NXP Semiconductors
User's Guide

Document identifier: CWARMv8TM
Rev. 11.5.0, 06/2020

CodeWarrior Development Studio for QorIQ LS series - ARM V8 ISA, Targeting Manual

Contents

Chapter 1
Introduction

This manual explains how to use the CodeWarrior Development Studio for QorIQ LS series - ARM V8 ISA product. This chapter presents an overview of the manual.

The topics in this chapter are:

- Release Notes - Lists new features, bug fixes, and incompatibilities
- About this Manual - Describes the contents of this manual
- Accompanying Documentation - Describes supplementary CodeWarrior documentation, third-party documentation, and references.

1.1 Release notes

Release notes lists new features, bug fixes, and incompatibilities.

Before using the CodeWarrior IDE, read the developer notes. These notes contain important information about last-minute changes, bug fixes, incompatible elements, or other topics that may not be included in this manual.

NOTE

The release notes for specific components of the CodeWarrior IDE are located in the `ARMv8` folder in the CodeWarrior for CW4NET installation directory.

1.2 About this Manual

This topic lists each chapter of this manual, which describes a different area of software development.

The following table lists the contents of this manual.

Table 1. Manual contents

Chapter	Description
Introduction	This chapter.
Working with Projects	Lists the various project types and explains how to create projects.
ARMv8 Build Properties	Explains the CodeWarrior build tools and build tool configurations.
Preparing Target	Explains how to prepare for debug various target types.
Configuring Target	Explains Target Connection Configuration (TCC) feature.
FSL Debugger References	Explains debugger features.
Flash Programmer	Explains how to configure, start, and use flash programmer
Use Cases	Lists U-Boot debug, Linux application debug, and Linux kernel debug, and UEFI use cases.
Troubleshooting	Lists troubleshooting information.

1.3 Accompanying Documentation

The Documentation page describes the documentation included in this version of CodeWarrior Development Studio for QorIQ LS series - ARM V8 ISA.

You can access the Documentation page by:

- Opening START_HERE.html in *<CWInstallDir>\CW_ARMv8\ARMv8\Help* folder
- Selecting **Help > Documentation**.

To view the online help for the CodeWarrior tools select **Help > Help Contents** from the IDE menu bar.

Chapter 2
Working with Projects

This chapter lists the various project types and explains how to create and work with projects.

The topics in this chapter are:

- ARMv8 New Project wizard
- CodeWarrior Executable Importer wizard
- Creating projects
- Preprocess/Disassemble files
- Debugging projects

2.1 ARMv8 New Project wizard

The New Project wizard presents a selection of sample projects preconfigured for build using the bundled Linaro GCC toolchains.

Hello World projects for bareboard and Linux oriented (C, C++, ASM, static and shared library) build/debug scenarios are enclosed with the product. As compared to the existing CodeWarrior products, the New Project wizard functionality in CodeWarrior for ARMv8 has been refined to generating copies of the existing pre-configured projects.

All the debugger connection settings are refactored in the Target Connection Configuration dialog.

The ARMv8 New Project wizard enables you to create both bareboard and Linux Application projects in little endian format. To access the ARMv8 New Project wizard, in the Workbench window, select **File > New > ARMv8 Stationery**.

