BA22

32-bit RISC Processor

Beyond Studio Integrated Development Environment

February 2012

IP Product Version
4.0p1

Document Signature
BA22–BSUG–4.0p1–103

CAST, Inc.

CONFIDENTIAL

Document Version

This document with its associated Release Notes applies to the version(s) of the core specified on the cover. See the Release Notes for any updates and additional information not included here.

Document Version History

Version	Date	Person	Changes from Previous Version
1.00	19-Sep-2011	P.D.	First release
1.01	13-Dec-2011	P.D.	FAQ and debugging sections
1.02	08-Feb-2012	P.D.	Profiling and FAQ sections
1.03	22-Feb-2012	P.D.	Execution profile view

CAST, Inc.
11 Stonewall Court, Woodcliff Lake, NJ 07677
Phone: +1 201-391-8300 Fax: +1 201-391-8694
info@cast-inc.com www.cast-inc.com

Table of Contents

1. References 6
1.1 Related Documents 6
1.2 Web Sites 6
2. Introduction 7
3. Supplied Software 8
3.1 Linux Package 8
3.2 Windows Package 8
4. Installation 9
4.1 Beyond Studio IDE for Linux 9
4.1.1 Prerequisites 9
4.1.2 Package Extraction 9
4.2 Beyond Studio IDE for Windows 9
4.2.1 Prerequisites 9
4.2.2 Package Extraction 10
4.3 Updates Installation 10
5. Development Flow 11
5.1 Starting BA22 Eclipse IDE 11
5.2 Creating New Project for the BA22 12
5.3 Building your Application 13
5.4 Debugging 14
5.4.1 Breakpoints 17
5.4.2 Watchpoints 17
5.4.3 Profiling 19
5.4.3.1. Simulator State View 19
5.4.3.2. Execution profile view 20
5.4.3.3. Profiling with using b.nop instruction 21
5.4.3.4. Interpretation of Cycle Count Results 22
5.5 Default Settings 23
5.5.1 Compiler 23
5.5.2 Linker 23
5.5.3 Debugger: 23
6. Project Configuration 24
6.1 Overview 24
6.2 Launch Configuration Dialogs 24
6.2.1 Target Launch Main Tab 24
6.2.2 Target Launch Target Tab 25
6.2.2.1. Target Launch Simulator Options 25
6.2.2.2. Target Launch Hardware Device 26
6.2.2.3. Target Launch Custom Target 26
6.2.3 Target Launch Debugger Tab 26
6.2.4 Target Launch Console Tab 27
6.2.4.1. No console 28
6.2.4.2. Internal console 28
6.2.4.3. PuTTY console 29
6.2.4.4. File console 29
6.2.4.5. Xterm 29
6.2.4.6. Hardware Launch Console Parameters 29
6.2.5 Target Launch Common Tab 29
6.2.6 Debug vs. Run Configurations 30
6.3 HW Debugger 30
6.3.1 Beyond Studio for Windows 30
6.3.2 Beyond Studio for Linux 34
6.3.2.1. Setting Permissions From IDE 35

6.3.2.2. Fixing JTAG Key Permissions Manually 35
6.4 Linker 36
6.5 Compiler 37
6.6 Assembler 37

7. Libraries 38
7.1 newlib 38
7.2 Beyond BA Static Library 38

8. FAQ 39
8.1 Eclipse Cannot Be Launched 39
8.2 Can’t Set a Breakpoint 39
8.3 The ‘Registers’’ Values are Blank 40
8.4 How to Debug Disassembly? 40
8.5 How to Check a Single Module’s Size? 41
8.5.1 Beyond Studio Linux Version 41
8.5.2 Beyond Studio Windows Version 41
8.6 How to Generate a Disassembly File? 42
8.6.1 Beyond Studio Linux Version 42
8.6.2 Beyond Studio Windows Version 42
8.7 Eclipse Workspace in Use or Cannot Be Created 42
8.8 The Hardware Application Cannot Be Re-launched 43
8.9 How to Monitor the Value of a Memory Location/Register 43
8.10 How to Compile a Static Library in Beyond Studio? 44
8.11 How to Use a Static Library in the C/C++ project in Beyond Studio IDE? 47
8.12 How to Turn On the Line Numbers in the Beyond Studio Editor 48

9. Support 49

List of Figures

Figure 1 Beyond Studio configuration file - eclipse.ini 9
Figure 2 Workspace Launcher 11
Figure 3 Welcome Screen 11
Figure 4 Project Definition 12
Figure 5 Project Configuration 13
Figure 6 Project Compilation 14
Figure 7 Debugging Selection 15
Figure 8 Debugger- Configuration Selection 15
Figure 9 Debugger – Perspective Switch 16
Figure 10 Eclipse debugger – initial view 16
Figure 11 Debugger session – re-launching 17
Figure 12 Adding Watchpoints 18
Figure 13 Conditional Watchpoint 18
Figure 14 Debug session with the simulator state view 19
Figure 15 Execution Profile View 20
Figure 16 Console Window with Cycle Counts 22
Figure 17 Launching Simulator Console Window 22
Figure 18 Project settings 24
Figure 19 Launch Debug Configuration – ‘Main’ tab 25
Figure 20 Launch Debug Configuration – ‘Target’ tab 25
Figure 21 Launch Debug Configuration – ‘Debugger’ tab 26
Figure 22 Launch Debug Configuration – ‘Consoles’ tab 28
Figure 23 Launch Debug Configuration – Internal Console 28

Figure 24 Launch Debug Configuration – ‘Common’ tab 30
Figure 25 JTAG Key Driver Installation 31
Figure 26 JTAG Key Installation Wizard 31
Figure 27 JTAG Key Installation Wizard – Device Selection 32
Figure 28 JTAG Key Installation Wizard – INF File 33
Figure 29 JTAG Key Installation Wizard – Finalizing Driver Installation 34
Figure 30 Windows Security Dialog 34
Figure 31 Fixing permissions from IDE 35
Figure 32 Listing USB devices 35
Figure 33 Permissions Fixed 36
Figure 34 Linker options – customized linker script 37
Figure 35 Static Library 38
Figure 36 BA22 Eclipse IDE cannot be launched 39
Figure 37 Setting up a breakpoint 40
Figure 38 Switching to Disassembly mode 41
Figure 39 Workspace cannot be created 43
Figure 40 Memory Monitor 44
Figure 41 Accessing Memory through GDB 44
Figure 42 Beyond BA Static Library 45
Figure 43 Library source import to the project 45
Figure 44 Import MATH library sources to the project 46
Figure 45 Project configuration with the necessary dependencies 46
Figure 46 Static BA library 47
Figure 47 Linker settings 48

1. References

1.1 Related Documents

[1] BA22 Command Line SDK

1.2 Web Sites

CAST Inc, – http://www.cast-inc.com

Beyond Semiconductor – http://www.beyondsemi.com

2. Introduction

Beyond Studio Integrated Development Environment (IDE) integrates the development tools for the BA22uP into a single platform based on the Eclipse open development environment. The Beyond Studio IDE offers easy access to the compilation and debugging tools which are essential in modern software development.

The built-in BA22 Architecture Simulator enables effective and fast code debugging. Together with the SW debugging, the user can connect a hardwareBA22 implementation and continue the debugging process.

The current Beyond Studio version contains a complete Beyond BA toolchain and no separate installation is required.

3. Supplied Software

3.1 Linux Package

The Beyond Studio IDE for Linux is supplied in a single gzip package named BeyondStudio.version.tgz; for instance BeyondStudio.b5.linux-x86_64.tgz**.** The package includes the BA22 tools integrated into the Eclipse platform.

3.2 Windows Package

The Beyond Studio IDE for Windows is supplied in a single Zip package named BeyondStudio.version.zip. The package includes the BA22 tools integrated into the Eclipse platform.

4. Installation

Administrative privileges are not required to install the Beyond Studio IDE in Linux or Windows environment.

4.1 Beyond Studio IDE for Linux

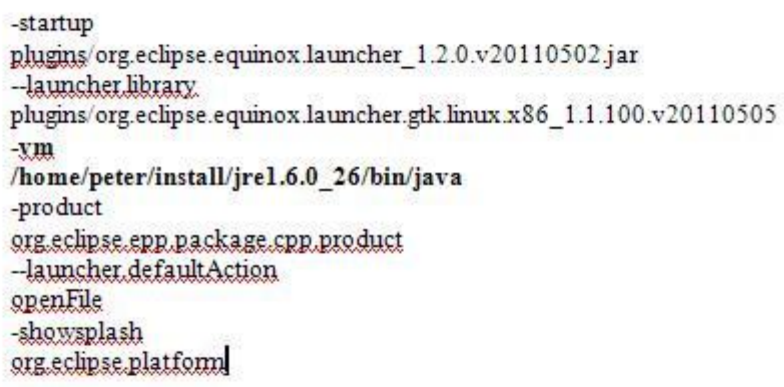
4.1.1 Prerequisites

To run the Beyond Studio IDE the Java JRE is required to be installed. The Java JRE version has to be 1.6.0_25 or newer. Please note that some Linux versions may include by default an outdated Java version.

To check the Java version enter *java –version* in the Linux shell.

The recent Java version can be downloaded from the Java website at: http://www.java.com/en/download/index.jsp

If additional Java version is required, it's necessary to make it default on your system or enter the path to supported version into eclipse.ini (found in the eclipse top-level directory) like shown on Figure 1 Beyond Studio configuration file - eclipse.ini.



```
-startup
plugins/org.eclipse.equinox.launcher_1.2.0.v20110502.jar
--launcher.library
plugins/org.eclipse.equinox.launcher.gtk.linux.x86_1.1.100.v20110505
-vm
/home/peter/install/jre1.6.0_26/bin/java
-product
org.eclipse.epp.package.cpp.product
--launcher.defaultAction
openFile
-showsplash
org.eclipse.platform
```

Figure 1 Beyond Studio configuration file - eclipse.ini

The modification to eclipse.ini should be made only when Beyond Studio is executed on a Java VM which is not default on the used system. The eclipse.ini modification is possible only when the installation package is extracted; for more details see the Package Extraction section.

4.1.2 Package Extraction

The Beyond Studio IDE can be generally installed anywhere in your Linux system however it's recommended to install it the /home/<user name>/BeyondSemi directory and omit any spaces in the directory path. After creating the /home/<user name>/BeyondSemi directory, the installation package (BeyondStudio.version.tgz) should be copied there and the extracting command is to be executed:

tarxvfzBeyondStudio-linux-32/64.tgz

After the successful installation, the Beyond Studio IDE should be available in the home/<user name>/BeyondSemi/BeyondStudio/

4.2 Beyond Studio IDE for Windows

4.2.1 Prerequisites

To run the Beyond Studio IDE the Java JRE is required to be installed. The Java JRE version has to be 1.6.0_25 or newer.

To check the Java version enter *java –version* in the Windows command window.

The recent Java version can be downloaded from the Java website at:
http://www.java.com/en/download/index.jsp

4.2.2 Package Extraction

The Beyond Studio IDE can be generally installed anywhere in your Windows system however it's recommended to install it a directory tree without any spaces. For instance C:\BeyondSemi\

After the successful extraction, the Beyond Studio IDE can be launched by executing eclipse.exe from the C:\BeyondSemi\BeyondStudio\ directory.

4.3 Updates Installation

Beginning from the version 1.0.2, the user can automatically update Beyond Studio IDE. To do so, click "Help" and "Check for updates". During the update process, the user will be asked to confirm installation, confirm the license and also confirm the security exception related to accessing the repository website.

It should be noted that the automatic updates feature has been implemented starting from the 1.0.2 version. In case of an older version, the Beyond Studio IDE should be updated manually.

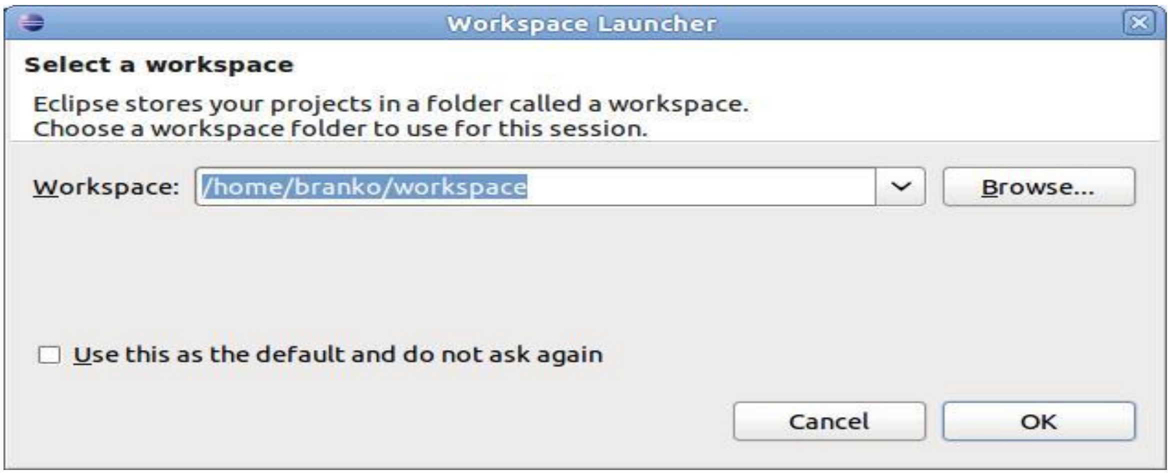
5. Development Flow

5.1 Starting BA22 Eclipse IDE

The BA22 IDE in the Linux version can be launched directly from a console (./eclipse) or by creating a shortcut, etc.

The Windows version can be executed by the eclipse.exe binary.

After the successful launch, Eclipse confirms the chosen workspace as shown on Figure 2 Workspace Launcher. Depending on the user preferences; the default location can accordingly be changed or left unchanged[1]. To skip the 'Workspace prompt', the "Use this as the default and do not ask again" checkbox can be checked.


Workspace Launcher

Select a workspace

Eclipse stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: /home/branko/workspace

☐ Use this as the default and do not ask again

Figure 2 Workspace Launcher

Afterwards, Eclipse presents a Welcome Screen as shown below on Figure 3 Welcome Screen, where the user can learn general information about Eclipse like editor's settings, creating projects, etc.

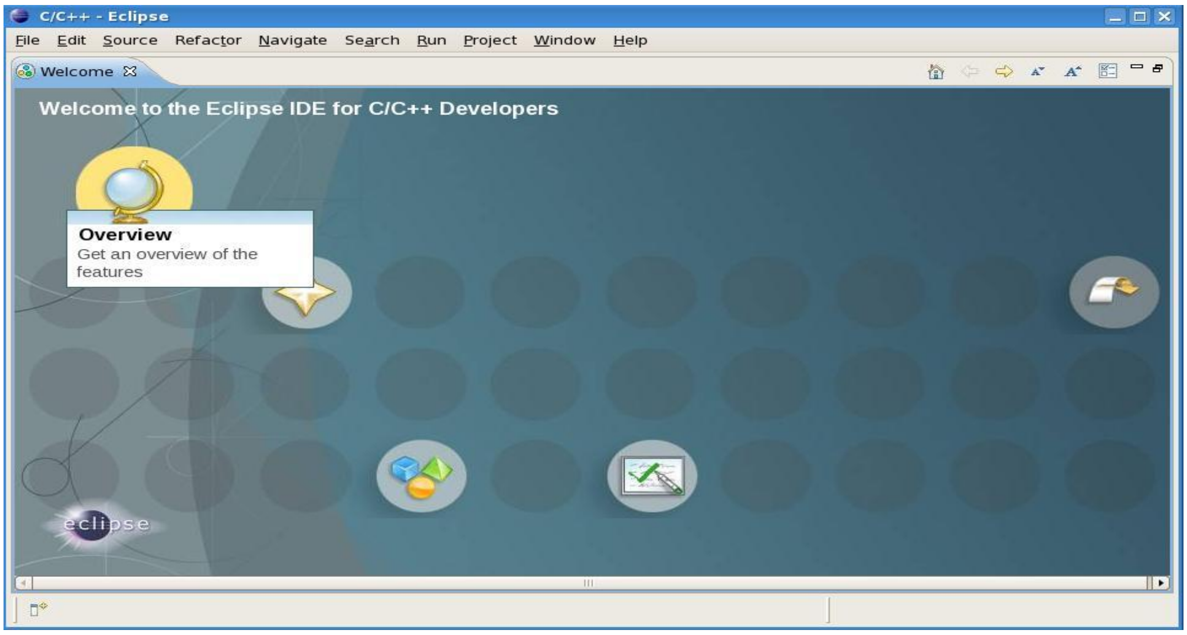

C/C++ - Eclipse
File Edit Source Refactor Navigate Search Run Project Window Help
Welcome
Welcome to the Eclipse IDE for C/C++ Developers
Overview
Get an overview of the features
eclipse

Figure 3 Welcome Screen

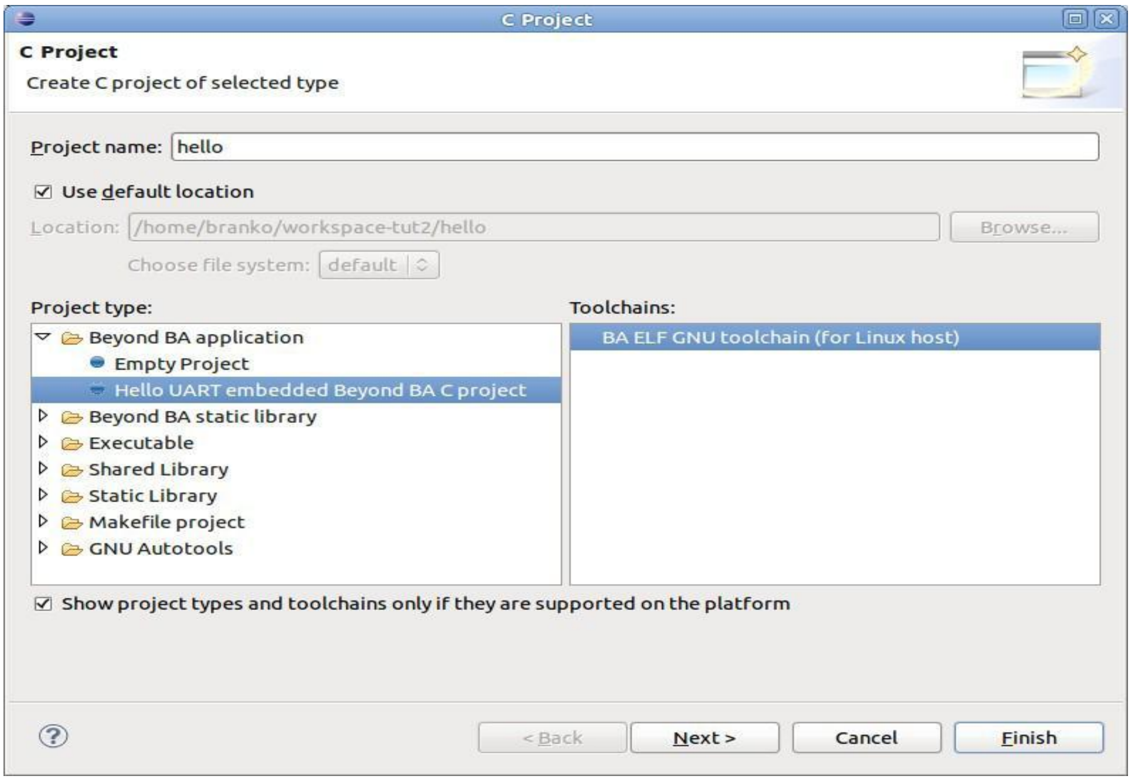
[1] On Windows XP the default location is in »Documents and Settings« folder. Eclipse CDT, the foundation of Beyond Studio has some problems with spaces in file and directory names, so it's best to avoid them. It's recommended to create a workspace or workspaces directory on root of a disk and not to use spaces in folder or source file names.

5.2 Creating New Project for the BA22

To create a new project for BA22, the option “File->New” then “C or C++ project” should be chosen. On the next page, it’s essential to select “Beyond BA application” or “Beyond BA static library” project type. If anything else is selected, the project will not target the BA22 architecture.

The “Beyond static library” project type is discussed further in the Beyond BA Static Library section, while the discussion about the BA22 application project follows.

After selecting the sample “Hello Uart” project, the user has to specify the project’s name and click “Next” as depicted on Figure 4 Project Definition


C Project

C Project
Create C project of selected type

Project name: hello

☑ Use default location

Location: /home/branko/workspace-tut2/hello

Choose file system: default

Project type:
- Beyond BA application
  - Empty Project
  - Hello UART embedded Beyond BA C project
- Beyond BA static library
- Executable
- Shared Library
- Static Library
- Makefile project
- GNU Autotools

Toolchains:
- BA ELF GNU toolchain (for Linux host)

☑ Show project types and toolchains only if they are supported on the platform

Figure 4 Project Definition

On the next screen, the user will be asked to specify two parameters for the code generator.

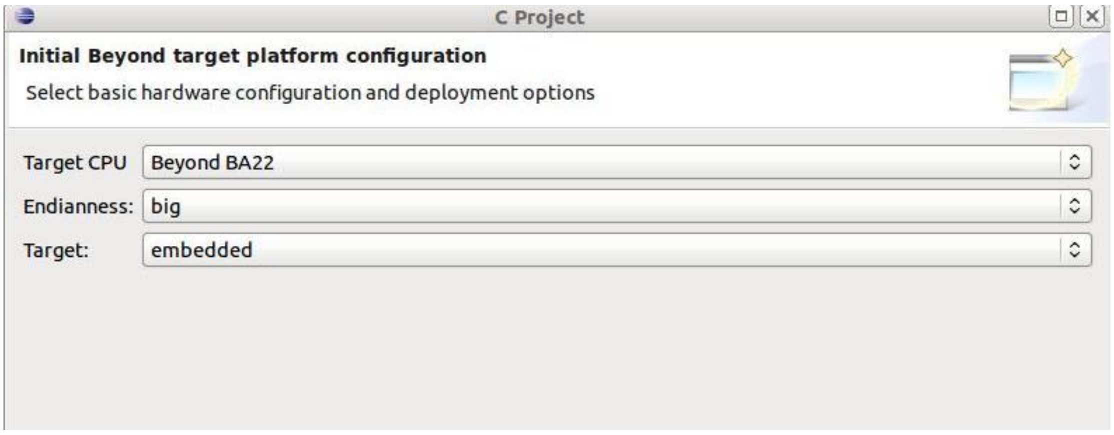
The first parameter is the RAM size needed for the linker script generation. After this step, the generated script can be used with the Instruction Set Simulator (ISS) or the demo board. In the “RAM Size” field, the user should specify the amount of memory on the BA22 board; the RAM is located in continuous region from address 0x0. The size can be entered in bytes (e.g. 8192), kilobytes (as for example 16k) or megabytes – for instance 1M. When the RAM size is entered in bytes, the size has to be divisible by 4.

