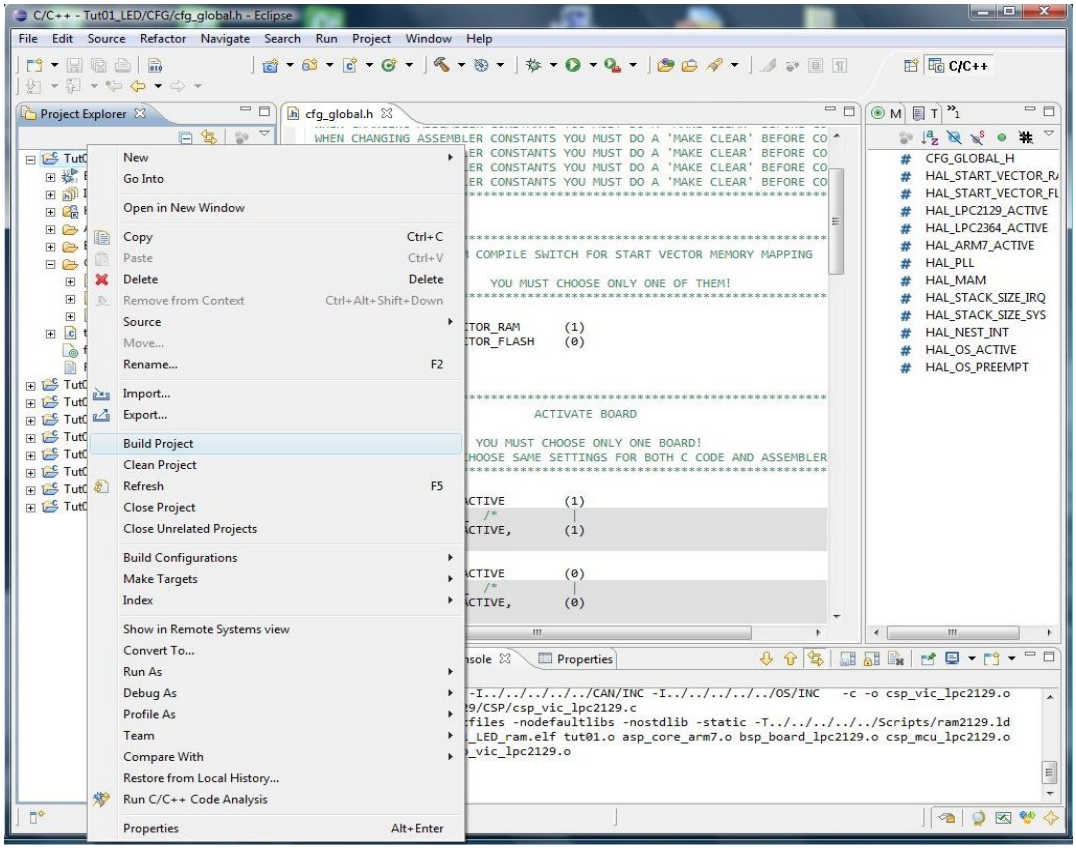
Have a look into file 'SRC/CFG/cfg_globall.h' and check that the memory mapping flag 'HAL_START_VECTOR_RAM' is set to '1' and 'HAL_START_VECTOR_FLASH' is set to '0' for a RAM compilation:



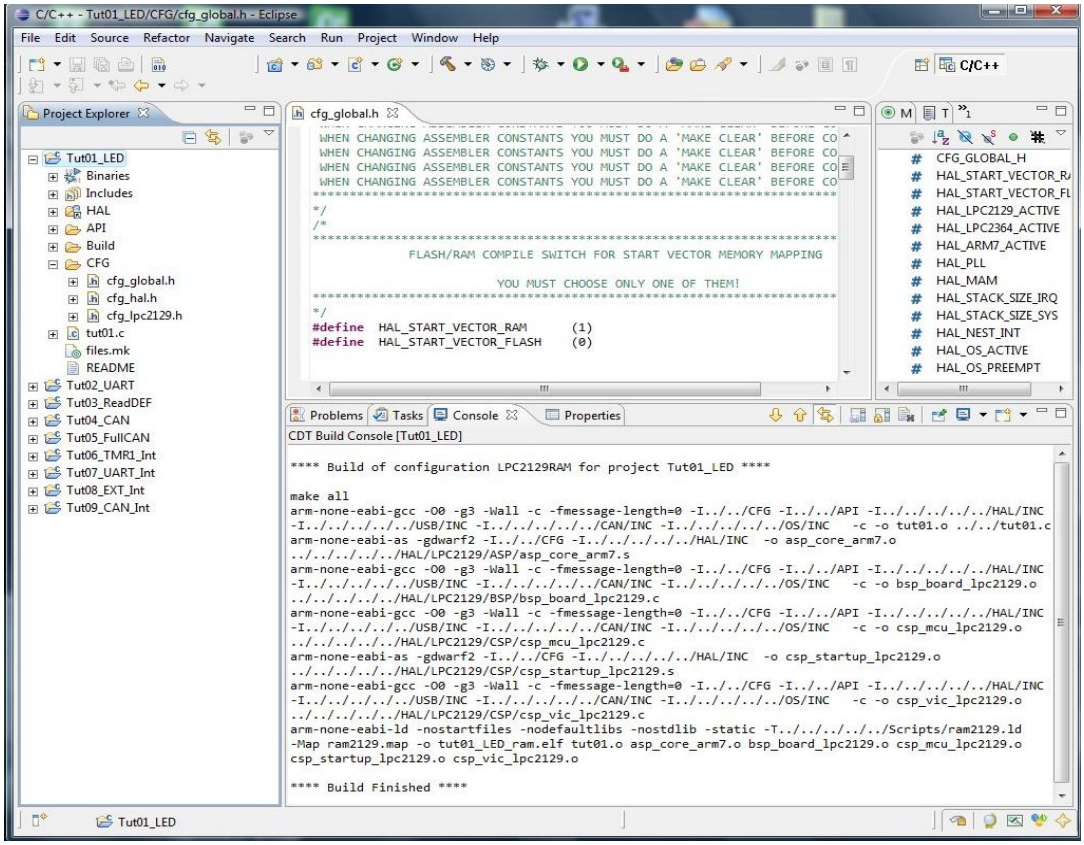
Check again Your project Properties settings. Click the right mouse button on Your project directory and choose 'Clean Project':



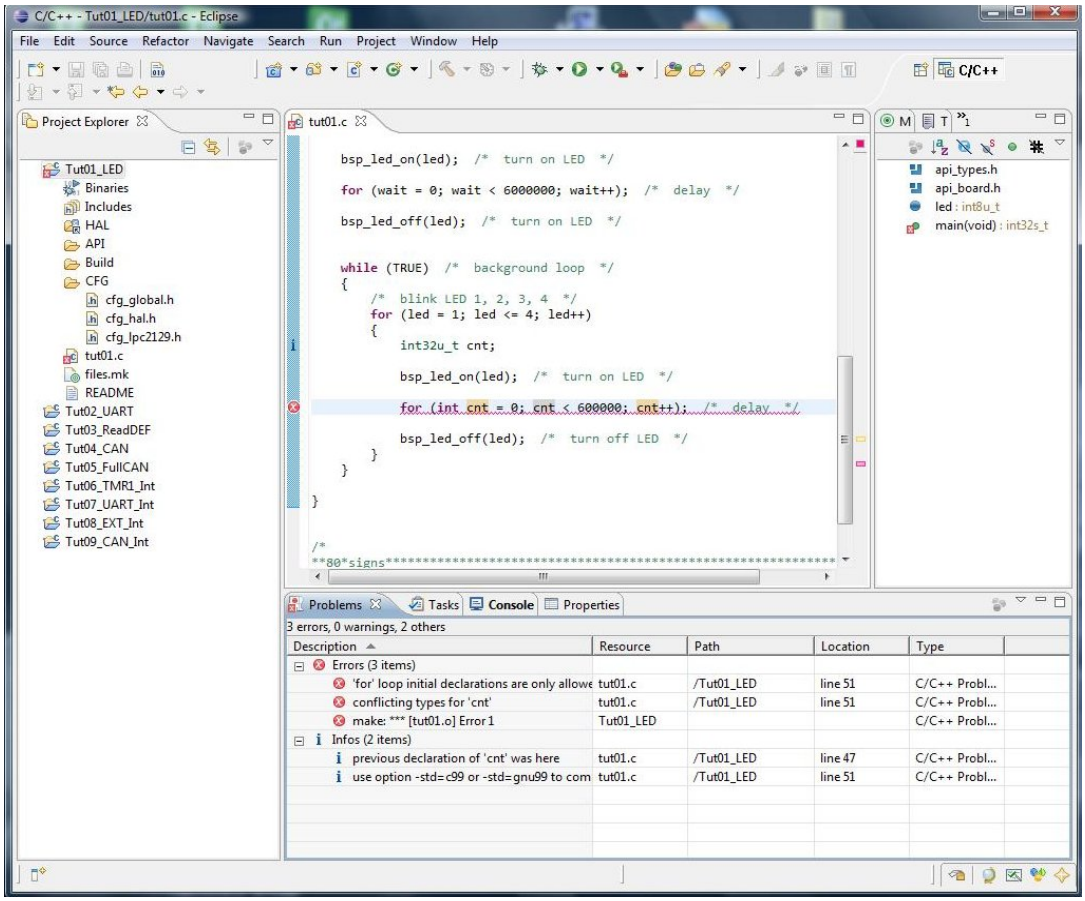
Click the right mouse button on Your project and choose 'Build Project':



Now the ARM assembler, C compiler and linker should work for a short moment. You can watch the output in the Eclipse IDE 'Console' output slider:



After the build You can read a list of all errors and warnings in the 'Problems' slider:



5 Running the JTAG Debugger

Make sure, Your debugging adapter is supported by OpenOCD. You can read which devices are supported by OpenOCD newest at

http://openocd.sourceforge.net/supported-jtag-interfaces/

Here's an excerpt of this list:

USB FT2232 Based:

- ✗ *usbjtag*
- ✗ *jtagkey*
- ✗ *jtagkey2*
- ✗ *oocdlink*
- ✗ *signalyzer*
- ✗ *Luminary ICDI*
- ✗ *olimex-jtag*
- ✗ *flyswatter*
- ✗ *turtelizer2*
- ✗ *comstick*
- ✗ *stm32stick*
- ✗ *axm0432_jtag*
- ✗ *cortino*
- ✗ *dlp-usb1232h*

USB-JTAG / Altera USB-Blaster compatibles:

- ✗ *USB-JTAG Kolja Waschk's USB Blaster-compatible adapter*
- ✗ *Altera USB-Blaster*

USB JLINK based:

- ✗ *SEGGER JLINK*
- ✗ *IAR J-Link*

USB RLINK based:

- ✗ *Raisonance RLink*
- ✗ *STM32 Primer*
- ✗ *STM32 Primer2*

USB Other:

- ✗ *USBprog*
- ✗ *USB - Presto*
- ✗ *Versaloon-Link*
- ✗ *ARM-JTAG-EW*

IBM PC Parallel Printer Port Based:

- ✗ *Wiggler - There are many clones of this.*
- ✗ *Wiggler2*
- ✗ *Wiggler_ntrst_inverted*
- ✗ *arm-jtag*
- ✗ *chameleon*

Connect Your Debugging adapter to Your PC.

Your debugging adapter will only work, it it is supported by OpenOCD **AND** if it was compiled into the OpenOCD executable. Make sure that You compiled OpenOCD correctly or that Your provider did this for You.

Open a new terminal and type in the command line:

openocd-0.5.0.exe -f target/lpc2129.cfg -f interface/jtagkey.cfg

or

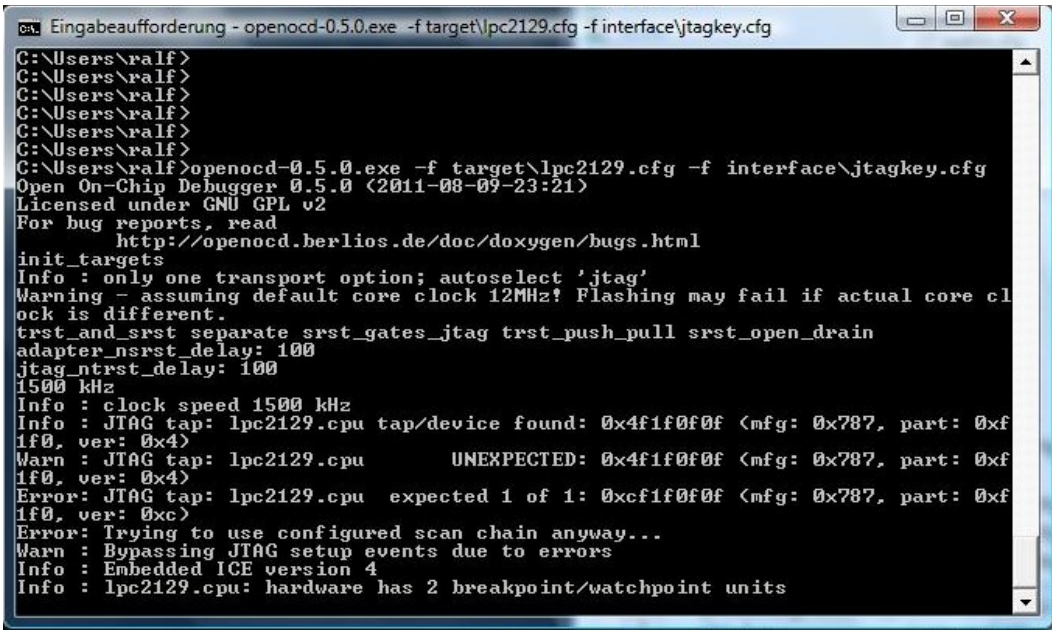
openocd-0.5.0.exe -f target/lpc2378.cfg -f interface/jtagkey2.cfg

This command can only work, if You installed OpenOCD correctly and added the environment variables correctly. If You didn't, look into the Windows/Linux Install Manual, chapter 'JTAG Debugger'.

Make sure, that You use the correct configuration files. Compare Your target configuration files with the files in 'Annex OpenOCD target config files'. The configuration files of Your installation are located in Your OpenOCD directory:

- Windows e.g.: C:\Bitrelle\openocd-0.5.0
- Linux e.g.: /usr/local/share/openocd/scripts/

When You start OpenOCD You might get an error message like that:

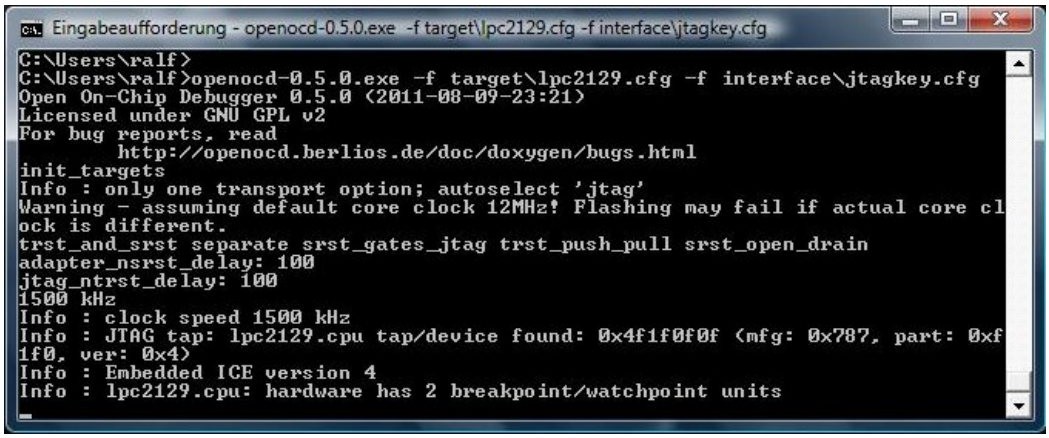


This is no important error as long as You only have one device in the JTAG scan chain, otherwise it is important. You can change the <cputapid> value in the 'lpc2129.cfg' file to prevent this error message:

setup_lpc2xxx lpc2129 0x4f1f0f0f 0x40000 lpc2000_v1 0x4000

The <cputapid> value is a number like 0x4f1f0f0f, 0x5f1f0f0f, 0xcf1f0f0f, the OpenOCD output error message tells You the right number to insert.

After You have inserted the <cputapid> into the 'lpc2129.cfg' file, start OpenOCD again. From now on You should get no error messages again:



To shutdown OpenOCD use the key combination

<CTRL>+C

OpenOCD can be controlled by GCC directly or via Eclipse. This will be explained in the later chapters. There is another method to get access to Your target by controlling OpenOCD with a Telnet connection.

If You don't have a Telnet client installed on Your Windows PC have a look into the Windows install manual annex to see how to install it. Linux users should know how to install a Telnet network client with the help of of one of their distributions software installation tools.

You start the Telnet connection when OpenOCD is running by opening a new command line window and typing:

telnet localhost 4444

or

telnet 127.0.0.1 4444

Both commands mean the same. 4444 is the default Telnet port of OpenOCD and 127.0.0.1 is simply the IP address of 'localhost'.

The OpenOCD Telnet connection will open and You can type commands into the OpenOCD command line. Find the table at the end of this chapter to see the most important OpenOCD commands. The 'usage' command shows You all possible commands and with 'help [command]' You can read more about it.

As an example You can study the following sequence of OpenOCD commands:

pwd

add_script_search_dir c:\Bitrelle\Data\Targets\Tutorials\LPC2129\Tut01_LED

halt

arm core_state arm

load_image Build/LPC2129RAM/tut01_LED_ram.elf

step 0x40000000

resume

halt

bp 0x400001b0 4

bp 0x400001b0 4 hw

bp 0x400001e8 4 hw

bp 0x40000468 4

bp 0x40000480 4

bp

resume

There are two hardware watchpoint/breakpoint units on the target CPU chip available. The (pseudo-code) sequence:

bp [address1] 4

bp [address1] 4 hw

bp [address2] 4 hw

bp [address3] 4

bp [address4] 4

etc.

enables the first watchpoint/breakpoint unit for software breakpoints. Now, it is possible to set in RAM as many software breakpoints as You need. This is not possible if You are debugging a program in the FLASH memory.

A program in the Flash memory can only use 2 breakpoints, because in the FLASH memory it is not possible to use software breakpoints.

This is not an OpenOCD issue. It is the case for all debuggers working with ARM 7 CPU's.

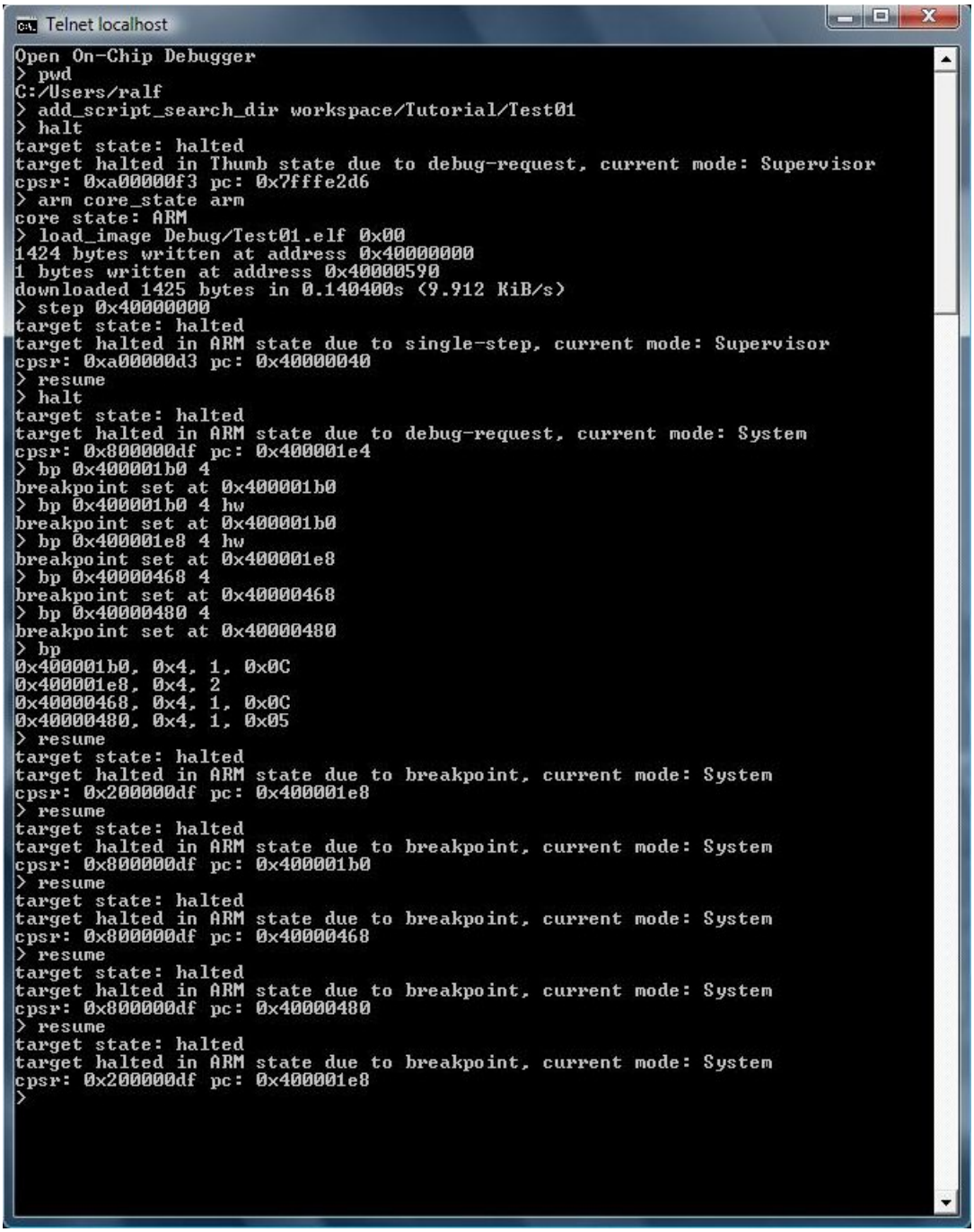
The sequence:

bp [address1] 4 hw

bp [address2] 4 hw

sets two hardware breakpoints. Now it's not possible to set any more breakpoints, especially no software breakpoints, even if You are working in the RAM, because software breakpoints need one of the two hardware watchpoint/breakpoint units to manage them.

This is a typical Telnet session with OpenOCD and a program residing in the RAM, where You can set as many software breakpoints as You want (it's similar to the sequence above):

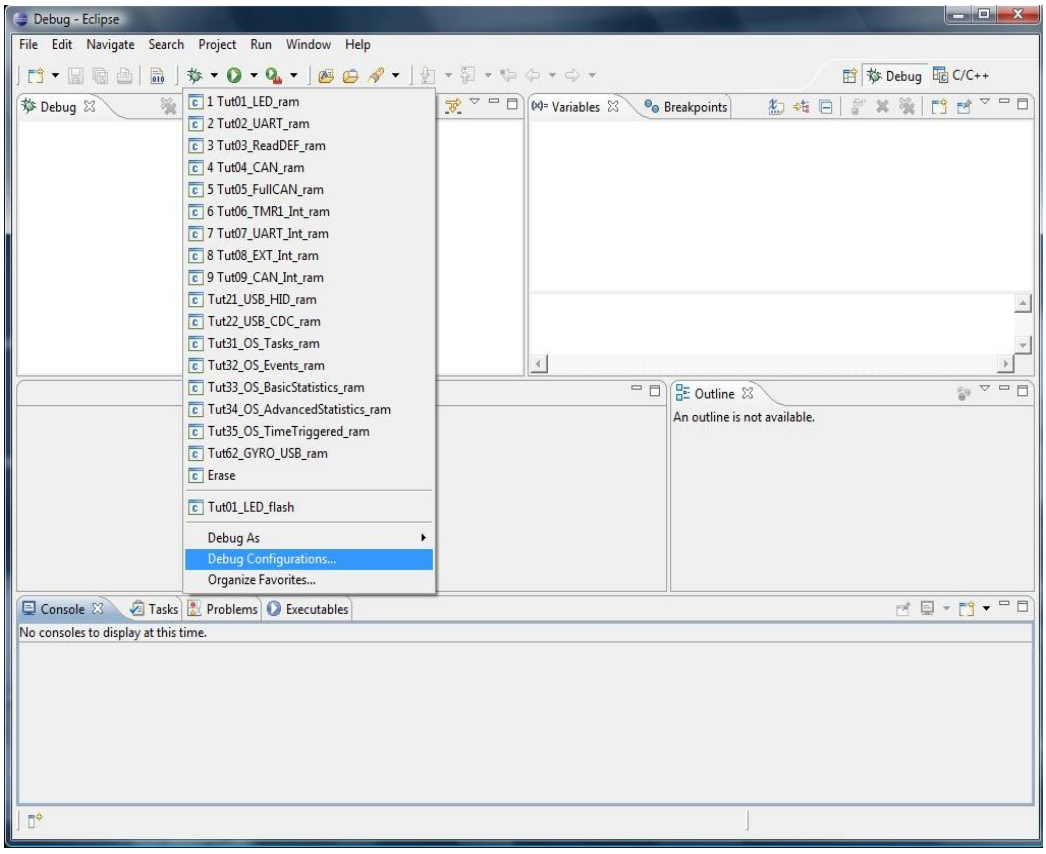


The most important OpenOCD command line instructions

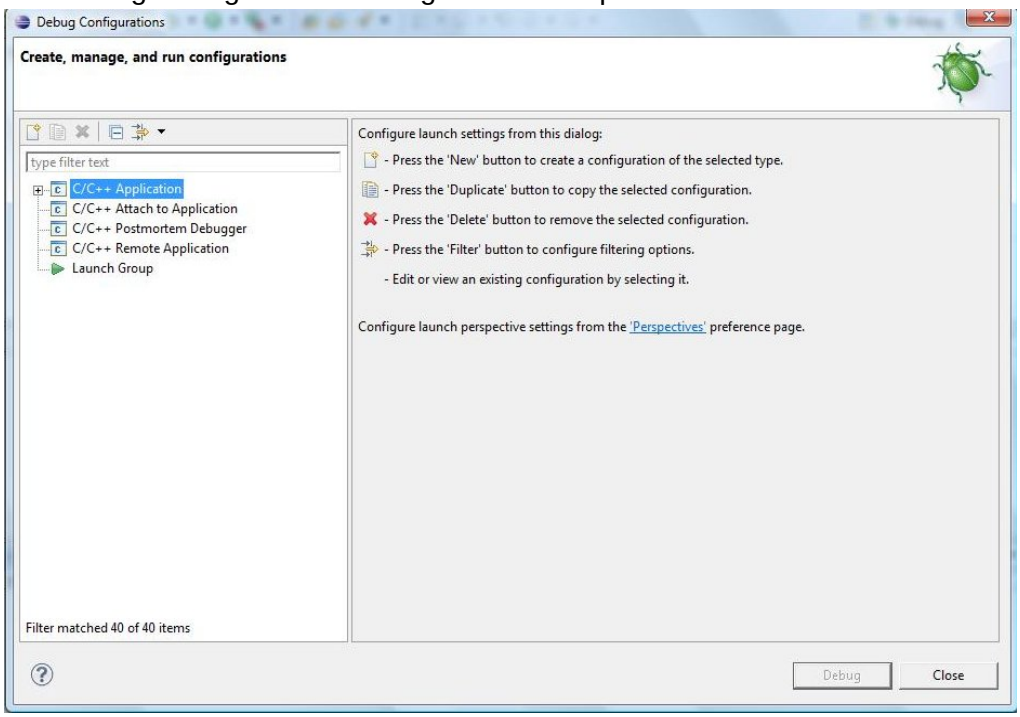
Command	Command, alternative	Explanation
exit		leave OpenOCD command line
usage	usage [command]	show commands
help [command]		more detailed help regarding command
pwd		print working directory
add_script_search_dir [directory path]		add directory path for searching script or binary files
load_image [image] [offset]		load a binary image to the address provided with the linker script and the additional offset
soft_reset_halt		stop CPU and set program counter back to reset vector
halt		halt JTAG debugger
resume	resume [address]	resume running program or resume running program at address
step	step [address]	step one machine instruction or step to address
bp		show list of breakpoints
bp [address] [size]		set soft breakpoint at instruction address, size is 4 for ARM and 2 for Thumb mode
bp [address] [size] hw		set hardware breakpoint at instruction address, size is 4 for ARM and 2 for Thumb mode
rbp [address]		remove breakpoint at address
poll [on/off]		poll target for state changes
arm reg		show registers and values
arm core_state arm		set CPU to ARM mode (not Thumb mode)