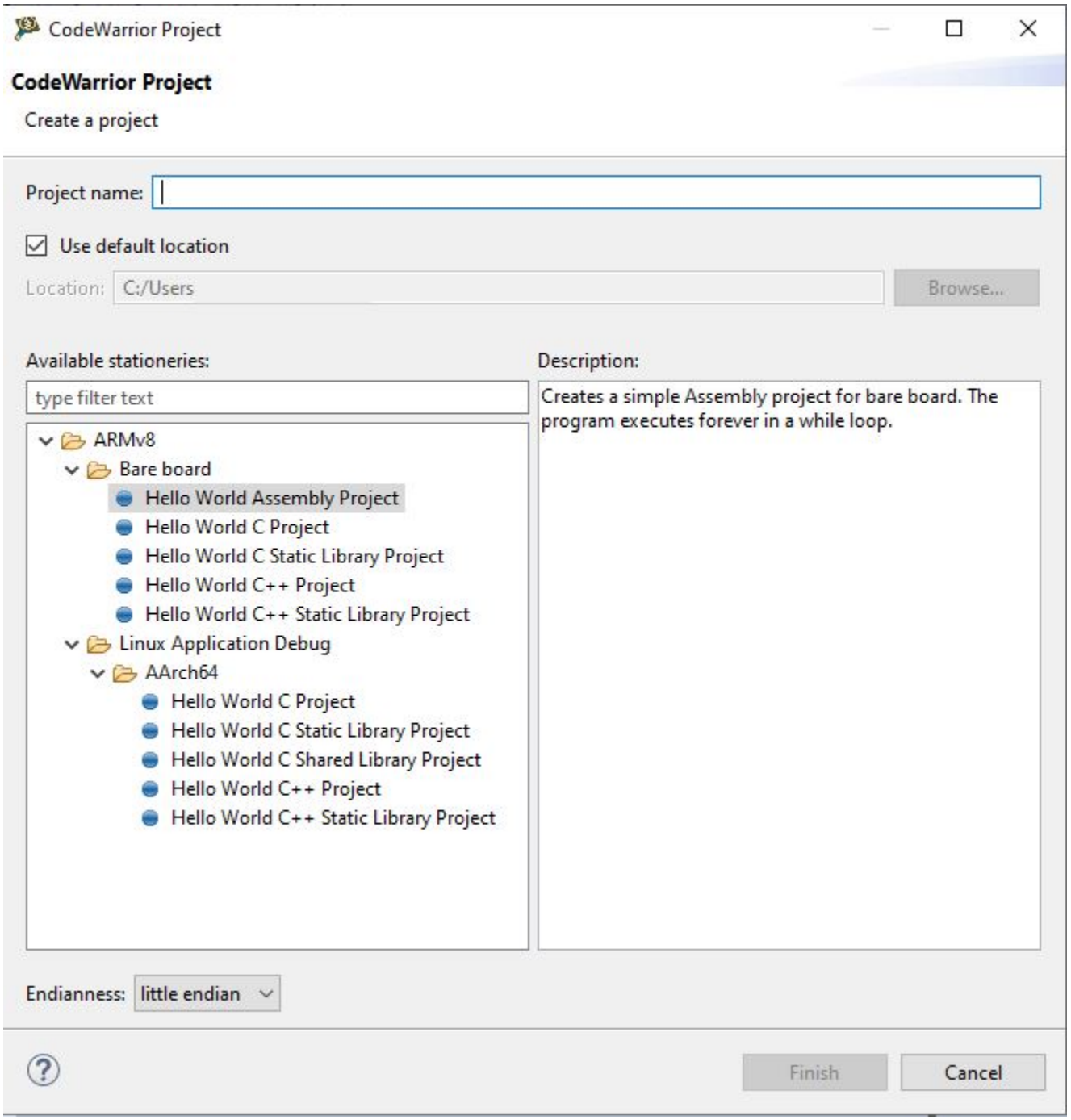


Figure 1. ARMv8 Project wizard

The left panel of the wizard displays a list of available sample projects and the right side panel provides short description for each of the stationery.

The table lists and explains the ARMv8 New Project wizard options.

Table 2. ARMv8 New Project wizard options

Option	Description
Project name	Enter the name for the new project in this text box. **Note:** Do not use the reserved/special characters/symbols such as < (less than), > (greater than), : (colon), " (double quote), / (forward slash), \ (backslash), | (vertical bar or pipe), ? (question mark), @ (at), * (asterisk) in the project name. The special characters/symbols in the project name may result in an unexpected behavior.
Use default location	Stores the files required to build the program in the current workspace directory. The project files are stored in the default location. **Clear the Use default location** checkbox and click **Browse** to select a new location.
Location	Specifies the directory that contains the project files. Click Browse to navigate to the desired directory. This option is available only when **Use default location** checkbox is clear.
Available Stationeries	List the various stationeries available for you to create a project. The stationeries are categorized under: Bareboard and Linux Application Debug.

2.2 CodeWarrior Executable Importer wizard

The CodeWarrior Executable Importer wizard allows users to import CodeWarrior ELF images of various types.

- Linux Application
- Bare-board
- Linux Kernel
- U-boot
- UEFI

You can access the wizard from **File > New > CodeWarrior Executable Importer**.