The “RAM size” parameter should never exceed the total amount of RAM available in the hardware implementation. The RAM field can be also left with its default value, if the user doesn’t plan on any further hardware testing.

The second field contains the greeting’s text specific only to the “Hello UART” example application.

The succeeding step can be just confirmed by clicking “Next” since the choice between “Debug” and “Release” configurations is constantly present during the application development.

At the next page, the user has to select the hardware configuration for the project as depicted on Figure 5 Project Configuration



C Project

Initial Beyond target platform configuration

Select basic hardware configuration and deployment options

Target CPU: Beyond BA22
Endianness: big
Target: embedded

Figure 5 Project Configuration

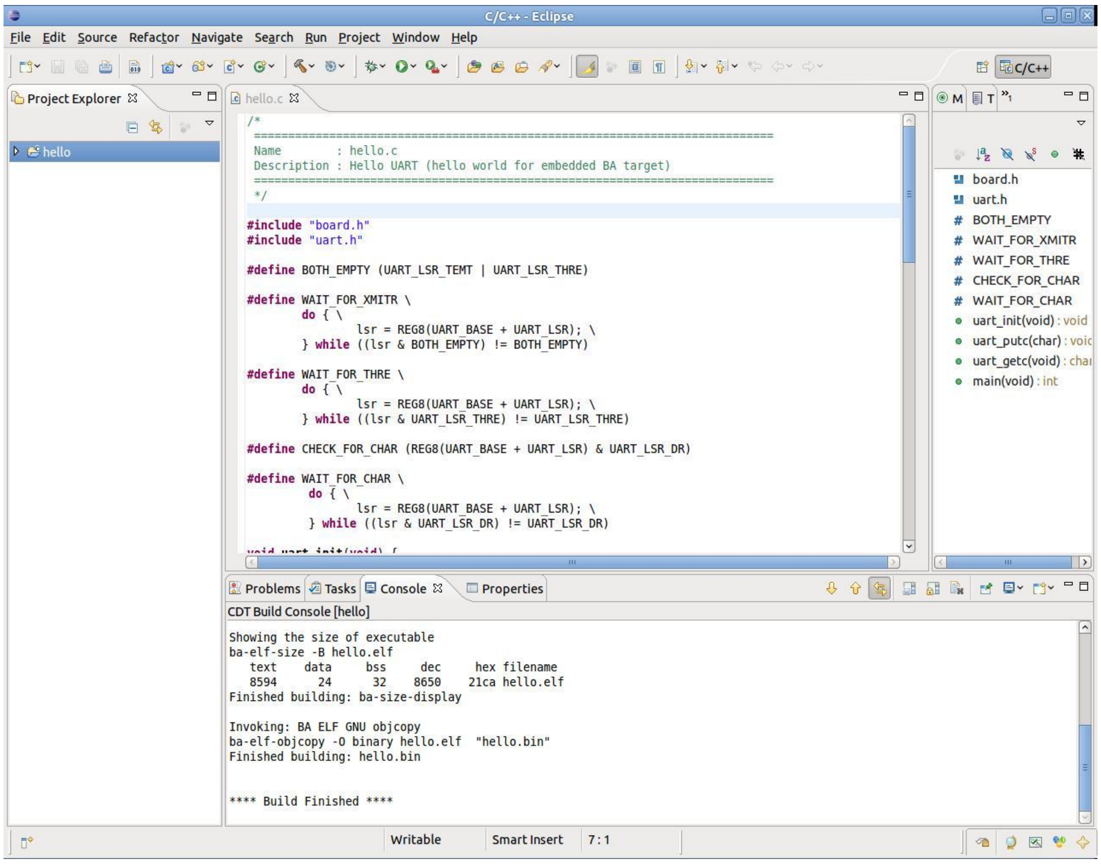
If the project is dedicated to target a physical BA22 implementation, it's strongly recommended to choose a configuration which matches the HW (Target CPU and endianness)

After the configuration is specified, the "Finish" button ends the configuration project phase. At that moment, the wizard will create a few source files and open the main source file of the new project. Afterwards, the code can be freely modified, new files can be added and the project settings can be changed if needed.

5.3 Building your Application

The project's compilation can be launched either by clicking the "hammer" icon or simply by choosing "Build Project" from the project menu. In either case, the project should be active during the application building.

The output messages of the compilation process are displayed in the Console page; that page is located by default below the source window as shown on Figure 6 Project Compilation. The console window also displays the executable's size. Since the displayed size applies to the complete application, it's highly recommended to get acquainted with information that is added by the default to the user project. This topic is described in detail in the Default Settings section



```
/*
 ==========================================================================
 Name        : hello.c
 Description : Hello UART (hello world for embedded BA target)
 ==========================================================================
 */

#include "board.h"
#include "uart.h"

#define BOTH_EMPTY (UART_LSR_TEMT | UART_LSR_THRE)

#define WAIT_FOR_XMITR \
        do { \
                lsr = REG8(UART_BASE + UART_LSR); \
        } while ((lsr & BOTH_EMPTY) != BOTH_EMPTY)

#define WAIT_FOR_THRE \
        do { \
                lsr = REG8(UART_BASE + UART_LSR); \
        } while ((lsr & UART_LSR_THRE) != UART_LSR_THRE)

#define CHECK_FOR_CHAR (REG8(UART_BASE + UART_LSR) & UART_LSR_DR)

#define WAIT_FOR_CHAR \
         do { \
                lsr = REG8(UART_BASE + UART_LSR); \
         } while ((lsr & UART_LSR_DR) != UART_LSR_DR)
```

```
Showing the size of executable
ba-elf-size -B hello.elf
   text    data     bss     dec     hex filename
   8594      24      32    8650    21ca hello.elf
Finished building: ba-size-display

Invoking: BA ELF GNU objcopy
ba-elf-objcopy -O binary hello.elf  "hello.bin"
Finished building: hello.bin


**** Build Finished ****
```

Figure 6 Project Compilation

By default, the C/C++ projects have two configurations: Debug (default) and Release. The Debug configuration contains all the data which are needed for debugging, but the code is not optimized. This configuration is recommended for debugging or code testing.

To create a size-optimized executable, the Release configuration should be selected by pressing an icon right to the “build hammer” icon. Afterwards, the application can be built again.

If it’s desired to fully rebuild the project, the project has to be selected and the - “Project ->Clean” option must be chosen. With the next application building, all the files will be created from scratch.

5.4 Debugging

After successful compilation, the user may want to debug his/her source code using the BA22 software debugger. This process can be initiated by pressing the “bug” icon in a toolbar; at that moment the project has to be active or one of its sources has to be edited.

Afterwards, the user has to go over two dialogs windows as shown on Figure 7 Debugging Selection and Figure 8 Debugger- Configuration Selection. In the first window the “Beyond BA application” should be chosen, and any choice is fine in the second one.[2]

[2]Both dialog windows are planned to be removed in the future Beyond Studio IDE releases

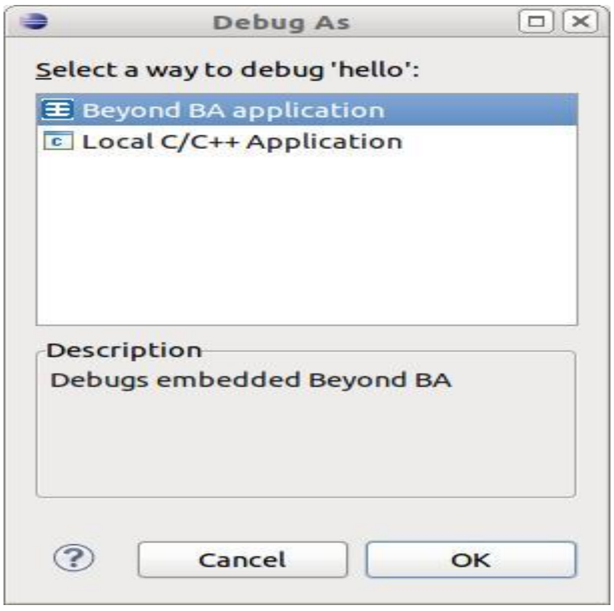


Figure 7 Debugging Selection

It is strongly suggested to follow the above choice since selecting anything else will not deploy the application properly

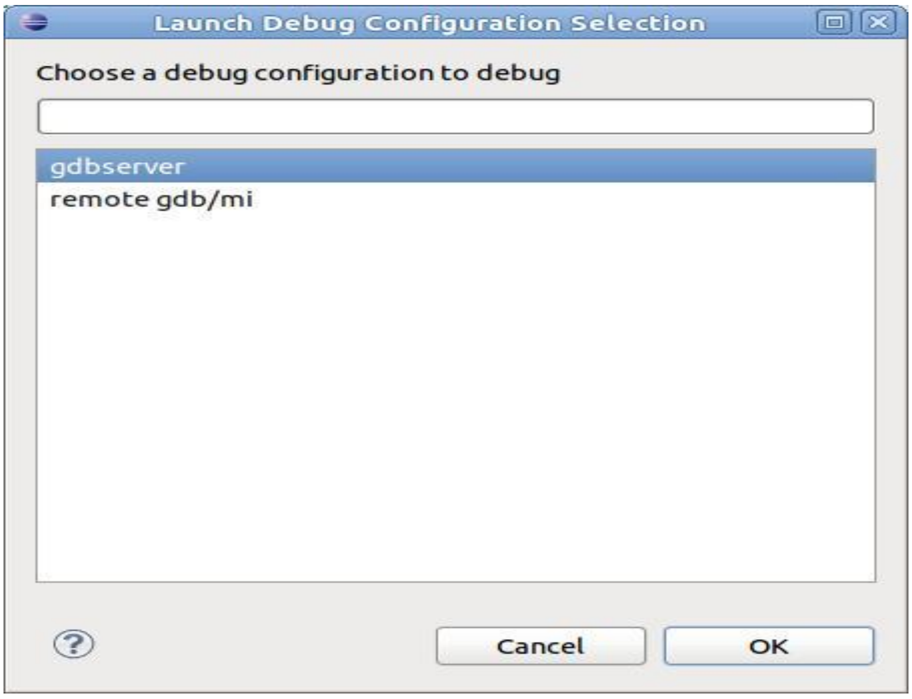
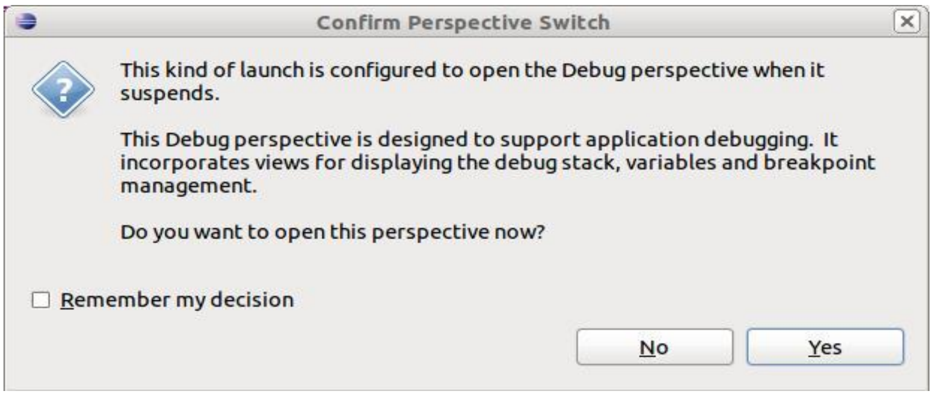

Launch Debug Configuration Selection
Choose a debug configuration to debug
gdbserver
remote gdb/mi
Cancel
OK

Figure 8 Debugger- Configuration Selection

When the debugging process is launched for the first time (in a new workspace), another dialog will be presented as shown in Figure 9 Debugger – Perspective Switch At that moment, the Beyond Studio IDE offers switching to the Debug perspective. Perspectives in Eclipse based IDEs represent layout of windows, options that are available and so on. During debugging users usually want to enter the Debug perspective, so it’s recommended to just check “Remember my decision” and select the “Yes” button.



Confirm Perspective Switch

This kind of launch is configured to open the Debug perspective when it suspends.

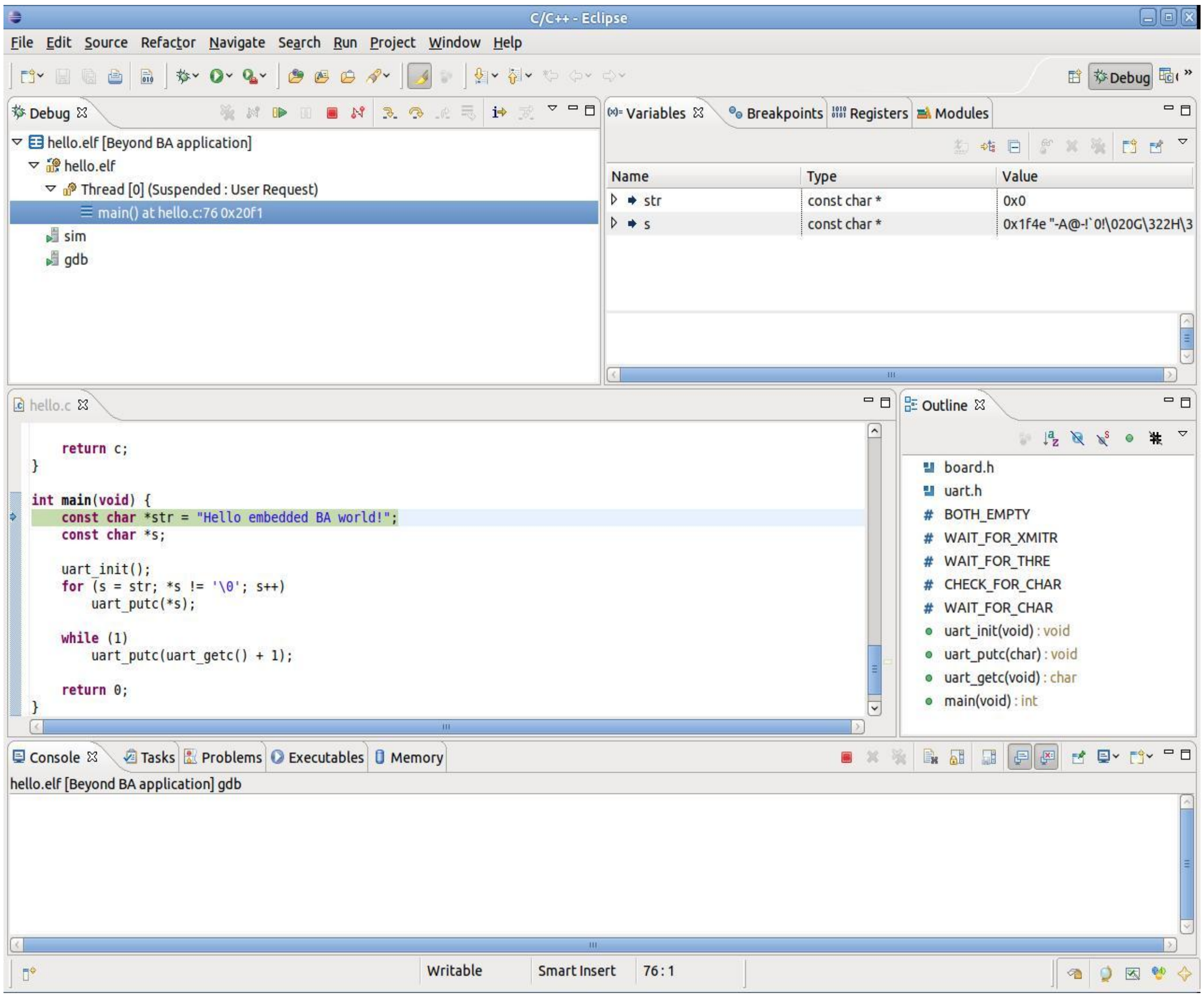
This Debug perspective is designed to support application debugging. It incorporates views for displaying the debug stack, variables and breakpoint management.

Do you want to open this perspective now?

☐ Remember my decision

Figure 9 Debugger – Perspective Switch

After the “debugger perspective switch”, the default debugger perspective is open and the program is stopped at the “main” function by default - Figure 10 Eclipse debugger – initial view. From that moment, the user can click the “play” icon (Resume-F8), or use the Step Into (F5) or Step Over(F6) to execute the code. The user can also set a breakpoint in C/C++ or assembler code and click Resume. This will result in the code execution till the set breakpoint.



```
hello.elf [Beyond BA application]
  hello.elf
    Thread [0] (Suspended : User Request)
      main() at hello.c:76 0x20f1
  sim
  gdb
```

Name	Type	Value
str	const char *	0x0
s	const char *	0x1f4e "-A@-!`0!\020G\322H\3

```
    return c;
}

int main(void) {
    const char *str = "Hello embedded BA world!";
    const char *s;

    uart_init();
    for (s = str; *s != '\0'; s++)
        uart_putc(*s);

    while (1)
        uart_putc(uart_getc() + 1);

    return 0;
}
```

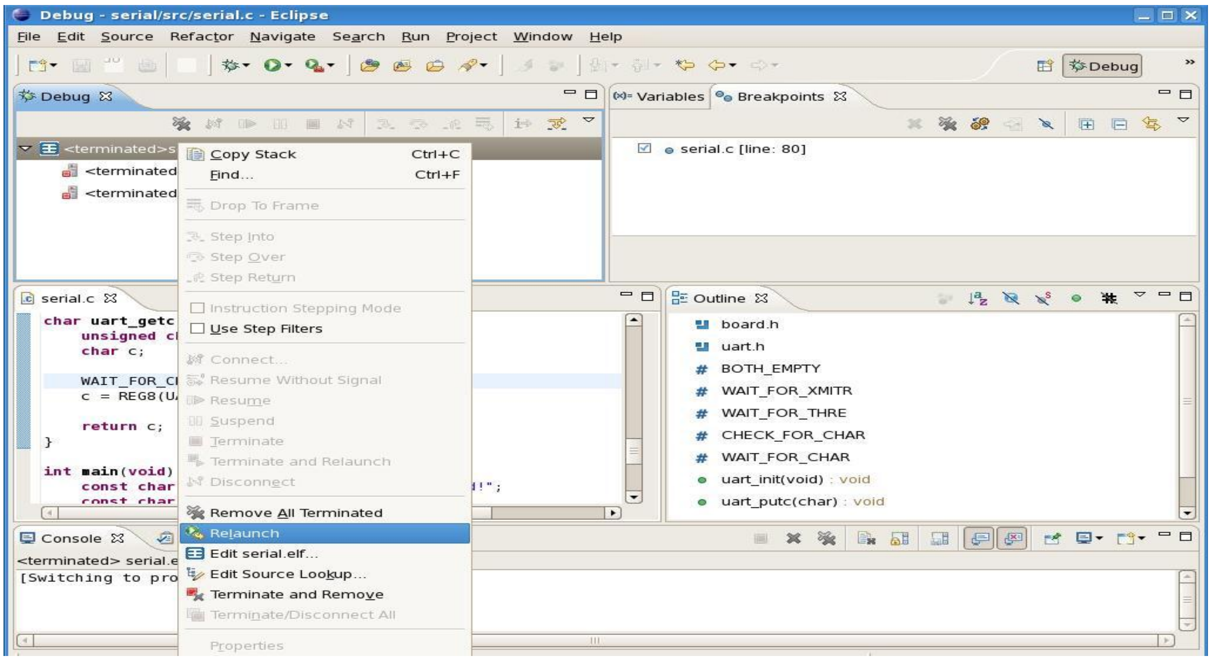
```
board.h
uart.h
BOTH_EMPTY
WAIT_FOR_XMITR
WAIT_FOR_THRE
CHECK_FOR_CHAR
WAIT_FOR_CHAR
uart_init(void) : void
uart_putc(char) : void
uart_getc(void) : char
main(void) : int
```

```
hello.elf [Beyond BA application] gdb
```

Figure 10 Eclipse debugger – initial view

The debug tab contains icons that let step through the source code, step through assembly instructions, run program etc. The “Stop” (red) button stops the debugger session whereas the “I with arrow” icon toggles instruction stepping mode and opens the disassembly window. The present view can be easily customized by adding new views by using the Window->ShowView option.

The debugging session can be stopped by pressing the Stop button (Ctrl+F2). At that moment, the user can re-launch the debugging session as depicted on Figure 11 Debugger session – re-launching or jump to the “C/C++” perspective to modify the code. To go back to the “C/C++” perspective, the user has to press the “C/C++” button at the top right corner of the Eclipse window or through “Window” / “Open perspective” / “C/C++” option. It’s also worth to remember that the debugging session can be suspended when needed.



```
char uart_getc
unsigned c
char c;

WAIT_FOR_C
c = REG8(U

return c;
}

int main(void)
const char
const char
```

```
<terminated> serial.e
[Switching to pro
```

Figure 11 Debugger session – re-launching

5.4.1 Breakpoints

In Beyond Studio, the breakpoints are set by double clicking in the margin area left of the source code. The same effect can be achieved through the menu by clicking: Run->Toggle Breakpoint or the Ctrl-Shift-B key combination.

Setting a breakpoint to a line of code means that the program execution will be automatically suspended when the program flow hits that particular line of code. When the execution is suspended at a breakpoint, the user can examine the program state and then resume its execution.

A breakpoint can be easily removed by double click on it. It can also be temporarily disabled by right click on its symbol and selecting disable (and later enable).

To set a conditional breakpoint, the user should make right click on the breakpoint and select the “Breakpoint properties” option. For instance, the condition can be set like i%5 == 0. This would cause the breakpoint to suspend the execution only when that condition is true. The breakpoints that have set additional rules are displayed with additional question mark next to circle.

In the debug perspective, all the breakpoints are shown in their separate panel. This view can be used to easily set the breakpoints properties.

5.4.2 Watchpoints

Watchpoints can be used to monitor variables and read or write accesses to a particular memory location. To set up a watchpoint, the user needs to press down an arrow at the right corner of the breakpoint window as shown on Figure 12 Adding Watchpoints

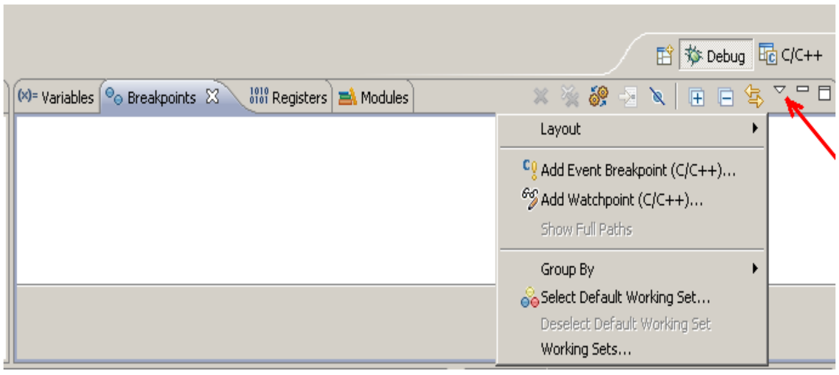

Debug
C/C++
Variables
Breakpoints
Registers
Modules
Layout
Add Event Breakpoint (C/C++)...
Add Watchpoint (C/C++)...
Show Full Paths
Group By
Select Default Working Set...
Deselect Default Working Set
Working Sets...