6 Eclipse Debug Settings

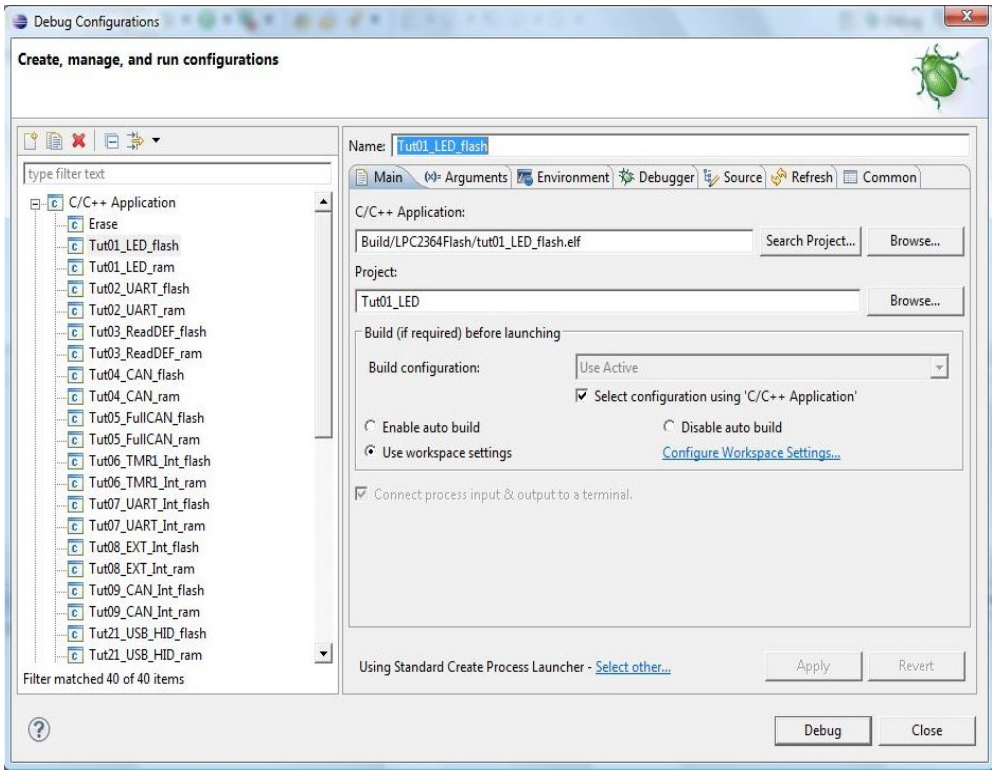
Before we can go on running and debugging programs on the target we first have to do some settings regarding the Eclipse debugging environment. Click on the right error near the bug symbol in the menu bar and choose 'Debug Configurations...' in the opened pop-down menu:



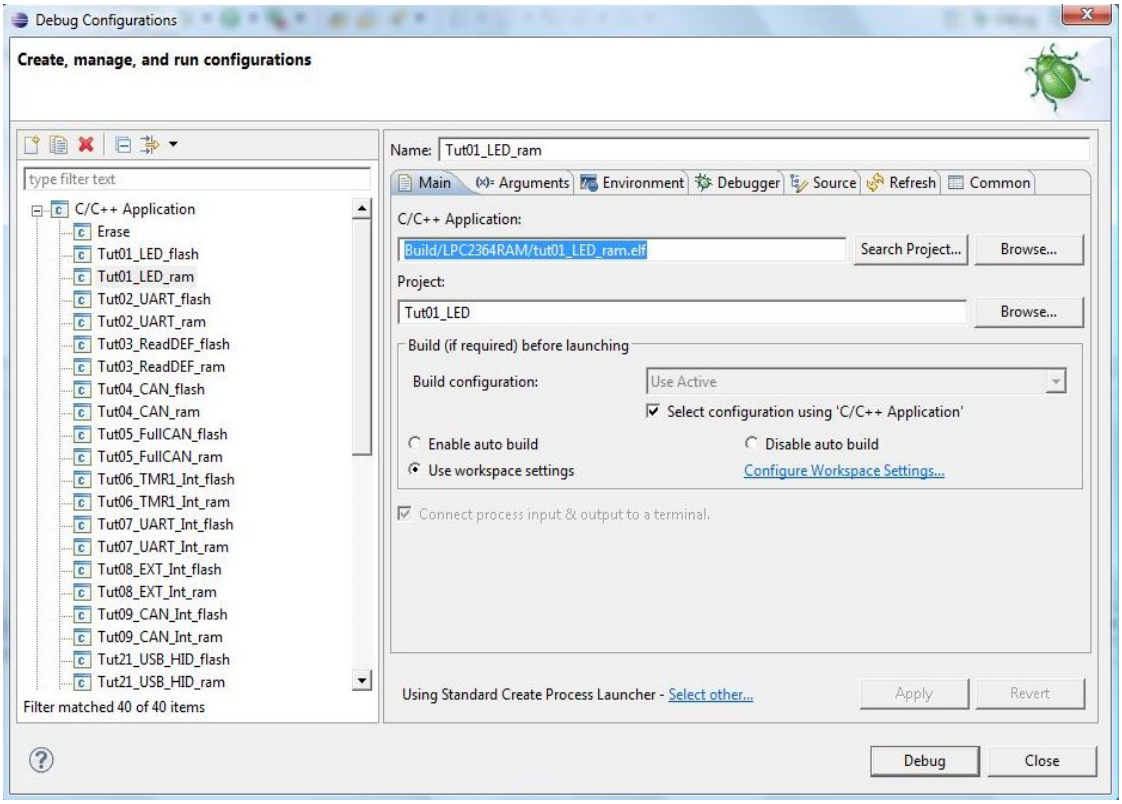
The 'Debug Configurations' dialogue window opens:



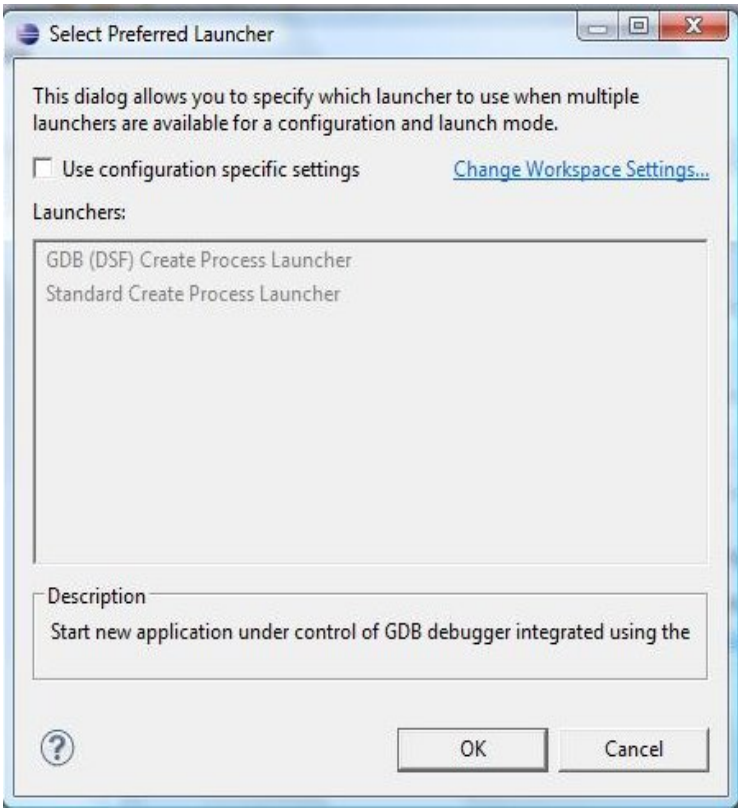
Mark the 'C/C++ Application' configuration type in the left menu and choose the 'New launch configuration' symbol (sheet of paper) on the upper left side. A new 'C/C++ Application' configuration opens. Input the correct name and click Apply:



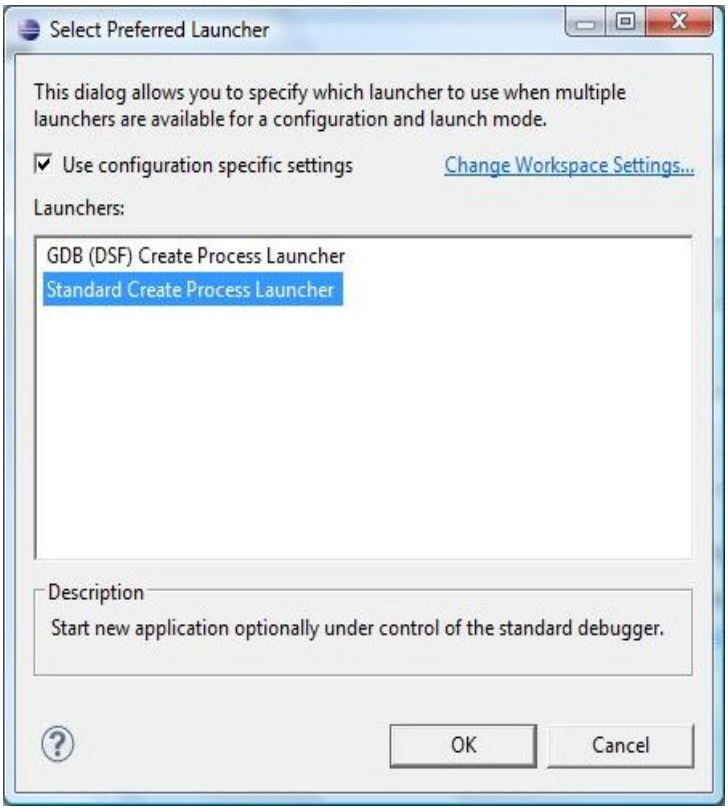
Select the correct 'C/C++ Application' relative path, click Apply:



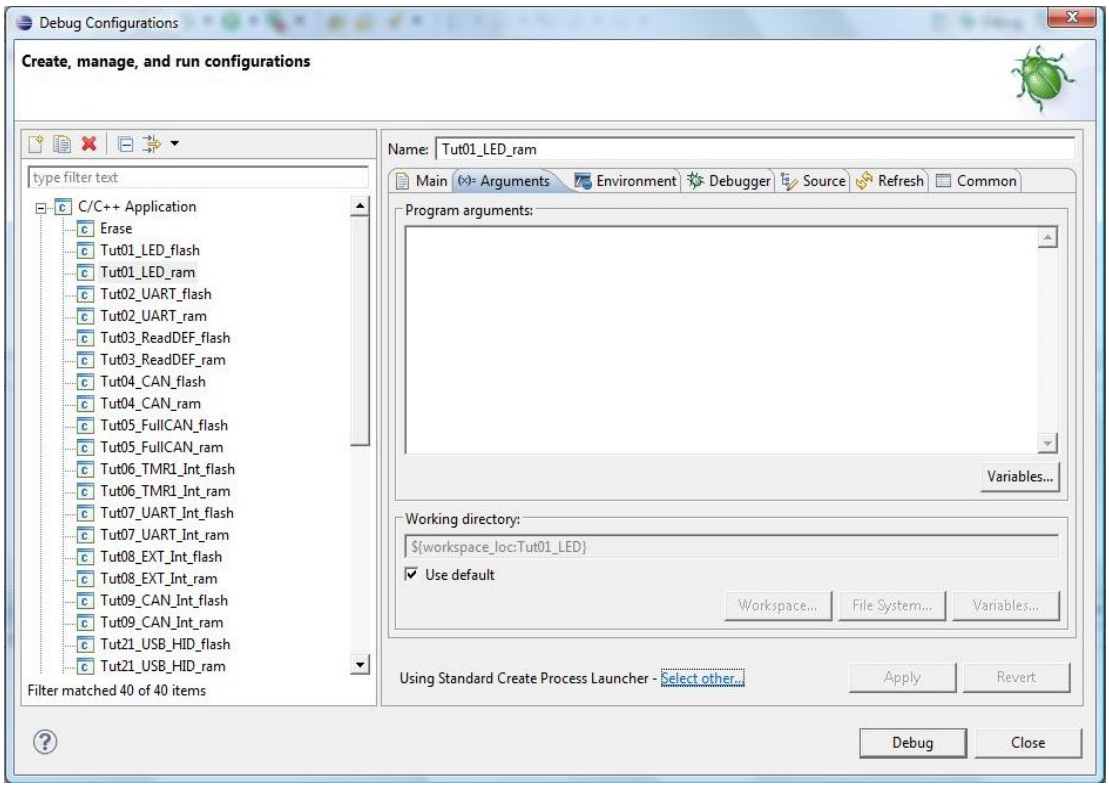
Click on the 'Using [..] Create Process Launcher - Select other...' item in the lower area of the window, a new menu opens:



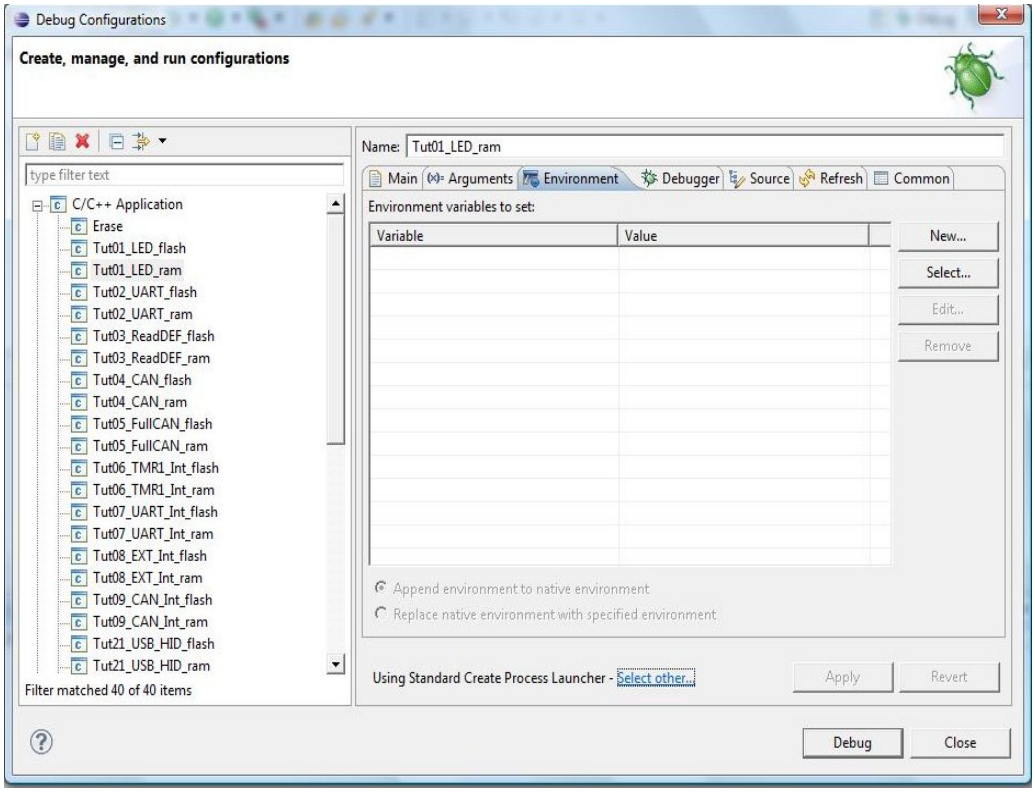
Select the 'Standard Create Process Launcher' (**IMPORTANT**) in the 'Select Preferred Launcher' window and click OK:



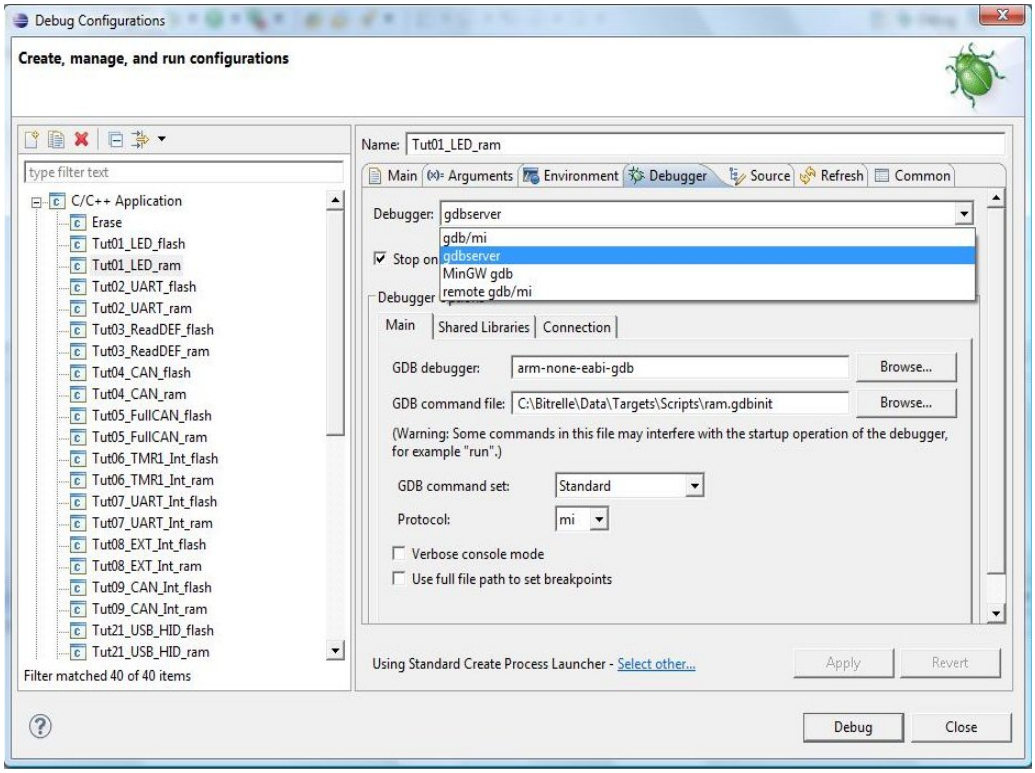
Go to the 'Arguments' slider and check that the 'Program arguments' field is empty and that the 'Use default' checkbox below 'Working directory' is activated, click Apply:



Go to the 'Environment' slider and check that it is empty, click Apply:



Go to the 'Debugger slider' and choose 'gdbserver' (**IMPORTANT**) in the 'Debugger' pop-down menu. Click Apply:



When OpenOCD runs, it is not only accessible via its Telnet server on port 4444 (default), it is also a GDB server on port 3333 (default) and we want to connect to it using the GDB client functionality of Eclipse and the GNU ARM cross tool chain.

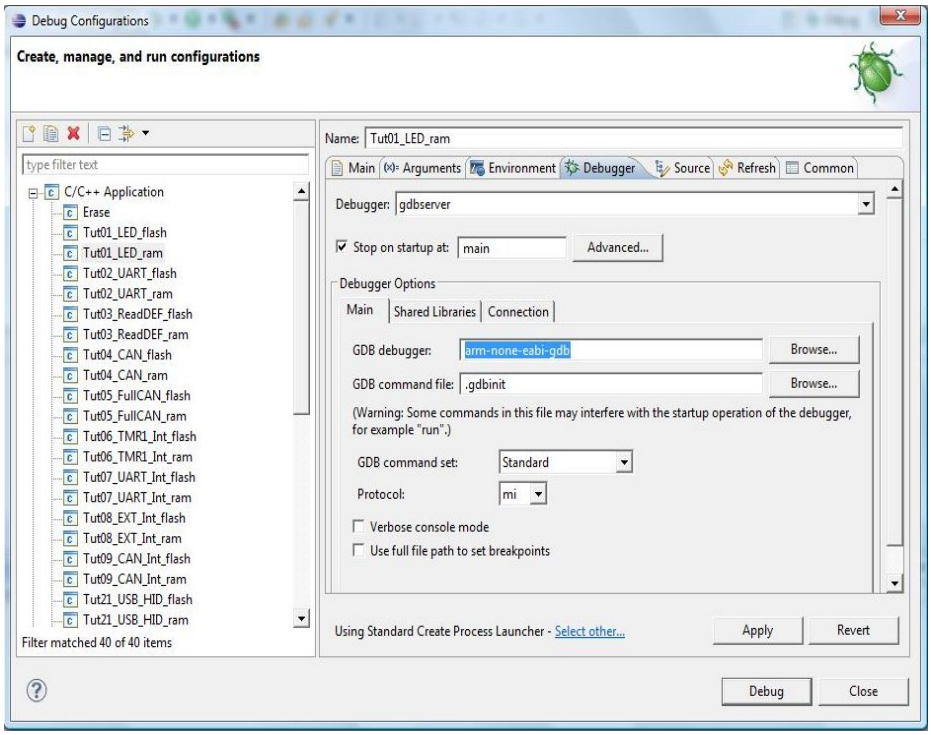
Do not activate the 'Use full file path to set breakpoints' checkbox (**IMPORTANT**). Add prefix

arm-none-eabi-

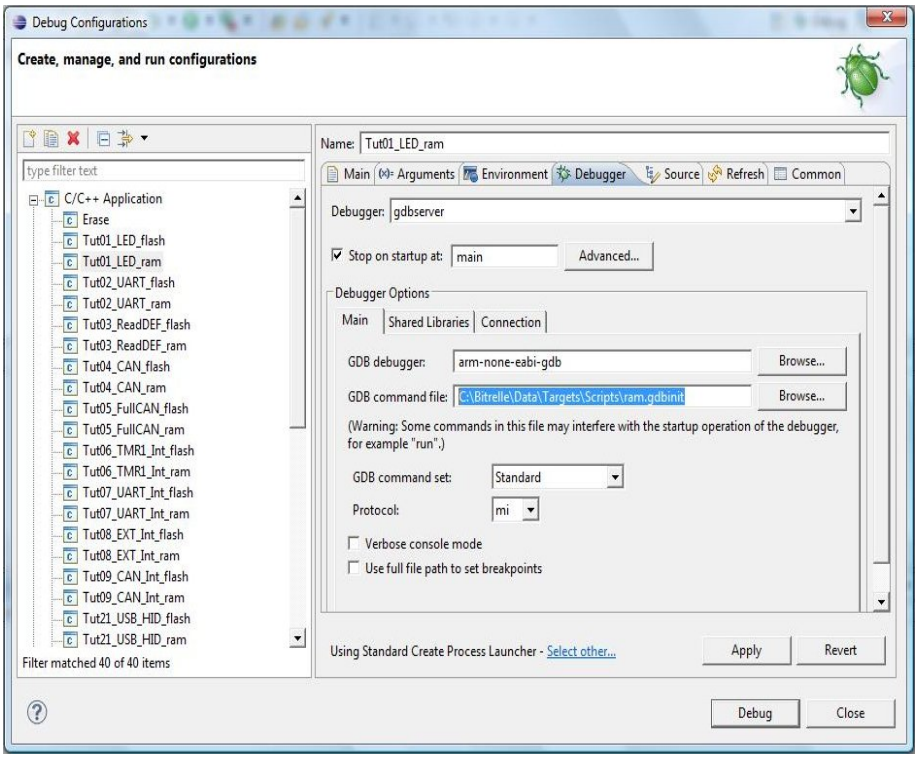
on Windows or otherwise

arm-elf-

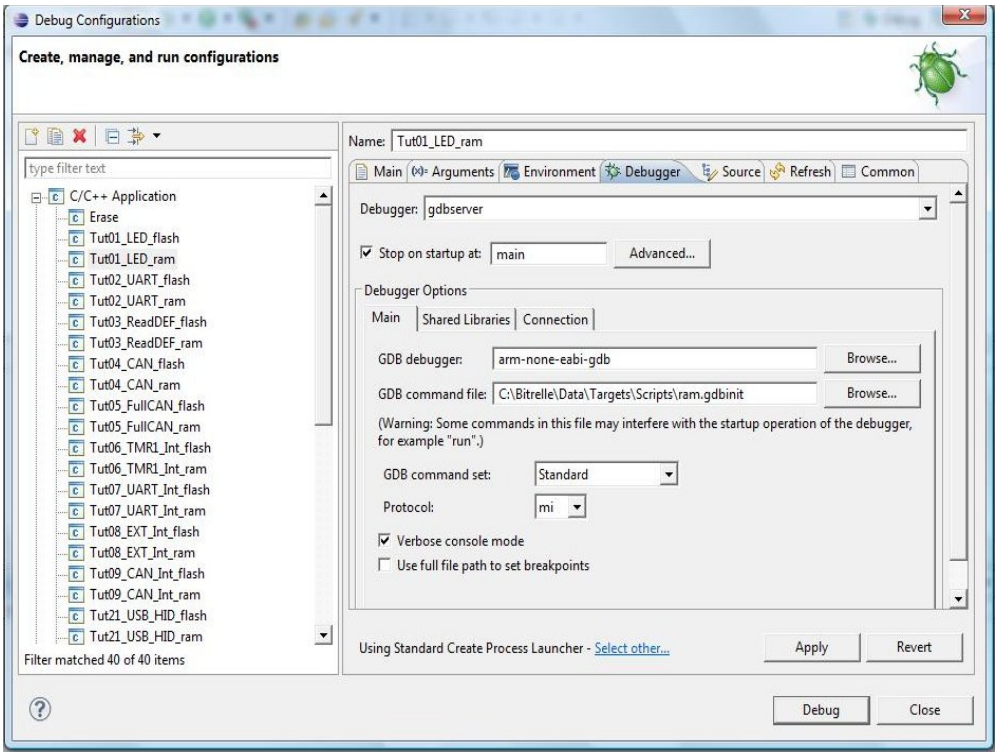
to the default 'GDB debugger' gdb (**IMPORTANT**) and click Apply:



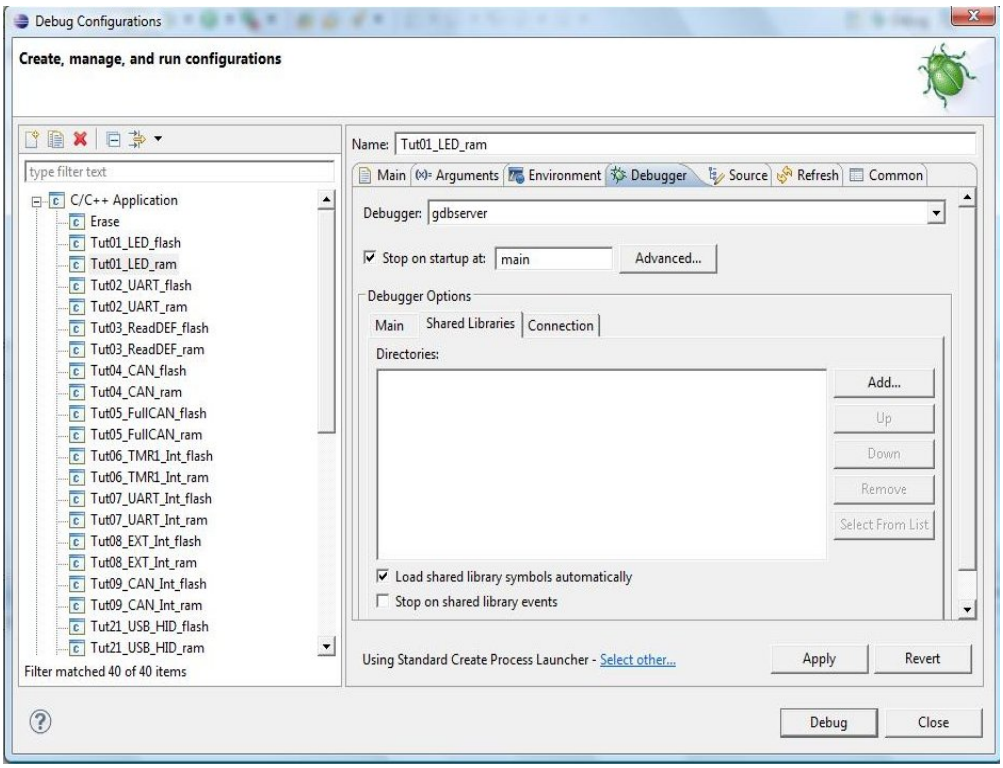
Be sure that the 'Stop on startup at' checkbox is activated and 'main'. Insert 'ram.gdbinit' as the 'GDB command file' (**IMPORTANT**) and click Apply:



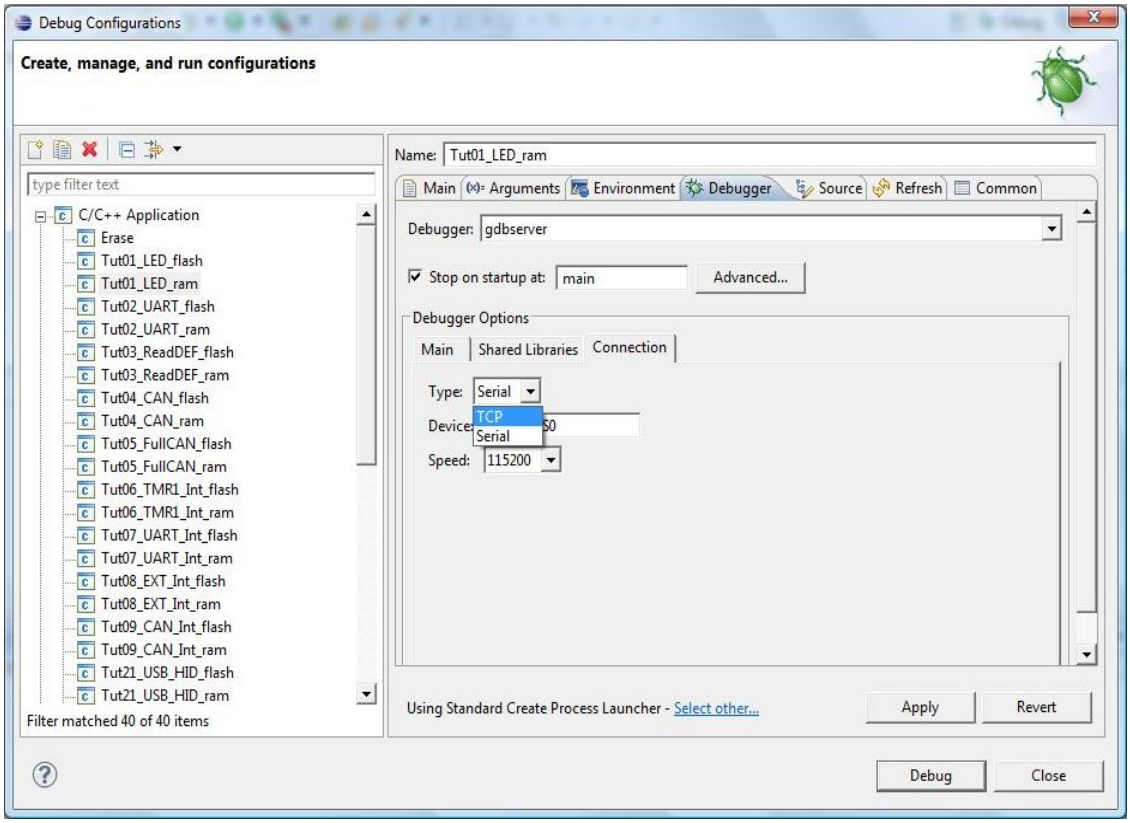
If You want to be able to see all GDB server messages mark checkbox 'Verbose console mode' and click Apply:



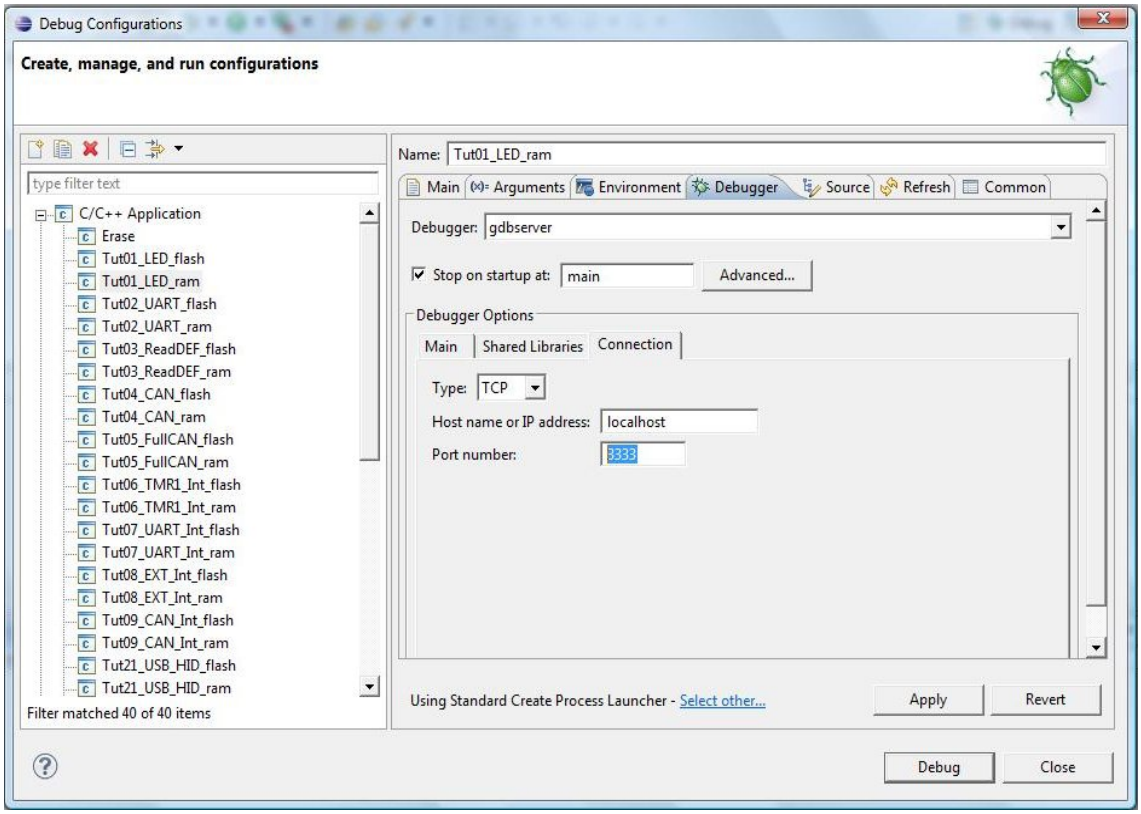
Go to the 'Shared Libraries' sub-slider in the 'Debugger' slider and check that it remains with default values, click Apply if applicable:



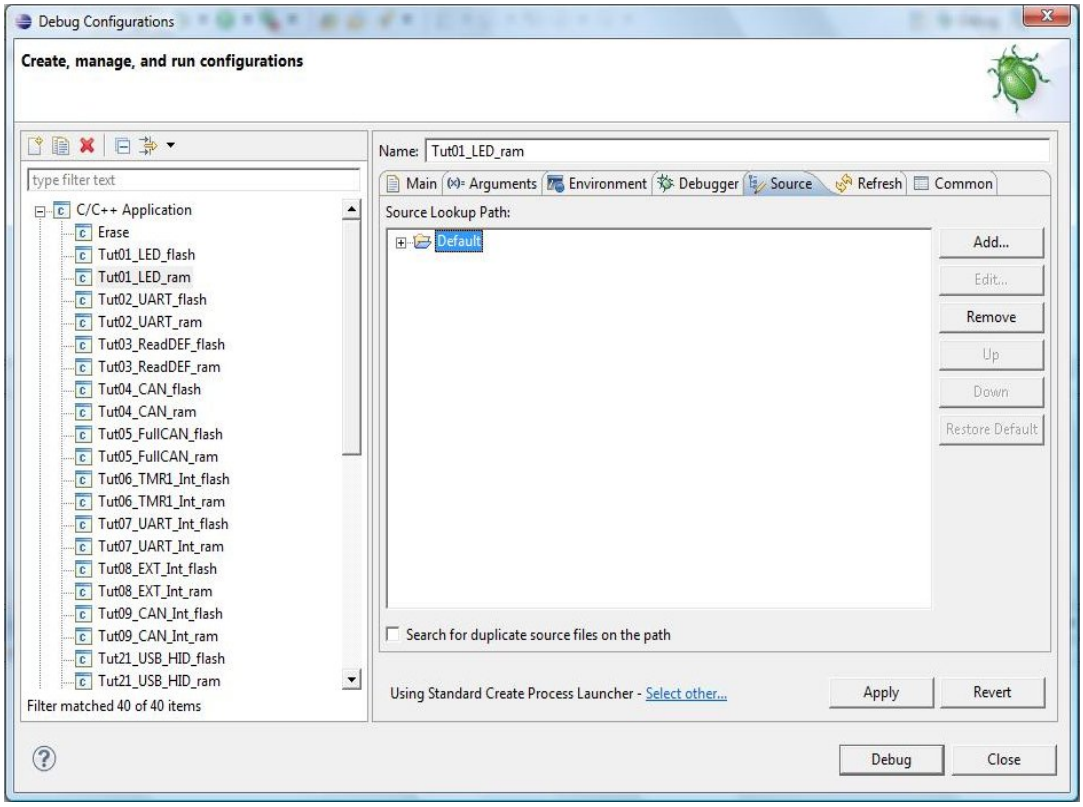
Go to the 'Connection' sub-slider in the 'Debugger' slider and choose 'TCP' in the 'Type' pop-down menu (**IMPORTANT**). Click Apply:



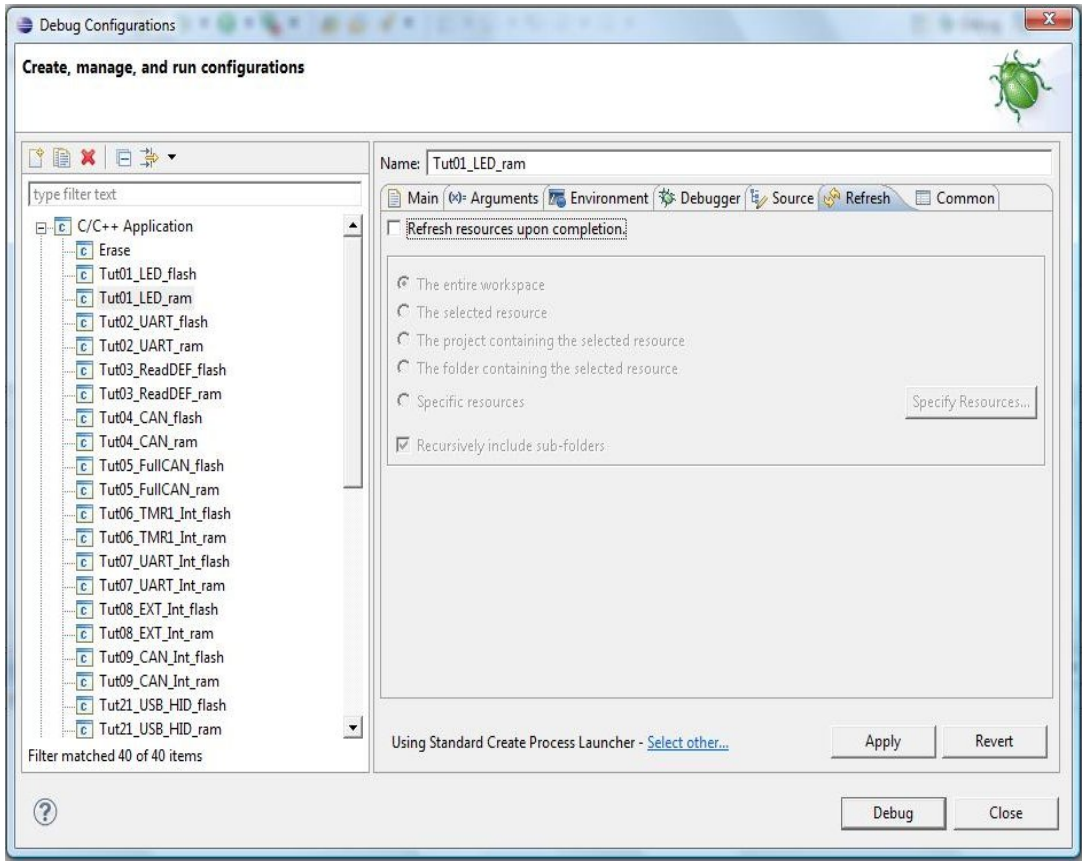
Change 'Port number' to 3333 (**IMPORTANT**). 3333 is the OpenOCD default value, You have to choose another value, if You have changed the GDB port in OpenOCD. Click Apply:



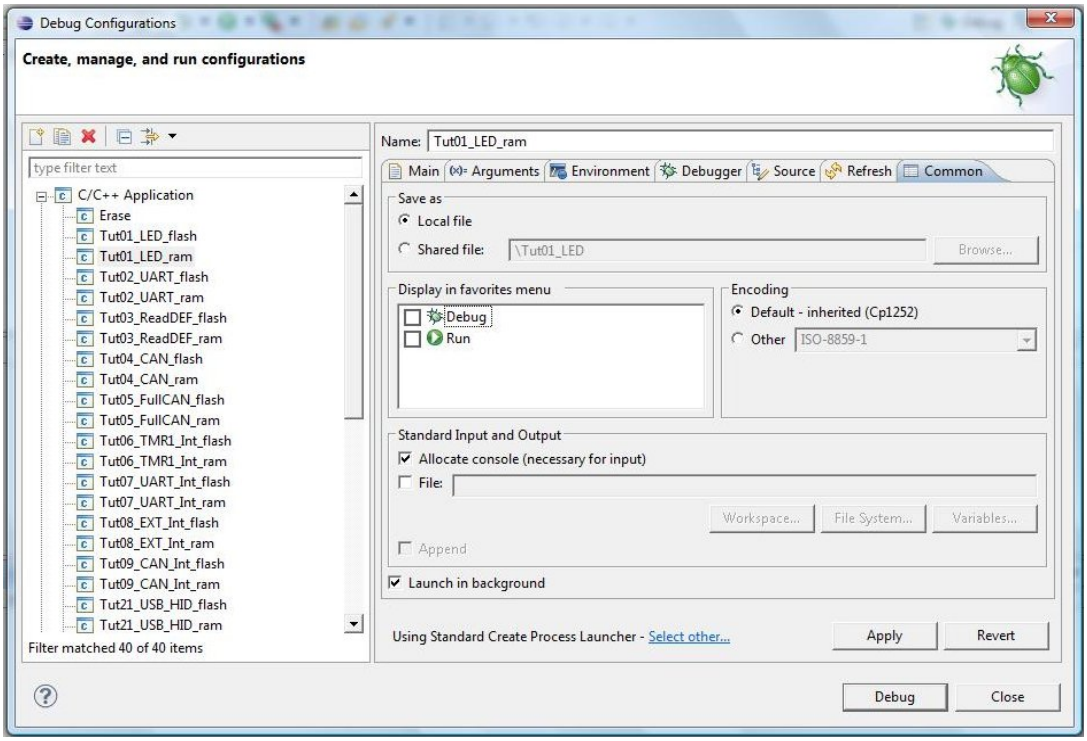
Go to the 'Source' slider, check that it has its default values and click Apply if applicable:



Go to the 'Refresh' slider and check that it is empty, click Apply if applicable:



Go to the 'Common' slider and check its default values. You can disable the 'Launch in background' checkbox if You like. Then You will see a progress bar when launching the debugger. Click Apply if applicable. Then click Close.



7 Debugging with Eclipse

First, You must be sure, that You have build Your project successfully. If not go back to chapter 'Building Projects' check all preconditions and do all steps precisely.

Without correct debug settings it is also not possible to run and debug the project. So if You didn't so far, go to chapter 'Eclipse Debug Settings' and do all debug settings steps precisely.

Open a new terminal and type in the command line:

openocd-0.5.0.exe -f target/lpc2129.cfg -f interface/jtagkey2.cfg

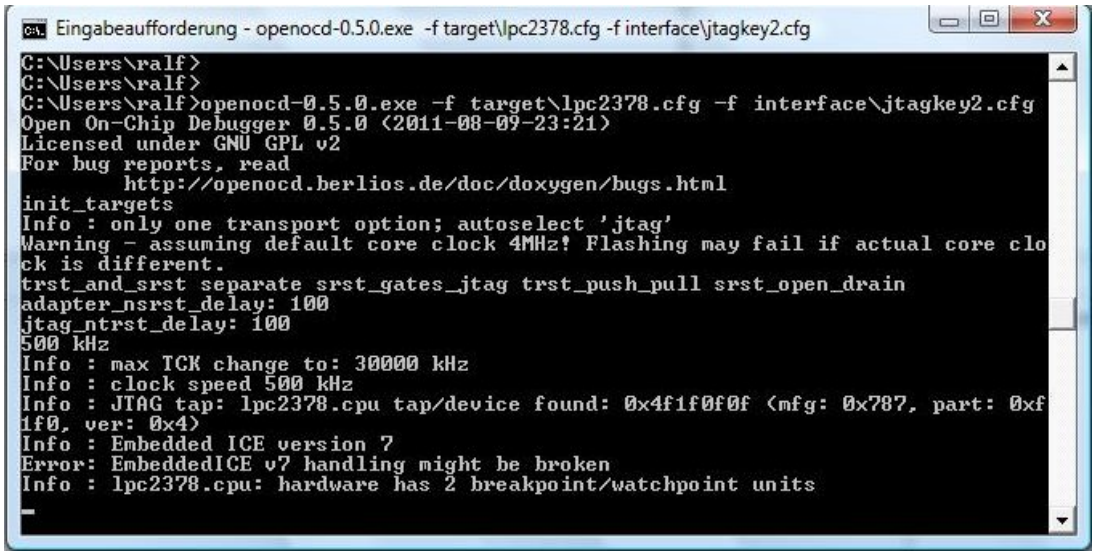
or

openocd-0.5.0.exe -f target/lpc2378.cfg -f interface/jtagkey2.cfg

This will start OpenOCD. Check if it is running in the background (i.e. the prompt did not return). Be sure that:

- JTAG device is connected
- drivers for JTAG device have been installed correctly

If You have problems, see chapter 'Running JTAG debugger' or read again the Windows/Linux Install Manual, chapter 'JTAG Debugger'.



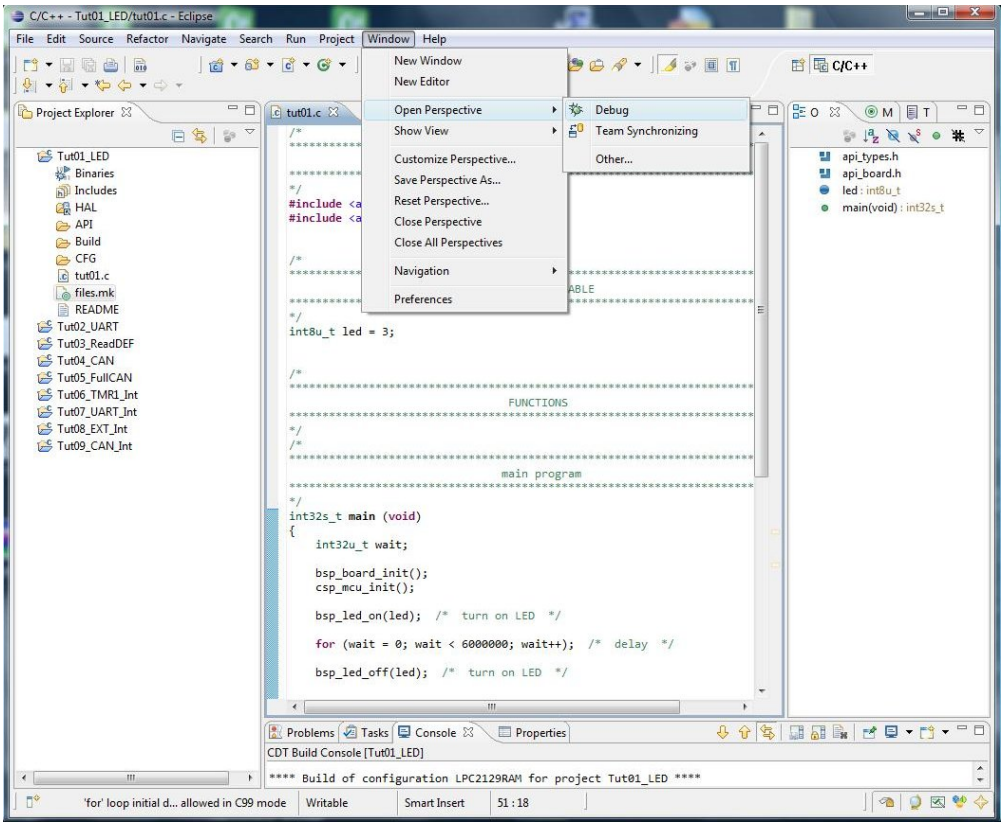
If OpenOCD is running make additionally sure that Your

- RAM build has finished successfully with the correct settings
- RAM image file 'Build/LPC2129RAM/tut01_LED_ram.elf' exists
- 'Tut01_LED_ram' debug settings are correct
- 'ram.gdbinit' is correct (see 'Annex GDB Init Files')

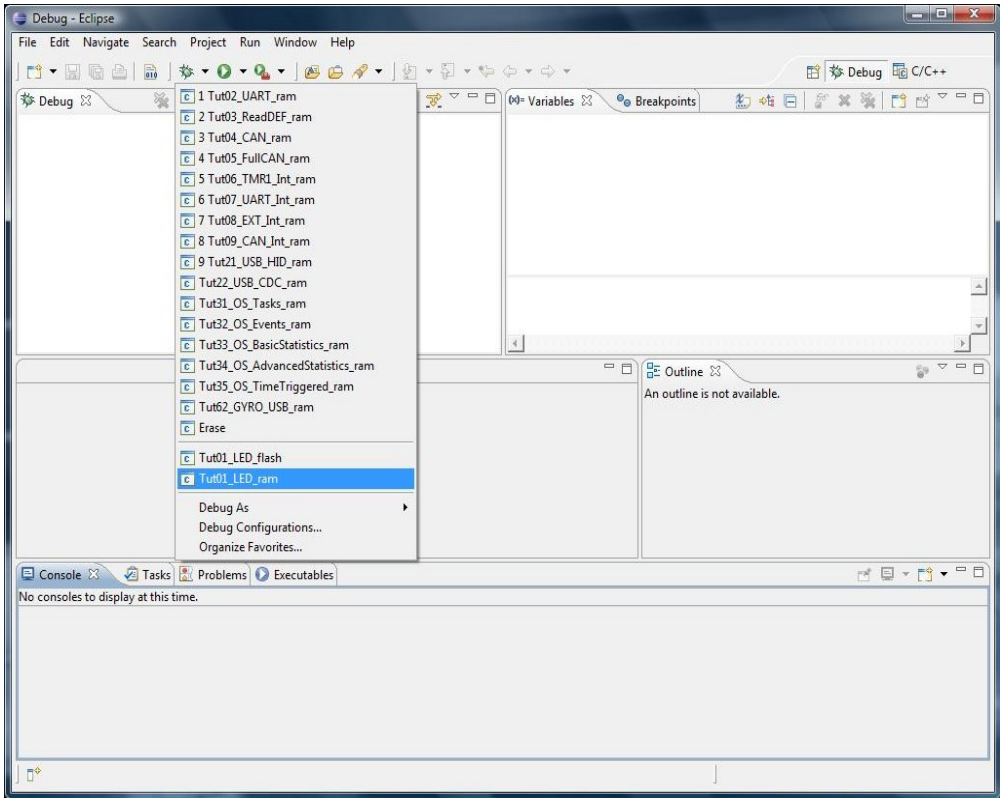
BE SURE THAT YOU HAVE BUILD THE PROJECT, HAVE DONE THE DEBUG SETTINGS AND HAVE STARTED OPENOCD. OTHERWISE THE PROGRAM CAN NOT BE DEBUGGED.

Now, You are ready to proceed.