Figure 12 Adding Watchpoints

In the pop-up menu, the user can enter an expression to watch; usually a variable. Besides a variable, it can be also a memory location, which is entered as a pointer.

Sample watch expressions are as follows:

- x (or name of any other variable)
- *(short*) 0x2488 (or any other pointer)

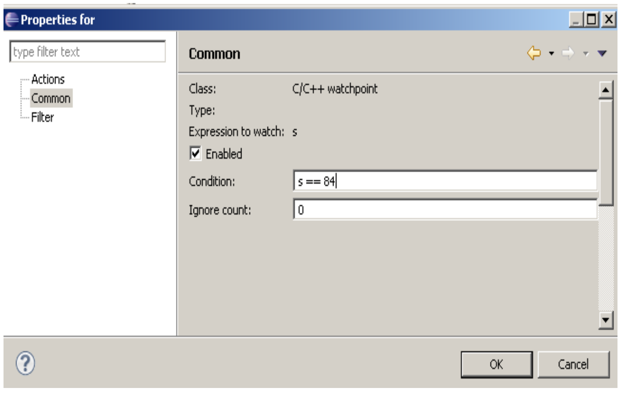
The first expression assigns a watchpoint to a variable *x*. If the contents of the memory location where *x* is stored, is changed by the program, the watchpoint will trigger.

The second expression creates a watchpoint for the contents of 16 bit (short) memory location at 0x2488[3].

Watchpoints are triggered whenever the variable or memory location defined by the watch expression has been modified. Watchpoints are triggered immediately; the cause is usually a previous instruction or a statement, not the one where the debugger stops at.

Regular watchpoints trigger only when a variable or memory location is modified. However, watchpoints that trigger when a memory location is accessed are also feasible when the BA22's hardware debug unit is configured to support watchpoints.

Similarly as in the case of breakpoints, an additional condition can be assigned to a watchpoint. For instance: the user may set a watchpoint to variable *s*, and then add a condition to trigger it only when its value becomes 84 - Figure 13 Conditional Watchpoint



Class: C/C++ watchpoint
Type:
Expression to watch: s
☑ Enabled
Condition: s == 84
Ignore count: 0

Figure 13 Conditional Watchpoint

[3] Using only "0x2488" would never trigger since it'd be treated as a constant.

The condition can be added by right-click on the watchpoint in the “breakpoints” window; then the “breakpoint properties” should be selected.

When the watchpoints are not supported by the hardware debug unit, they slow down the program execution. If the number of set watchpoints exceeds the number of watchpoints implemented in hardware, a software watchpoint will be used instead.

Currently, there are a few known issues which should be taken into account when using watchpoints. Those issues will be addressed in the future versions of Beyond Studio IDE.

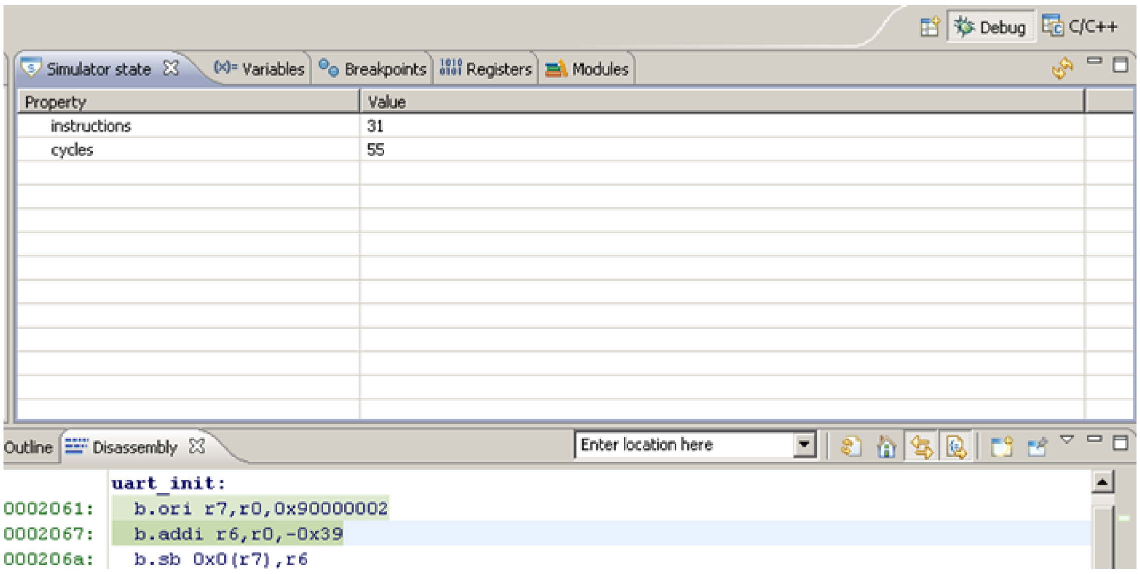
- When the debug session is restarted, watchpoints are not properly reinserted. The current workaround is to set a breakpoint in the program, then disable and re-enable each all the watchpoints.
- If a read (or read/write) watchpoint is set, and the user tries “stepping” after it is hit, the watchpoint will immediately get triggered again. The workaround is to disable and re-enable that kind of watchpoint after it has been triggered.

5.4.3 Profiling

Starting from the 1.0.2 version, the Beyond Studio IDE supports cycle accurate simulation of QMEM targets. This chapter describes features introduced in version 1.1.0 so it’s recommended to make sure that the latest version is used.

5.4.3.1. Simulator State View

The easiest way to see the cycle counts is by using the simulator state view. To have it enabled, select “Windows”->”Shows View” -> Other” -> Beyond BA custom views” -> “Simulator state”. The cycle count window can be placed at any place in the debug perspective view as shown on Figure 14 Debug session with the simulator state view



Property	Value
instructions	31
cycles	55

```
uart_init:
0002061:   b.ori r7,r0,0x90000002
0002067:   b.addi r6,r0,-0x39
000206a:   b.sb 0x0(r7),r6
```

Figure 14 Debug session with the simulator state view

The debug session should be started in simulator. There are two fields showed in the simulator state view:

- *Instructions* means the number of instructions executed in the current session

- *Cycles* is the number[4] of the CPU cycles[5] spent in current debug session.

Then, the user can proceed with the debug session as usually. Whenever the debugger stops (on breakpoint, after a single step, etc.) the counters will be updated with new values.

A sample guideline on how to measure the time spent on executing a foo() function is provided below:

1. Put a breakpoint at the function call location.
2. Start the debug session and run to that breakpoint. Write down the value of the counters.
3. Use »Step over (F6)« to execute that function.
4. Calculate the difference of counters; that's the number of instructions and cycles used by that function.

The exectuion time spend by the entire program can be achieved by simply running the SW (in debug or run mode) while the simulator state view is shown.

When the program is normally terminated[6], the counters will be updated and shown on a green background.

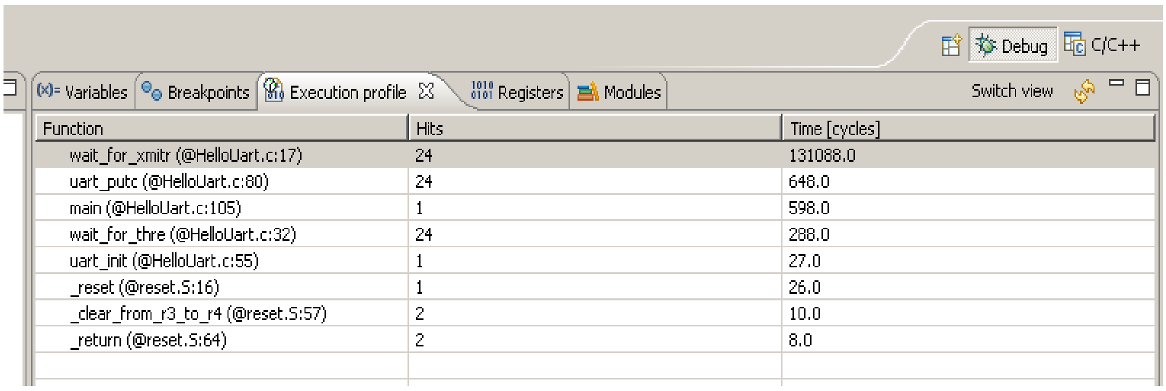
The background color of the simulator counters can be as follows:

- Green: the program has terminated normally and the shown values are the final measurement for the complete program.
- Red: the program has terminated by some other means. The shown values are final, but do not represent the complete program.
- White: the session is still active (the program is running). The values are refreshed every time the debugger is paused. The value of the counters can be refreshed manually by pressing the refresh button at the right top corner of the simulator state view.

5.4.3.2. Execution profile view

When the code is run in Instruction Set Simulator, at the same moment its execution profile is being collected. The execution profile view can be added to the current perspective by selecting “Window -> Show View -> Other -> Beyond BA custom views -> Execution profile”. The view can be placed anywhere in the current perspective by dragging it with a mouse.

Figure 15 Execution Profile View depicts a typical view of the profiling window.



Function	Hits	Time [cycles]
wait_for_xmitr (@HelloUart.c:17)	24	131088.0
uart_putc (@HelloUart.c:80)	24	648.0
main (@HelloUart.c:105)	1	598.0
wait_for_thre (@HelloUart.c:32)	24	288.0
uart_init (@HelloUart.c:55)	1	27.0
_reset (@reset.S:16)	1	26.0
_clear_from_r3_to_r4 (@reset.S:57)	2	10.0
_return (@reset.S:64)	2	8.0

Figure 15 Execution Profile View

The profiling view can show the number of clock cycles spent on executing:

[4] This count does not include cycles CPU spent stalled (waiting for debugger). Also note that the cycle count is calculated for a system using QMEM memory and that only CPU is simulated in cycle accurate manner.
[5] To calculate time spent at target frequency divide cycle count with frequency (in Hz).
[6] Program is »terminated« on simulator when it runs to »b.nop 1« assembly instruction.

- Each function
- Each line of the source code
- Each instruction

To switch among these modes, the user can click the “Switch view” button which is located in the top right corner of the execution profile view. Only the functions, lines and instructions that have already been executed are presented on the profiling list. The list is sorted by the cycles spent from the code segment that used most time downwards.

By double clicking at the line in the profile view, the source code editor will open the respective software location.

Remarks:

- Only the code that was actually executed is shown in profile view
- The calculated time (cycles) does NOT include any time spent in the code called from that location. The time shown is spent in that function/line/instruction alone.
- The hit count is calculated considering the first machine instruction of each code block. This may give a bit unexpected results, as for example in a typical for loop, the first instruction of the loop is run only once.
- Some lines of the source code (like variables’ declarations) may result in no code at all. For this reason those lines will never show up in the execution profile.
- The BA22 compiler uses various techniques to optimize the code including code restructuring (e.g. loop unfolding, move loop invariant code etc). The profile execution results are being calculated for the code that is executed and sometimes it can take some effort to understand the results.

The execution profile view uses the same color code as the simulator state view. The green background means that the shown profile is the final measurement for a program that executed completely. (until »b.nop 1«). The red background means the final profile is shown for a program that has been terminated differently than “b.nop 1” and the white background is used to display a profile for a program that is still executing.

5.4.3.3. Profiling with using b.nop instruction

The user can also profile the code by placing the »b.nop 5« assembly instructions at the locations where the clock count counters are desired to be printed, and then simply running whole program through.

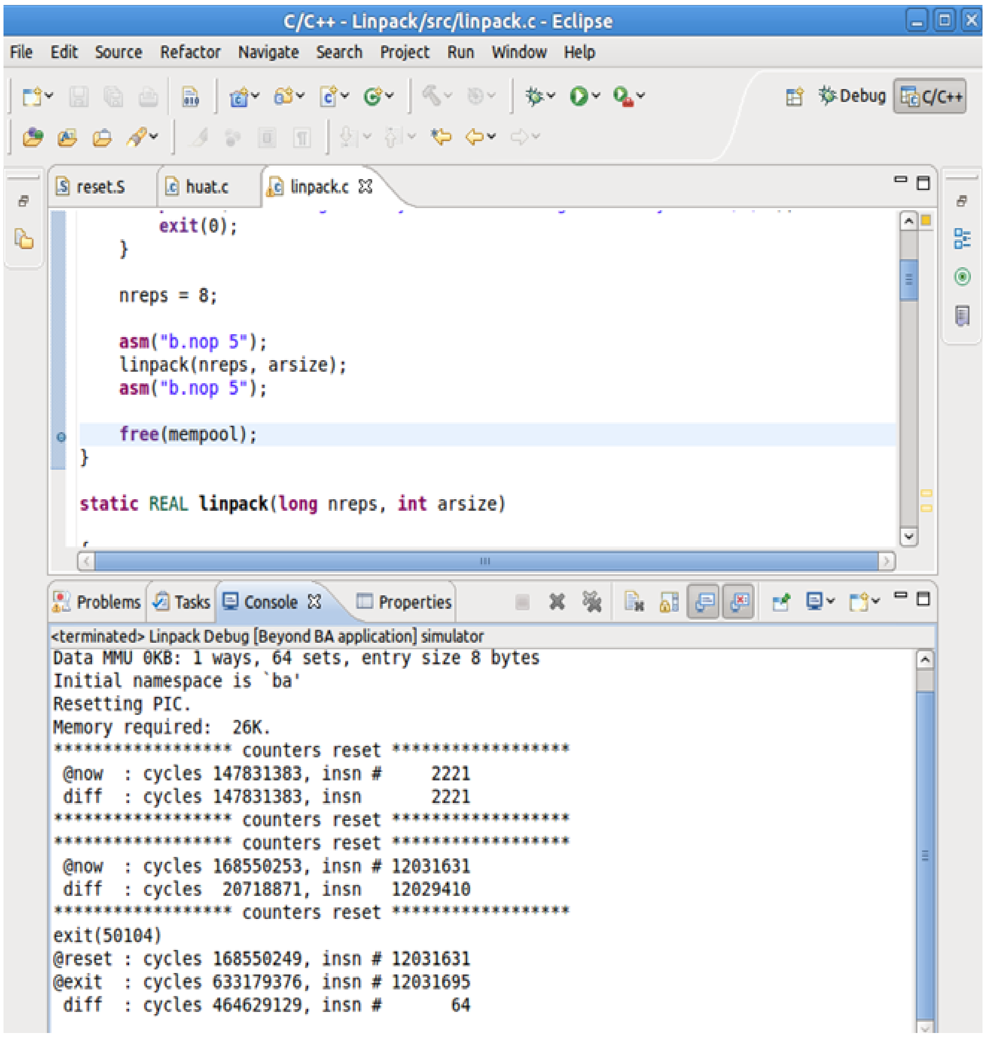
Every time, where the cycle timer needs to be reset, the `“asm("b.nop 5");”` instruction has to be added. It is absolutely required to place this instruction at the beginning of area which is to be benchmarked. It should be noted that the CPU starts running before the user program is started and without an appropriate reset, the printed value would show cycles spent on uploading binary through the debug interface, etc.

In most cases, the benchmarking/profiling can be done as follows:

```
asm("b.nop 5");
/* some code you want to benchmark */
asm("b.nop 5");
```

Each of the special “b.nop 5” instructions tells the simulator to print the current instruction and cycle counts, and then reset those values.

The results are obtained from simulator console window as shown on Figure 16 Console Window with Cycle Counts


C/C++ - Linpack/src/linpack.c - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Debug
C/C++
reset.S
huat.c
linpack.c
Problems
Tasks
Console
Properties

```
        exit(0);
    }

    nreps = 8;

    asm("b.nop 5");
    linpack(nreps, arsize);
    asm("b.nop 5");

    free(mempool);
}

static REAL linpack(long nreps, int arsize)
```

```
<terminated> Linpack Debug [Beyond BA application] simulator
Data MMU 0KB: 1 ways, 64 sets, entry size 8 bytes
Initial namespace is `ba'
Resetting PIC.
Memory required:  26K.
****************** counters reset ******************
 @now  : cycles 147831383, insn #     2221
 diff  : cycles 147831383, insn       2221
****************** counters reset ******************
****************** counters reset ******************
 @now  : cycles 168550253, insn # 12031631
 diff  : cycles  20718871, insn   12029410
****************** counters reset ******************
exit(50104)
@reset : cycles 168550249, insn # 12031631
@exit  : cycles 633179376, insn # 12031695
 diff  : cycles 464629129, insn #       64
```

Figure 16 Console Window with Cycle Counts

If the simulator console is not automatically displayed, it can be selected manually by clicking down the arrow next to the monitor icon in “Console” view toolbar. Then the console tagged with “<name of application running>simulator” should be chosen as depicted on Figure 17 Launching Simulator Console Window

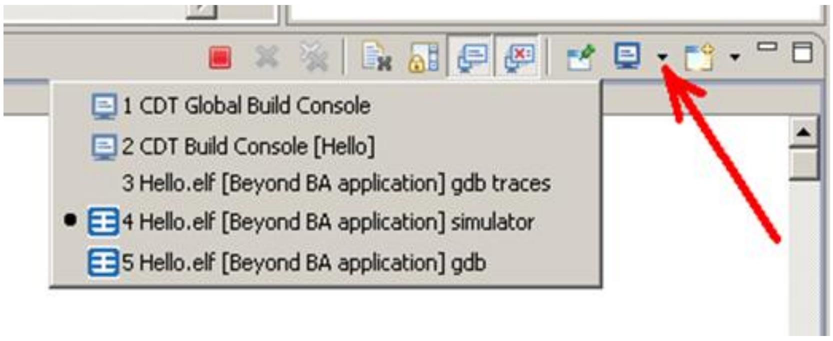

1 CDT Global Build Console
2 CDT Build Console [Hello]
3 Hello.elf [Beyond BA application] gdb traces
4 Hello.elf [Beyond BA application] simulator
5 Hello.elf [Beyond BA application] gdb

Figure 17 Launching Simulator Console Window

5.4.3.4. Interpretation of Cycle Count Results

Each “b.nop 5” prints out the difference in instructions and cycles from the previous “b.nop 5” statement. On Figure 16 Console Window with Cycle Counts, the following text is depicted:

****************** counters reset ******************

```
 @now  : cycles 147831383, insn #     2221
 diff  : cycles 147831383, insn       2221
****************** counters reset ******************
****************** counters reset ******************
 @now  : cycles 168550253, insn # 12031631
 diff  : cycles  20718871, insn   12029410
****************** counters reset ******************
```

The first “counters reset” block represents the cycles spent by the simulator on uploading the binary, waiting for modules to set up, etc.

The second “counters reset” shows more interesting values from the user point of view. The line starting with “diff” presents the difference in cycles and instruction count from the previous “b.nop 5”. Since the previous “b.nop” was inserted just before the function targeted to be benchmarked, this difference effectively shows the clock count spent in that function.

The difference is 20,718,871 cycles and 12,029,410 instructions[7].

5.5 Default Settings

The BA22 Eclipse IDE default settings automatically apply with every new project. With time however, the user might want to change some of the default Eclipse settings.

5.5.1 Compiler

The default “Release Configuration” is compiled with the size optimization - “-Os” switch, whereas the default “Debug Configuration” is compiled without any optimization: “O0” switch.

More information about the compilers settings can be found in the Compiler section.

Any default configuration includes “startfiles” which add about 8KB of code to address the necessary uP initial configuration like clearing the memory sections, setting up the stack pointer etc. After every compilation process, the size of the executable is shown in the “Console” window. This result also includes the size of the “startfiles”.

It’s possible to check out the size of a single module (C/C++ or assembler source file) compiled for BA22. That procedure is described in the How to Check a Single Module’s Size? section.

5.5.2 Linker

The default linker script sets the memory size for 32MB. More information about how to use a customized linker script (ram.ld) can be found in the Linker section.

5.5.3 Debugger:

The default Eclipse configuration sets the breakpoint at the “main” function. More information about how to change the debugger settings can be found in the Target Launch Debugger Tab section.

[7] From the cycles count, it can be also calculated how much time would be required for the benchmarked function to run on actual CPU. For example: on 63 MHz CPU, that would take 20718871/63000000 or approximately 0.33 seconds.

6. Project Configuration

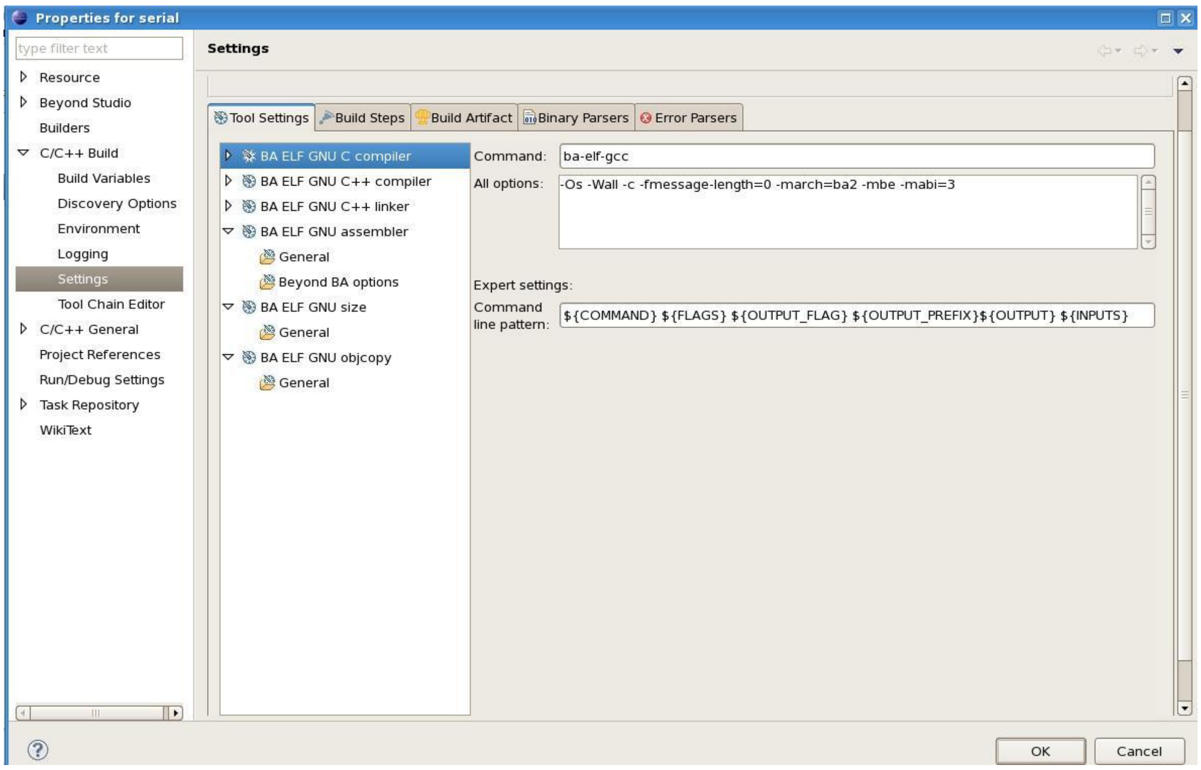
6.1 Overview

The Beyond Studio IDE similarly to all Eclipse CDT based IDEs supports the notion of launch configurations. A launch configuration is a collection of settings which define how application is being launched (run, debugged, etc.). Each project may have many launch configurations associated with it; for instance one configuration launches application in a simulator while another in hardware.