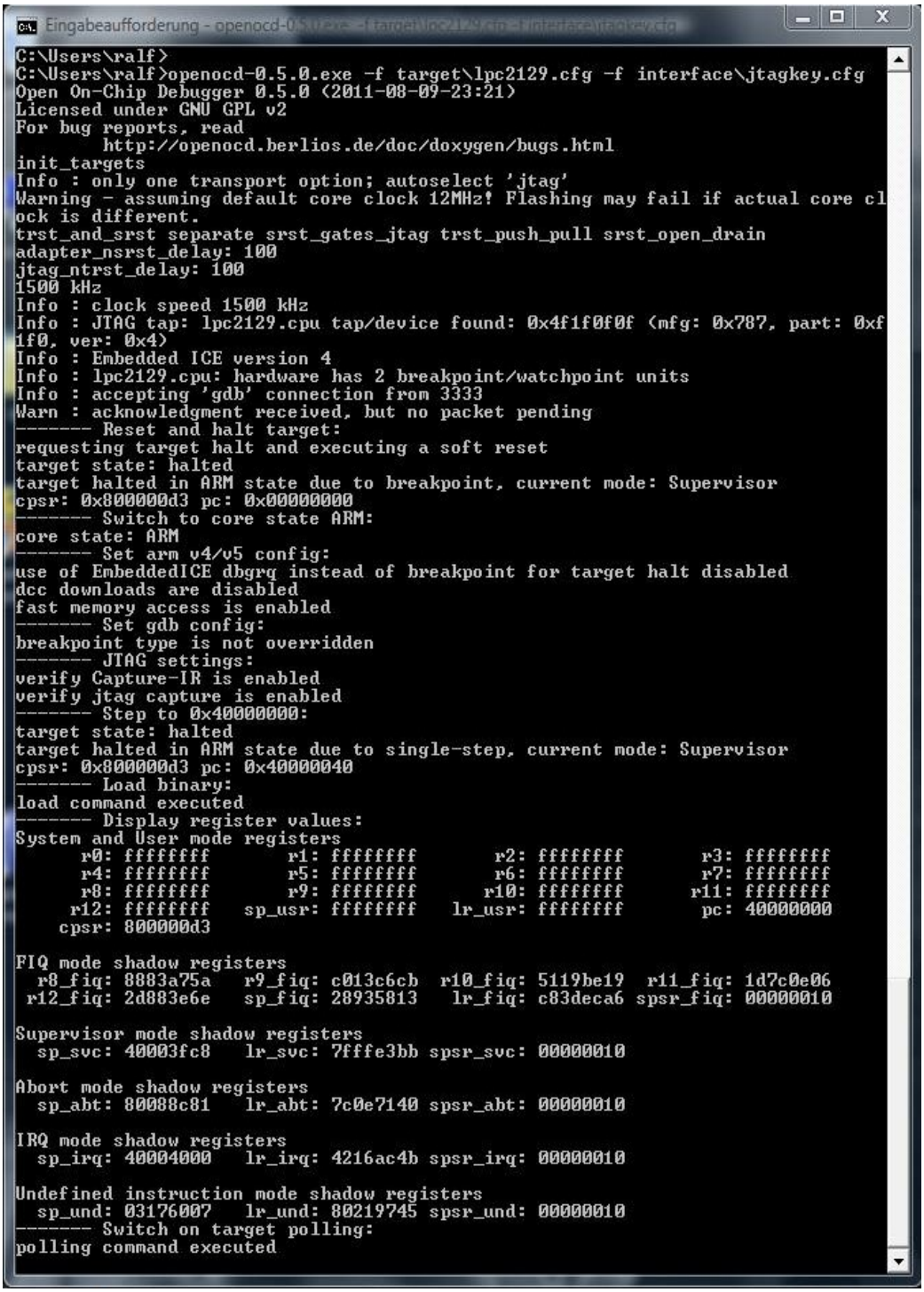
Go to the 'Debug' perspective button on the upper right side and click. If it is not applicable then go to the 'Windows' menu and choose 'Open Perspective' -> 'Debug':



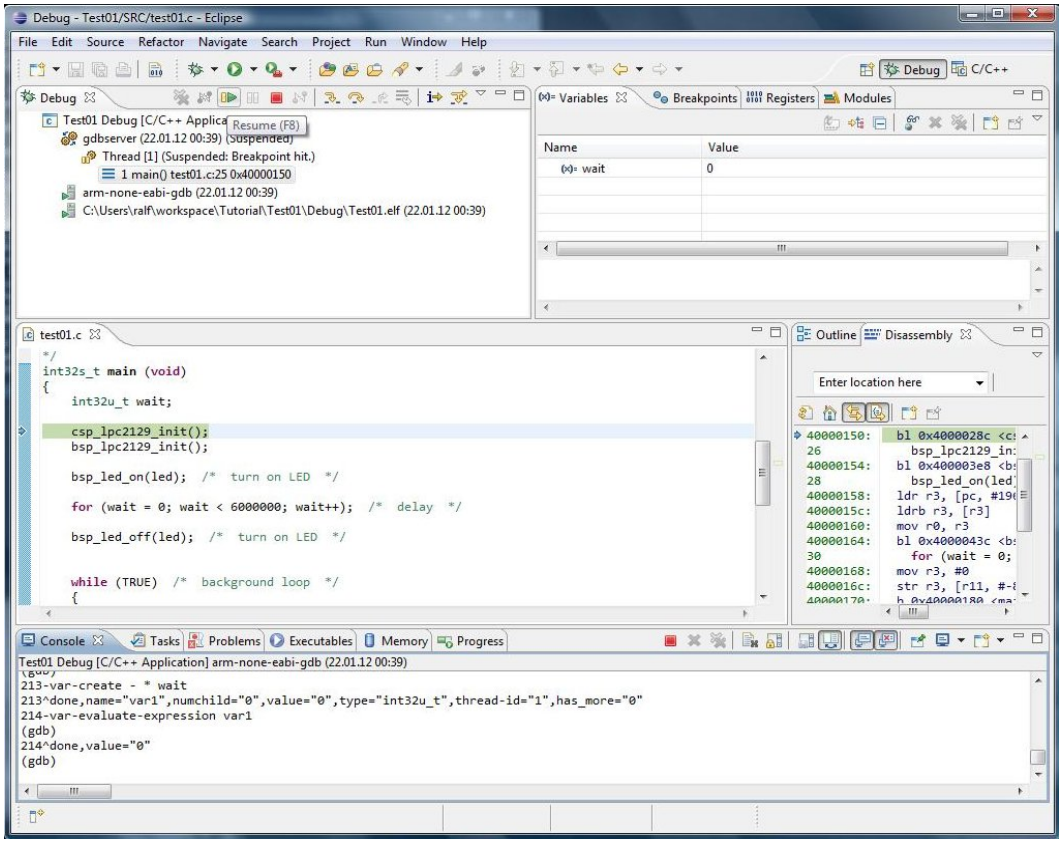
The 'Debug' perspective opens and should look like that. Click the bug button or the down-arrow right of the bug button and choose Your debug configuration. In our example this is 'Tut01_LED_ram':



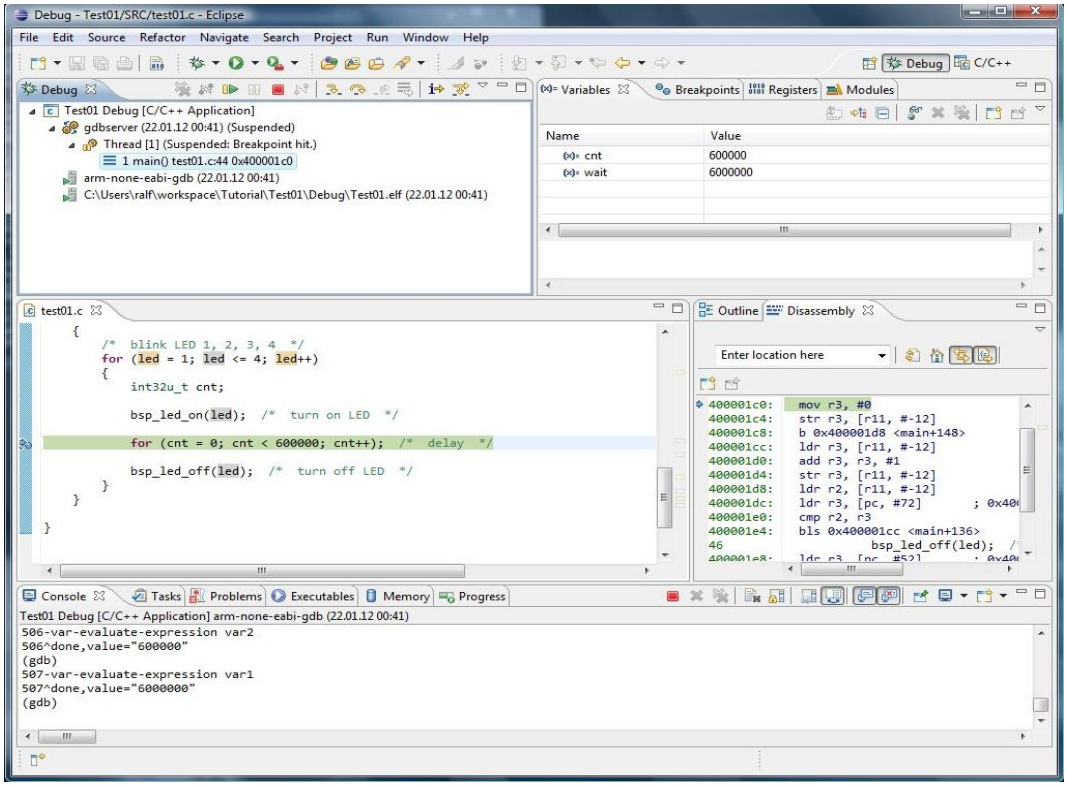
Have a look to the OpenOCD window. OpenOCD should output something like that and should not return the prompt, i.e. the program should remain in the background:



A debug session opens in the 'Debug' window and the program stops on startup at the first instruction of the main program. You can start execution with clicking on the green 'Resume' arrow or by hitting the F8 key:



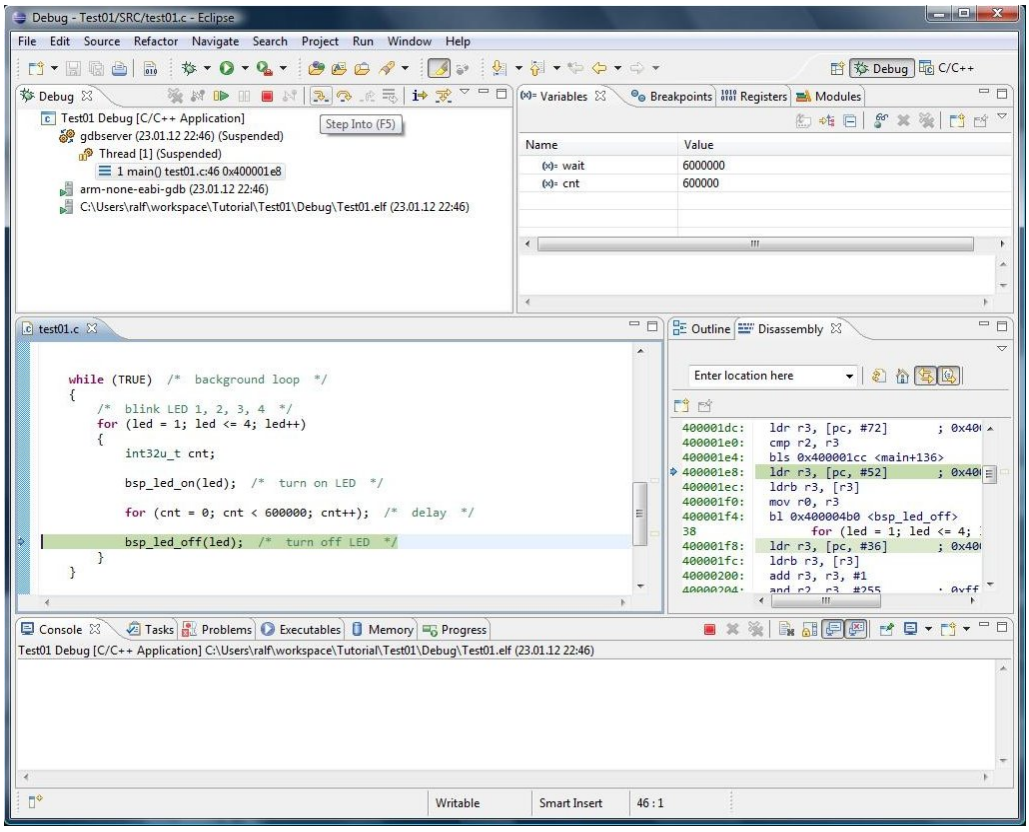
You can add or remove breakpoints by double-clicking on the left bar:



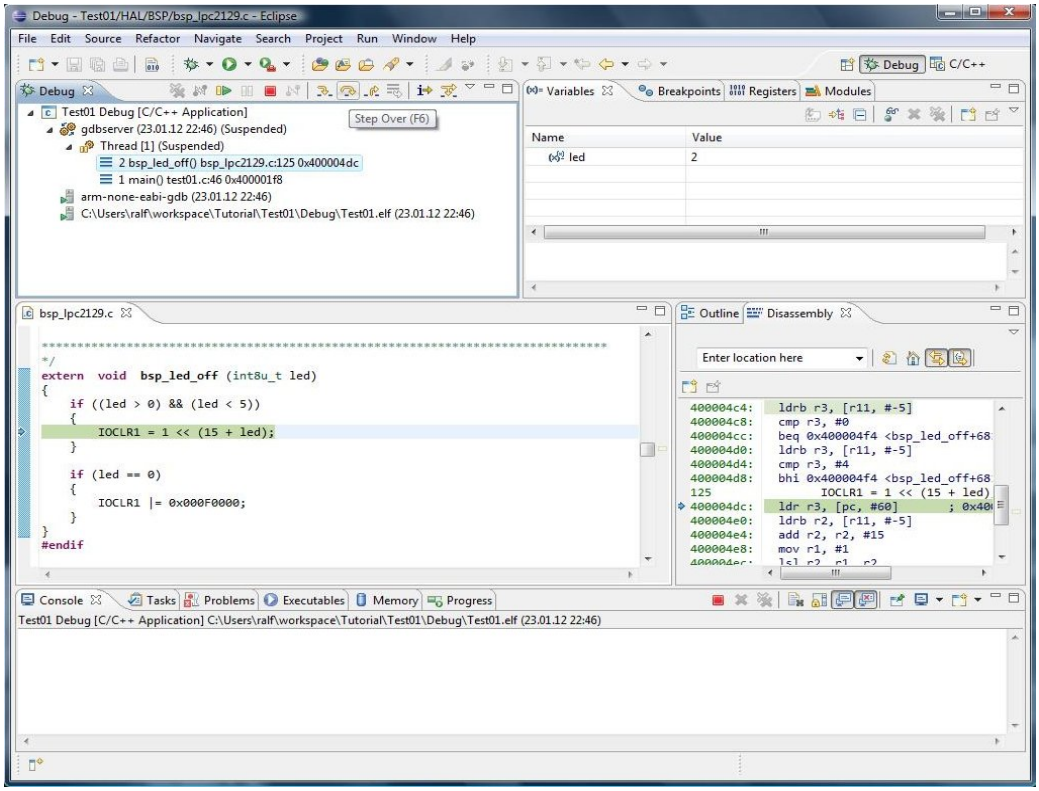
Next we will show You how to

- step into C functions
- step over single C commands
- return out of C functions
- view the ARM7 registers
- view variables values
- run to a particular command line
- terminate and remove a debug session

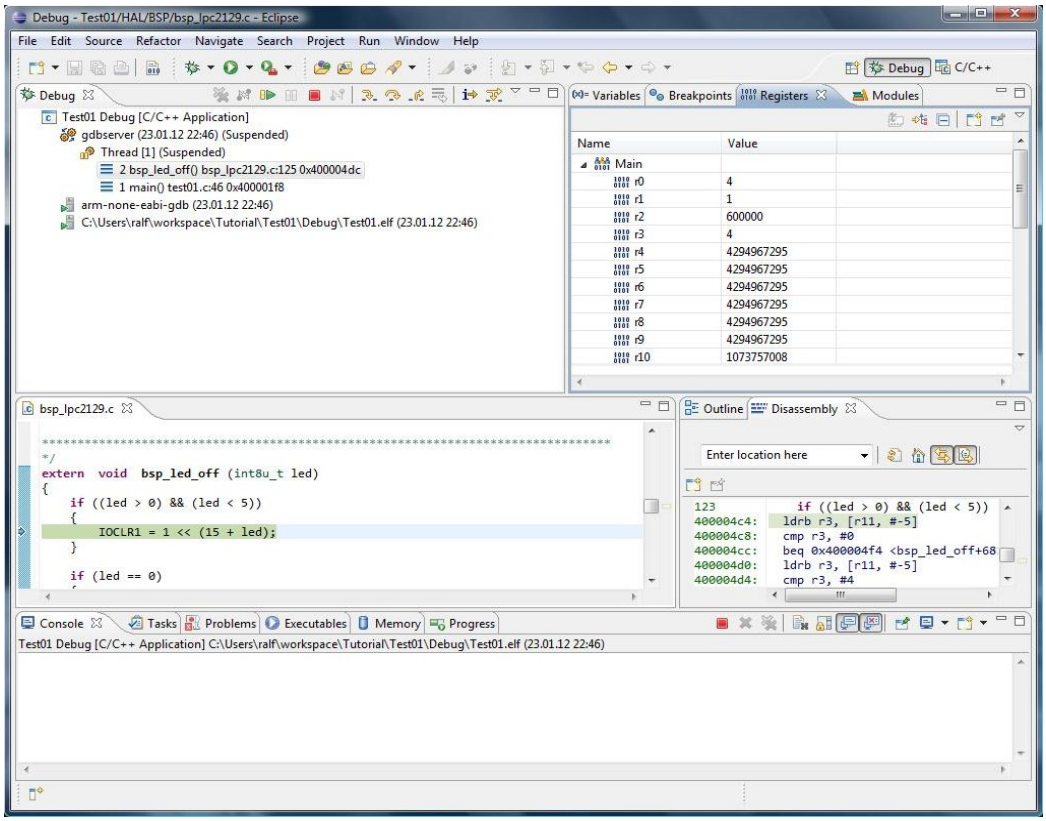
If You stop on a C function call, You can step into the function with the yellow 'Step Into' arrow or by hitting the F5 key.



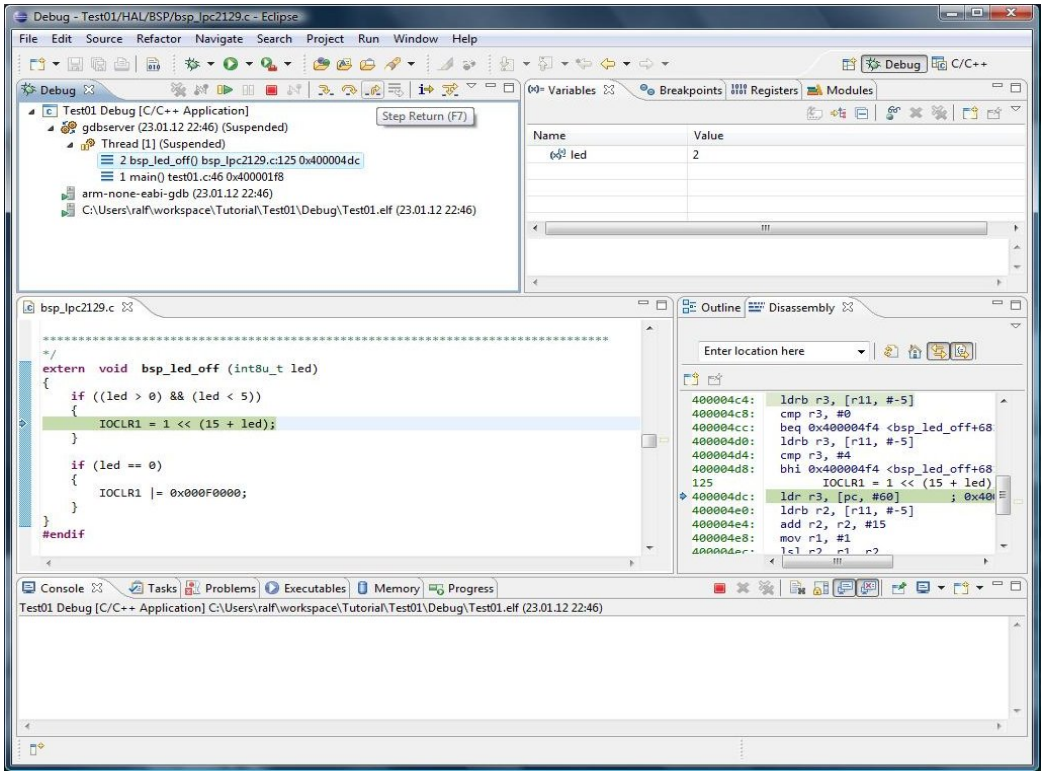
You can step over single C commands by clicking the yellow 'Step over' arrow or by hitting the F6 key:



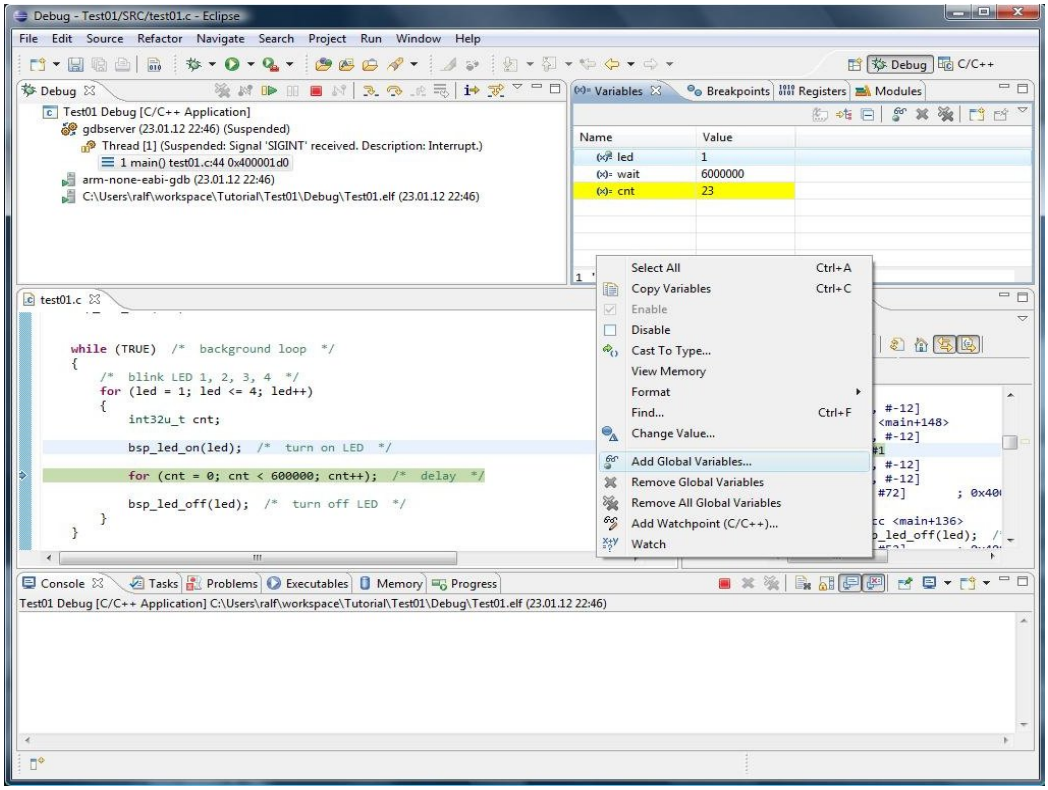
If You click on the 'Registers' slider You can see the ARM7 register values at any time:



You can leave any C function immediately by clicking on the yellow 'Step Return' arrow or by hitting the F7 key:



If You click on the 'Variables' slider You can see all local variables of the current context at one sight. You can add global variables to watch by clicking on the variables window with the right mouse button and choosing 'Add Global Variables...'. Then a menu opens where You can choose the global variable to watch:



Next we will show You the 'Run to Line' functionality of the debugger. Be aware that the 'Run to Line' functionality is not as stable as the other functions.

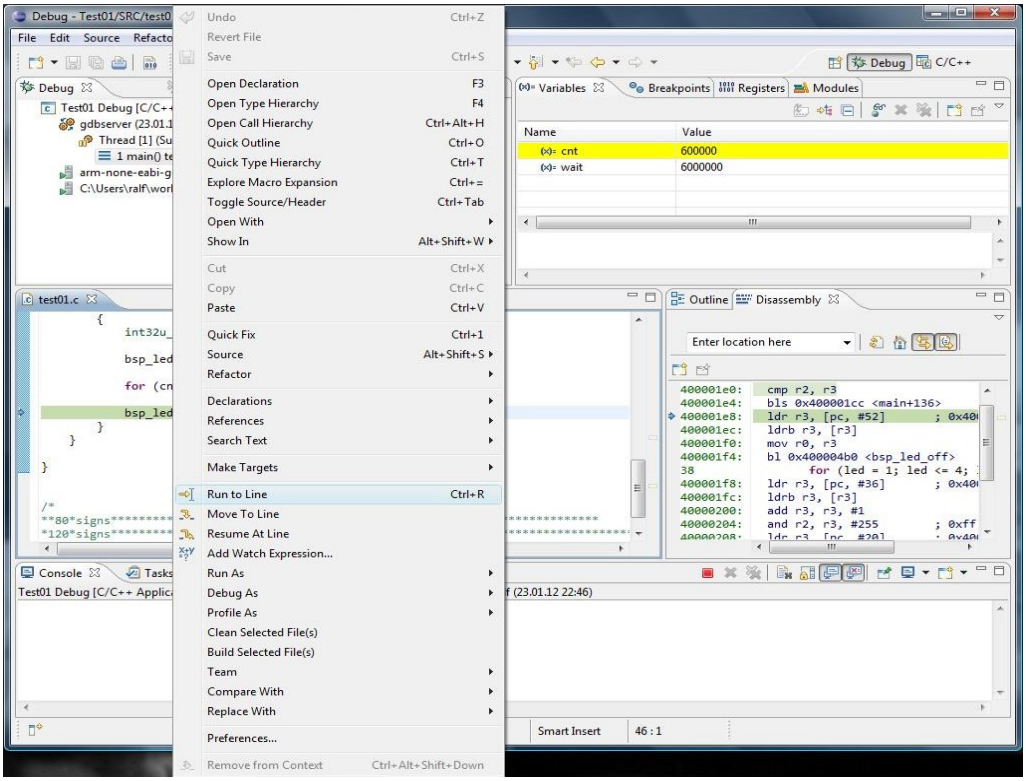
After the program is halted either by clicking the yellow 'Suspend' bars symbol or by hitting on a breakpoint, You are able to let the program run to a defined command line.

Now, You mark the line. Its getting light blue. Press the right mouse button on this light blue highlighting bar and choose 'Run to Line':

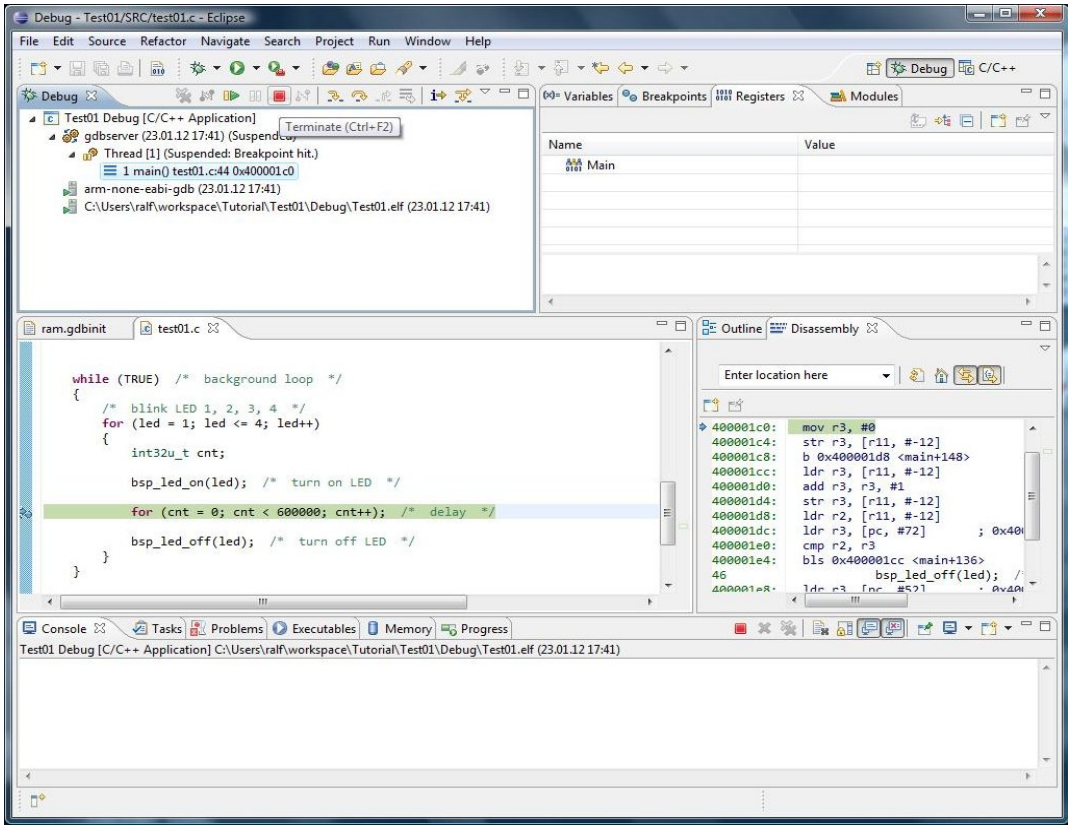
If 'Run to Line' doesn't work follow the restart procedure at the end of this chapter. Repeat this procedure as many times until 'Run to Line' works properly.

If 'Run to Line' works once, it normally works well until the end of the debugging session.

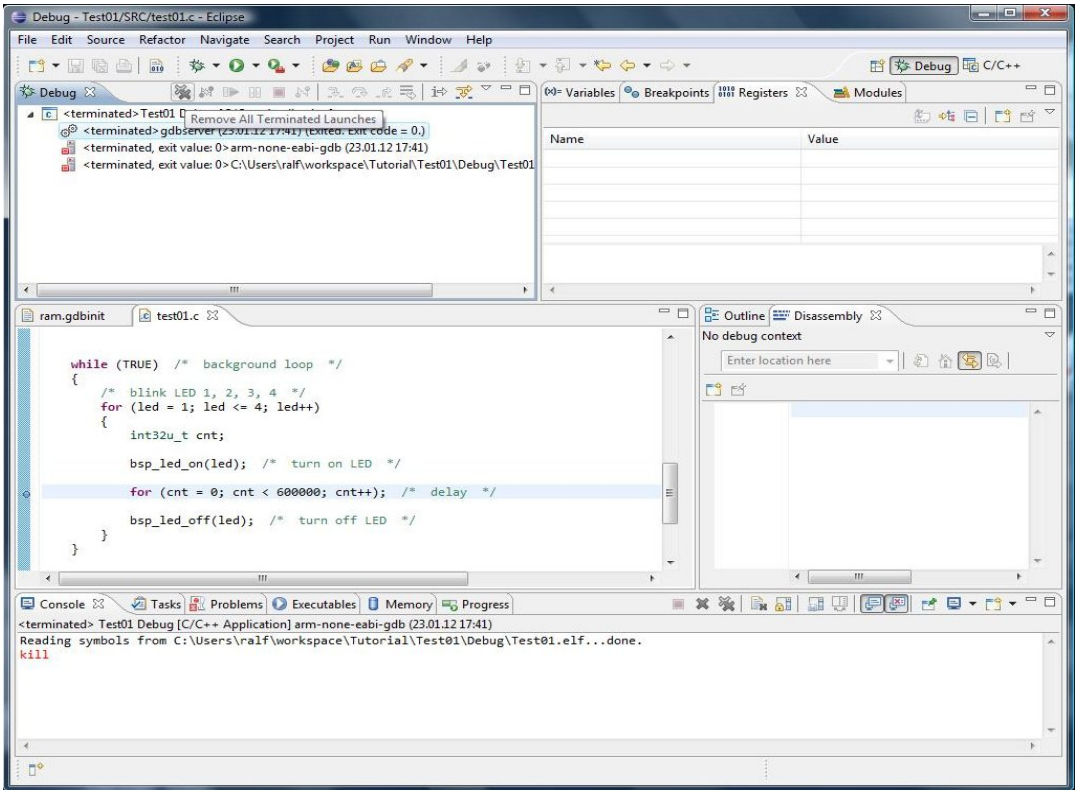
Tip: To increase the stability of the 'Run to Line' function it is advantageous that You have marked at least three breakpoints in the code.



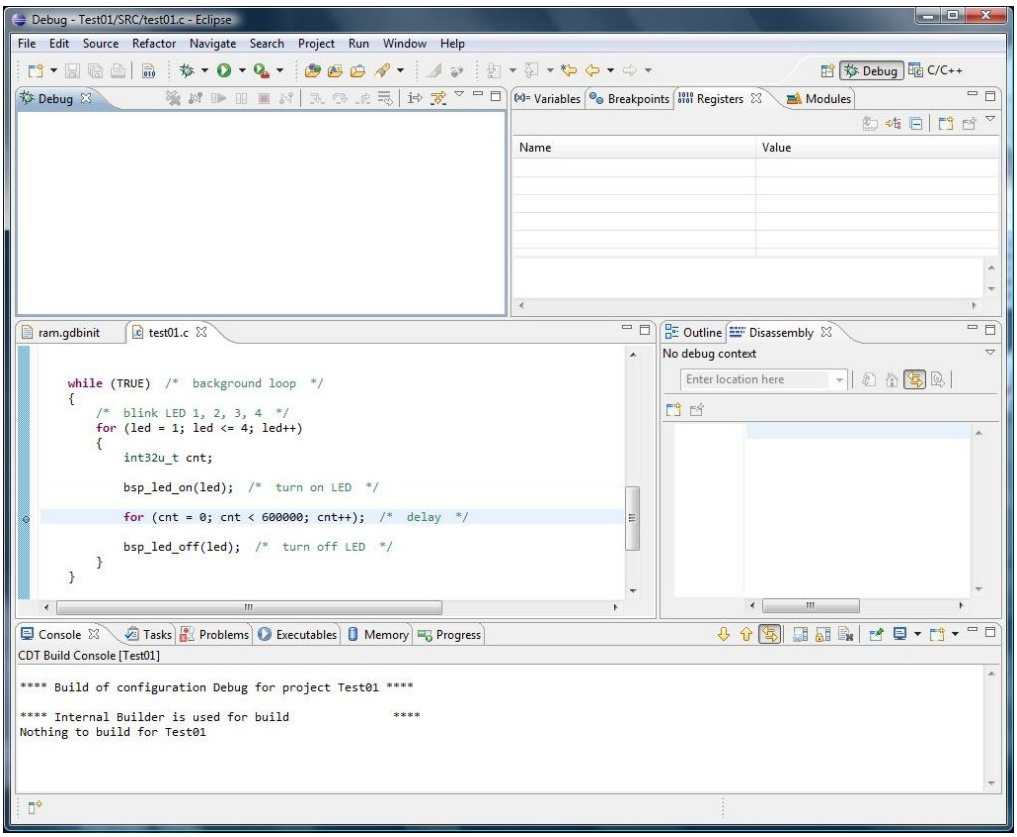
To stop a debug session You can click on the red 'Terminate' rectangle :



To delete a debug session from the 'Debug' window You must click on the 'Remove all Terminated Launches' double-cross:



The 'Debug' session window should be empty after clicking the double-cross. If not, try again to stop and remove the debug session, until it is empty. Sometimes it helps to stop first OpenOCD by clicking CTRL-C in the OpenOCD window:



If the debugger stalls or has an odd behavior, it is time to bring back the GDB debugger, the debug monitor (OpenOCD), the PC drivers and the JTAG device adapter to a well defined condition. Do the following restart procedure:

Restart procedure:

1. Terminate (red rectangle) and remove (gray double-cross) the debugging session
2. Stop OpenOCD with the key combination CTRL-C
3. Unplug the Debugger USB device to the PC
4. Switch the target board off and wait a view seconds
5. Switch the target board on again
6. Replug the Debugger USB device to the PC and wait until the drivers have reconnected automatically (Windows user hear a ping)
7. Restart OpenOCD (press Up-Arrow and press Return in the OpenOCD command window)
8. Restart the Debug session by clicking on the bug icon (or click on one of Your debug configurations in the bug's pop-down menu)

This steps will cost You about 15 seconds, but in many cases it is sufficient to do only a subset of this restart procedure steps. Try out!

8 Debugging with GDB command line

If You ever get into a very deep embedded debugging problem where Eclipse seems not to provide the desirable debugging results, You don't have to give up hope. Actually You are a very lucky person, because beneath Your Eclipse Debugging system lies as a basis on of the most powerful debuggers available: the GDB command line debugger.

Open a command line terminal wherever You want and type into the command line:

openocd-0.5.0.exe -f target/lpc2129.cfg -f interface/jtagkey.cfg

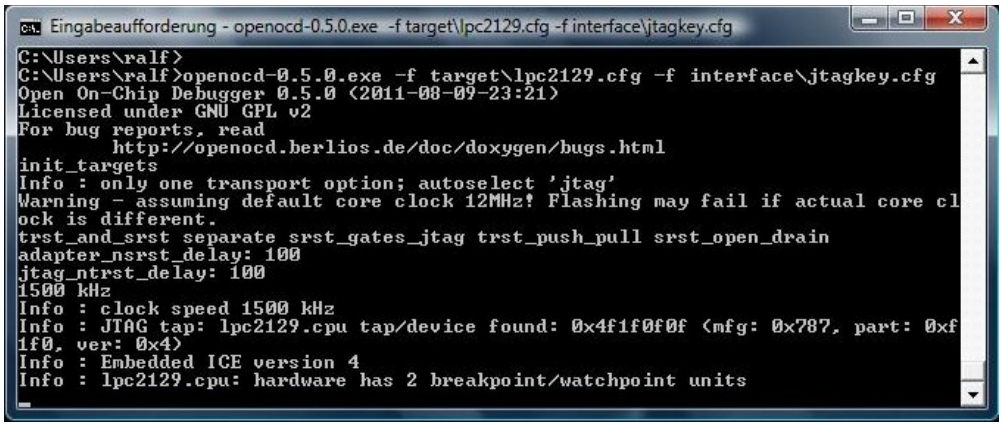
or

openocd-0.5.0.exe -f target/lpc2378.cfg -f interface/jtagkey2.cfg

This will start OpenOCD. Check if it is running in the background (i.e. the prompt did not return). Be sure that:

- JTAG device is connected
- drivers for JTAG device have been installed correctly

If You have problems, see chapter 'Running JTAG debugger' or read again the Windows/Linux Install Manual, chapter 'JTAG Debugger'.



If OpenOCD is running make additionally sure that Your

- RAM build has finished successfully with the correct settings
- RAM image file 'Build/LPC2129RAM/tut01_LED_ram.elf' exists
- 'ram.gdbinit' is correct (see 'Annex GDB Init Files')

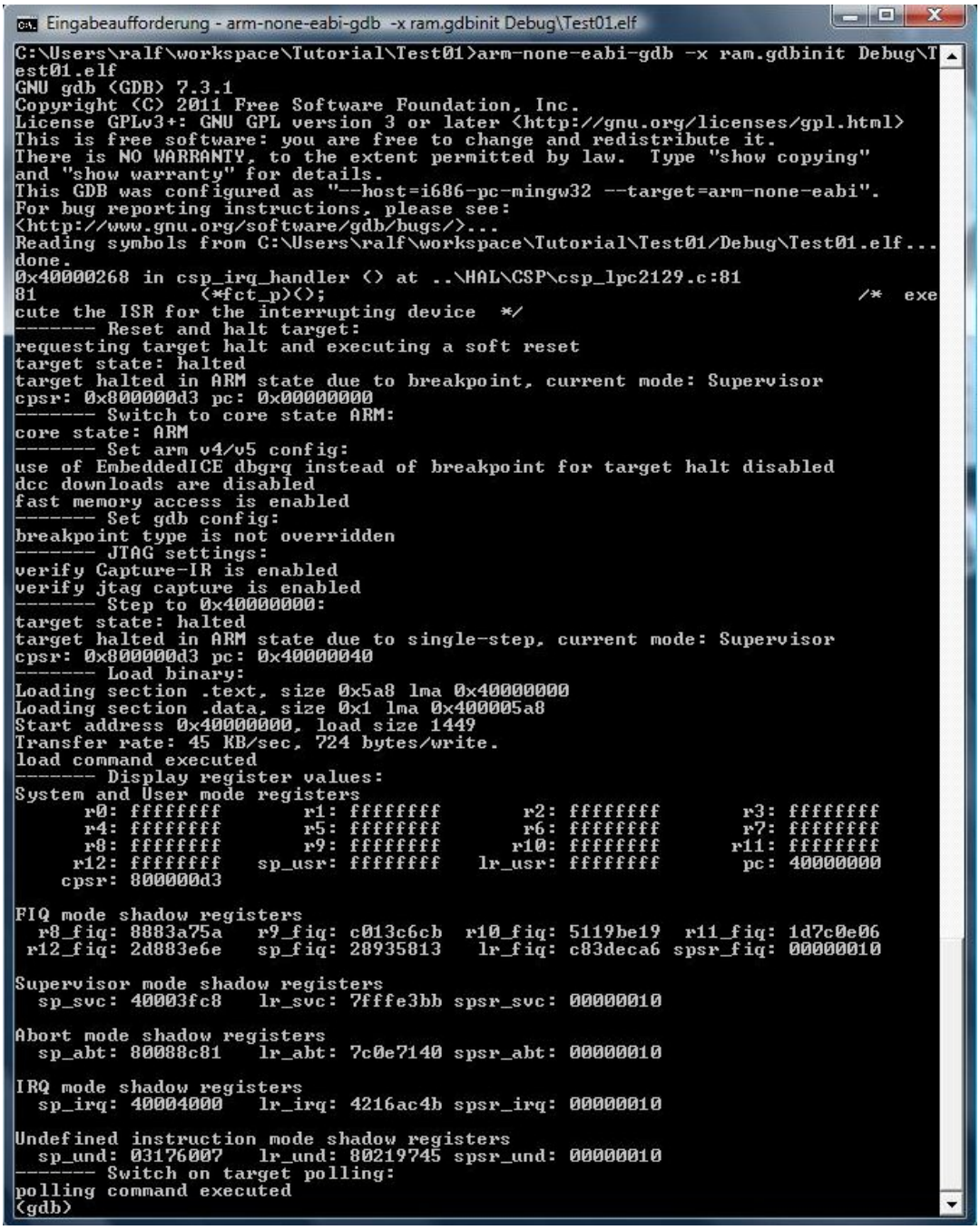
Open a new command line terminal and change the directory (with the cd command) to Your working directory, e.g. 'Tut01_LED' (**IMPORTANT**). If You are sure You are in the correct working directory type into the command line:

arm-none-eabi-gdb -x ram.gdbinit Build/LPC2129RAM/tut01_LED_ram.elf

on Windows or otherwise

arm-elf-gdb -x ram.gdbinit Build/LPC2129RAM/tut01_LED_ram.elf

GDB will start, read in the commands from the GDB initialization file 'ram.gdbinit' and output the results in the terminal. Then the GDB command prompt appears:



We will now show You some important GDB commands. See also the table at the end of this chapter. To stop the execution of the program at any time You use the key combination:

<CTRL> + c

To end the GDB debugging program You type 'quit' or

q

and confirm the command with 'y'.

Tip: If 'quit' doesn't work to end GDB go to the Windows Task manger with <CTRL> + <ALT> + <DEL> and kill the GDB process.

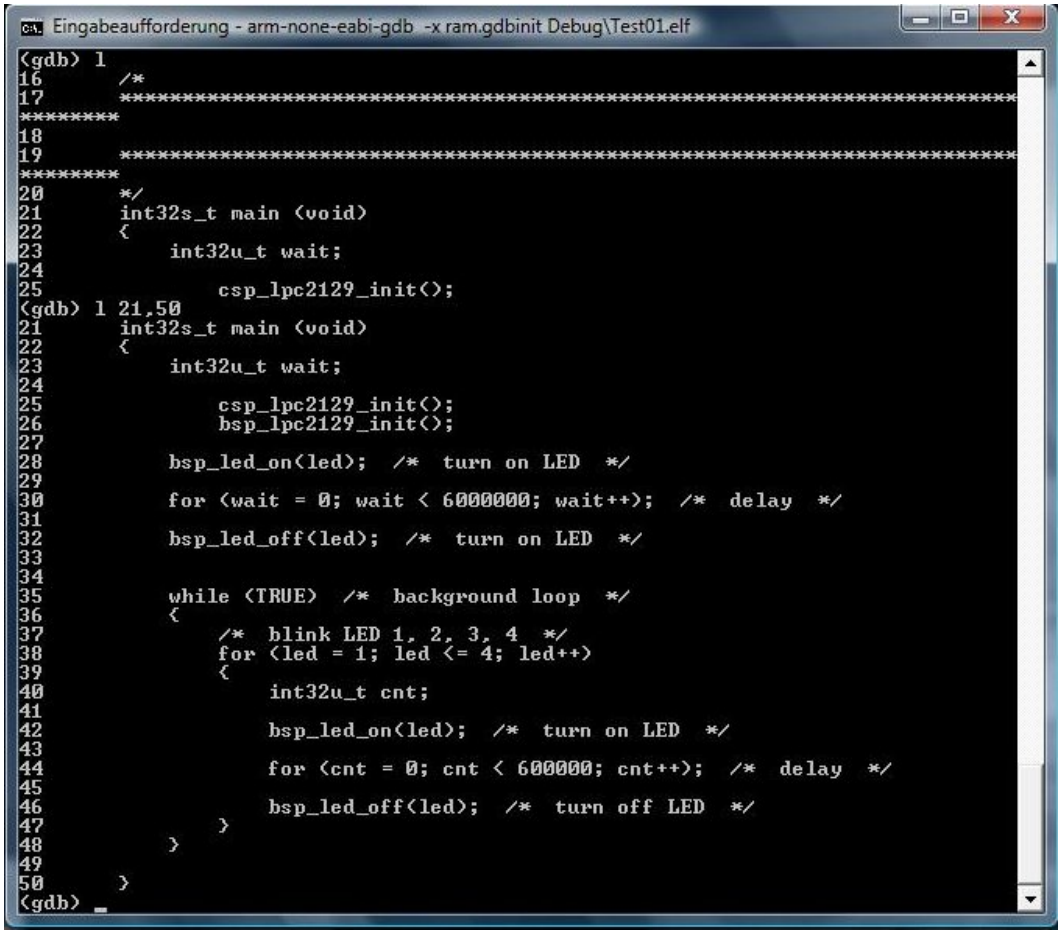
You can have a look on the source code by typing

l

for 'line' or

l [startline], [endline]

for showing the source code of a certain range of code lines:



You can insert breakpoints by typing

b [line]

or hardware breakpoints (max. 2) by typing

hb [line]

You can see a list of all breakpoints and their states via

i b

that means 'info breakpoint' You can start or continue the program by typing

c

You can delete a breakpoint with

d [number]

or clear the breakpoint at a certain code line with

cl [line]

You can disable a breakpoint by typing

dis [number]

or You can type

en [number]

to enable the breakpoint again:

It's possible to let the program run **u**ntil a certain line with

u [line]

to step to the **n**ext command and **n**ot into substructures with

n

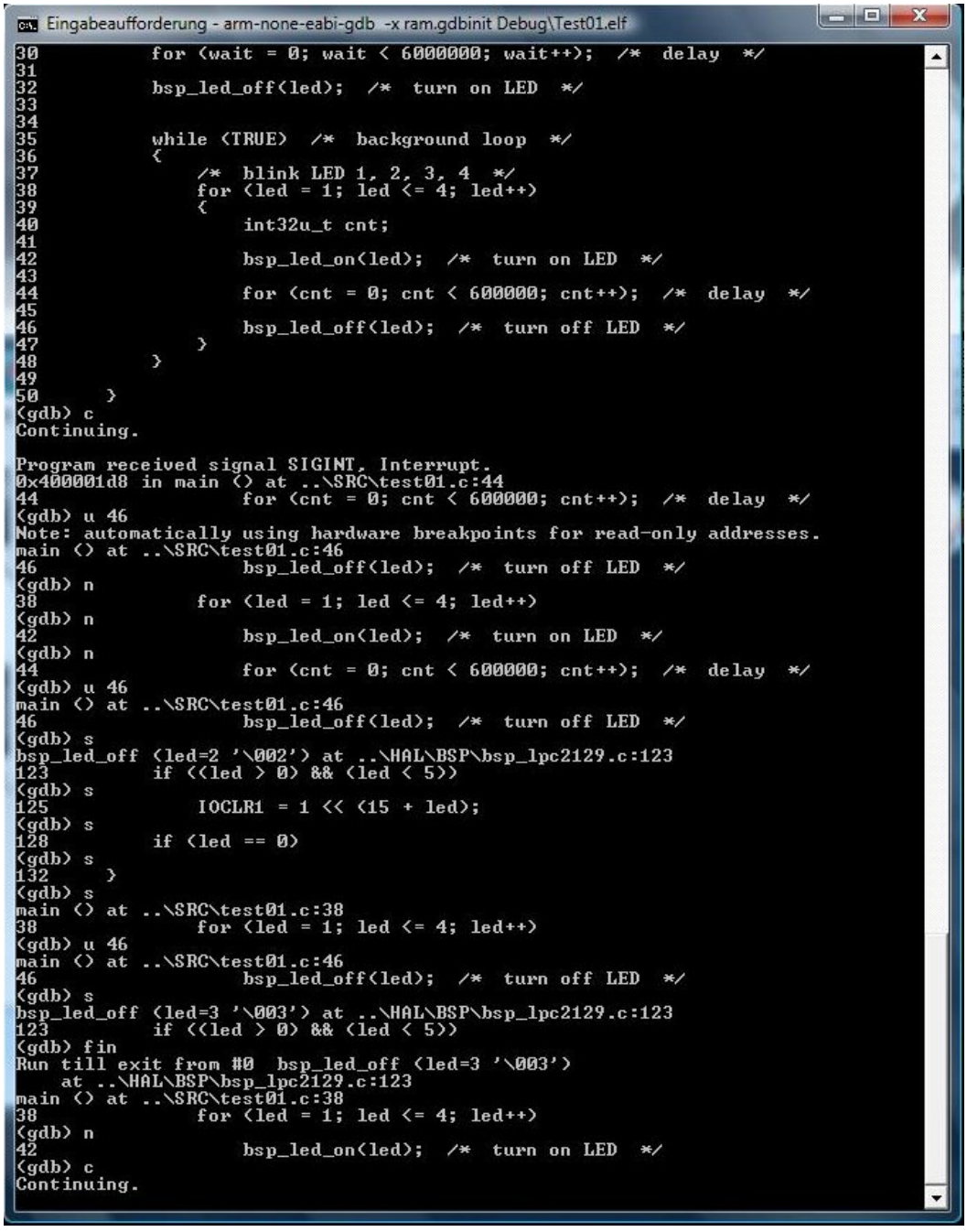
and to step into **s**ubstructures with

s

You can type

fin

to finish and leave any substructure at any time:



You have several commands for analyzing the code:

p [variable]

i lo

i s

i li

i r

See the table at the end of this chapter for a description and other important commands.

The most important GDB commands

Abbreviation	Command	Abbr., alternative	Command, alternative	Explanation
q	quit			Quit GDB
l	list	l 1,10	list 1,10	List source code from line 1 to line 10
<ctrl> + c				Stop running program
c	continue			Resume running program
b	break	b 33	break 33	Set normal breakpoint at line 33
hb	hbreak	hb 33	hbreak 33	Set hardware supported breakpoint at line 33
cl	clear	cl 33	clear 33	Delete breakpoint at line 33
i b	info break			Show all breakpoints and their states
d	delete	d 2	delete 2	Delete all breakpoints or delete breakpoint 2
dis	disable	dis 2	disable 2	Disable all breakpoints or disable breakpoint 2
en	enable	en 2	enable 2	Enable all breakpoints or enable breakpoint 2
s	step			Steps and jumps into **s**ubstructures (functions)
n	next			Steps and jumps **n**ot into functions
u	until	u 33	until 33	Steps until next source code instruction or steps until line 33
fin	finish			Finish function (e.g. after pressing s accidentaly)
i s	info stack	bt	backtrace	Show complete list of stack frames
p	print	p var1	print var1	Print value (variable)
i lo	info locals			Show local variable values
i r	info registers			Show register values
i li	info line			Infos about line (start-, endaddress)
i source	info source			Detailed info about current source file
i sources	info sources			Infos about all source files available
pwd				Print working directory
i	info			Show list of possible info features
h	help			Help function

9 Flashing Software with JTAG

The main difference between loading a program into RAM and loading it into Flash memory is of course that the program will reside in the latter after powering down. In detail there are some more differences that we will discuss now.

First, when building programs for the Flash memory You have to follow these rules:

- You must create a new build configuration like e.g. 'LPC2364Flash' (a)*
- You can copy the project settings for the 'LPC2364Flash' Flash build configuration from the 'LPC2364RAM' build configuration, but You have to make a few changes
- **You must use a special linker script 'flash.ld' (b)**
- **Set the define 'HAL_START_VECTOR_FLASH' in the projects './CFG/cfg_global.h' file to '1' and the 'HAL_START_VECTOR_RAM' to '0'. This will set the NXP LPC2xxx memory mapping register to start from Flash (c)**

Second, when running and debugging programs in Flash notice that:

- You must create a new debug configuration like e.g. 'Tut01_LED_flash' (e)
- You can copy the debug settings for the 'Tut01_LED_flash' from the 'Tut01_LED_ram' debug configuration, but You have to make a few changes
- **You must use another GDB initialization script 'flash.gdbinit' (f)**
- **Software breakpoints doesn't work with a program that is located in the Flash memory. You can only set two hardware breakpoints, not more (g)**

Third, there are two general differences:

- The program in the Flash runs much slower than that in the RAM. But it is possible to compensate this by enabling the memory acceleration module (h)
- **If You want to erase a program in Flash You have to overwrite the memory banks with zeros (0x00000000) or ones (0xFFFFFFFF) otherwise it will reside in the memory and can interfere with newer programs**

Before You start with the Flash software You must be sure, that You have a working RAM build configuration 'LPC2364RAM' and a working RAM debug configuration 'Tut01_LED_ram'. If not go back to the chapters 'Setting Project Properties' and 'Building Projects' for the working build configuration and 'Eclipse Debug Settings' and 'Debugging with Eclipse' for the working debug configuration.

You also should be sure, that Your linker script 'flash.ld' is ok, see therefore 'Annex Linker Scripts', and that Your GDB initialization file 'flash.gdbinit' is ok, see therefore 'Annex GDB Init Files'.

Are You sure that Your JTAG debugging system is working? If not, see chapter 'Running the JTAG Debugger'.

Now You are ready to proceed!

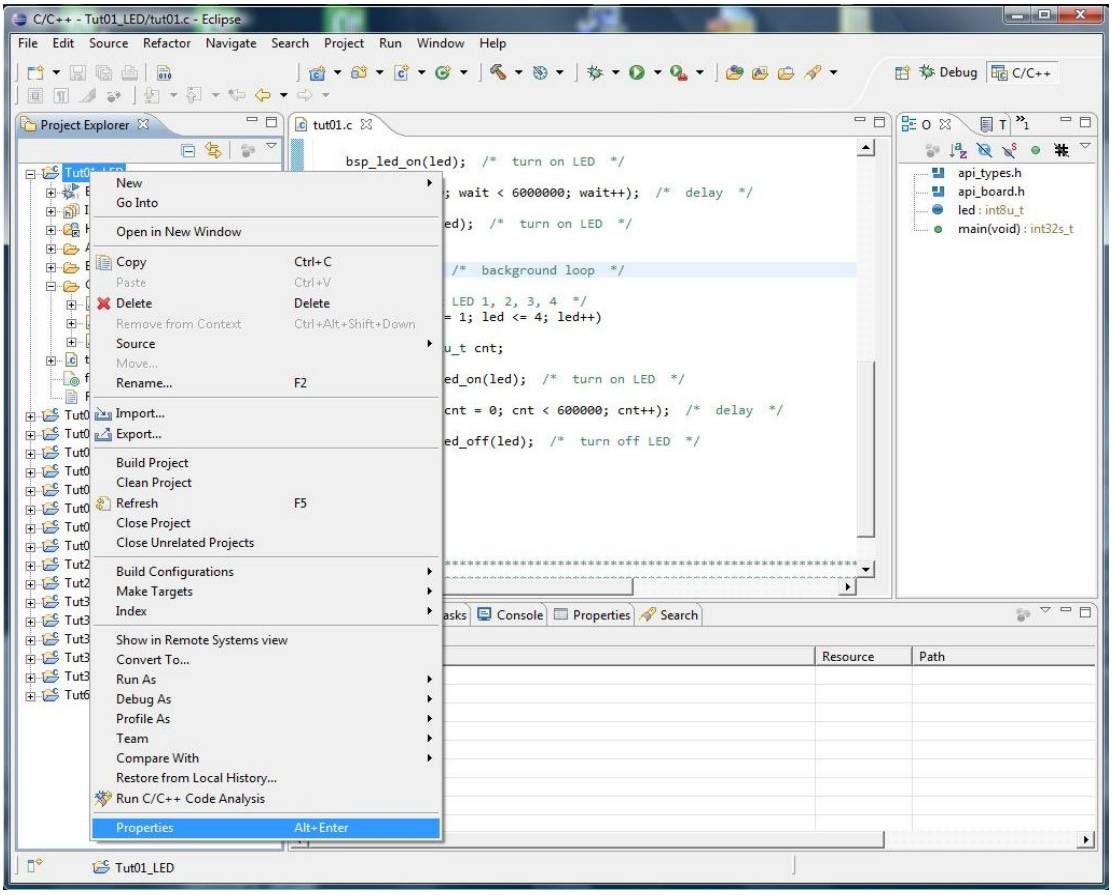
---

* The letters a, b, c, .. refer to the following sub-chapters

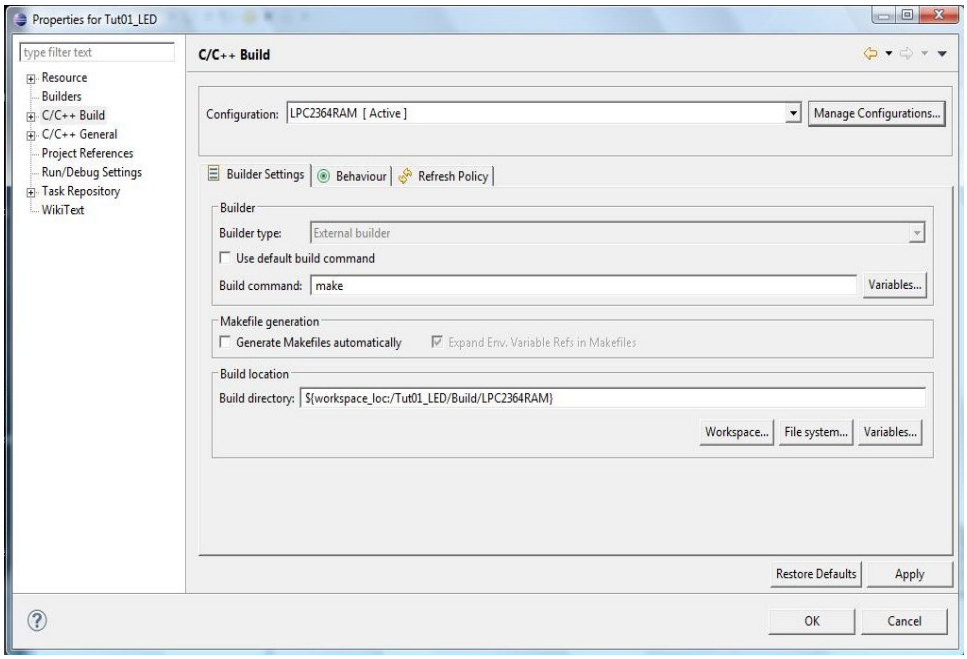
a) Creating a Build Configuration for Flash Memory

Let's start with creating a new build configuration that is used for building Flash images. While creating the new build configuration we will copy all settings from the existing RAM configuration.