When the “debug” or “run” shortcut is selected for a first time in a project, a default launch configuration is automatically created. By default, the applications are executed on the simulator, which is configured to match the system built for according to the user preferences. In other words: the simulator simulates the hardware which was selected for a particular project.

During a debugging session, a program is executed up to “main” function, and then suspended, while in the “run” mode the program keeps going until is it stopped by the user.

Besides the Launch configuration, the user can change a wide range of compiler, linker and assembler options (they are presented further in this section). To do it, the user has to launch the project properties by choosing Project->Properties->C/C++ Build->Settings



Command: ba-elf-gcc

All options: -Os -Wall -c -fmessage-length=0 -march=ba2 -mbe -mabi=3

Expert settings:

Command line pattern: ${COMMAND} ${FLAGS} ${OUTPUT_FLAG} ${OUTPUT_PREFIX}${OUTPUT} ${INPUTS}

Figure 18 Project settings

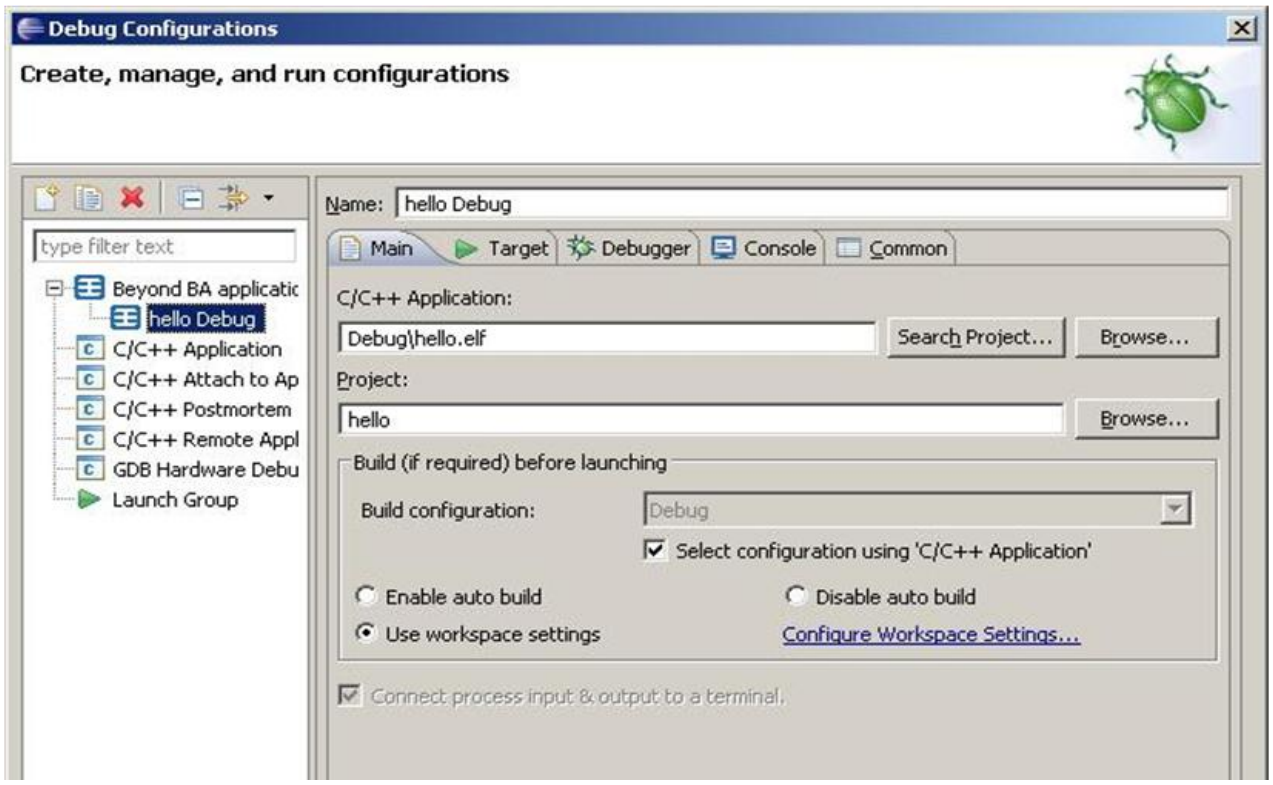
6.2 Launch Configuration Dialogs

The debug and run configurations can be configured by pressing a down arrow next to the “debug” or “run” icons and then selecting the “Debug configurations” or “Run configurations” in the context menu.

After choosing the configuration, the default view is set at the “Beyond BA application”. This configuration type is customized to the embedded BA22 programming. It consists of five tabs: Main, Target, Debugger, Console and Common. All of them are reviewed in the following sections.

6.2.1 Target Launch Main Tab

On the “Main” tab, the project and binary associated with a launch are selected. The default settings are sufficient for most situations. In case of launching a project which doesn’t have a binary (because it was not compiled yet), the “C/C++ Application” field may be blank. In that event, the project binary is chosen by the “Search Project” button.


Debug Configurations

Create, manage, and run configurations

Name: hello Debug

C/C++ Application: Debug\hello.elf

Project: hello

Build (if required) before launching

Build configuration: Debug

[✓] Select configuration using 'C/C++ Application'

( ) Enable auto build ( ) Disable auto build

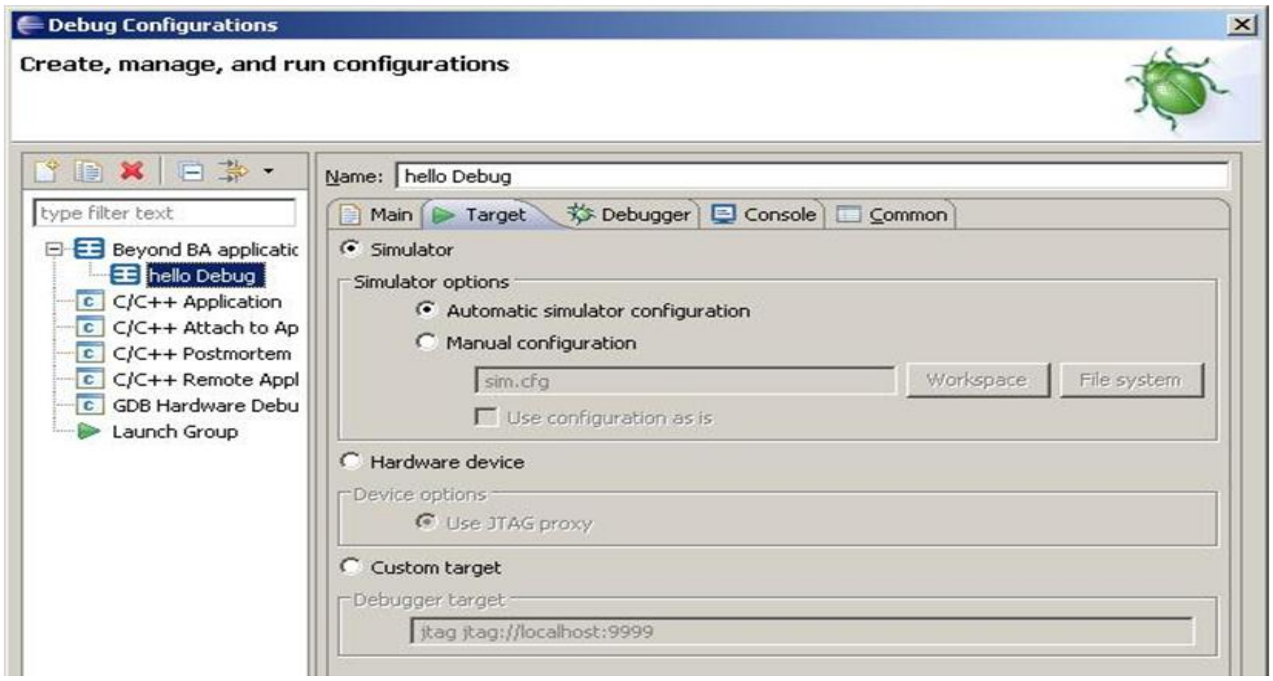
(●) Use workspace settings

[✓] Connect process input & output to a terminal.

Figure 19 Launch Debug Configuration – 'Main' tab

6.2.2 Target Launch Target Tab

The "Target" tab allows specifying how the application should be deployed. It defaults to simulator with automatic configuration.


Debug Configurations

Create, manage, and run configurations

Name: hello Debug

(●) Simulator

Simulator options

(●) Automatic simulator configuration

( ) Manual configuration

sim.cfg

[ ] Use configuration as is

( ) Hardware device

Device options

(●) Use JTAG proxy

( ) Custom target

Debugger target

jtag jtag://localhost:9999

Figure 20 Launch Debug Configuration – 'Target' tab

6.2.2.1. Target Launch Simulator Options

This option deploys the application to BA simulator (BASIM) and doesn't require any hardware. The simulator runs on the host machine and emulates various aspects of BA based embedded system (CPU with BA2 ISA, memories, MMUs, caches, UART, etc).

The simulator can be configured with automatic or manual settings.

Automatic configuration: This option is chosen by default. It means the simulator will be configured to the typical configuration while its minor features will be adapted to the project's configuration (CPU, endianness)

Manual configuration: This option allows specifying a customized simulator configuration file. For more details please contact CAST.

6.2.2.2. Target Launch Hardware Device

This option allows deploying application to the BA22-based SOC through JTAG. Currently this option supports the JTAG key connections to the board / device. For instance, the "Hardware device" option is suitable for the Raptor evaluation boards with a JTAG key cable. All options required to deploy application to the locally connected board are configured automatically.

More information about the JTAG connection can be found in the HW Debugger section.

6.2.2.3. Target Launch Custom Target

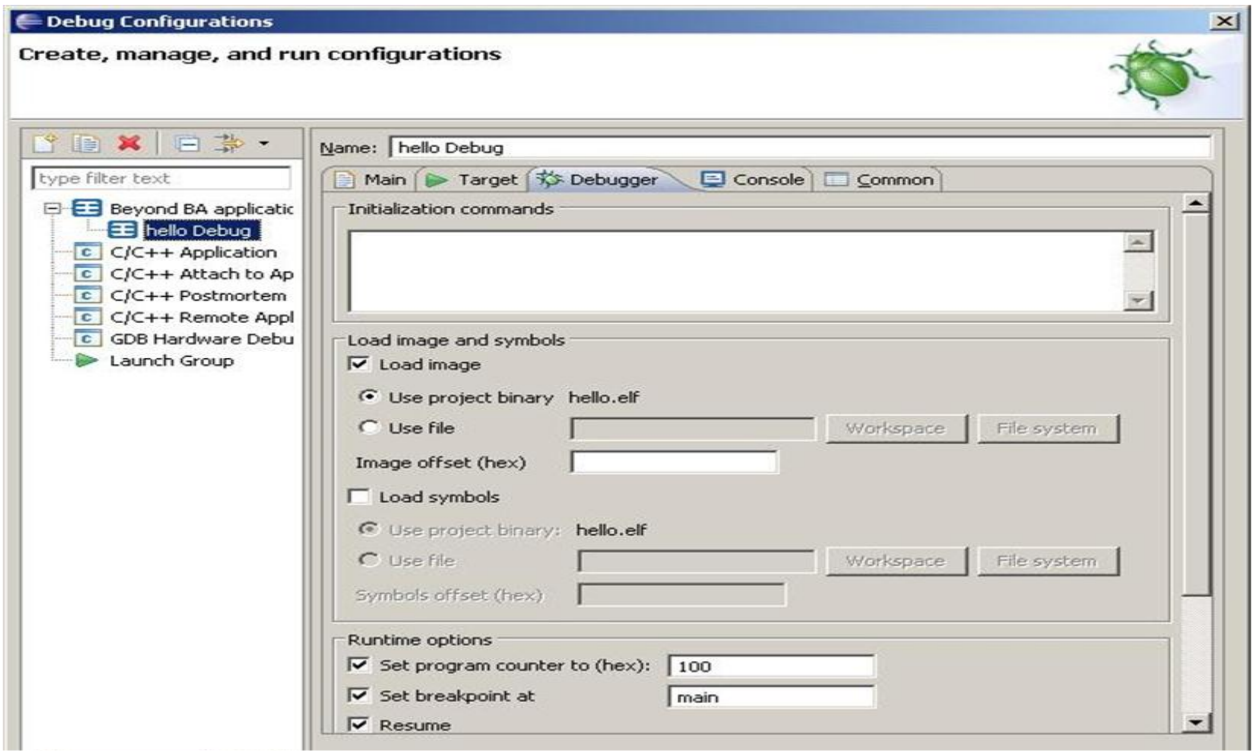
The "Custom target" allows specifying a custom target. It is used as an argument to the BA gdb debugger target command. This configuration can be used to connect to a device connected to other computer.

For example, a device is connected through the JTAG key cable to a machine named foo, and the jp3 JTAG proxy that is started there is set up to accept TCP connections on port 9999. In this case, the following argument can be used: jtag jtag://foo:9999 to connect to it. With the "Custom target" option, the user is fully responsible for all the necessary settings.

6.2.3 Target Launch Debugger Tab

This page presents the debugger's options, describes what should be loaded (binary, symbols), presents the startup options and custom commands that enable the debugger to initialize targets correctly.

For default targets, there is no need to specify or change any options.



Name: hello Debug

Initialization commands

Load image and symbols
[✓] Load image
(●) Use project binary hello.elf
( ) Use file
Image offset (hex)
[ ] Load symbols
(●) Use project binary: hello.elf
( ) Use file
Symbols offset (hex)

Runtime options
[✓] Set program counter to (hex): 100
[✓] Set breakpoint at main
[✓] Resume

Figure 21 Launch Debug Configuration – 'Debugger' tab

The **Initialization commands** text box is used to specify the gdb commands that will be executed by gdb at the beginning of a launch sequence. Those custom commands can be used to initialize the target. Default targets don't require any initialization.

A very similar field is located at the bottom of this tab (“**Run commands**”). Here, the gdb commands can be also entered, but they are run after everything else is set up (image loaded, PC set, etc).

The **Load image** segment allows specifying (optionally) an application binary that will be loaded to target as part of the launch session startup; however, it is highly recommended to load the project binary. When nothing is loaded, during the debugging session, the debugger will be attached to whatever is currently running in the simulator or the device.

The **Load symbols** segment specifies whether to load program symbols or not. With the current Eclipse implementation symbols are always loaded, if they are present in the binary, even if the check box is cleared. If another file is specified to load symbols from, it will override the symbols loaded from the binary.

The **Runtime options** option group configures the way how the program starts up in the debug mode.

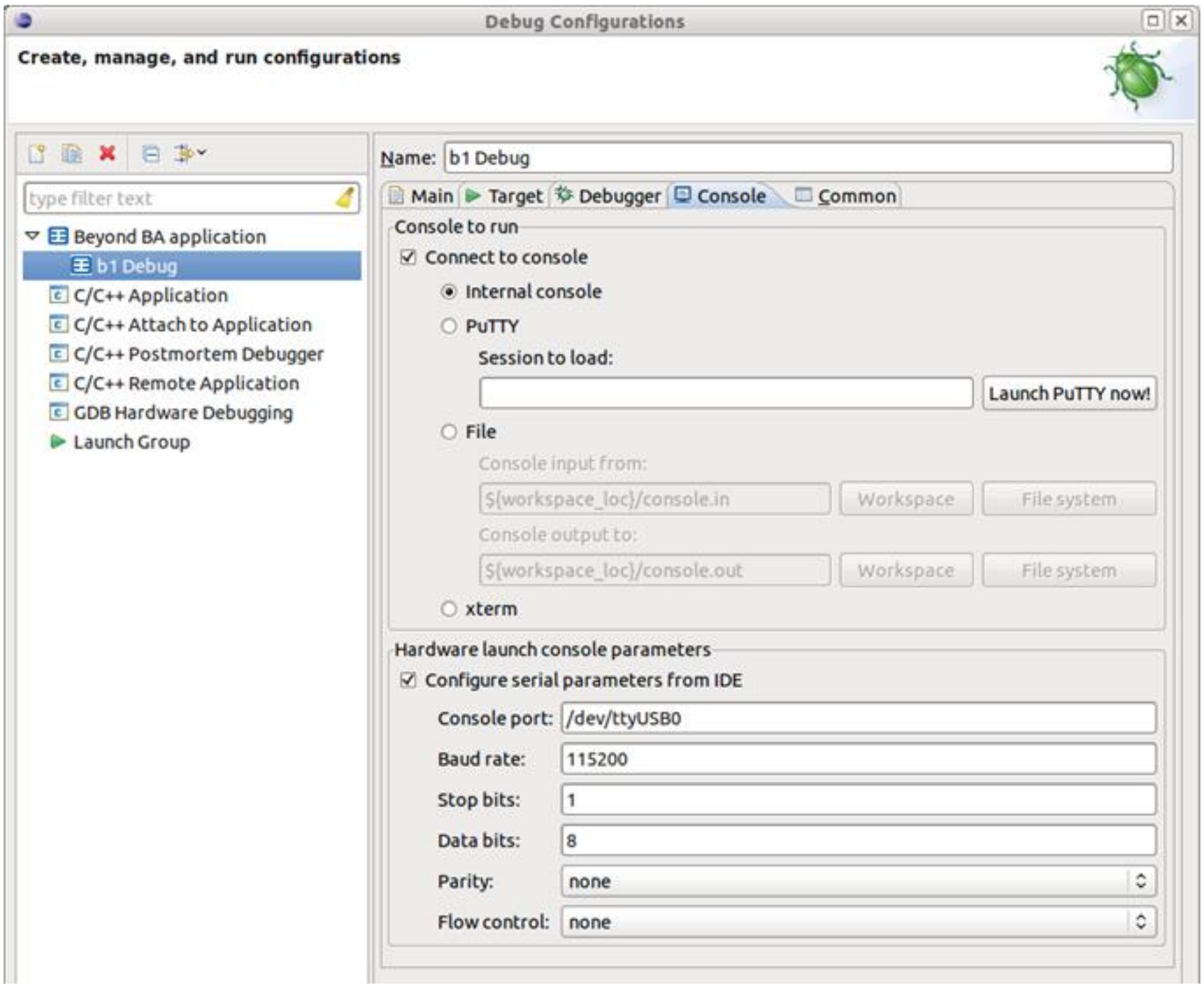
“**Set program counter to**”: it specifies the initial value of program counter for starting up the application. If it’s desired you want to start programs the usual way on the embedded BA SoC, it’s recommended to leave this at0x100. In case of a different value of program counter, part of the code can be skipped.

“**Set breakpoint at**”: specifies the function, when the application startup will be suspended at. When the field is clear, the debugger will not suspend program execution, when program is started. This “clear” option can be also utilized while the breakpoints are set in the editor window at required locations. Then, the program will just keep running until it hits one of those breakpoints.

“**Resume**”: When this option is checked, program will start executing when it is loaded to target otherwise it won’t. If the option is cleared, the user can still manually step through program or set the debugger to resume by selecting the “resume” button.

6.2.4 Target Launch Console Tab

This page presents the console options and its connection settings.



Name: b1 Debug

Console to run
☑ Connect to console
● Internal console
○ PuTTY
Session to load:
○ File
Console input from: ${workspace_loc}/console.in
Console output to: ${workspace_loc}/console.out
○ xterm

Hardware launch console parameters
☑ Configure serial parameters from IDE

Console port:	/dev/ttyUSB0
Baud rate:	115200
Stop bits:	1
Data bits:	8
Parity:	none
Flow control:	none

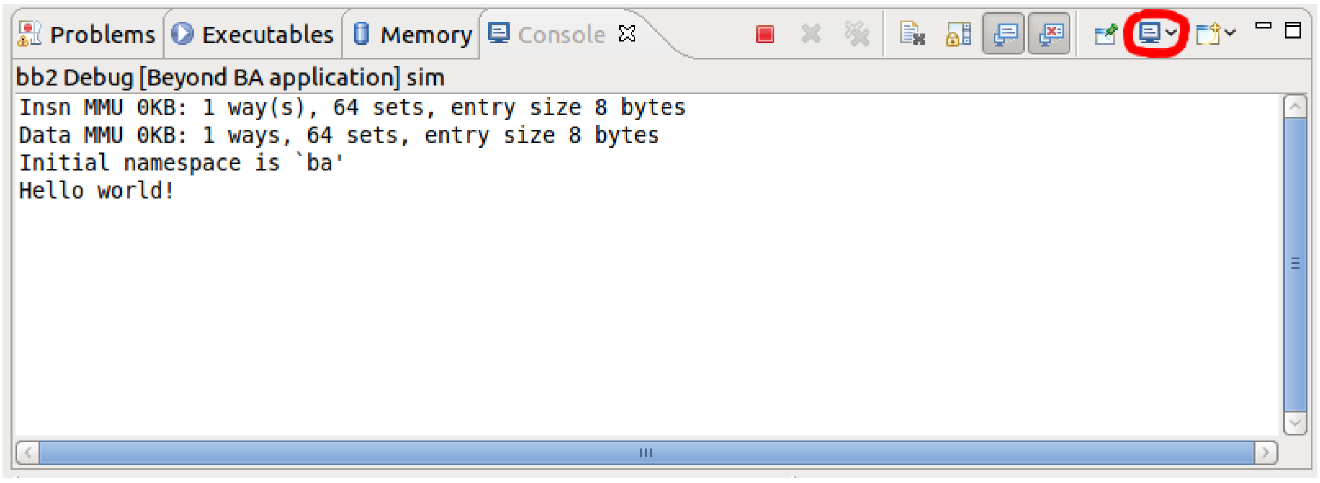
Figure 22 Launch Debug Configuration – ‘Consoles’ tab

6.2.4.1. No console

The first available option is not to have any console at all. Any console output is lost, but otherwise the program is fully functional and its behavior can be watched in the debugger. This mode is selected by clearing the “connect to console” checkbox.