Click with the right mouse button on the Project, in our example it is 'Tut01_LED' and choose 'Properties':

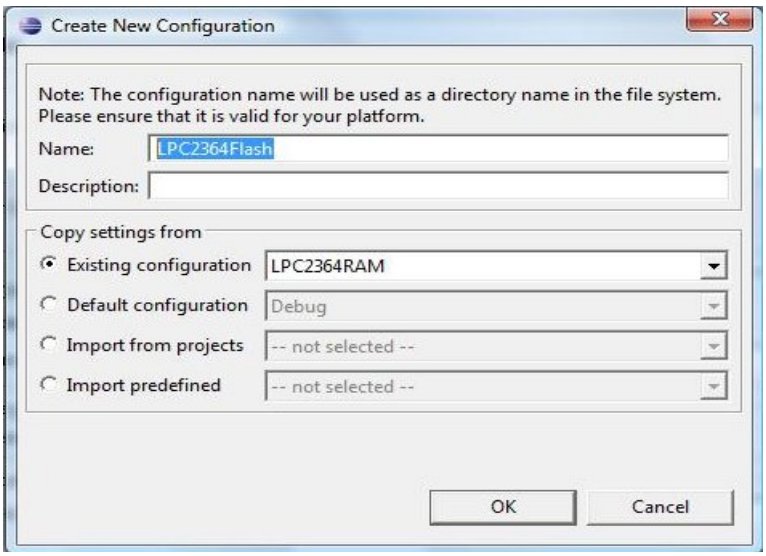


Here You click on the 'C/C++ Build' menu item. You should see this. Click on the 'Manage Configurations' button:

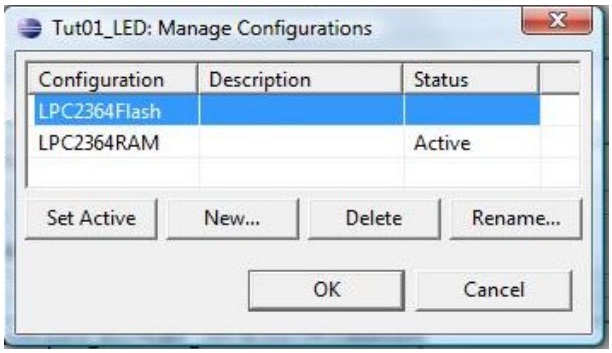


A new window opens. Click on 'New...':

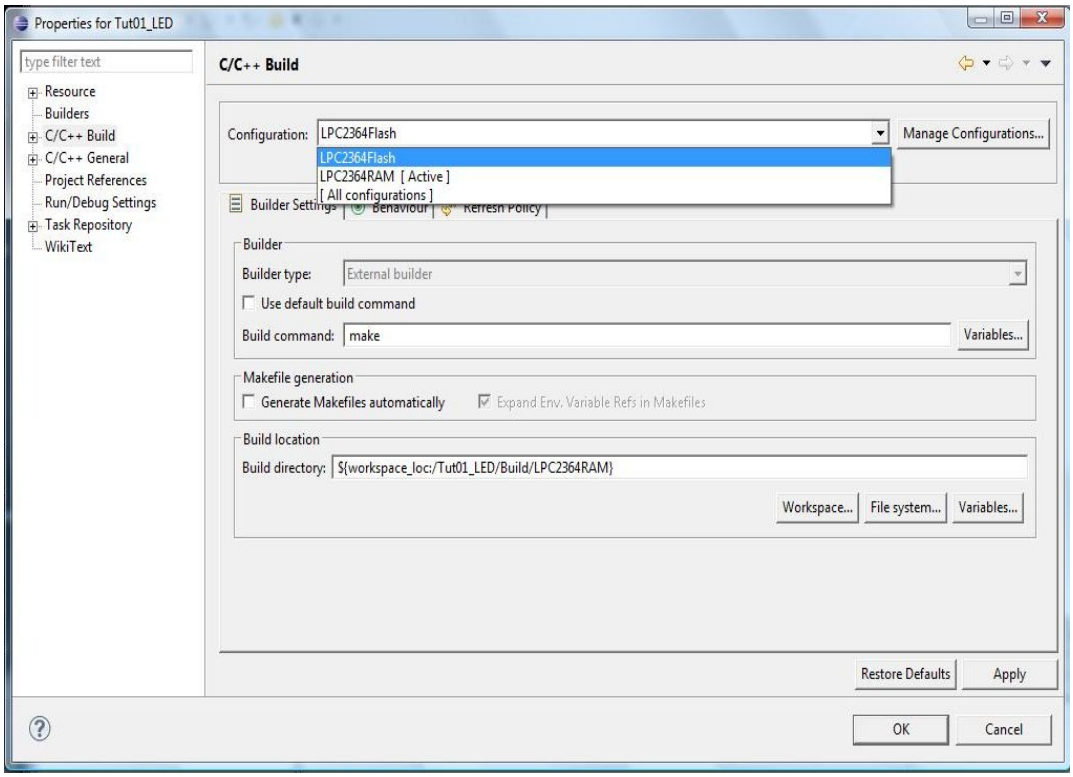
Another new window opens. Insert a new name, for example 'LPC2364Flash'. Under 'Copy settings from' choose 'Existing configuration' and choose in the corresponding pop-down menu 'LPC2364RAM'. Then click OK:



You get back to the former window. Here You see the new settings configuration. Click OK:



In the 'C/C++ Build' menu choose in the 'Configurations' pop-down menu our new 'LPC2364Flash' configuration. Click Apply:



This is it. Now You have a new build configuration for the Flash that has the same settings than Your RAM build configuration. Of course, this can not work properly. So let's proceed with the next step.

b) Adjusting the new Flash Build Configuration

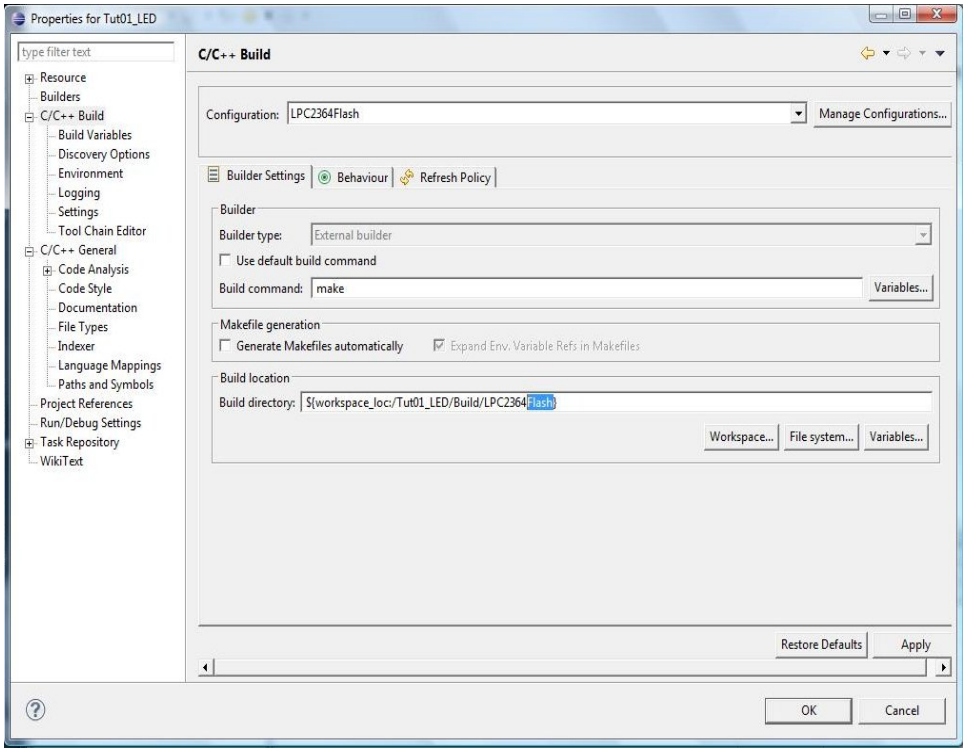
We have just created a new build configuration 'FlashDebug' and copied all setting from the 'Debug' configuration for RAM builds. Now we have to do two small changes:

First change the Build directory name from

./Build/LPC2364RAM

to

./Build/LPC2364Flash



The next change is actually no change, it is a correction. When copying the setting configuration the prefix in 'Discovery Options' gets lost. Open the 'C/C++ Build' folder in the left menu and go to 'Discovery Options'.

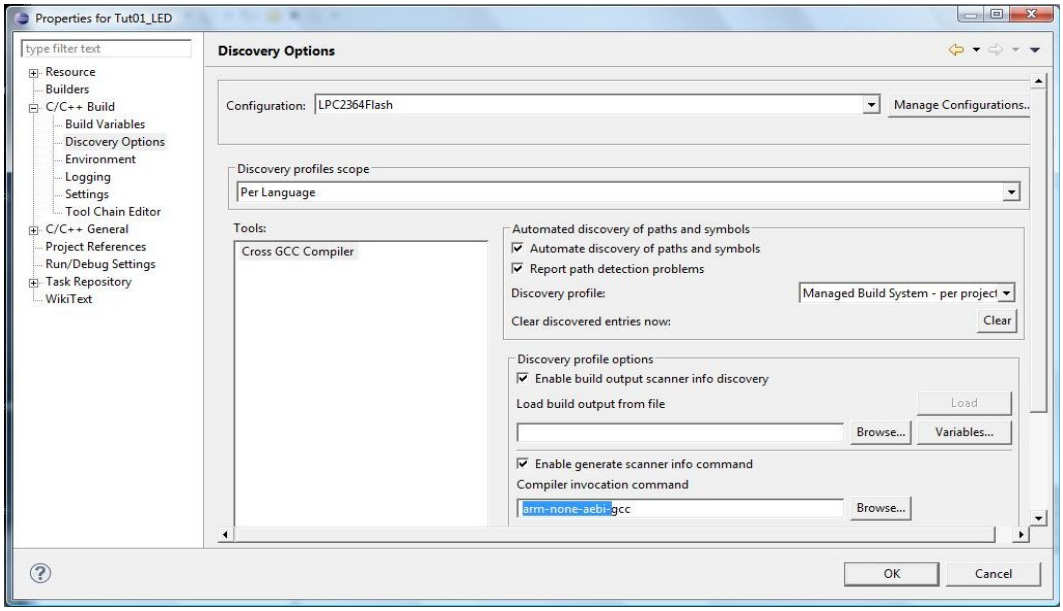
Here You add in front of 'gcc' the prefix

arm-none-eabi-

on Windows or otherwise

arm-elf-

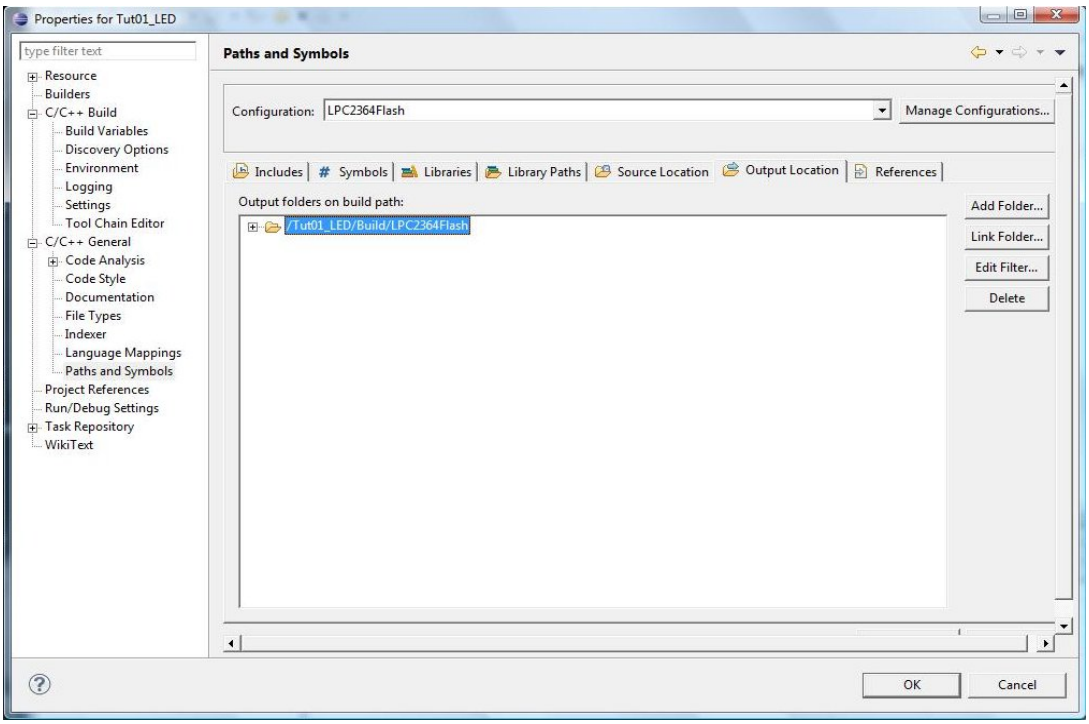
Click Apply:



Open the 'C/C++ General' folder and the 'Output Location' slider. Add the path

'/Tut01_LED/Build/LPC2364Flash'

and by psuhing the 'Add Folder..' button. Delete the old path with 'Delete'.



Now Your new Flash build configuration is ready!

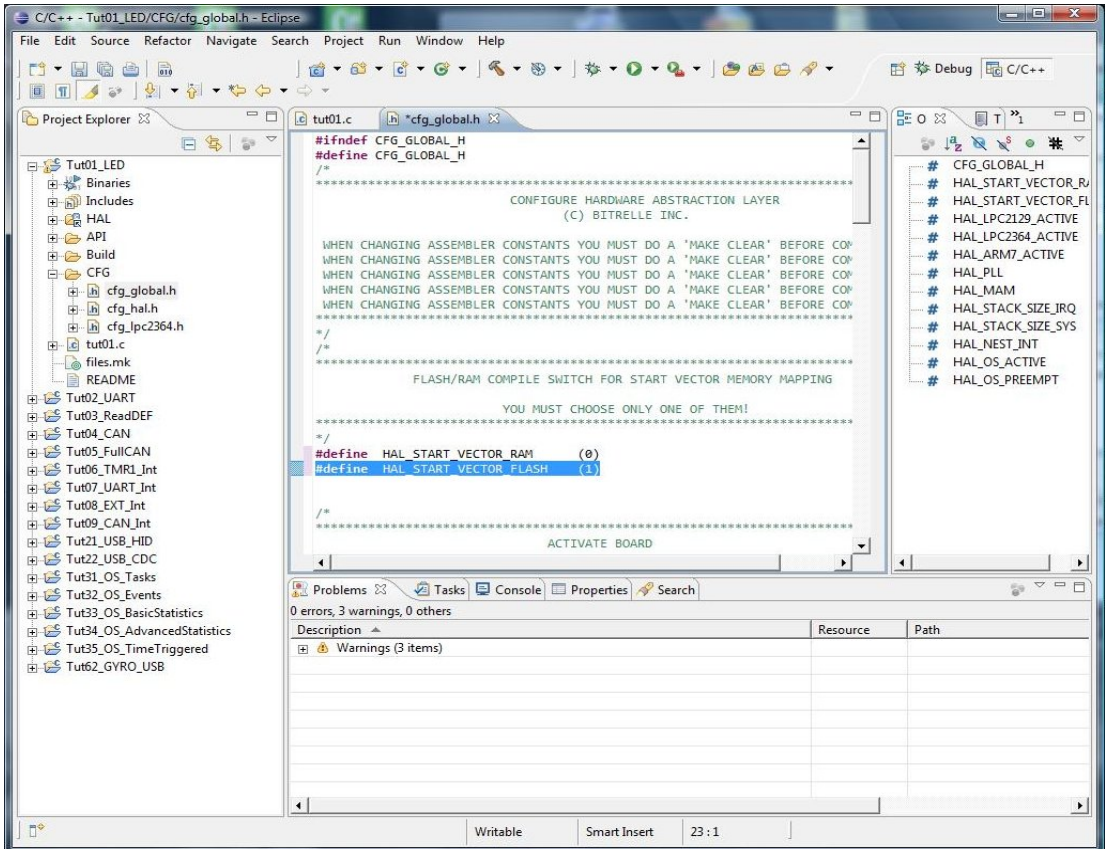
c) Rerouting Memory Mapping

Go to file 'CFG/cfg_global.h' and set the define

'HAL_START_VECTOR_FLASH' to '1'

and the

'HAL_START_VECTOR_RAM' to '0':

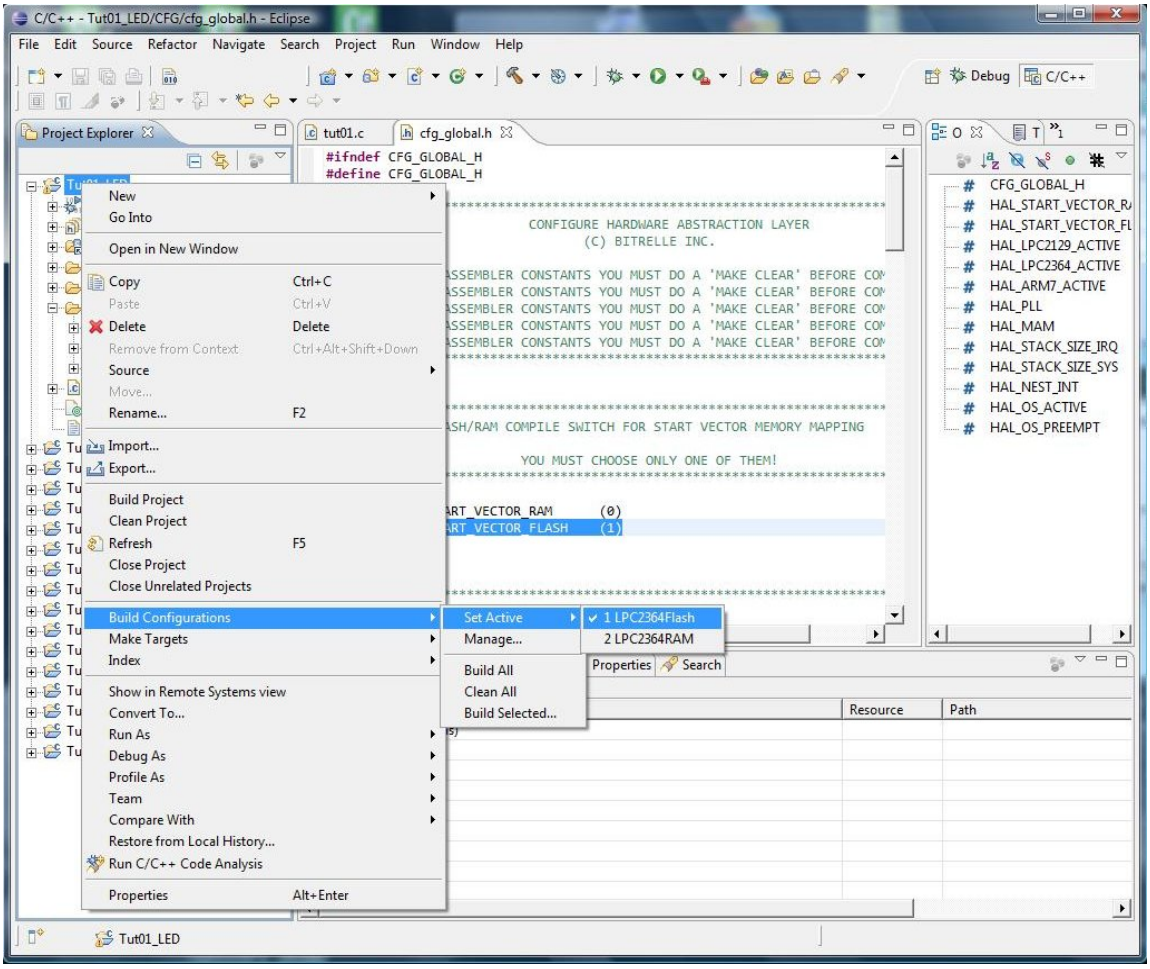


d) Building the Program for Flash Memory

Make 'LPC2364Flash' to Your active build configuration. Click with the right mouse button on the project to build. In our example this is 'Tut01_LED'. Click on

'Build Configurations' -> 'Set Active' -> 'LPC2364Flash'.

A check mark should indicate 'LPC2364Flash' as the active build configuration:

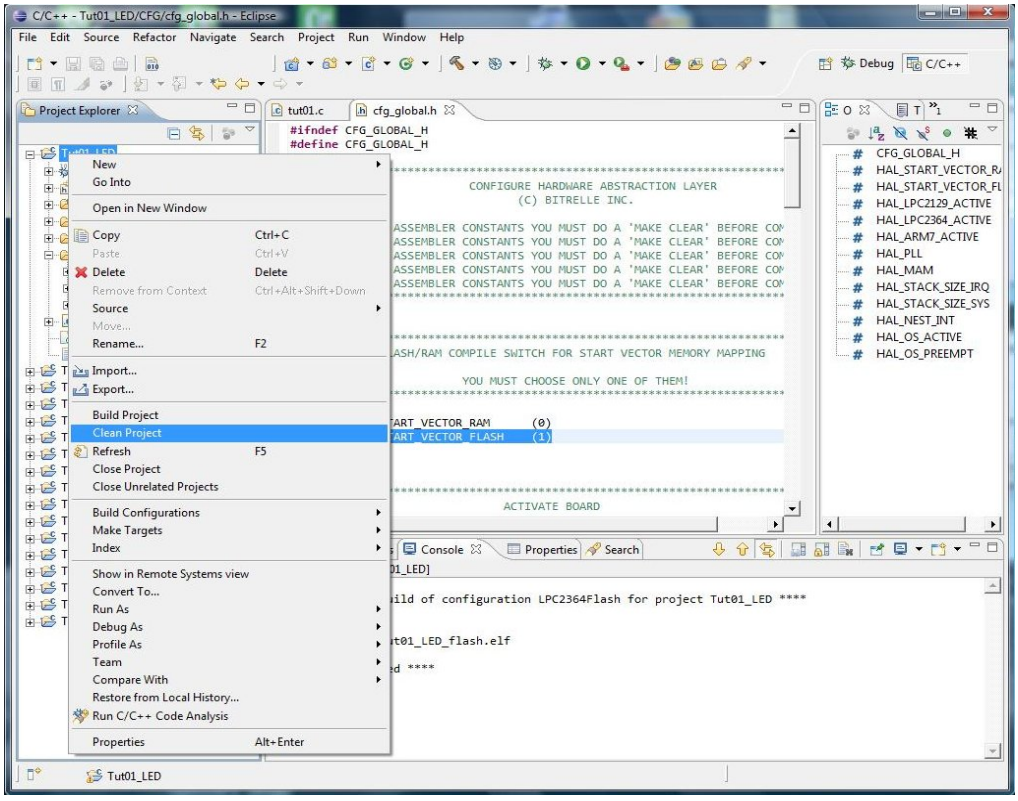


Before You proceed make sure that Your

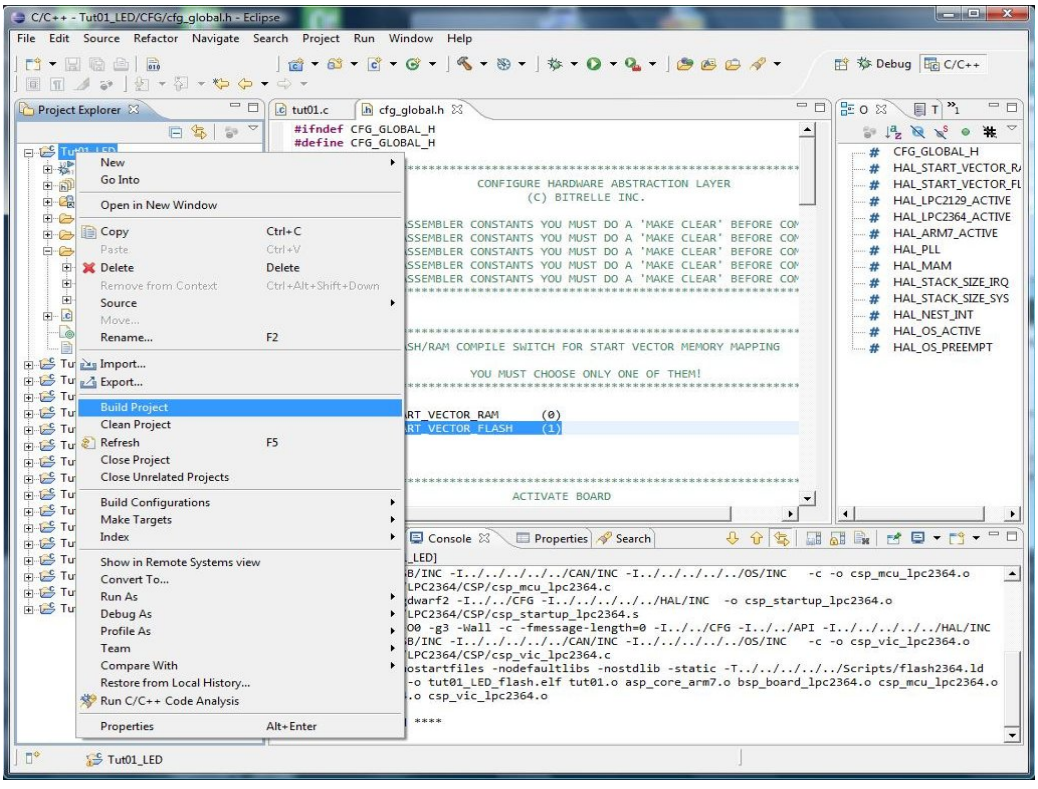
- build configuration is 'LPC2364Flash'
- project settings for 'LPC2364Flash' are correct
- linker script is 'flash.ld' and is correct
- memory mapping flag 'HAL_START_VECTOR_FLASH' is set to '1'
- memory mapping flag 'HAL_START_VECTOR_RAM' is set to '0':

Choose 'Clean Project' to clean the project. This is very important.