6.2.4.2. Internal console

By default, the console is enabled and the “Internal console” is selected. This console is just a TXT window within IDE as depicted on Figure 23 Launch Debug Configuration – Internal Console



bb2 Debug [Beyond BA application] sim

```
Insn MMU 0KB: 1 way(s), 64 sets, entry size 8 bytes
Data MMU 0KB: 1 ways, 64 sets, entry size 8 bytes
Initial namespace is `ba'
Hello world!
```

Figure 23 Launch Debug Configuration – Internal Console

If the “Console” window is not visible while in debugging mode, it can be re-opened by using the “Window ->Show view -> Console” option.

The “Console” window is also an interface to many separate consoles. The user can switch between various consoles using the console icon in the console window (circled in red on Figure 23 Launch Debug Configuration – Internal Console). By default, the console that has a new output is automatically activated.

For the simulator launches, the internal console output is shown in the simulator console. It initially shows some simulator start-up messages, and after that, the program's console output. The user can also enter text into this window. No configuration of internal console is required for simulator launches.

For the hardware launches, the console that is used to communicate with hardware is named - “remote console”. The “remote console: plays the same role as the simulator's internal console. Together with the “internal console” option the hardware launch parameters need to be set as described in the Hardware Launch Console Parameters section.

6.2.4.3. PuTTY console

Beyond Studio can use PuTTY terminal (see http://www.chiark.greenend.org.uk/~sgtatham/putty/) as a console for simulator and hardware runs. Putty is bundled with IDE for both Linux and Windows versions of Beyond Studio. Also “internal console” uses PuTTY tool plink to connect to the hardware.

For simulator runs, no configuration is strictly required, though the user may want to set up few things like local echo and so on, depending on the application. For the hardware launches, the user can configure basic parameters in IDE or use PuTTY's configuration dialogs to create a session to use.

6.2.4.4. File console

This option allows configuring two files, one acting as a console input and other as a console output. This option currently works only for the simulator launches.

If the console’s log files are required with the hardware launches, thePuTTY console can be used for that purpose.

6.2.4.5. Xterm

This option is available only for simulator launches on Linux. It’s similar to PuTTY console, but uses xterm terminal instead. It does not require any configuration.

6.2.4.6. Hardware Launch Console Parameters

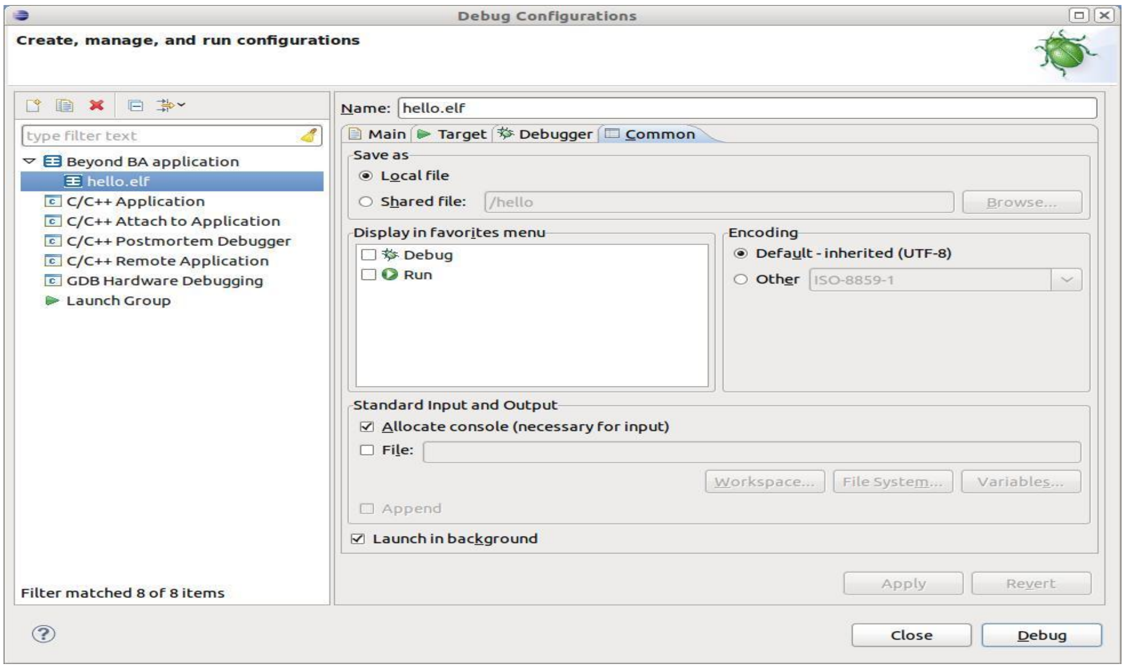
An additional parameter which is required for successful connections to hardware with PuTTY or internal console is the “console port”. This is the name of the serial port that is connected to the used board or device. Its name can vary, depending on configuration of the development PC, but on Windows its name is probably COM* (COM1, or COM3), and on Linux /dev/tty* (/dev/ttyS0 or /dev/ttyUSB0). Sample settings are depicted on Figure 22 Launch Debug Configuration – ‘Consoles’ tab Currently, this information can't be detected automatically; so the user has to be aware of the name of the serial port which is connected to the hardware BA22 implementation.

The remaining UART parameters, like the port baud rate, bits, parity etc, can also be customized. Note that the default values are valid for connections to the demo boards. If other boards or devices are used, the appropriate parameters have to be set.

6.2.5 Target Launch Common Tab

On this page, additional launch options can be specified. For instance the “Save as” option group allows specifying where the launch configuration is to be stored (e.g. on the team’s CVS). The “Display in favorites” option pins a launch configuration to “debug” or “run” action sub-menus for faster access.

Other options currently don't apply to the “Beyond BA Application” launches.


Debug Configurations
Create, manage, and run configurations

Name: hello.elf

Tabs: Main | Target | Debugger | Common

Save as
- [✓ Local file]
- Shared file: /hello (Browse...)

Display in favorites menu
- ☐ Debug
- ☐ Run

Encoding
- [✓ Default - inherited (UTF-8)]
- Other ISO-8859-1

Standard Input and Output
- ☑ Allocate console (necessary for input)
- ☐ File:
- Workspace... File System... Variables...
- ☐ Append

☑ Launch in background

Apply Revert
Close Debug

Figure 24 Launch Debug Configuration – ‘Common’ tab

6.2.6 Debug vs. Run Configurations

The Run and Debug configurations are shared. In case of editing a configuration through the “Debug configurations” menu shortcut, the user will be presented with full options, while in the “Run configurations” only a subset that is used while simply running an application is presented.

Changing options through either shortcut will also change the same options (if available) in other mode.

6.3 HW Debugger

6.3.1 Beyond Studio for Windows

For the hardware debugging purpose, it’s recommended to use the “Amontec JTAG Key”. Please contact CAST for more details.

The software JTAG Key drivers are built into the Windows Beyond Studio IDE release. After connecting the “Amontec JTAGkey” cable to the PC's USB port, any generic drivers suggested by Windows OS should be discarded. To install the drivers, the Beyond Studio should be launched and the “Window ->Preferences ->BeyondStudio->Hardware link” option selected as shown on Figure 25 JTAG Key Driver Installation

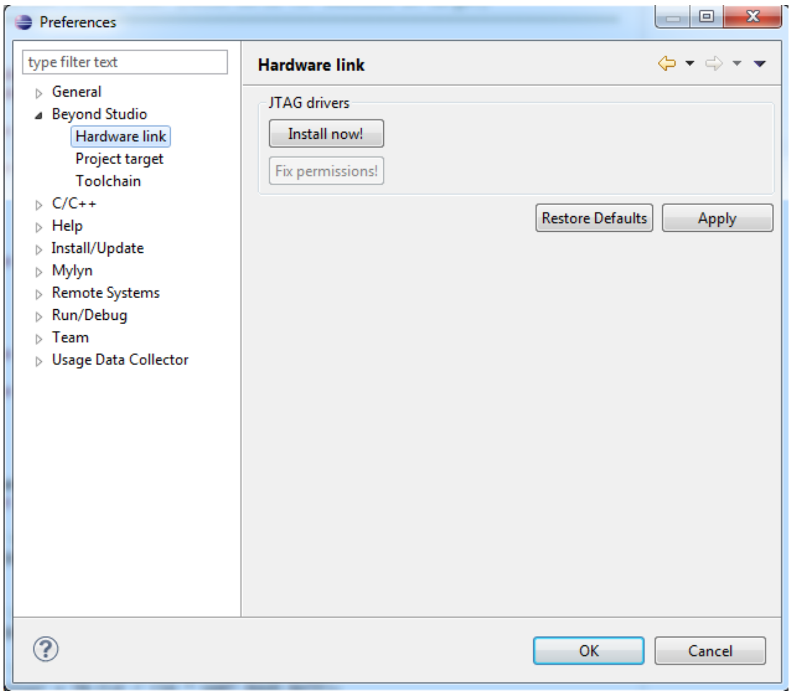
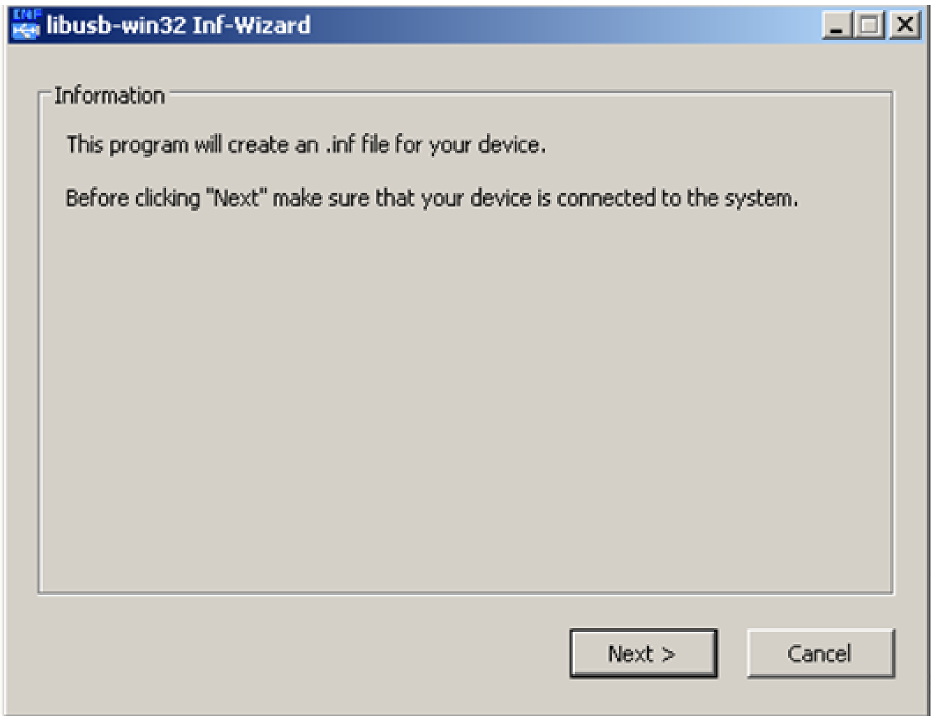


Figure 25 JTAG Key Driver Installation

The “Install now!” button should be pressed and the installation process is automatically launched as shown on Figure 26 JTAG Key Installation Wizard

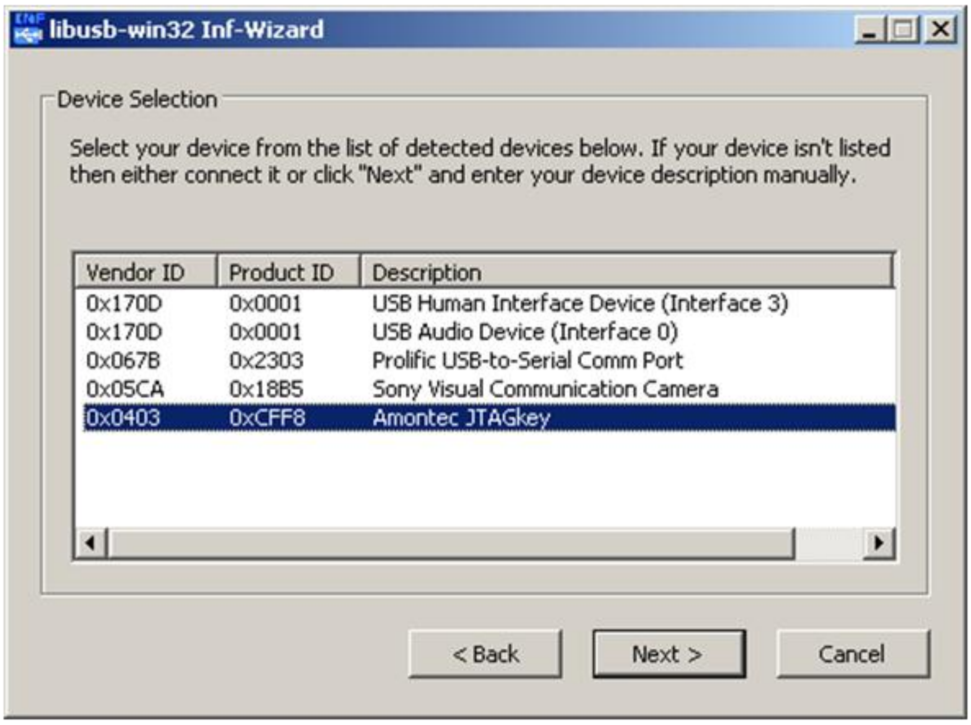


This program will create an .inf file for your device.

Before clicking "Next" make sure that your device is connected to the system.

Figure 26 JTAG Key Installation Wizard

Click ‘Next’ and then, the Amontec JTAG Key should be selected from the available devices as depicted on Figure 27 JTAG Key Installation Wizard – Device Selection

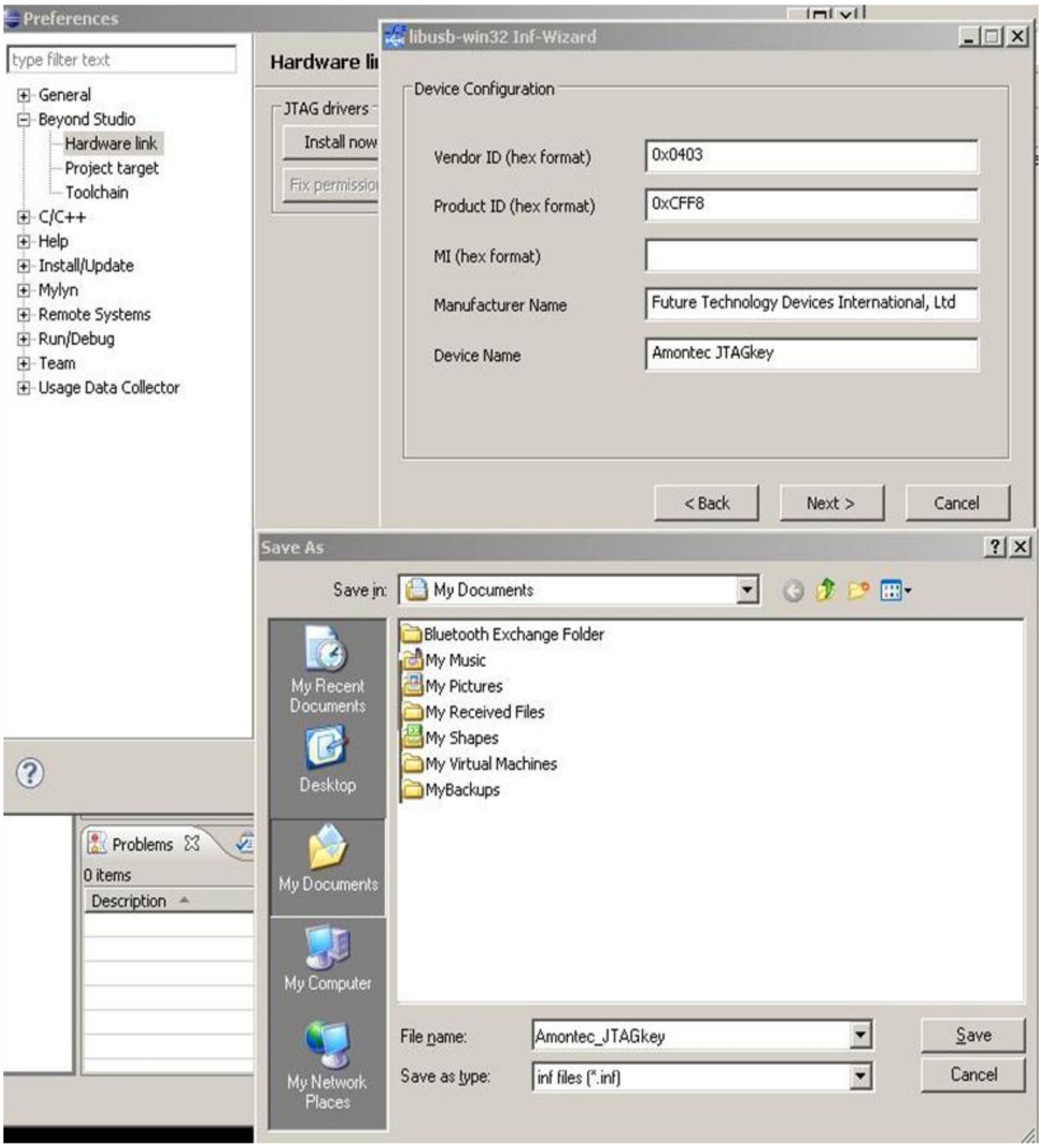


Select your device from the list of detected devices below. If your device isn't listed then either connect it or click "Next" and enter your device description manually.

Vendor ID	Product ID	Description
0x170D	0x0001	USB Human Interface Device (Interface 3)
0x170D	0x0001	USB Audio Device (Interface 0)
0x067B	0x2303	Prolific USB-to-Serial Comm Port
0x05CA	0x18B5	Sony Visual Communication Camera
0x0403	0xCFF8	Amontec JTAGkey

Figure 27 JTAG Key Installation Wizard – Device Selection

Afterwards, the “Next” and “Next” buttons should be pressed. At the next step, the user can save the “inf” file for further references as shown on Figure 28 JTAG Key Installation Wizard – INF File. The INF file can be deleted after the installation is finished.



Device Configuration

Vendor ID (hex format): 0x0403

Product ID (hex format): 0xCFF8

MI (hex format):

Manufacturer Name: Future Technology Devices International, Ltd

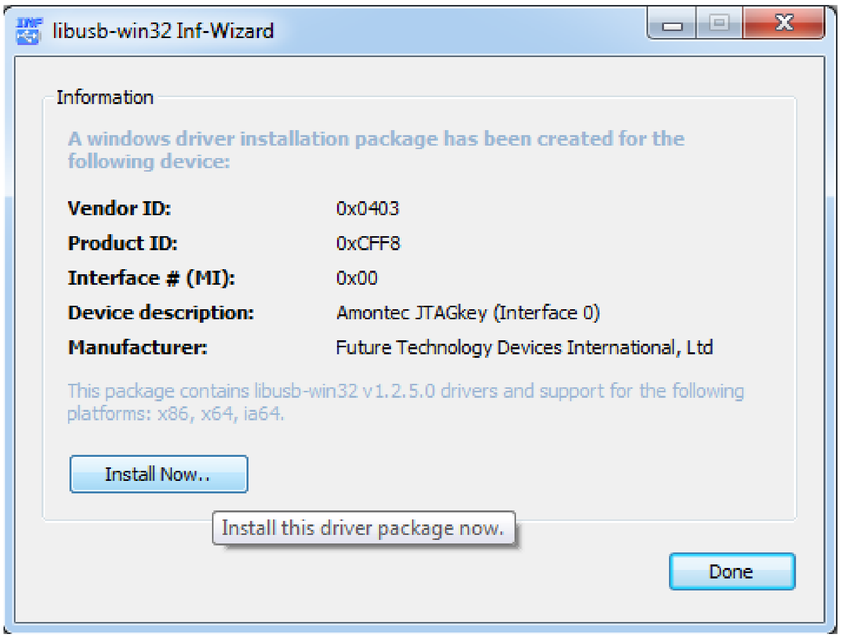
Device Name: Amontec JTAGkey

File name: Amontec_JTAGkey

Save as type: inf files (*.inf)

Figure 28 JTAG Key Installation Wizard – INF File

After saving the ".INF" file, the user should press "Install now" to install the JTAG Cable driver as depicted on Figure 29 JTAG Key Installation Wizard – Finalizing Driver Installation


libusb-win32 Inf-Wizard

Information

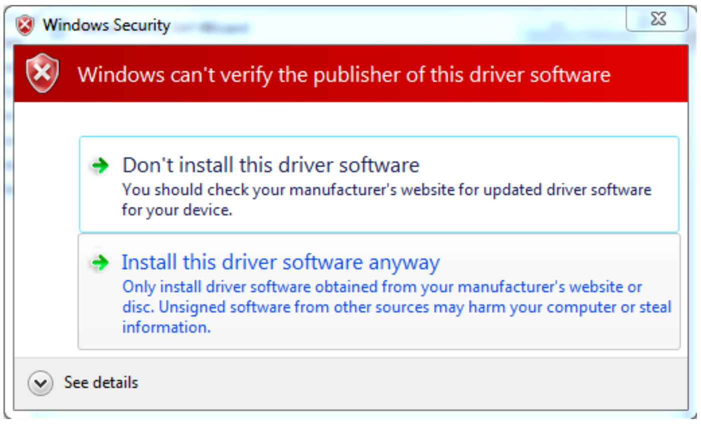
A windows driver installation package has been created for the following device:

Vendor ID:	0x0403
Product ID:	0xCFF8
Interface # (MI):	0x00
Device description:	Amontec JTAGkey (Interface 0)
Manufacturer:	Future Technology Devices International, Ltd

This package contains libusb-win32 v1.2.5.0 drivers and support for the following platforms: x86, x64, ia64.

Figure 29 JTAG Key Installation Wizard – Finalizing Driver Installation

In case of any "User Account Control" warnings - Figure 30 Windows Security Dialog, it's recommended to choose "Install this driver software anyway" even though it's not signed.


Windows Security

Windows can't verify the publisher of this driver software

Don't install this driver software
You should check your manufacturer's website for updated driver software for your device.

Install this driver software anyway
Only install driver software obtained from your manufacturer's website or disc. Unsigned software from other sources may harm your computer or steal information.

Figure 30 Windows Security Dialog

The final "Installation successful" message means that the JTAG Key cable has been successfully installed.

To use the JTAG Key connection during the hardware debugging session, the option "Hardware device->Use JTAG proxy" should be chosen as described in the Target Launch Hardware Device section.

The JTAG Key device supports two communication interfaces: 0 and 1; drivers for those two channels are installed separately. Although, only a single interface (0) is needed for the BA22 JTAG communication, it's also recommended to install the same driver for interface 1. In case of installing only a single interface, the Windows OS might keep asking for a new driver for JTAG Key every time, the JTAG Key is plugged in or the PC is rebooted.

To install interface 1, it's necessary to restart the entire driver installation process and repeat the installation procedure as described above except selecting interface 1 instead of interface 0.

6.3.2 Beyond Studio for Linux

There are no required drivers on Linux for the JTAG Key debugging cable however the user needs to take care of the correct files' permissions as described below.

6.3.2.1. Setting Permissions From IDE

Beyond Studio contains “udev” rules file to set the correct permissions for the JTAG connection. If the user’s Linux system has a recent version of kernel, this method should work. To proceed further, the IDE has to be launched as a super user. This can be done by issuing “*sudo eclipse*” from Beyond Studio directory in the shell window.

Then, the option “Windows ->Preferences -> Beyond Studio -> Hardware link” should be chosen and the button “Fix permissions!” pressed.

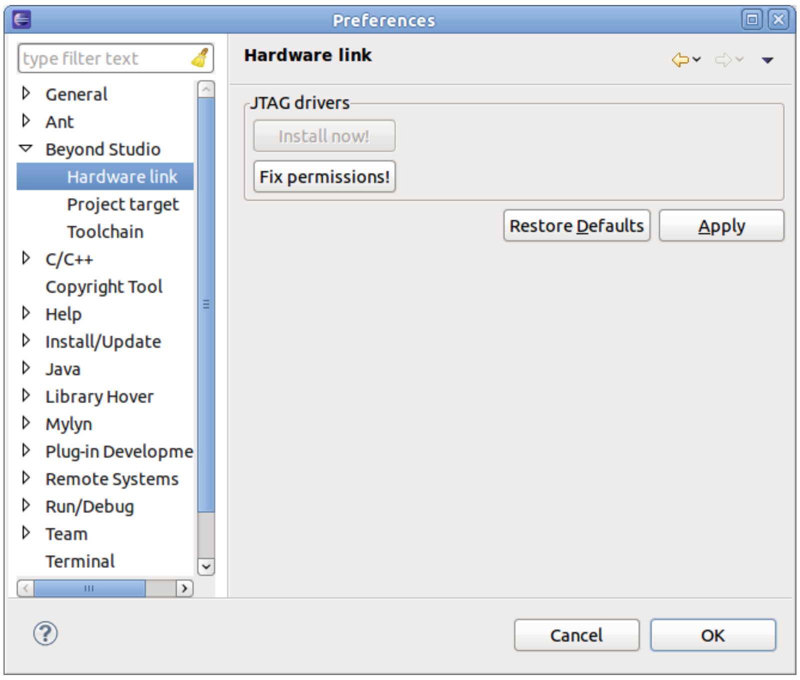
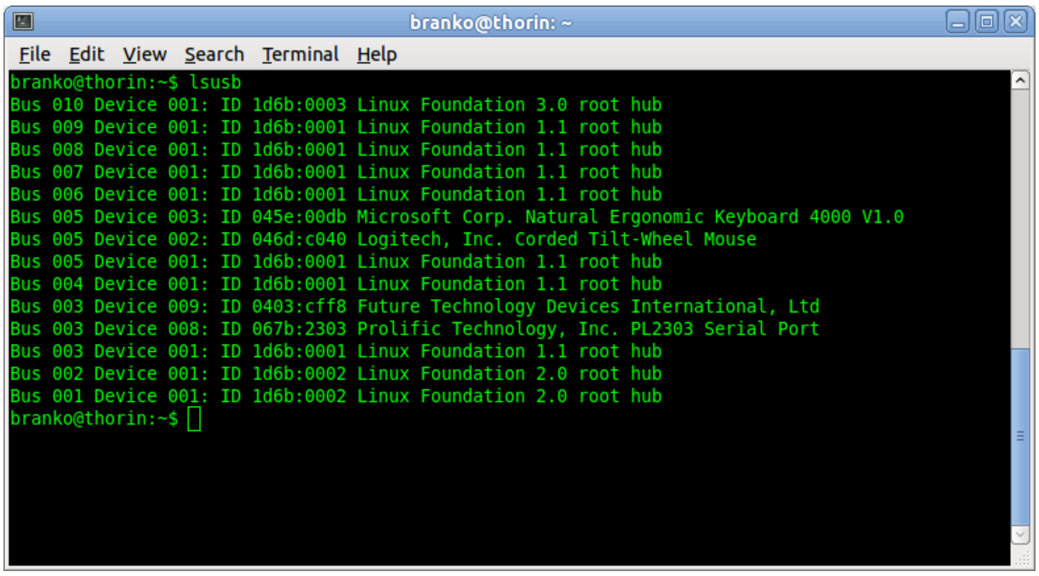


Figure 31 Fixing permissions from IDE

If there are no warnings generated by Linux, the permissions should be fixed and the change should be permanent. In this case, the JTAG Key cable should be detached and plugged again to the PC.

6.3.2.2. Fixing JTAG Key Permissions Manually

If the above method does not work, the user should try to manually change the permissions for the device. To do it, open a console and issue the *lsusb* command. The list of the USB devices and ports should be listed as shown on Figure 32 Listing USB devices.

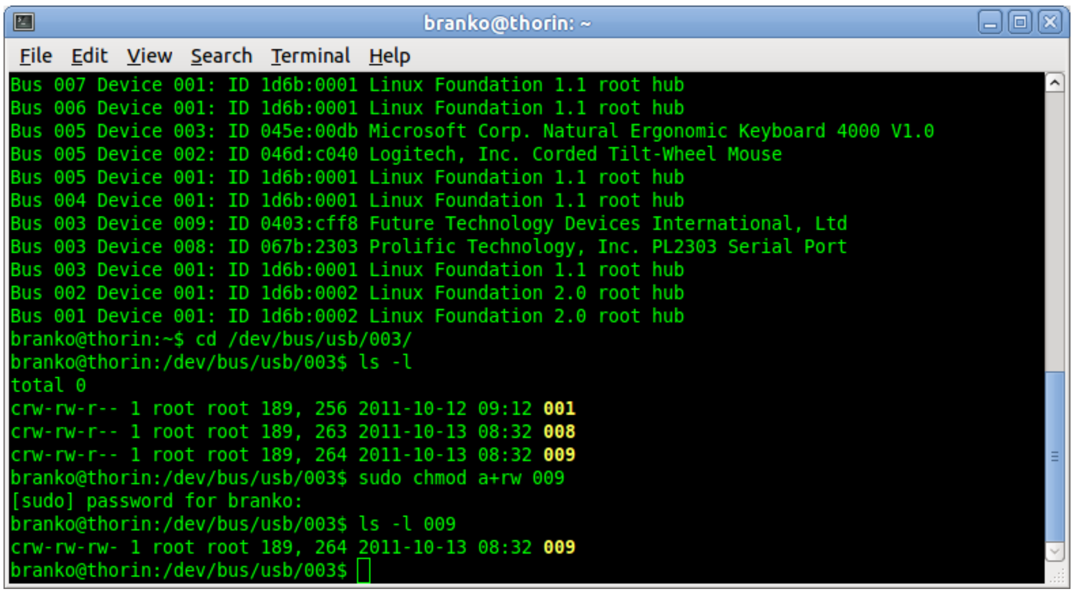


```
branko@thorin:~$ lsusb
Bus 010 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 009 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 008 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 007 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 006 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 005 Device 003: ID 045e:00db Microsoft Corp. Natural Ergonomic Keyboard 4000 V1.0
Bus 005 Device 002: ID 046d:c040 Logitech, Inc. Corded Tilt-Wheel Mouse
Bus 005 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 004 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 003 Device 009: ID 0403:cff8 Future Technology Devices International, Ltd
Bus 003 Device 008: ID 067b:2303 Prolific Technology, Inc. PL2303 Serial Port
Bus 003 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 002 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
branko@thorin:~$ 
```

Figure 32 Listing USB devices

The desired device should be shown as the “Future Technology Devices International, Ltd” together with its bus and device number. On the recent versions of Linux (2.6 and later), the device file is named */dev/bus/usb/<bus number>/<device number>*

On the Figure 32 Listing USB devices, it’s depicted that the bus number for the device is 003 and the device number is 009. Now, the user can do *lscd /dev/bus/usb/003* and list files (with *ls -l*) to see that the file is owned by root:root and other users can only read from it (crw-rw-r--). The permission has to be changed with *chmoda+rw 009*. This needs to be done as root, so the user has to issue *sudo* or *su*, as appropriate for the used system. The Figure 33 Permissions Fixed shows the fixed permission.



```
Bus 007 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 006 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 005 Device 003: ID 045e:00db Microsoft Corp. Natural Ergonomic Keyboard 4000 V1.0
Bus 005 Device 002: ID 046d:c040 Logitech, Inc. Corded Tilt-Wheel Mouse
Bus 005 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 004 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 003 Device 009: ID 0403:cff8 Future Technology Devices International, Ltd
Bus 003 Device 008: ID 067b:2303 Prolific Technology, Inc. PL2303 Serial Port
Bus 003 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 002 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
branko@thorin:~$ cd /dev/bus/usb/003/
branko@thorin:/dev/bus/usb/003$ ls -l
total 0
crw-rw-r-- 1 root root 189, 256 2011-10-12 09:12 001
crw-rw-r-- 1 root root 189, 263 2011-10-13 08:32 008
crw-rw-r-- 1 root root 189, 264 2011-10-13 08:32 009
branko@thorin:/dev/bus/usb/003$ sudo chmod a+rw 009
[sudo] password for branko:
branko@thorin:/dev/bus/usb/003$ ls -l 009
crw-rw-rw- 1 root root 189, 264 2011-10-13 08:32 009
branko@thorin:/dev/bus/usb/003$ 
```

Figure 33 Permissions Fixed

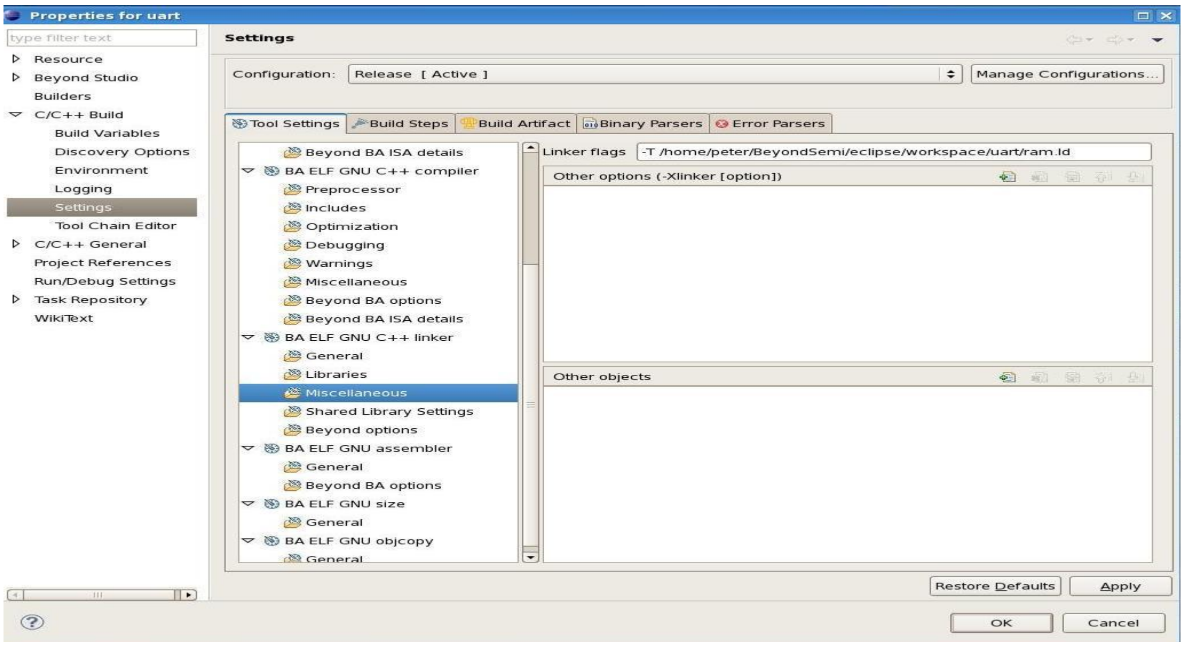
Permissions may change if a device is detached or if the machine is rebooted. In such a case the user needs to repeat the above procedure after every reboot and with the device attached to the PC. Another option is to use the udev rule.

6.4 Linker

The default linker settings have been briefly presented in the Default Settings section.

By using the linker options, the project may be linked with a customized linker script as shown on Figure 34 Linker options – customized linker script. More information about the linker script can be found in the “BA22 Command Line SDK” document.

For more details about the customized linker script please contact CAST.



Properties for uart

Settings

Configuration: Release [Active]

Linker flags: -T /home/peter/BeyondSemi/eclipse/workspace/uart/ram.ld

Other options (-Xlinker [option])

Other objects

Figure 34 Linker options – customized linker script

6.5 Compiler

In the Complier settings, the user can change the CPU configuration (endianness, architecture), type of optimization, and a wide range of the compilation parameters.

6.6 Assembler

The Assembler settings duplicate the Beyond Studio target settings. Additionally, the user can select the fdpic mode (for ucLinux).

In Beyond BA projects, the assembler is used directly to assemble files only with the .asm and .ASM extensions. Files with the .s and .S extensions are assembled through gcc and options specified for C compiler.

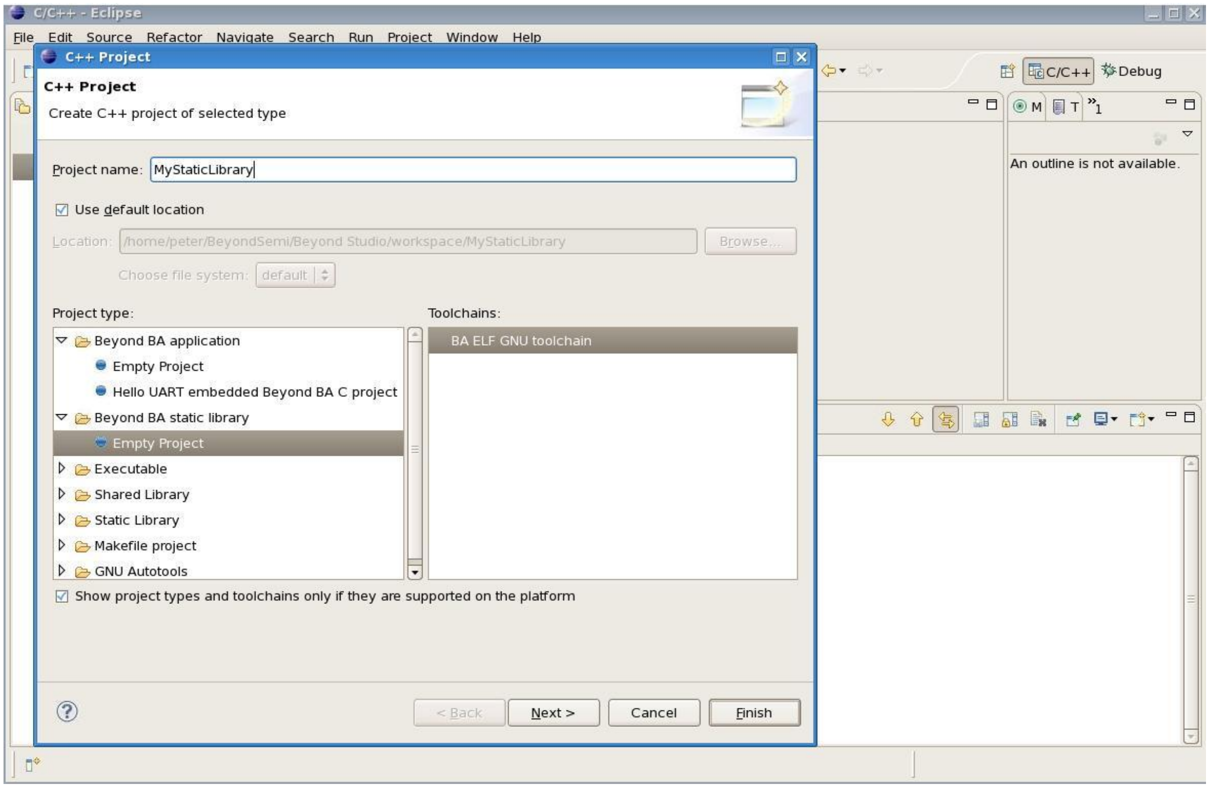
7. Libraries

7.1 newlib

For embedded target (ba-elf) the newlib is supported and already embedded into the toolchain. Its documentation can be found at: http://sourceware.org/newlib/

7.2 Beyond BA Static Library

In the BA22 Eclipse IDE project wizard, the user can choose between “Beyond BA application” or “Beyond BA static library” as shown on Figure 35 Static Library.


C/C++ - Eclipse
File Edit Source Refactor Navigate Search Run Project Window Help
C++ Project
C/C++
Debug
< Back
Next >
Cancel
Finish

C++ Project

Create C++ project of selected type

Project name: MyStaticLibrary

☑ Use default location

Location: /home/peter/BeyondSemi/Beyond Studio/workspace/MyStaticLibrary Browse...

Choose file system: default

Project type:
- Beyond BA application
  - Empty Project
  - Hello UART embedded Beyond BA C project
- Beyond BA static library
  - Empty Project
- Executable
- Shared Library
- Static Library
- Makefile project
- GNU Autotools

Toolchains:
- BA ELF GNU toolchain

☑ Show project types and toolchains only if they are supported on the platform

An outline is not available.

Figure 35 Static Library

When the user selects "Beyond BA application", the result of compilation is an application (.elf) file, which can be executed on the target uP. When selecting "Beyond BA static library", the result of compilation is a static library (.a) file, which contains he functions present in the source files.

When library functions are desired to be used on the target uP, an application still has to be built and the user must specify that the application uses a particular library (the library directory and its name have to be specified). The functions from the library which are used by the application are integrated into the resulting application (.elf) file on the development machine.

The result is the same as if the user would just compile the application together with the library source at once. The library does not get deployed to the target separately.

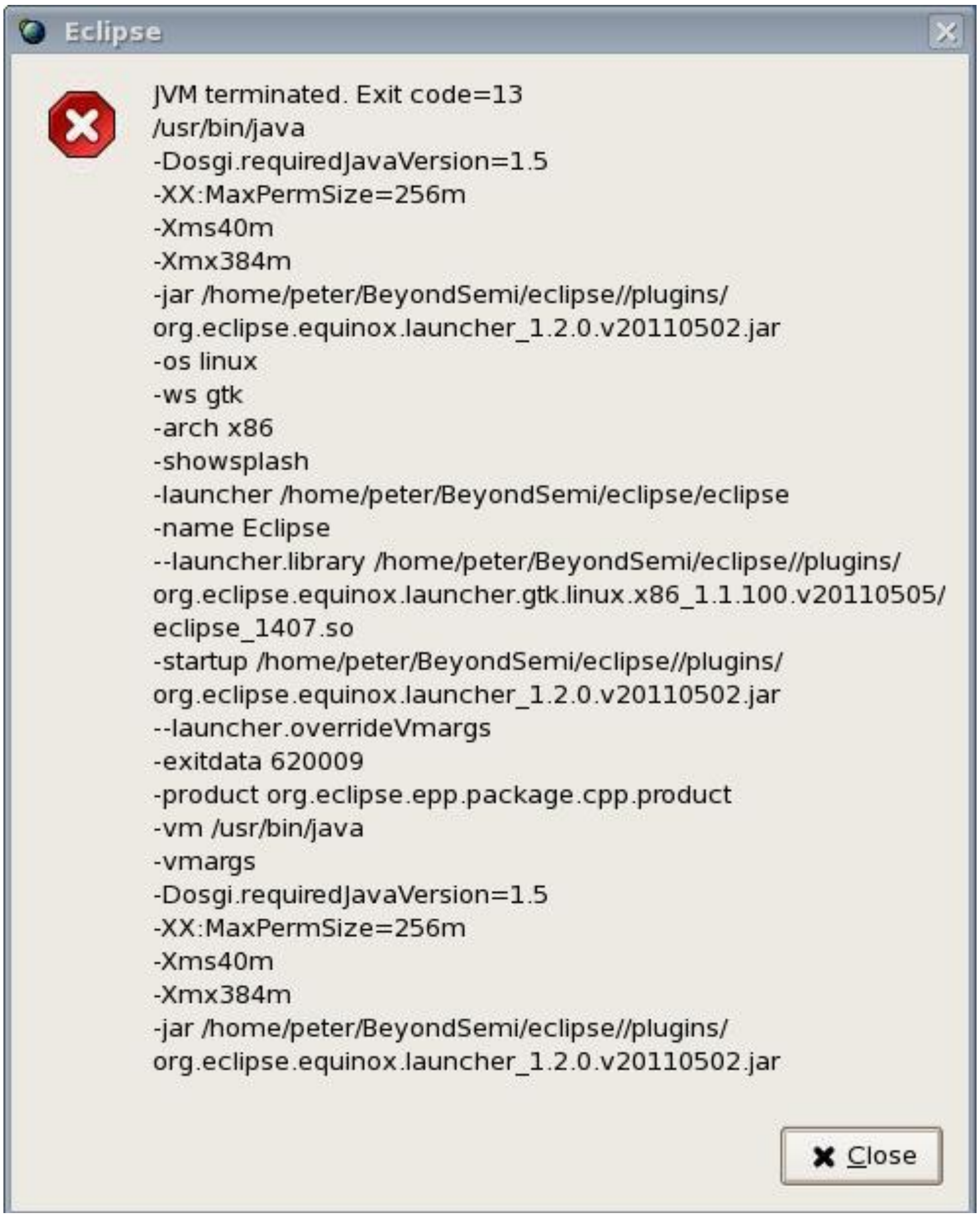
The only difference from building from a "monolithic" source lies in the development process. The user can build and maintain his libraries of common functionality, and the developer only needs the header and the library file to build an end-application.

A sample flow of static library compilation process can be found in the How to Compile a Static Library in Beyond Studio? section.