WITHOUT CLEANING THE PROJECT FIRST, ALL ASSEMBLER SETTINGS THAT CHANGED ARE IGNORED AND WILL NOT HAVE ANY EFFECT.

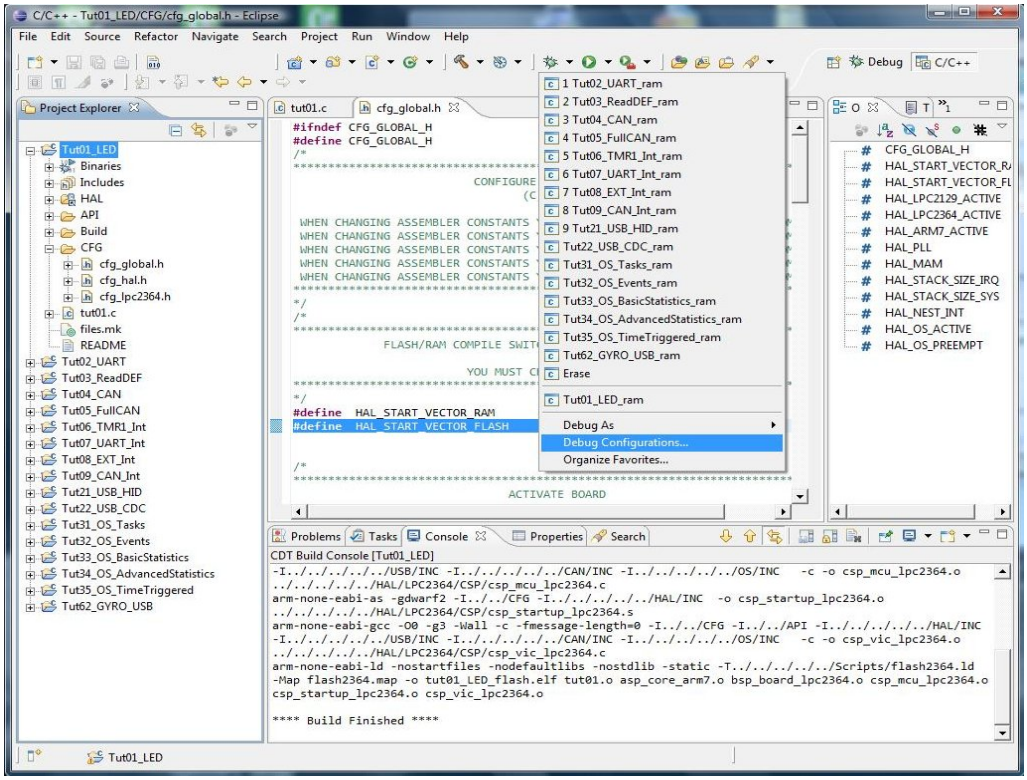


After the project has been cleaned click on 'Build Project' to build the Flash image:

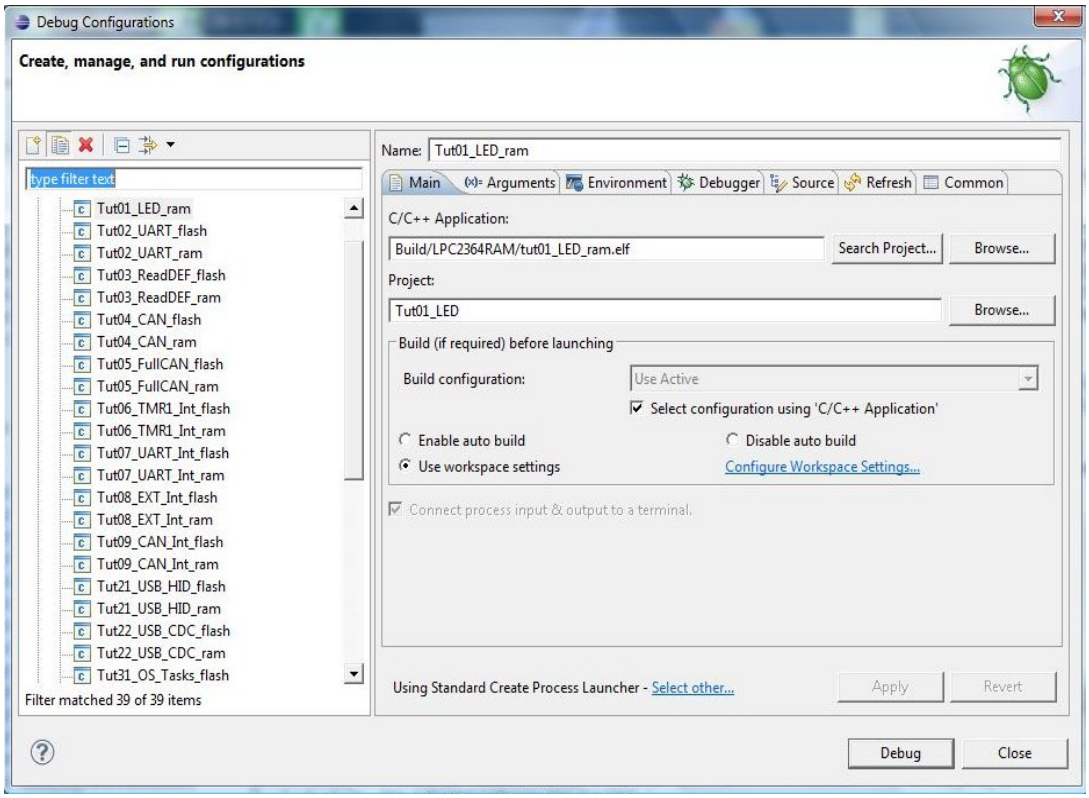


e) Creating the Debug Configuration for Flash Memory

Click on the down-arrow right of the bug symbol in the menu bar and choose 'Debug Configurations..':



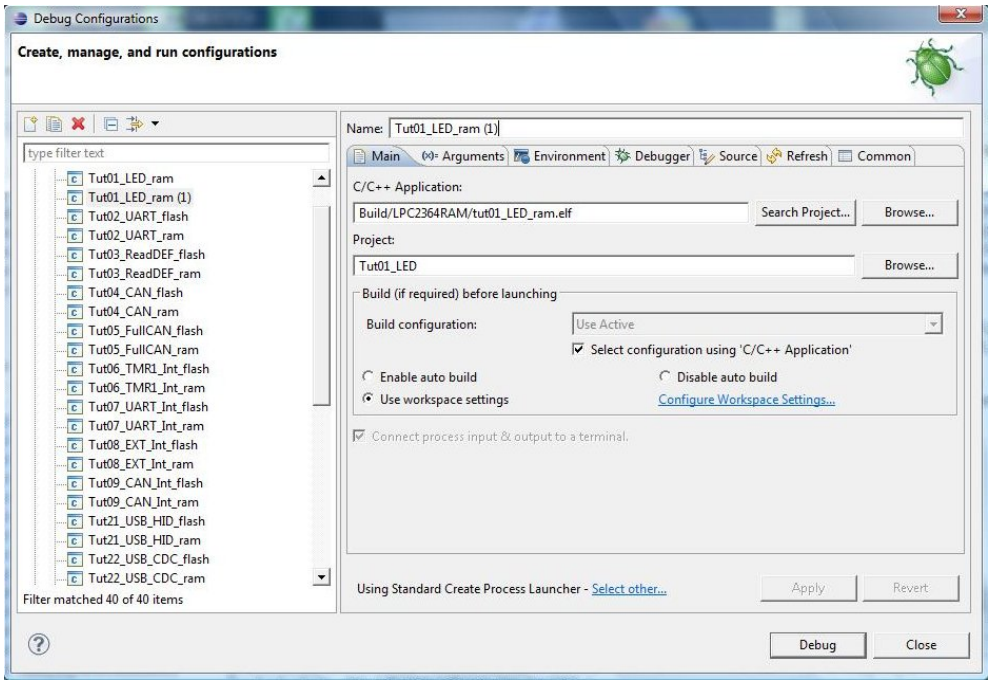
The 'Debug Configurations' window opens. Mark the 'Tut01_LED_ram' configuration and click on the 'Duplicates the currently selected launch configuration' symbol (the paper sheets):



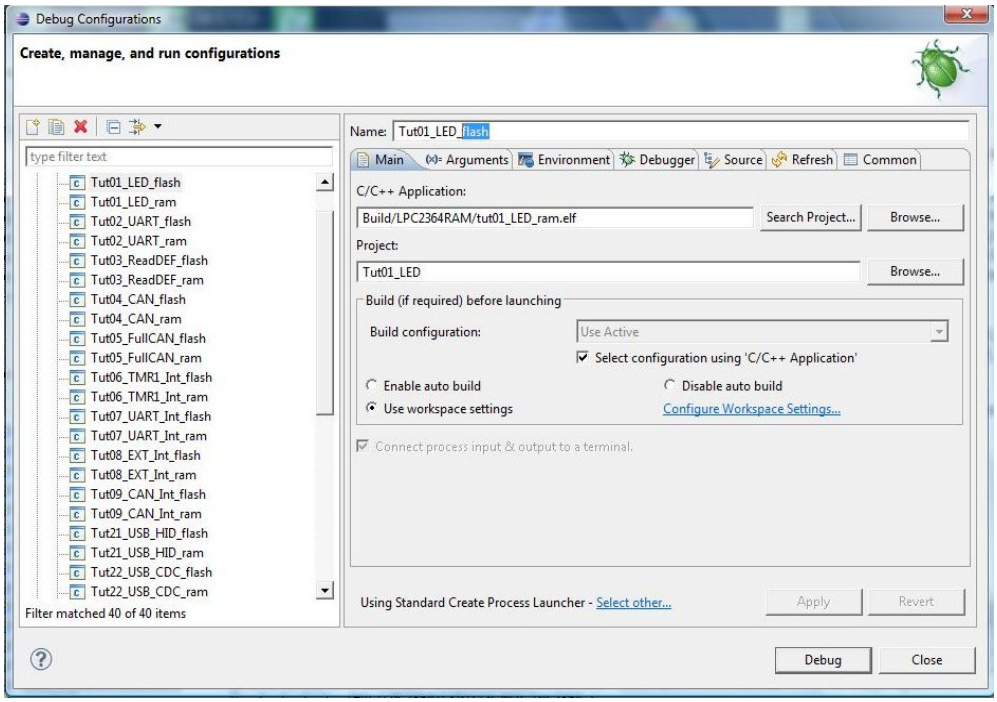
f) Adjusting the Flash Debug Configuration

In the last sub-chapter we learned how to create a new debug launch configuration from duplicating existing debug launch configurations.

The new debug configuration should now be indicated in the left menu under 'C/C++ Application' with the same name and a number one in braces. All settings in this new debug configuration are copied from the 'Tut01_LED_ram' launch configuration. We have to make four changes before we can start with loading and debugging on target.



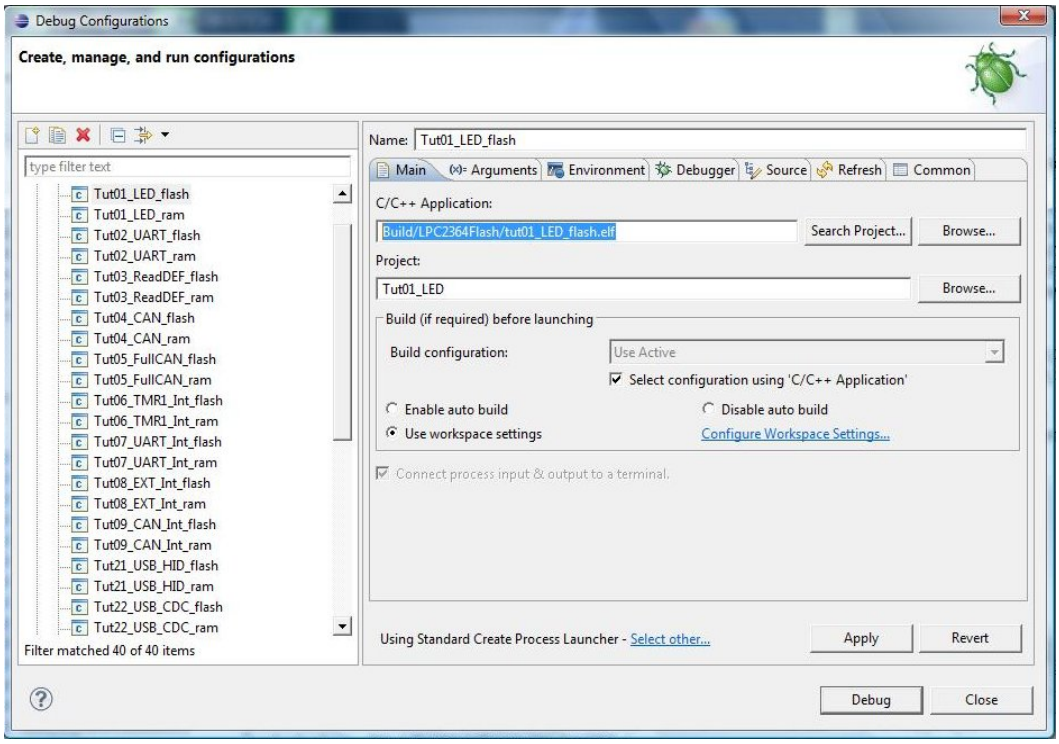
The first change is the name. Our debug configuration shall have its own name, we choose 'Tut01_LED_flash'. Click Apply.



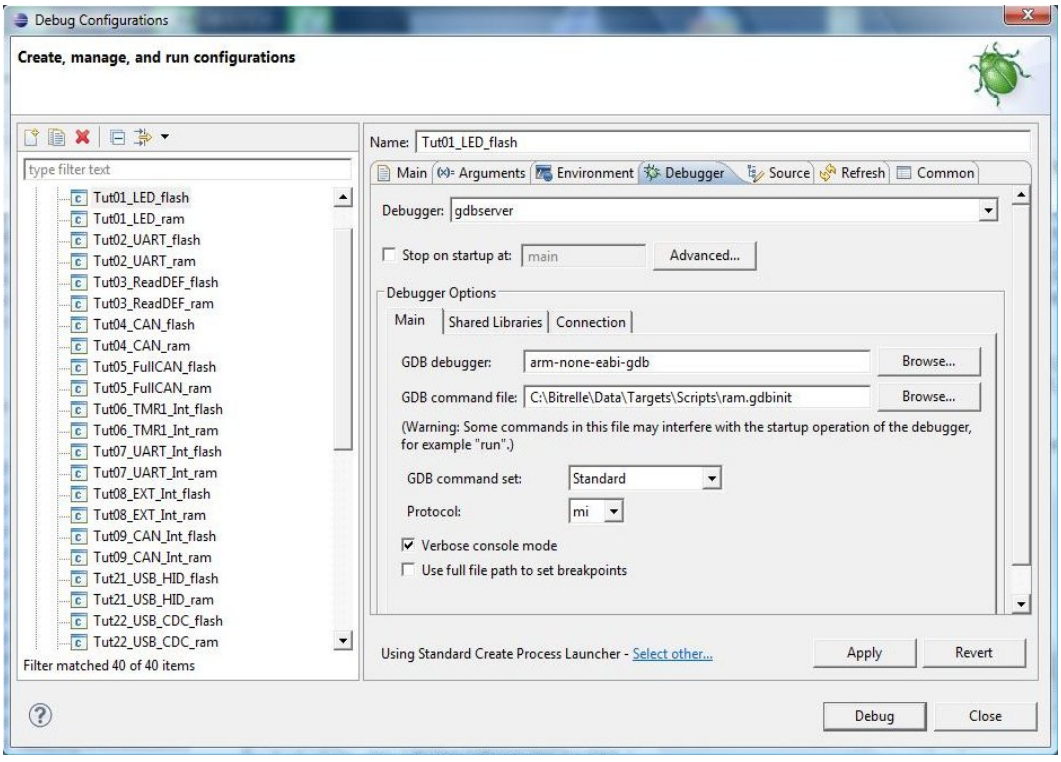
The second change is the 'C/C++ Application' path. Change it to

Build/LPC2364Flash/tut01_LED_flash.elf

Otherwise we will load the wrong image to the target. Click Apply:



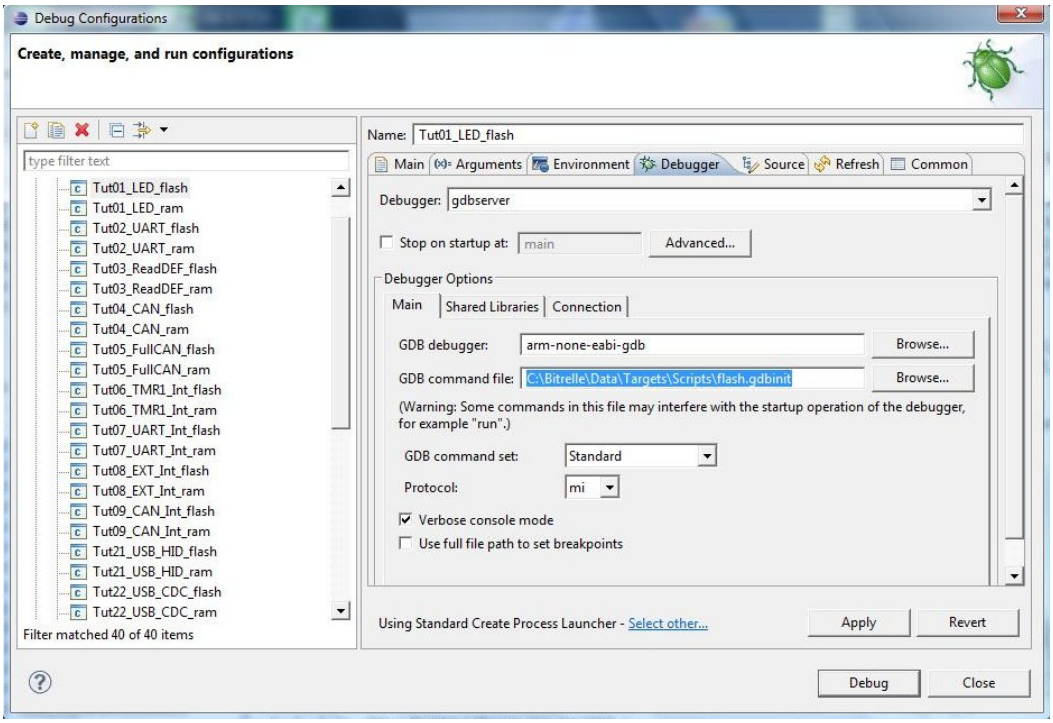
The third change You have to make in the 'Debugger' slider. Here, deactivate the 'Stop on startup at' checkbox. Otherwise Your debugging session would be limited to only one breakpoint, because one of the two available hardware breakpoints would always be used for trivial stopping on startup:



The fourth and last change is the GDB command file. Change it to

C:\Bitrelle\Data\Targets\Scripts\flash.gdbinit

by pressing the 'Browse..' button and browsing to the 'flash.gdbinit' file.



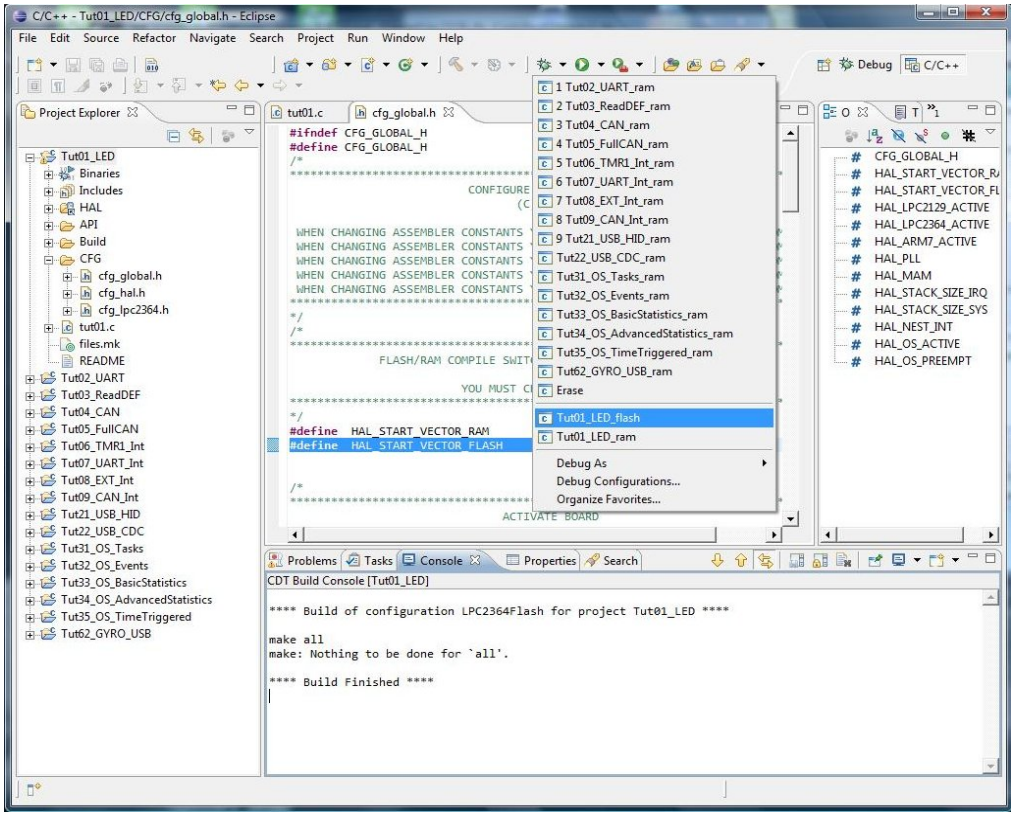
Now we are ready to start a debug session.

g) Debugging Programs in the Flash Memory

Before You proceed make sure that Your

- Flash build has finished successfully with the correct settings
- Flash image file 'Build/LPC2364Flash/tut01_LED_flash.elf' exists
- 'Tut01_LED_flash' debug settings are correct
- 'flash.gdbinit' is correct (see 'Annex GDB Init Files')
- JTAG device is connected
- drivers for JTAG device have been installed correctly
- OpenOCD is running in the background

For starting a debug session you click on the down-arrow right of the bug symbol and choose our debug configuration 'Tut01_LED_flash':

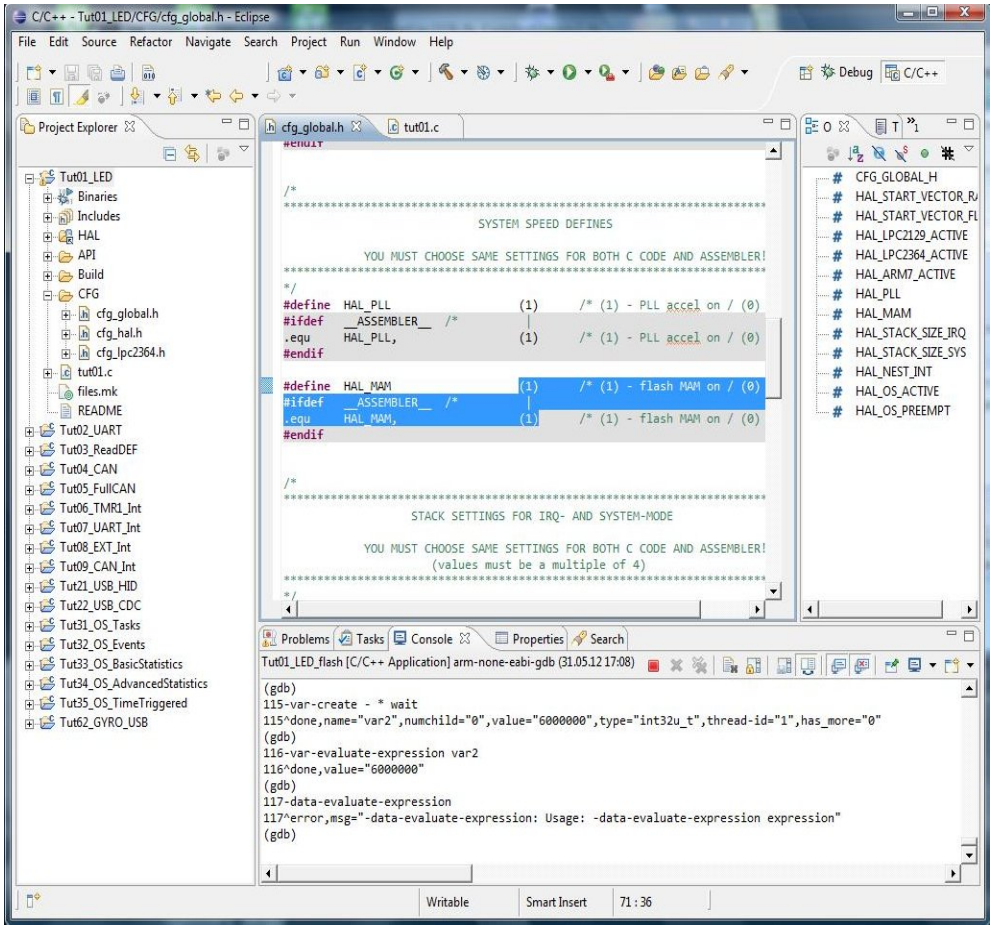


As we have mentioned before, software breakpoints doesn't work with a program that is located in the Flash memory. You can only set two hardware breakpoints, not more. All other debugging features we have learned in chapter 'Debugging with Eclipse' work in the same way.

h) Speed Acceleration of Programs in Flash Memory

It is possible to increase the speed of programs located in the Flash memory by activating the memory acceleration module. This hardware acceleration module is a kind of parallelization. While the first command is executed, the next command is read out of the memory. The execution speed, that was decelerated heavily by slow Flash memory is thereby nearly doubled.