8. FAQ

8.1 Eclipse Cannot Be Launched

The BA22 Eclipse is launched and all what is visible is shown on Figure 36 BA22 Eclipse IDE cannot be launched



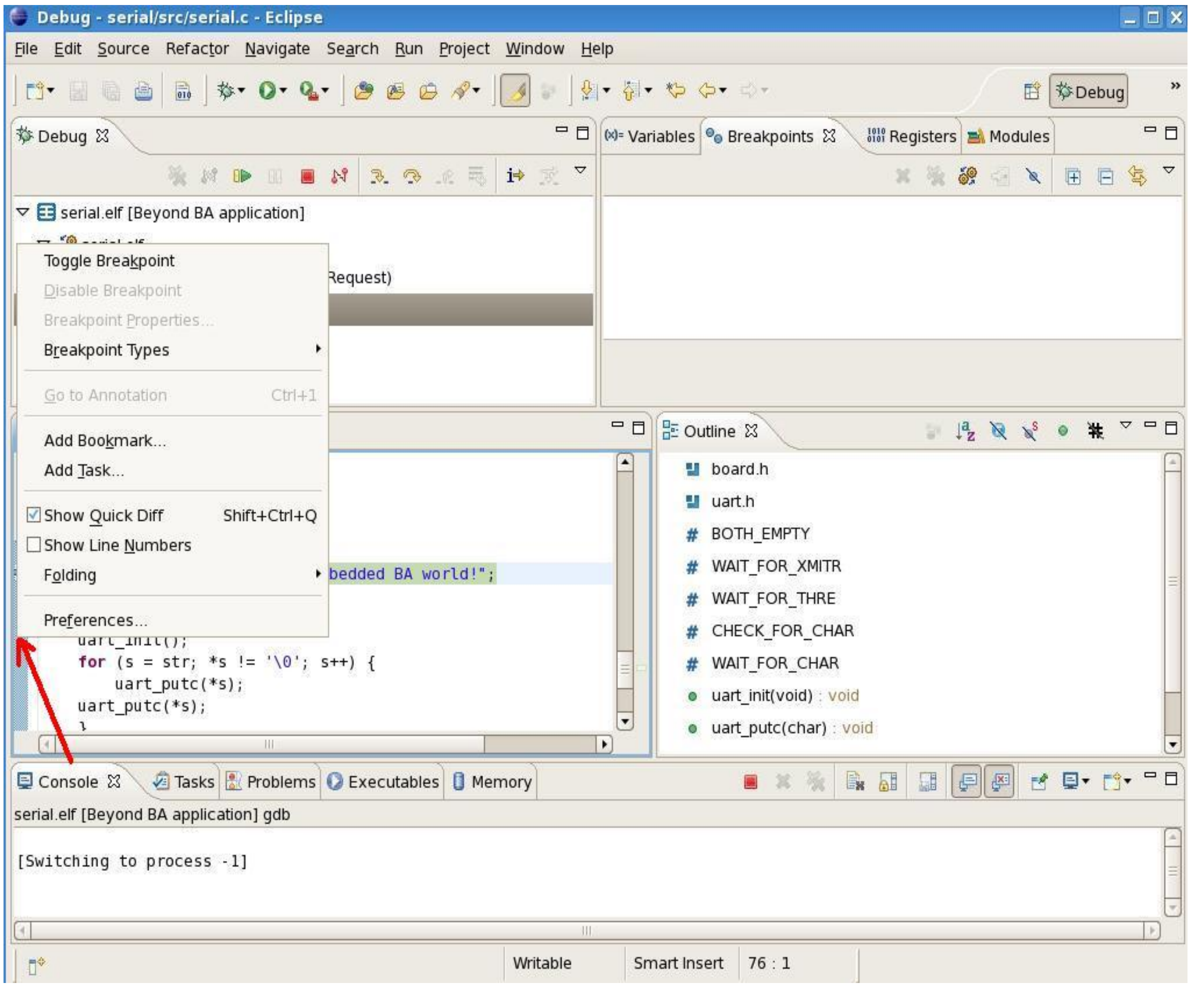
```
JVM terminated. Exit code=13
/usr/bin/java
-Dosgi.requiredJavaVersion=1.5
-XX:MaxPermSize=256m
-Xms40m
-Xmx384m
-jar /home/peter/BeyondSemi/eclipse//plugins/
org.eclipse.equinox.launcher_1.2.0.v20110502.jar
-os linux
-ws gtk
-arch x86
-showsplash
-launcher /home/peter/BeyondSemi/eclipse/eclipse
-name Eclipse
--launcher.library /home/peter/BeyondSemi/eclipse//plugins/
org.eclipse.equinox.launcher.gtk.linux.x86_1.1.100.v20110505/
eclipse_1407.so
-startup /home/peter/BeyondSemi/eclipse//plugins/
org.eclipse.equinox.launcher_1.2.0.v20110502.jar
--launcher.overrideVmargs
-exitdata 620009
-product org.eclipse.epp.package.cpp.product
-vm /usr/bin/java
-vmargs
-Dosgi.requiredJavaVersion=1.5
-XX:MaxPermSize=256m
-Xms40m
-Xmx384m
-jar /home/peter/BeyondSemi/eclipse//plugins/
org.eclipse.equinox.launcher_1.2.0.v20110502.jar
```

Figure 36 BA22 Eclipse IDE cannot be launched

It is very possible that the installed Java version is not correct. More details about how this can be solved is available in the Prerequisites section.

8.2 Can't Set a Breakpoint

To set a breakpoint, the user has to click parallel to the code line however not inside the editor window, but a little bit on the left side of it as it is depicted on Figure 37 Setting up a breakpoint.



```
bedded BA world!";
    uart_init();
    for (s = str; *s != '\0'; s++) {
        uart_putc(*s);
    uart_putc(*s);
```

serial.elf [Beyond BA application] gdb

```
[Switching to process -1]
```

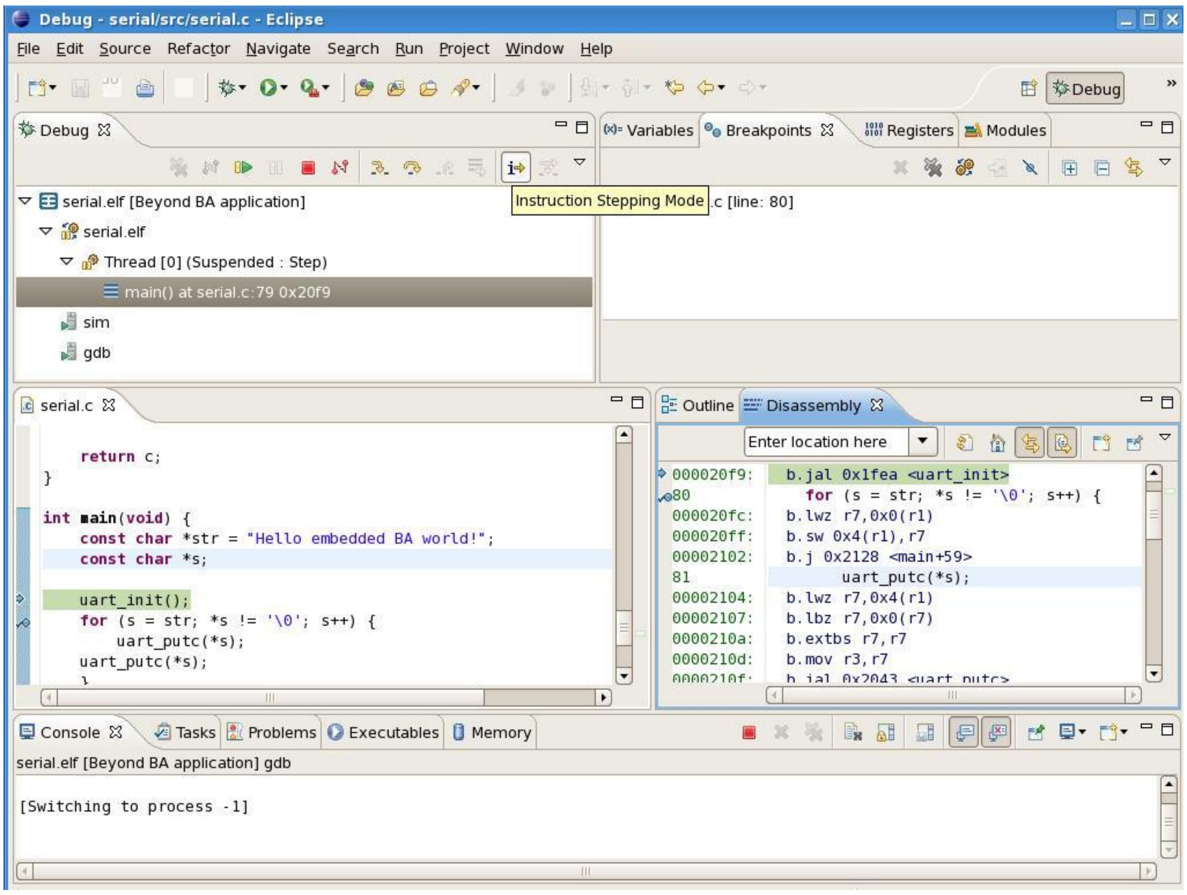
Figure 37 Setting up a breakpoint

8.3 The 'Registers'' Values are Blank

When the application is being debugged in the "Resume" mode, the user may see that the "Registers" view has blank values. Such situation can be noticed in the sample UART example project which is available in the BA22 Eclipse IDE. It's caused by the fact that the debugger refreshes its state only when the program execution is paused that's after every single step, breakpoint, etc… In case of the "UART" application, the user can simply press the "Suspend" button and the registers values are refreshed. Afterwards, the "Resume" button can be pressed again.

8.4 How to Debug Disassembly?

To debug the disassembly code, the user has to click the "i with arrow" icon as shown on Figure 38 Switching to Disassembly mode.


Debug - serial/src/serial.c - Eclipse
File Edit Source Refactor Navigate Search Run Project Window Help
Debug
Debug
Variables
Breakpoints
Registers
Modules
Instruction Stepping Mode
.c [line: 80]
serial.elf [Beyond BA application]
serial.elf
Thread [0] (Suspended : Step)
main() at serial.c:79 0x20f9
sim
gdb
serial.c
Outline
Disassembly
Enter location here
Console
Tasks
Problems
Executables
Memory

```
    return c;
}

int main(void) {
    const char *str = "Hello embedded BA world!";
    const char *s;

    uart_init();
    for (s = str; *s != '\0'; s++) {
        uart_putc(*s);
    uart_putc(*s);
```

```
000020f9:   b.jal 0x1fea <uart_init>
80            for (s = str; *s != '\0'; s++) {
000020fc:   b.lwz r7,0x0(r1)
000020ff:   b.sw 0x4(r1),r7
00002102:   b.j 0x2128 <main+59>
81                 uart_putc(*s);
00002104:   b.lwz r7,0x4(r1)
00002107:   b.lbz r7,0x0(r7)
0000210a:   b.extbs r7,r7
0000210d:   b.mov r3,r7
0000210f:   b.jal 0x2043 <uart_putc>
```

```
serial.elf [Beyond BA application] gdb

[Switching to process -1]
```

Figure 38 Switching to Disassembly mode

8.5 How to Check a Single Module's Size?

By default, the executable size information, shown in Beyond Studio after every compilation, include the size of startfiles as explained in the Default Settings section. Sometimes however, the user may want to check the size of a single, compiled module.

8.5.1 Beyond Studio Linux Version

Any Beyond Studio IDE release includes an embedded toolchain, which can be used separately from the IDE. These tools are typically located in:

./BeyondStudio/plugins/com.beyondsemi.base.toolchains.linux_1.0.0.201110031144/root/ba-elf/bin/ba-elf-size

Since the accurate path can differ with every release, it's recommended to search for ba-elf-size executable to find the correct path. The toolchain path can be later added to the Linux PATH variable to enable fast ba-elf-size launch

To retrieve the module's size, the ba-elf-size should be executed with the module's object file as a parameter. For instance:

ba-elf-sizeuart.o

More information about the ba-elf-size tool can be found in the "BA22 Command Line SDK" document.

8.5.2 Beyond Studio Windows Version

Any Beyond Studio IDE release includes an embedded toolchain which can be used separately from the IDE. These tools are typically located in

.\BeyondStudio\plugins\com.beyondsemi.base.toolchains.windows_1.0.0.201110061620\root\ba-elf\bin\

Since the accurate path can differ with every release, it's recommended to search for ba-elf-size.exe executable to find the correct path. The toolchain path can be added to the Windows PATH variable to enable fast ba-elf-size.exe launch

To retrieve the module's size, the ba-elf-size.exe should be executed with the module's object file as a parameter. For instance:

ba-elf-size.exe uart.o

More information about the ba-elf-size tool can be found in the "BA22 Command Line SDK" document.

8.6 How to Generate a Disassembly File?

During the debugging phase, the user has an option to debug the disassembly file as presented in section How to Debug Disassembly? Sometimes however, it might be useful for the user to generate a separate disassembly file for each of the sub-modules or from the final module.

8.6.1 Beyond Studio Linux Version

Any Beyond Studio IDE release includes an embedded toolchain, which can be used separately from the IDE. These tools are typically located in:

./BeyondStudio/plugins/com.beyondsemi.base.toolchains.linux_1.0.0.201110031144/root/ba-elf/bin/ba-elf-objdump

Since the accurate path can differ with every release, it's recommended to search for ba-elf-objdump executable to find the correct path. The toolchain path can be later added to the Linux PATH variable to enable fast ba-elf-objdump launch

To generate a disassembly file, the ba-elf-objdump should be executed with the module's object file as a parameter. For instance:

ba-elf-objdump –d uart.o > uart.d

More information about the ba-elf-objdump tool can be found in the "BA22 Command Line SDK" document.

8.6.2 Beyond Studio Windows Version

Any Beyond Studio IDE release includes an embedded toolchain which can be used separately from the IDE. These tools are typically located in

.\BeyondStudio\plugins\com.beyondsemi.base.toolchains.windows_1.0.0.201110061620\root\ba-elf\bin\

Since the accurate path can differ with every release, it's recommended to search for ba-elf-objdump.exe executable to find the correct path. The toolchain path can be added to the Windows PATH variable to enable fast ba-elf-objdump.exe launch

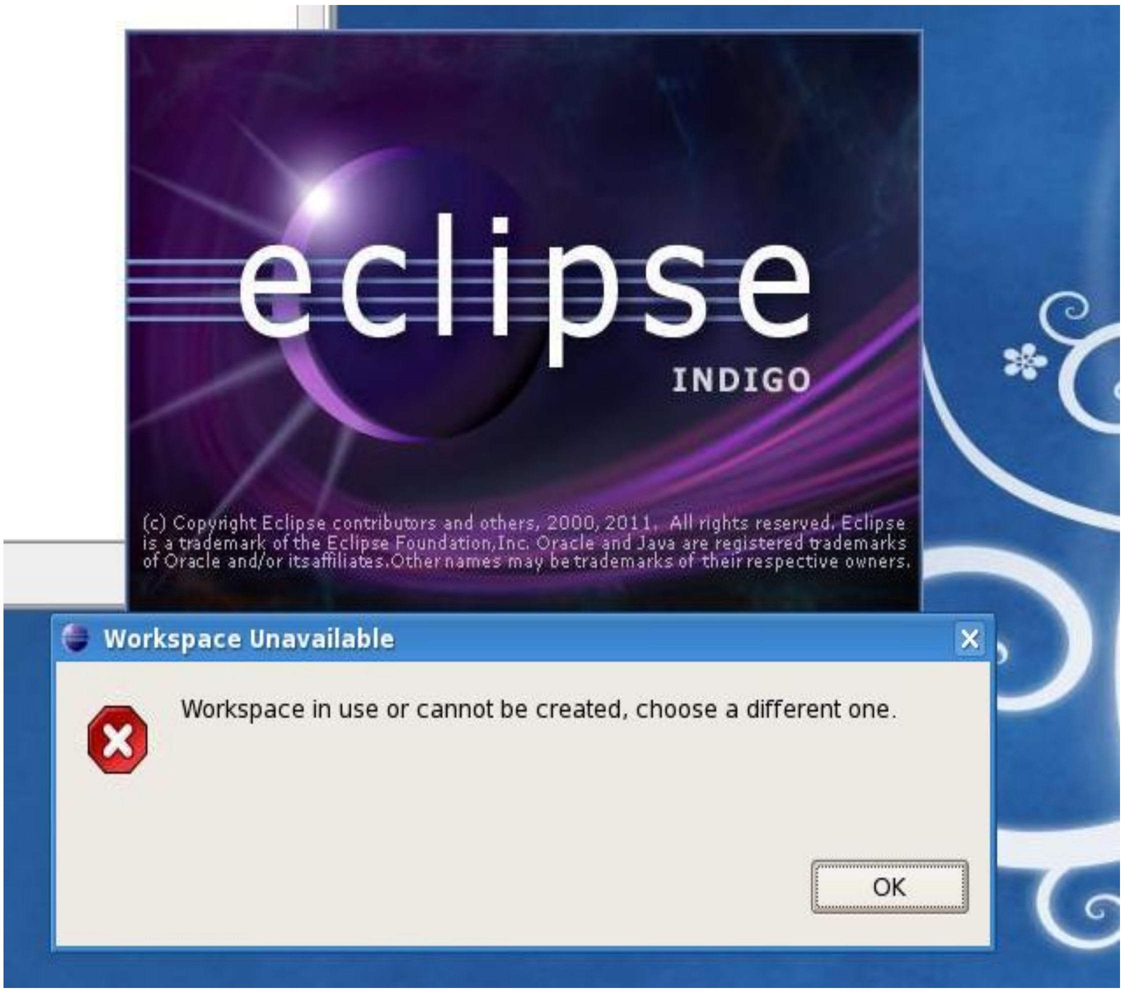
To generate a disassembly file, the ba-elf-objdump.exe should be executed with the module's object file as a parameter. For instance:

ba-elf-objdump.exe –d uart.o > uart.d

More information about the ba-elf-objdump tool can be found in the "BA22 Command Line SDK" document.

8.7 Eclipse Workspace in Use or Cannot Be Created

Sometimes, when Eclipse directory is moved to another OS or machine, the user might notice information as shown on Figure 39 Workspace cannot be created In this case, the user should click "OK" and choose an appropriate location for Workspace.


eclipse
INDIGO
(c) Copyright Eclipse contributors and others, 2000, 2011. All rights reserved. Eclipse is a trademark of the Eclipse Foundation,Inc. Oracle and Java are registered trademarks of Oracle and/or its affiliates.Other names may be trademarks of their respective owners.

Workspace Unavailable

Workspace in use or cannot be created, choose a different one.

Figure 39 Workspace cannot be created

8.8 The Hardware Application Cannot Be Re-launched

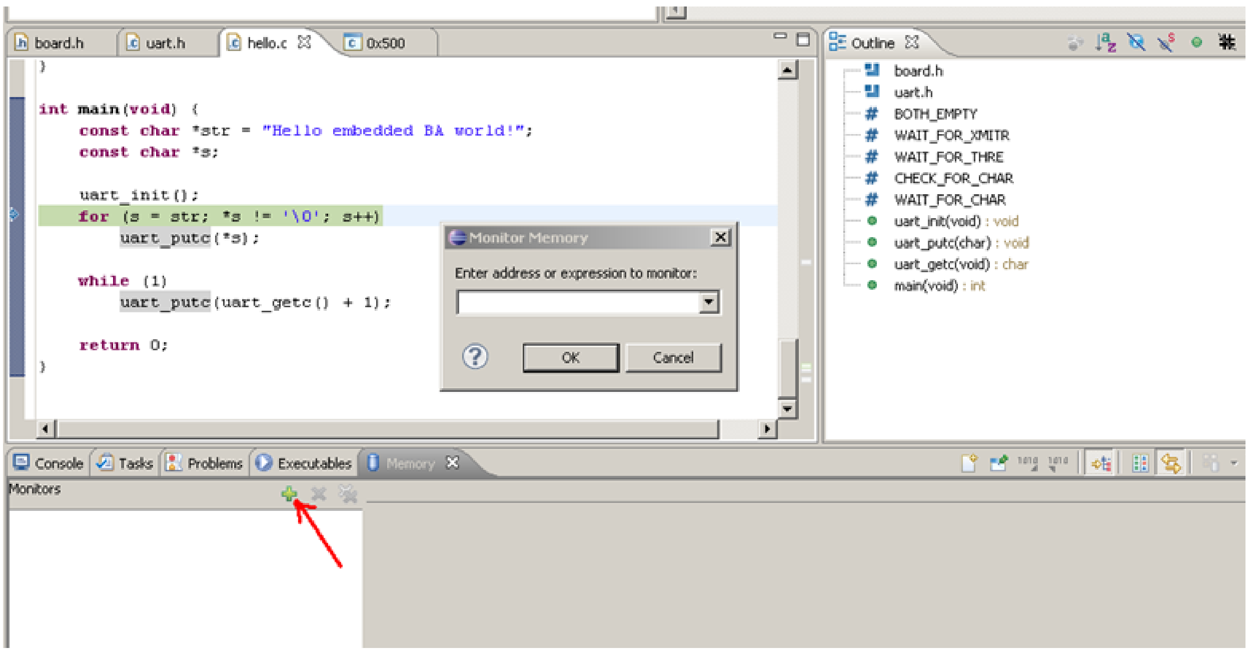
The application has been successfully launched and run on hardware, but the repeated launch repeatedly failed.

This probably means that board has been left in some state, from which gdb can't recover and load new program. It's recommended to manually reset the board via reset switch or turn the power on/off.

8.9 How to Monitor the Value of a Memory Location/Register

The memory location can be monitored by using "Monitor Memory" option[8] in the Beyond Studio IDE as shown on Figure 40 Memory Monitor

[8] This option may not be fully implemented in the early versions of the Beyond Studio IDE



```
}

int main(void) {
    const char *str = "Hello embedded BA world!";
    const char *s;

    uart_init();
    for (s = str; *s != '\0'; s++)
        uart_putc(*s);

    while (1)
        uart_putc(uart_getc() + 1);

    return 0;
}
```

Figure 40 Memory Monitor

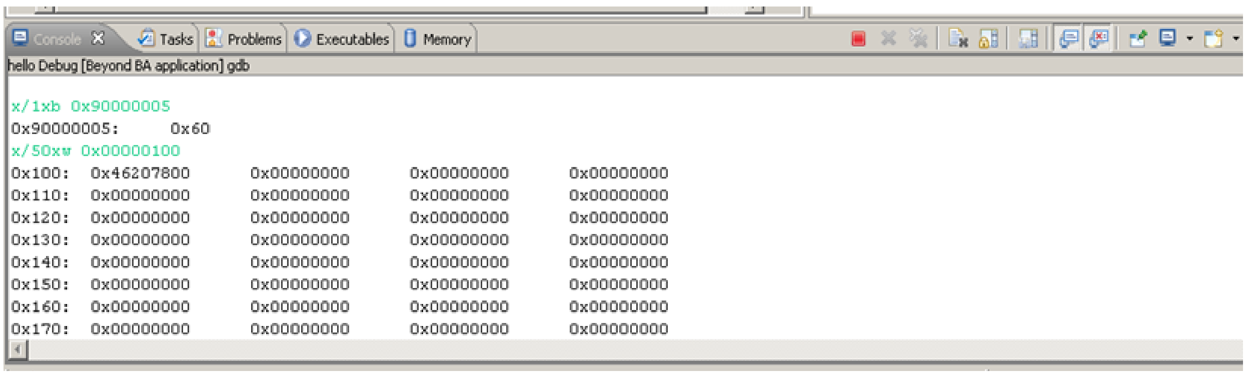
An alternative method to access the memory locations like peripherals' registers or data is by using the GDB command *x/NFS* where:

N:	number of registers; integer value
F:	data format; b – binary, d –decimal, h – hex
S: –	data size; b – byte, h – half word (2 bytes), w – word (4 bytes)

For instanceto check the register located at 0x90000005 and read its value the following command can be used

x/1xb 0x90000005

To check the memory contents at 0x00000100 (typical location of the reset handler in BA22) and read 50 consecutive registers of the word size the following command can be used:*x/50xw 0x00000100*



```
hello Debug [Beyond BA application] gdb

x/1xb 0x90000005
0x90000005:     0x60
x/50xw 0x00000100
0x100:  0x46207800      0x00000000      0x00000000      0x00000000
0x110:  0x00000000      0x00000000      0x00000000      0x00000000
0x120:  0x00000000      0x00000000      0x00000000      0x00000000
0x130:  0x00000000      0x00000000      0x00000000      0x00000000
0x140:  0x00000000      0x00000000      0x00000000      0x00000000
0x150:  0x00000000      0x00000000      0x00000000      0x00000000
0x160:  0x00000000      0x00000000      0x00000000      0x00000000
0x170:  0x00000000      0x00000000      0x00000000      0x00000000
```

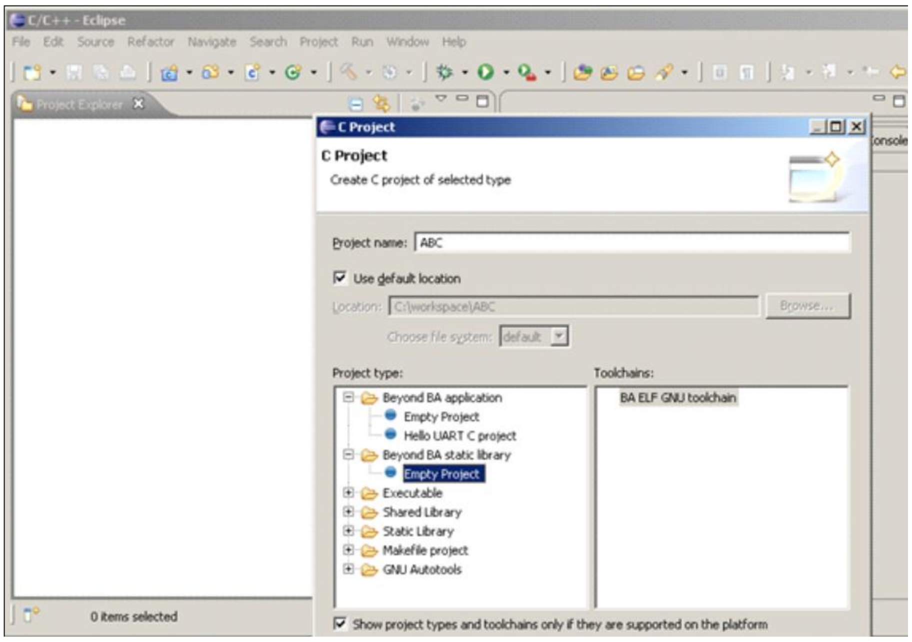
Figure 41 Accessing Memory through GDB

8.10 How to Compile a Static Library in Beyond Studio?

In this section, a sample static library compilation flow using the example of the math package which is part of the newlib library.

Download and extract newlib package from ftp://sources.redhat.com/pub/newlib/index.html For instance, the 2010-12-16: newlib-1.19.0.tar.gz version can be downloaded

Create "Beyond BA Static Library" project as shown on Figure 42 Beyond BA Static Library where a library named "ABC" is to be created. It should be carefully checked that the library target platform settings are the same as the BA22 uP configuration (endianess).


C Project

Create C project of selected type

Project name: ABC

☑ Use default location

Location: C:\workspace\ABC

Choose file system: default

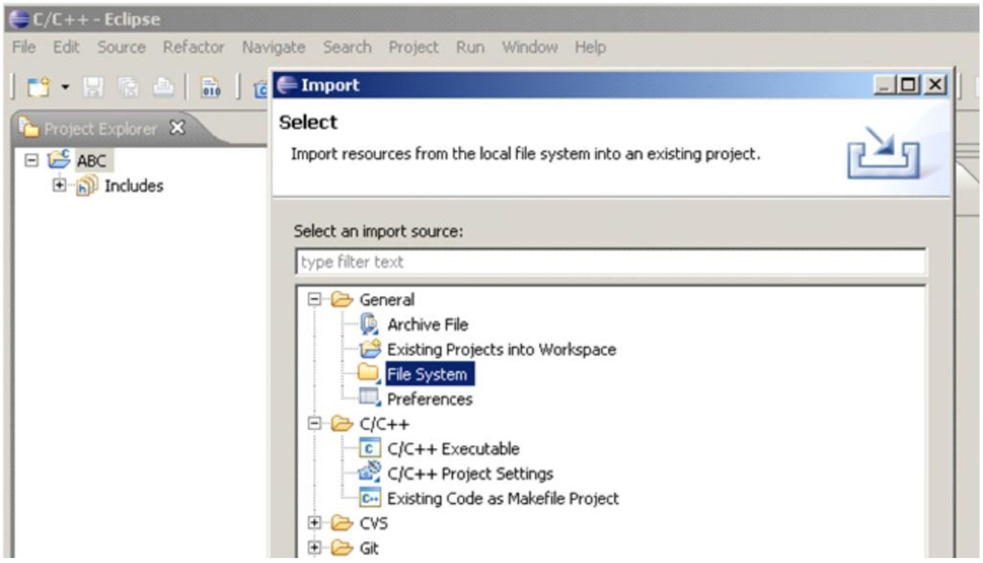
Project type:
- Beyond BA application
  - Empty Project
  - Hello UART C project
- Beyond BA static library
  - Empty Project
- Executable
- Shared Library
- Static Library
- Makefile project
- GNU Autotools

Toolchains:
- BA ELF GNU toolchain

☑ Show project types and toolchains only if they are supported on the platform

Figure 42 Beyond BA Static Library

Add library sources to the created library project: the project name is to be highlighted in the “Project Navigator” then click at File->Import and choose “General->File System” as depicted on Figure 43 Library source import to the project


Import

Select

Import resources from the local file system into an existing project.

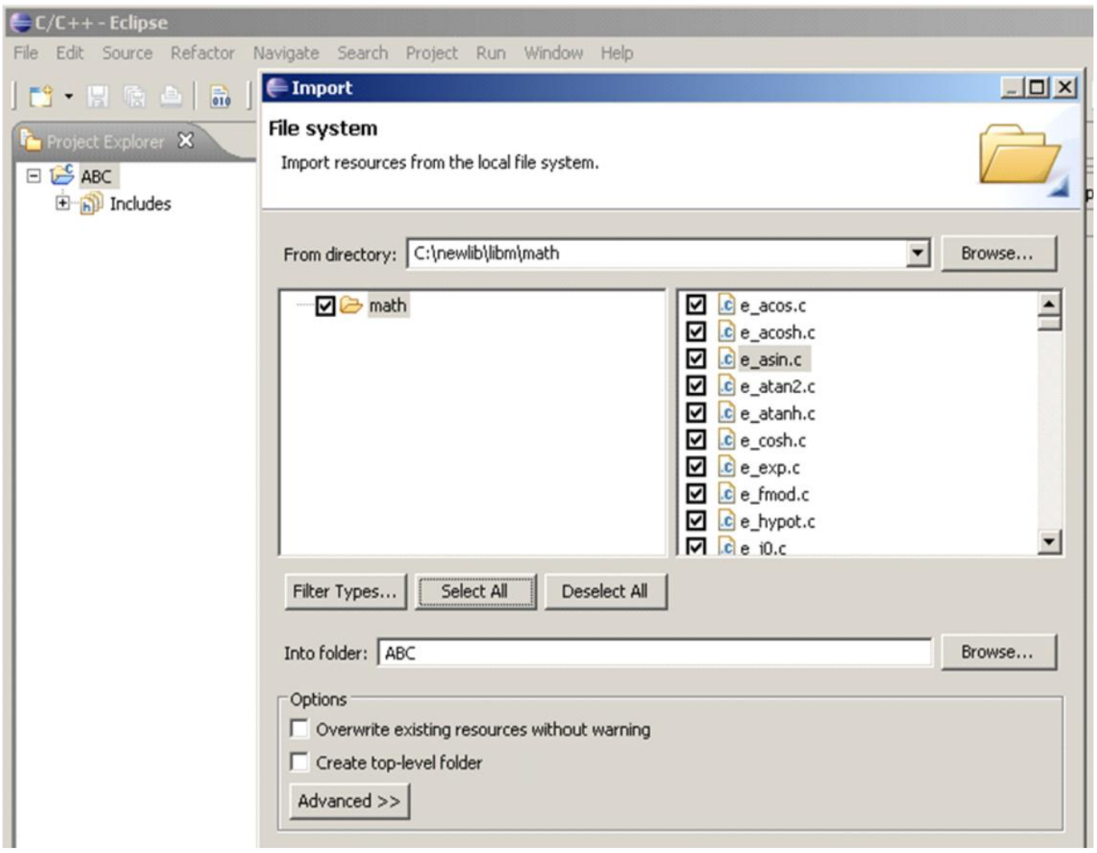
Select an import source:

type filter text

- General
  - Archive File
  - Existing Projects into Workspace
  - File System
  - Preferences
- C/C++
  - C/C++ Executable
  - C/C++ Project Settings
  - Existing Code as Makefile Project
- CVS
- Git

Figure 43 Library source import to the project

Click Next and then browse for the source files. The browser is launched by the upper “Browse” button. To compile the MATH library from newlib, all the math sources should be imported from ./newlib/libm/math as shown on Figure 44 Import MATH library sources to the project



From directory: C:\newlib\libm\math

☑ math

☑ e_acos.c
☑ e_acosh.c
☑ e_asin.c
☑ e_atan2.c
☑ e_atanh.c
☑ e_cosh.c
☑ e_exp.c
☑ e_fmod.c
☑ e_hypot.c
☑ e_j0.c

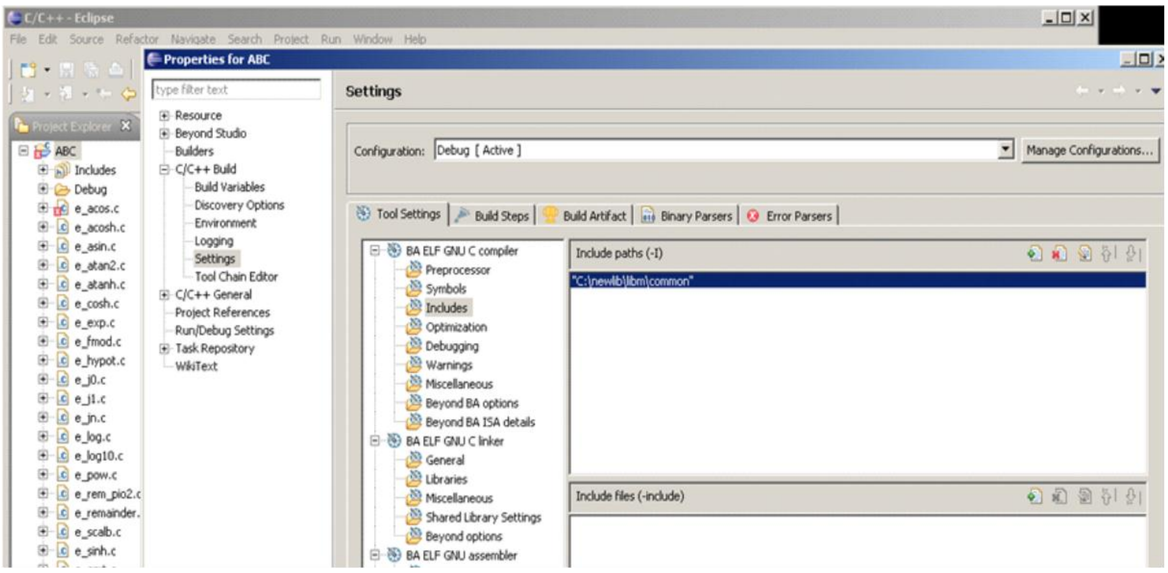
Into folder: ABC

☐ Overwrite existing resources without warning
☐ Create top-level folder

Figure 44 Import MATH library sources to the project

All the imported sources should be selected by clicking the “Select All” button. Then click Finish.

Before proceeding further, the user has to make sure that all the SW dependencies are included in the project. For instance, if the user tries to compile the MATH library, a following error might be displayed “e_acos.c:38:20: fatal error: fdlibm.h: No such file or directory”. The header file “fdlibm.h” is located in ./newlib/libm/common/ and that path has to be added to the project settings as shown on Figure 45 Project configuration with the necessary dependencies



Configuration: Debug [Active]

Include paths (-I)
"C:\newlib\libm\common"

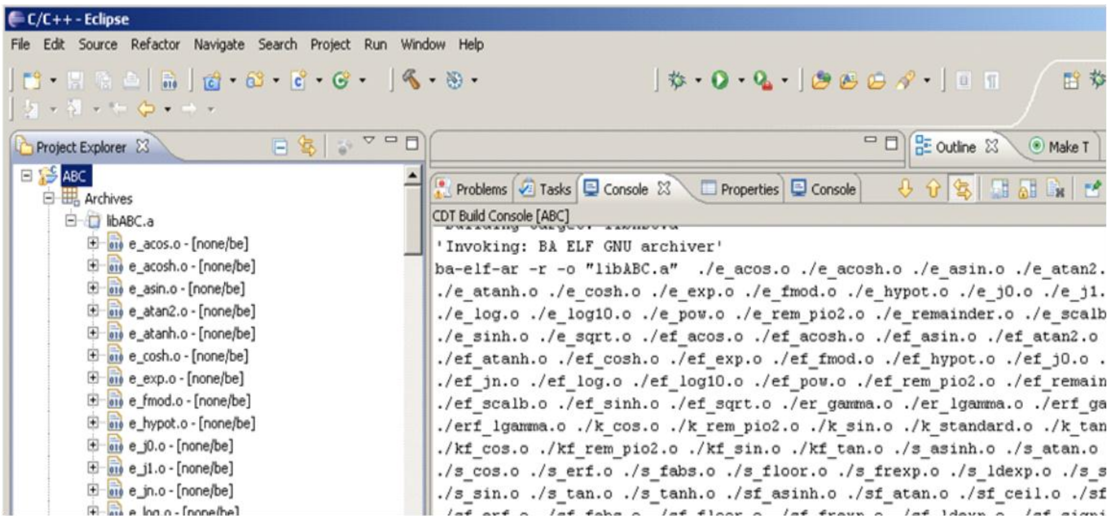
Include files (-include)

Figure 45 Project configuration with the necessary dependencies

It should be confirmed that the same BA22 Hardware configuration as the final BA22 implementation is chosen. For instance, if the FPU module is included, the library should be compiled with the FPU support

– Project Properties->C/C++ Build->Settings->BA ELF GNU C Compiler -> Beyond BA ISA details -> “User hardware FP unit”. If other switches are used by the user in his final project, a similar set of switches should be used to compile the library.

After setting the library project configuration, the library can be built by pressing the “Hammer” icon. As a result, the library archive named libABC.a is created as depicted on Figure 46 Static BA library



```
ABC
  Archives
    libABC.a
      e_acos.o - [none/be]
      e_acosh.o - [none/be]
      e_asin.o - [none/be]
      e_atan2.o - [none/be]
      e_atanh.o - [none/be]
      e_cosh.o - [none/be]
      e_exp.o - [none/be]
      e_fmod.o - [none/be]
      e_hypot.o - [none/be]
      e_j0.o - [none/be]
      e_j1.o - [none/be]
      e_jn.o - [none/be]
```

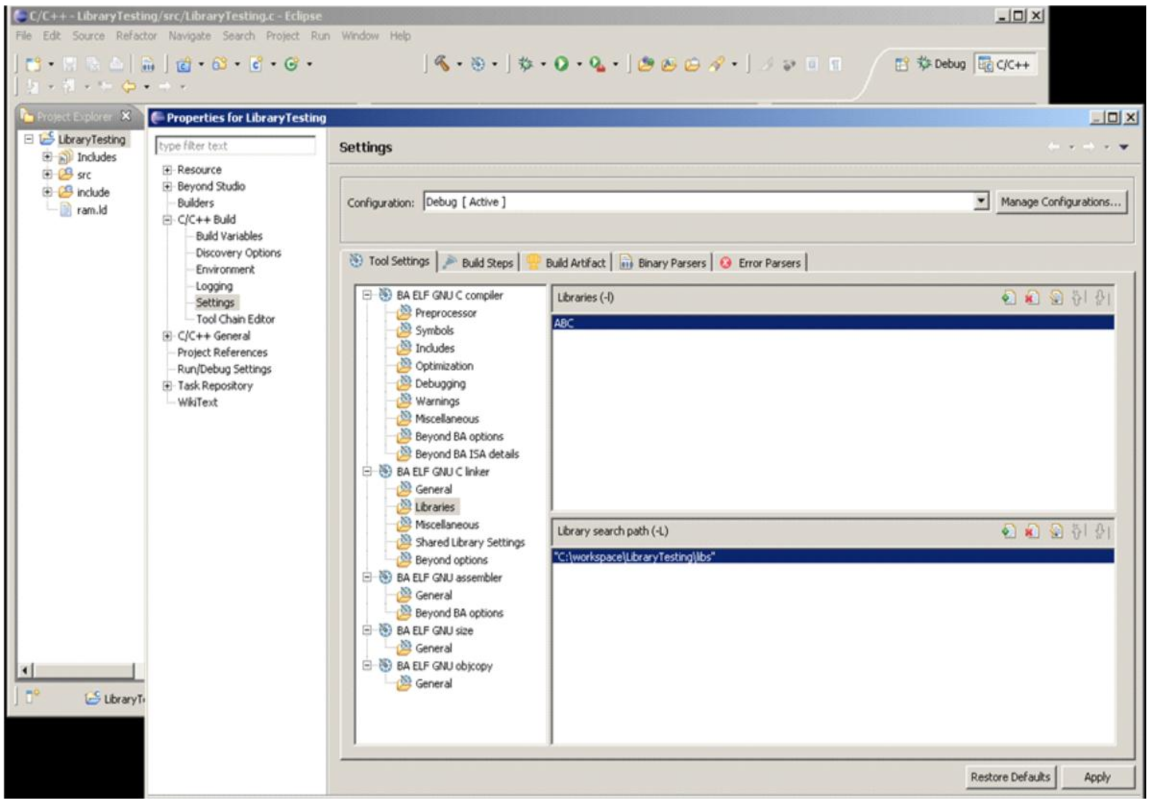
```
CDT Build Console [ABC]
'Invoking: BA ELF GNU archiver'
ba-elf-ar -r -o "libABC.a"  ./e_acos.o ./e_acosh.o ./e_asin.o ./e_atan2.
./e_atanh.o ./e_cosh.o ./e_exp.o ./e_fmod.o ./e_hypot.o ./e_j0.o ./e_j1.
./e_log.o ./e_log10.o ./e_pow.o ./e_rem_pio2.o ./e_remainder.o ./e_scalb
./e_sinh.o ./e_sqrt.o ./ef_acos.o ./ef_acosh.o ./ef_asin.o ./ef_atan2.o
./ef_atanh.o ./ef_cosh.o ./ef_exp.o ./ef_fmod.o ./ef_hypot.o ./ef_j0.o .
./ef_jn.o ./ef_log.o ./ef_log10.o ./ef_pow.o ./ef_rem_pio2.o ./ef_remain
./ef_scalb.o ./ef_sinh.o ./ef_sqrt.o ./er_gamma.o ./er_lgamma.o ./erf_ga
./erf_lgamma.o ./k_cos.o ./k_rem_pio2.o ./k_sin.o ./k_standard.o ./k_tan
./kf_cos.o ./kf_rem_pio2.o ./kf_sin.o ./kf_tan.o ./s_asinh.o ./s_atan.o
./s_cos.o ./s_erf.o ./s_fabs.o ./s_floor.o ./s_frexp.o ./s_ldexp.o ./s_s
./s_sin.o ./s_tan.o ./s_tanh.o ./sf_asinh.o ./sf_atan.o ./sf_ceil.o ./sf
```

Figure 46 Static BA library

The presented Math library example from the newlib package doesn’t require adding the header (math.h) file into the project. This is because that file is already embedded into the BA tools package. In case of any other libraries, the user should take care of the appropriate library’s header files.

8.11 How to Use a Static Library in the C/C++ project in Beyond Studio IDE?

First, a new C/C++ project in Beyond Studio IDE should be created. Then make a directory in the project workspace and copy there the required library. Afterwards, the linker settings should be appropriately configured by adding the library name and its search path as shown on Figure 47 Linker settings. It should be noted that the library name has to be defined without the “lib” prefix and the “.a” extension which are automatically added during the “Static Library project”.



Configuration: Debug [Active]

Libraries (-l): ABC

Library search path (-L): "C:\workspace\LibraryTesting\libs"

Figure 47 Linker settings

During the project development, the user should include the library header as for any other libraries. For the MATH library, the user can use #include <math.h> statement.

It might happen that the user gets involved into unintentional mixed C/C++ linking annoyance. In this case, it's recommended to use "extern C" keyword in the C++ program to include a C library.

Instead of using just:
#include "my_c_library.h"

Use:
extern "C"
{ #include "my_c_library.h" }

More information about C/C++ mix can be found at: http://www.parashift.com/c++-faq-lite/mixing-c-and-cpp.html

8.12 How to Turn On the Line Numbers in the Beyond Studio Editor

To turn on the lines numbers click Window->Preferences->General->Editors->TextEditors and mark the option "Show line numbers".

9. Support

Every effort has been made to ensure that this core functions correctly. If a problem is encountered, contact:

CAST, Inc.
11 Stonewall Court
Woodcliff Lake, New Jersey 07677 USA
Technical Support Hotline: +1-201-391-8300 ext. 2

Fax:	+1-201-833-2682
E-mail:	support@cast-inc.com e
URL:	www.cast-inc.com