To activate the memory acceleration module You go to file 'CFG/cfg_global.h' and set the memory acceleration flag 'HAL_MAM' to '1' for both entries, the C preprocessor define and for the assembler constant:

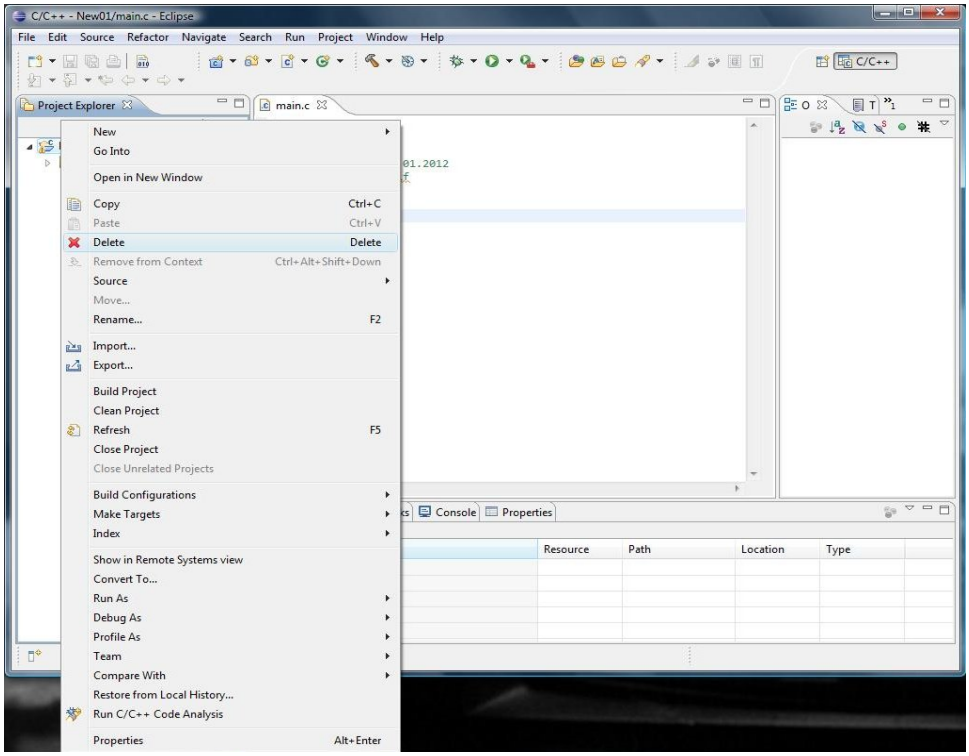


10 Document History

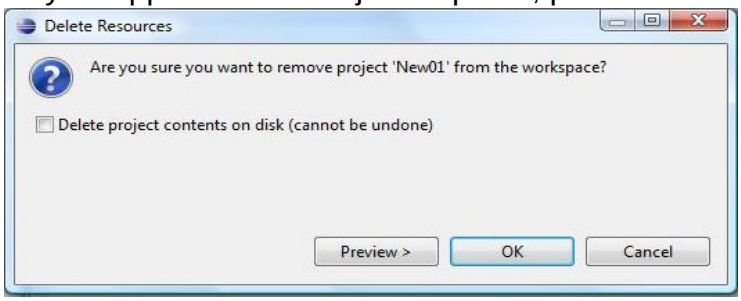
Date	Description	Revision Number	Author(s)
31.1.2012	Initial Revision	1.0	R. Schaefer
31.5.2012	New chapter Project Properties, Path information updated	1.1	R. Schaefer

11 Annex Deleting Projects

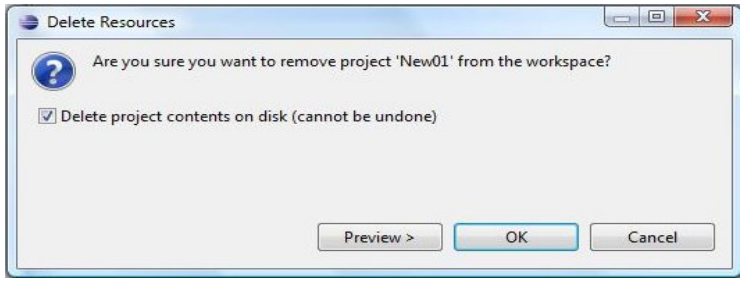
For deleting a project You have to click on the new project with the right mouse button and select 'Delete':



A pop-up opens and You can choose to leave the project on hard disk. This means it will only disappear in the Project Explorer, press OK:

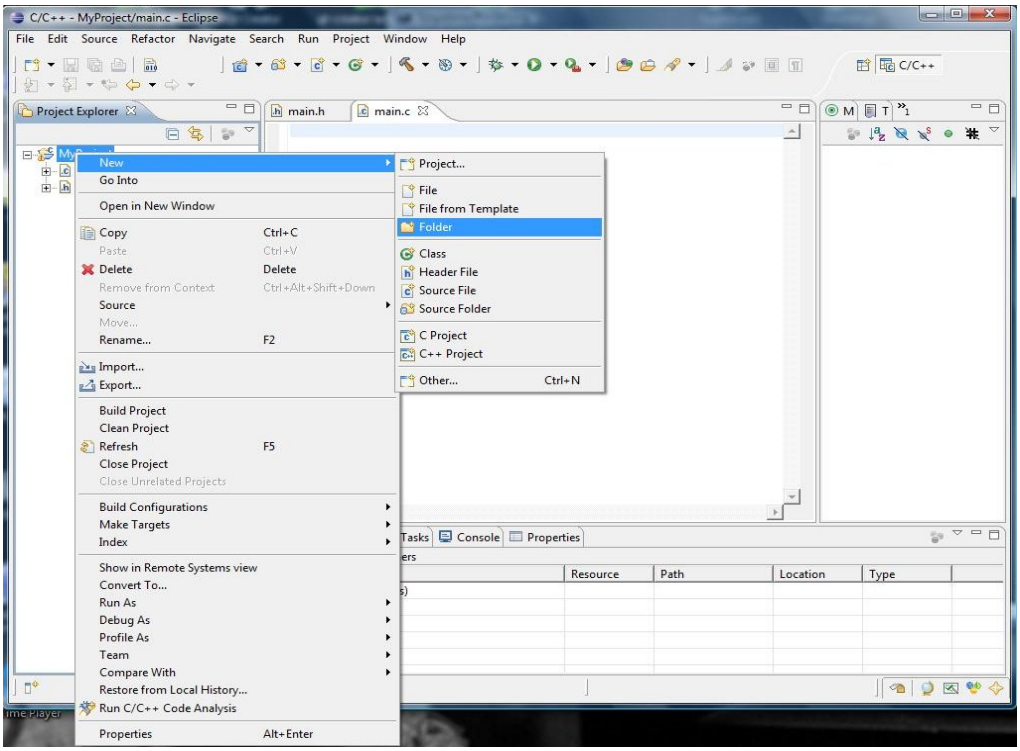


Or You can choose to delete the project completely, press OK:



12 Annex Adding Directories to Your Project

If You want to add a directory to Your project, click on the project witch the right mouse button and select 'New' → 'Folder'



A menu opens where You can enter the name of the directory, then press Finish:

13 Annex Makefiles

a) Makefile for RAM

```
#
# Makefile for RAM and Debug
#


#
# general path settings
#
TRUNK_PATH = ../../../../..
PROJ_PATH  = ../..
HAL_PATH   = $(TRUNK_PATH)/HAL
USB_PATH   = $(TRUNK_PATH)/USB
CAN_PATH   = $(TRUNK_PATH)/CAN
OS_PATH    = $(TRUNK_PATH)/OS


#
# includes
#
include $(PROJ_PATH)/files.mk


#
# general linker settings
#
ELF_NAME  = $(OUT)_ram
SCRIPT_NAME = ram2129


#
# source paths
#
vpath  tut%       $(PROJ_PATH)
vpath  asp_%      $(HAL_PATH)/LPC2129/ASP
vpath  bsp_%      $(HAL_PATH)/LPC2129/BSP
vpath  csp_%      $(HAL_PATH)/LPC2129/CSP
vpath  usb_%      $(USB_PATH)/USB20
vpath  can_%      $(CAN_PATH)/CANopen
vpath  osp_%      $(OS_PATH)/OS10/OSP
vpath  esp_%      $(OS_PATH)/OS10/ESP
vpath  stp_%      $(OS_PATH)/OS10/STP


#
# include paths
#
CC_INC = -I$(PROJ_PATH)/CFG   \
         -I$(PROJ_PATH)/API   \
         -I$(HAL_PATH)/INC    \
         -I$(USB_PATH)/INC    \
         -I$(CAN_PATH)/INC    \
         -I$(OS_PATH)/INC
```

```
AS_INC = -I$(PROJ_PATH)/CFG   \
         -I$(HAL_PATH)/INC


#
# toolchain settings
#
TOOLCHAIN = arm-none-eabi-

CC = $(TOOLCHAIN)gcc

AS = $(TOOLCHAIN)as

LD = $(TOOLCHAIN)ld

CFLAGS  = -O0 -g3 -Wall -c -fmessage-length=0 $(CC_INC)

ASFLAGS = -gdwarf2 $(AS_INC)

LDFLAGS = -nostartfiles -nodefaultlibs -nostdlib -static      \
          -T$(TRUNK_PATH)/Scripts/$(SCRIPT_NAME).ld -Map $(SCRIPT_NAME).map

#
# make targets
#
TARGET = $(ELF_NAME).elf

OBJ = $(PRO) $(ASP) $(BSP) $(CSP) $(USB) $(CAN) $(OS)

$(TARGET): $(OBJ) Makefile
      $(LD) $(LDFLAGS) -o $(TARGET) $(OBJ)

.PHONY : all
all: $(TARGET)

.PHONY : clean
clean:
      rm -f *.o *.map $(TARGET)
```

b) Makefile for Flash

```
#
# Makefile for Flash and Debug
#


#
# general path settings
#
TRUNK_PATH = ../../../../..
PROJ_PATH  = ../..
HAL_PATH   = $(TRUNK_PATH)/HAL
USB_PATH   = $(TRUNK_PATH)/USB
CAN_PATH   = $(TRUNK_PATH)/CAN
OS_PATH    = $(TRUNK_PATH)/OS


#
```

```
# includes
#
include $(PROJ_PATH)/files.mk


#
# general linker settings
#
ELF_NAME  = $(OUT)_flash
SCRIPT_NAME = flash2129


#
# source paths
#
vpath  tut%       $(PROJ_PATH)
vpath  asp_%      $(HAL_PATH)/LPC2129/ASP
vpath  bsp_%      $(HAL_PATH)/LPC2129/BSP
vpath  csp_%      $(HAL_PATH)/LPC2129/CSP
vpath  usb_%      $(USB_PATH)/USB20
vpath  can_%      $(CAN_PATH)/CANopen
vpath  osp_%      $(OS_PATH)/OS10/OSP
vpath  esp_%      $(OS_PATH)/OS10/ESP
vpath  stp_%      $(OS_PATH)/OS10/STP


#
# include paths
#
CC_INC = -I$(PROJ_PATH)/CFG   \
         -I$(PROJ_PATH)/API   \
         -I$(HAL_PATH)/INC    \
         -I$(USB_PATH)/INC    \
         -I$(CAN_PATH)/INC    \
         -I$(OS_PATH)/INC

AS_INC = -I$(PROJ_PATH)/CFG   \
         -I$(HAL_PATH)/INC


#
# toolchain settings
#
TOOLCHAIN = arm-none-eabi-

CC = $(TOOLCHAIN)gcc

AS = $(TOOLCHAIN)as

LD = $(TOOLCHAIN)ld

CFLAGS  = -O0 -g3 -Wall -c -fmessage-length=0 $(CC_INC)

ASFLAGS = -gdwarf2 $(AS_INC)

LDFLAGS = -nostartfiles -nodefaultlibs -nostdlib -static      \
          -T$(TRUNK_PATH)/Scripts/$(SCRIPT_NAME).ld -Map $(SCRIPT_NAME).map

#
# make targets
```

```
#
TARGET = $(ELF_NAME).elf

OBJ = $(PRO) $(ASP) $(BSP) $(CSP) $(USB) $(CAN) $(OS)

$(TARGET): $(OBJ) Makefile
	$(LD) $(LDFLAGS) -o $(TARGET) $(OBJ)

.PHONY : all
all: $(TARGET)

.PHONY : clean
clean:
	rm -f *.o *.map $(TARGET)
```

c) files.mk

```
#
# object files to build and link
#
PRO = tut01.o

ASP = asp_core_arm7.o

BSP = bsp_board_lpc2129.o

CSP = csp_mcu_lpc2129.o      \
      csp_startup_lpc2129.o  \
      csp_vic_lpc2129.o

USB =

CAN =

OS  =


#
# output file name
#
OUT = tut01_LED
```

14 Annex Linker Scripts

a) ram2129.ld

```
/* ld linker script for ram image */

ENTRY(_startup_begin)

FORCE_COMMON_ALLOCATION

_STACK_TOP_ADDR_ = 0x40004000;

MEMORY
{
  CODE (rwx) : ORIGIN = 0x40000000, LENGTH = 0x00003000
  DATA (rw)  : ORIGIN = 0x40003000, LENGTH = 0x00001000
}


SECTIONS
{
  . = ALIGN(4);

  /* section .text which is used for code */
  .text :
  {
    *csp_startup_lpc2129.o (.text)
    *test*.o (.text)
    *asp*.o (.text)
    *bsp*.o (.text)
    *csp*.o (EXCLUDE_FILE (*csp_startup_lpc2129.o) .text)
    *(.glue_7)
    *(.glue_7t)
  } >CODE =0
  . = ALIGN(4);
  _text_end_ = . ;

  /* .rodata section which is used for read-only data (constants) */
  .rodata :
  {
    *(.rodata)
  } >CODE
  . = ALIGN(4);
  _rodata_end_ = . ;

  _CODE_ = . ;


  /* .data section which is used for read-write initialized data */
  .data : AT (_rodata_end_)
  {
    _data_start_ = . ;
    _data = .;
    *(.data)
    SORT(CONSTRUCTORS)
  } >DATA
  . = ALIGN(4);
  _data_end_ = . ;
```

```
  _edata = .;

  /* .bss section which is used for read-write uninitialized/zero-initialized
data */
  .bss :
  {
    _bss_start_ = . ;
    __bss_start__ = . ;
    *(.bss)
    *(COMMON)
  } >DATA
  . = ALIGN(4);
  _bss_end_ = . ;
  __bss_end__ = . ;

  _DATA_ = . ;
  _end = . ;

  /* 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) }
}
```

b) flash2129.ld

```
/* ld linker script for flash image */

ENTRY(_startup_begin)

FORCE_COMMON_ALLOCATION

_STACK_TOP_ADDR_ = 0x40004000;

MEMORY
{
  CODE (rwx) : ORIGIN = 0x00000000, LENGTH = 0x0003E000
  DATA (rw)  : ORIGIN = 0x40000000, LENGTH = 0x00004000
}


SECTIONS
{
  . = ALIGN(4);

  /* section .text which is used for code */
  .text :
  {
    *csp_startup_lpc2129.o (.text)
    *(EXCLUDE_FILE (*csp_startup_lpc2129.o) .text)
    *(.glue_7)
    *(.glue_7t)
  } >CODE =0
  . = ALIGN(4);
  _text_end_ = . ;

  /* .rodata section which is used for read-only data (constants) */
  .rodata :
  {
    *(.rodata)
  } >CODE
  . = ALIGN(4);
  _rodata_end_ = . ;

  _CODE_ = . ;


  /* .data section which is used for read-write initialized data */
  .data : AT (_rodata_end_)
  {
    _data_start_ = . ;
    _data = .;
    *(.data)
    SORT(CONSTRUCTORS)
  } >DATA
  . = ALIGN(4);
  _data_end_ = . ;
  _edata = .;

  /* .bss section which is used for read-write uninitialized/zero-initialized
data */
  .bss :
  {
    _bss_start_ = . ;
```

```
    __bss_start__ = . ;
    *(.bss)
    *(COMMON)
  } >DATA
  . = ALIGN(4);
  _bss_end_ = . ;
  __bss_end__ = . ;

  _DATA_ = . ;
  _end = . ;

  /* 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) }
}
```

c) ram2364.ld

```
/* ld linker script for ram image */

ENTRY(_startup_begin)

FORCE_COMMON_ALLOCATION

_STACK_TOP_ADDR_ = 0x7FD02000;

MEMORY
{
  CODE  (rwx) : ORIGIN = 0x40000000, LENGTH = 0x00002000
  CODE2 (rwx) : ORIGIN = 0x7FE00000, LENGTH = 0x00004000
  DATA  (rwx) : ORIGIN = 0x7FD00000, LENGTH = 0x00002000
}


SECTIONS
{
  . = ALIGN(4);

  /* section .text which is used for code */
  .text :
  {
    *csp_startup_lpc2364.o (.text)
    *(.glue_7)
    *(.glue_7t)
  } >CODE =0
  . = ALIGN(4);
  _text_end_ = . ;

  /* .rodata section which is used for read-only data (constants) */
  .rodata :
  {
    *(.rodata)
  } >CODE
  . = ALIGN(4);
  _rodata_end_ = . ;

  _CODE_ = . ;


  /* section .text2 which is used for code */
  .text2 :
  {
    *test*.o (.text)
    *asp*.o (.text)
    *bsp*.o (.text)
    *csp*.o (EXCLUDE_FILE (*csp_startup_lpc2364.o) .text)
    *stp*.o (.text)
    *usb*.o (.text)
  } >CODE2 =0
  . = ALIGN(4);
  _text2_end_ = . ;
  _etext = . ;

  _CODE2_ = . ;
```

```
  /* .data section which is used for read-write initialized data */
  .data : AT (_rodata_end_)
  {
    _data_start_ = . ;
    _data = .;
    *(.data)
    SORT(CONSTRUCTORS)
  } >DATA
  . = ALIGN(4);
  _data_end_ = . ;
  _edata = .;

  /* .bss section which is used for read-write uninitialized/zero-initialized
data */
  .bss :
  {
    _bss_start_ = . ;
    __bss_start__ = . ;
    *(.bss)
    *(COMMON)
  } >DATA
  . = ALIGN(4);
  _bss_end_ = . ;
  __bss_end__ = . ;

  _DATA_ = . ;
  _end = . ;

  /* 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) }
}
```

d) flash2364.ld

```
/* ld linker script for flash image */

ENTRY(_startup_begin)

FORCE_COMMON_ALLOCATION

_STACK_TOP_ADDR_ = 0x7FD02000;

MEMORY
{
  CODE  (rwx) : ORIGIN = 0x00000000, LENGTH = 0x00020000
  DATA  (rw)  : ORIGIN = 0x40000000, LENGTH = 0x00002000
  DATA2 (rw)  : ORIGIN = 0x7FD00000, LENGTH = 0x00002000
  DATA3 (rw)  : ORIGIN = 0x7FE00000, LENGTH = 0x00004000
}


SECTIONS
{
  . = ALIGN(4);

  /* section .text which is used for code */
  .text :
  {
    *csp_startup_lpc2364.o (.text)
    *(EXCLUDE_FILE (*csp_startup_lpc2364.o) .text)
    *(.glue_7)
    *(.glue_7t)
  } >CODE =0
  . = ALIGN(4);
  _text_end_ = . ;

  /* .rodata section which is used for read-only data (constants) */
  .rodata :
  {
    *(.rodata)
  } >CODE
  . = ALIGN(4);
  _rodata_end_ = . ;

  _CODE_ = . ;


  /* .data section which is used for read-write initialized data */
  .data : AT (_rodata_end_)
  {
    _data_start_ = . ;
    _data = .;
    *(.data)
    SORT(CONSTRUCTORS)
  } >DATA
```

```
  . = ALIGN(4);
  _data_end_ = . ;
  _edata = .;

  /* .bss section which is used for read-write uninitialized/zero-initialized
data */
  .bss :
  {
    _bss_start_ = . ;
    __bss_start__ = . ;
    *(.bss)
    *(COMMON)
  } >DATA
  . = ALIGN(4);
  _bss_end_ = . ;
  __bss_end__ = . ;

  _DATA_ = . ;
  _end = . ;

  /* 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) }
}
```

15 Annex OpenOCD Target Config Files

a) lpc2129.cfg

```
# NXP LPC2129 ARM7TDMI-S with 256kB flash and 16kB SRAM, clocked with 12MHz
crystal

source [find target/lpc2xxx.cfg]

# parameters:
# - core_freq_khz - frequency of core in kHz during flashing, usually equal to
connected crystal or internal oscillator, e.g. 12000
# - adapter_freq_khz - frequency of debug adapter in kHz, should be 8x slower
than core_freq_khz, e.g. 1000

proc setup_lpc2129 {core_freq_khz adapter_freq_khz} {
      # 256kB flash and 16kB SRAM
      # setup_lpc2xxx <chip_name> <cputapid> <flash_size> <flash_variant>
<workarea_size> <core_freq_khz> <adapter_freq_khz>
      setup_lpc2xxx lpc2129 0xcf1f0f0f 0x40000 lpc2000_v1 0x4000 $core_freq_khz
$adapter_freq_khz
}

proc init_targets {} {
      # default to core clocked with 12MHz crystal
      echo "Warning - assuming default core clock 12MHz! Flashing may fail if
actual core clock is different."

      # setup_lpc2129 <core_freq_khz> <adapter_freq_khz>
      setup_lpc2129 12000 1500
}
```

b) lpc2378.cfg

```
# NXP LPC2378 ARM7TDMI-S with 512kB flash (8kB used by bootloader) and 56kB SRAM
(16kB for ETH, 8kB for DMA), clocked with 4MHz internal oscillator

source [find target/lpc2xxx.cfg]

# parameters:
# - core_freq_khz - frequency of core in kHz during flashing, usually equal to
connected crystal or internal oscillator, e.g. 12000
# - adapter_freq_khz - frequency of debug adapter in kHz, should be 8x slower
than core_freq_khz, e.g. 1000

proc setup_lpc2378 {core_freq_khz adapter_freq_khz} {
      # 504kB flash and 32kB SRAM
      # setup_lpc2xxx <chip_name> <cputapid> <flash_size> <flash_variant>
<workarea_size> <core_freq_khz> <adapter_freq_khz>
      setup_lpc2xxx lpc2378 0x4f1f0f0f 0x7e000 lpc2000_v2 0x8000 $core_freq_khz
$adapter_freq_khz
}

proc init_targets {} {
      # default to core clocked with 4MHz internal oscillator
      echo "Warning - assuming default core clock 4MHz! Flashing may fail if
actual core clock is different."

      # setup_lpc2378 <core_freq_khz> <adapter_freq_khz>
```

```
	setup_lpc2378 4000 500
}
```

c) lpc2xxx.cfg

```
# Common setup for the LPC2xxx parts

# parameters:
# - chip_name - name of the chip, e.g. lpc2103
# - cputapids - TAP IDs of the core, should be quoted if more than one, e.g.
0x4f1f0f0f or "0x3f0f0f0f 0x4f1f0f0f"
# - flash_size - size of on-chip flash (available for code, not including the
bootloader) in bytes, e.g. 0x8000
# - flash_variant - "type" of LPC2xxx device, lpc2000_v1 (LPC22xx and older
LPC21xx) or lpc2000_v2 (LPC213x, LPC214x, LPC210[123], LPC23xx and LPC24xx)
# - workarea_size - size of work-area in RAM for flashing procedures, must not
exceed the size of RAM available at 0x40000000, e.g. 0x2000
# - core_freq_khz - frequency of core in kHz during flashing, usually equal to
connected crystal or internal oscillator, e.g. 12000
# - adapter_freq_khz - frequency of debug adapter in kHz, should be 8x slower
than core_freq_khz, e.g. 1000

proc setup_lpc2xxx {chip_name cputapids flash_size flash_variant workarea_size
core_freq_khz adapter_freq_khz} {
	reset_config trst_and_srst

	# reset delays
	adapter_nsrst_delay 100
	jtag_ntrst_delay 100

	adapter_khz $adapter_freq_khz

	foreach i $cputapids {
		append expected_ids "-expected-id " $i " "
	}

	eval "jtag newtap $chip_name cpu -irlen 4 -ircapture 0x1 -irmask 0xf
$expected_ids"

	set _TARGETNAME $chip_name.cpu
	target create $_TARGETNAME arm7tdmi -chain-position $_TARGETNAME

	$_TARGETNAME configure -work-area-phys 0x40000000 -work-area-size
$workarea_size -work-area-backup 0

	if { $flash_size > 0 } {
		# flash bank <name> lpc2000 <base> <size> 0 0 <target#> <variant>
<clock> [calc checksum]
		set _FLASHNAME $chip_name.flash
		flash bank $_FLASHNAME lpc2000 0x0 $flash_size 0 0 $_TARGETNAME
$flash_variant $core_freq_khz calc_checksum
	}
}

proc init_targets {} {
	# FIX!!! read out CPUTAPID here and choose right setup. In addition to the
	# CPUTAPID some querying of the target would be required.
	return -error "This is a generic LPC2xxx configuration file, use a
specific target file."
}
```

16 Annex GDB Init Files

a) ram.gdbinit

```
### GDB init file for ram image



### architecture specific commands

set arm apcs32 on
   # tell gdb to use arm 32 bit mode



### connect to openocb server (tcp:3333)

target remote localhost:3333
   # tell gdb to connect to gdbserver at localhost:3333



### reset target

monitor echo "------- Reset and halt target:"
monitor soft_reset_halt
   # IMPORTANT: stop CPU and set PC back to reset vector, no reset



### openocd arm v4/v5 settings

monitor echo "------- Switch to core state ARM:"
monitor arm core_state arm
   # IMPORTANT: switch to core state ARM

monitor echo "------- Set arm v4/v5 config:"
monitor arm7_9 dbgrq disable
   # breakpoints instead of entry into ICE debug mode, unsafe
monitor arm7_9 dcc_downloads disable
   # do not write larger amounts of memory, unsafe
monitor arm7_9 fast_memory_access enable
   # disables completion check, huge speed increase, unsafe at low clock speeds
   # IMPORTANT: disable when working with very low clock speeds



### openocd gdb settings

monitor echo "------- Set gdb config:"
monitor gdb_breakpoint_override disable
   # IMPORTANT: do not force breakpoint type, gdb knows which type and when



### openocd jtag settings

monitor echo "------- JTAG settings:"
```

```
monitor verify_ircapture enable
   # verify values during IRCAPTURE, increases stability
monitor verify_jtag enable
   # verification of DR and IR scans, increases stability



### step to RAM section

monitor echo "------- Step to 0x40000000:"
monitor step 0x40000004
   # step to RAM section



### load program

monitor echo "------- Load binary:"
load
   ### load the command line option specified binary
monitor echo "load command executed"



### misc

monitor echo "------- Display register values:"
monitor arm reg
   # display a table of all banked core registers

monitor echo "------- Switch on target polling:"
monitor poll on
   # poll target for state changes
monitor echo "polling command executed"
```

b) flash.gdbinit

```
### GDB init file for flash image



### architecture specific commands

set arm apcs32 on
   # tell gdb to use arm 32 bit mode



### connect to openocb server (tcp:3333)

target remote localhost:3333
   # tell gdb to connect to gdbserver at localhost:3333



### reset target

monitor echo "------- Reset and halt target:"
monitor soft_reset_halt
   # IMPORTANT: stop CPU and set PC back to reset vector, no reset



### openocd arm v4/v5 settings

monitor echo "------- Switch to core state ARM:"
monitor arm core_state arm
   # IMPORTANT: switch to core state ARM

monitor echo "------- Set arm v4/v5 config:"
monitor arm7_9 dbgrq disable
   # breakpoints instead of entry into ICE debug mode, unsafe
monitor arm7_9 dcc_downloads disable
   # do not write larger amounts of memory, unsafe
monitor arm7_9 fast_memory_access enable
   # disables completion check, huge speed increase, unsafe at low clock speeds
   # IMPORTANT: disable when working with very low clock speeds



### openocd gdb settings

monitor echo "------- Set gdb config:"
monitor gdb_breakpoint_override disable
   # IMPORTANT: do not force breakpoint type, gdb knows which type and when



### openocd jtag settings

monitor echo "------- JTAG settings:"
monitor verify_ircapture enable
   # verify values during IRCAPTURE, increases stability
monitor verify_jtag enable
   # verification of DR and IR scans, increases stability
```

```
### step to flash section

monitor echo "------- Step to 0x00000000:"
monitor step 0x00000000
   # step to Flash section



### load program

monitor echo "------- Load binary:"
monitor flash erase_sector  0  0 last
monitor flash info 0
   ### erase flash bank 0
load
   ### flash the command line option specified binary
monitor echo "load command executed"



### misc

monitor echo "------- Display register values:"
monitor arm reg
   # display a table of all banked core registers

monitor echo "------- Switch on target polling:"
monitor poll on
   # poll target for state changes
monitor echo "polling command executed"
```