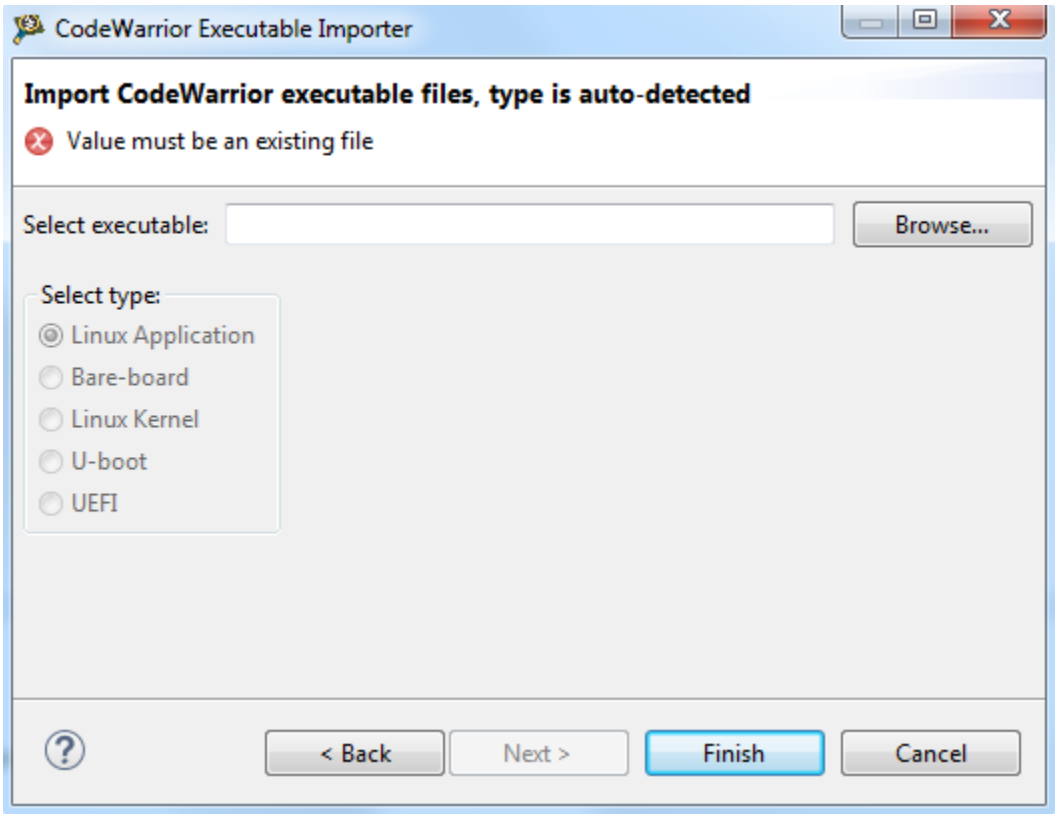


Figure 2. CodeWarrior Executable Importer, Select executable

Once the executable is selected the image type is auto-detected based on the symbol table. The user can overwrite the value by selecting another type.

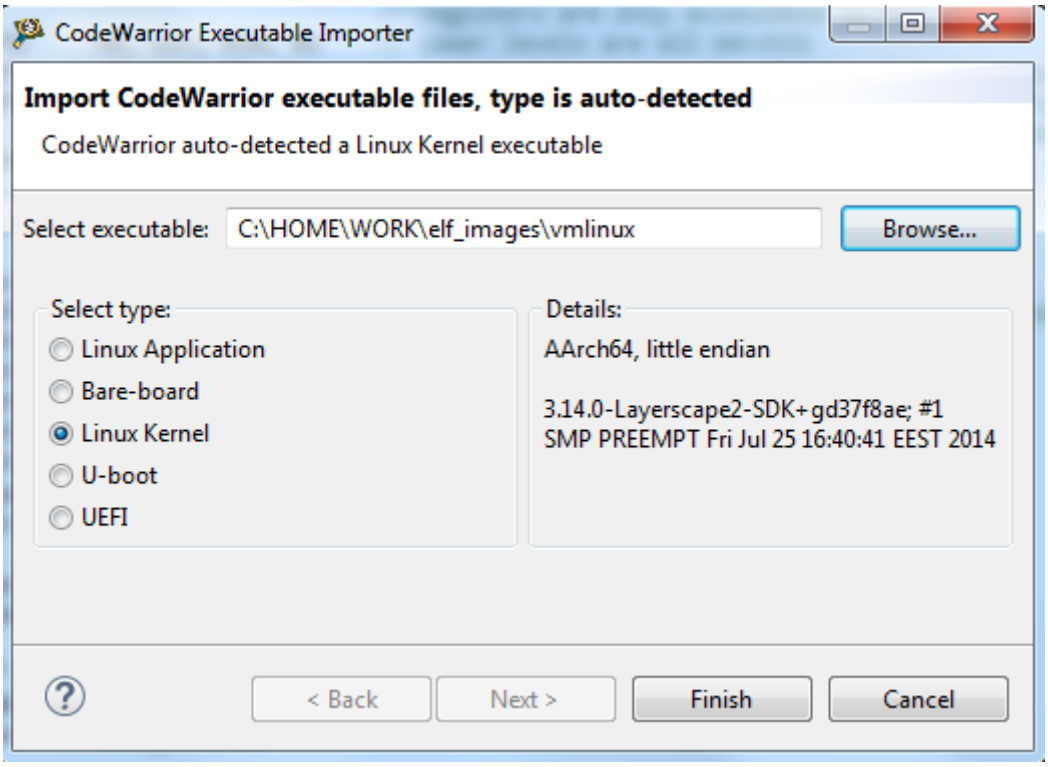


Figure 3. CodeWarrior Executable Importer, type is auto-detected

An error message is displayed and the user is not allowed to finish the project creation if the selected executable does not have the ELF or UEFI format. The created project contains the executable image as a linked resource and also a default launch configuration file with all the setup ready to debug.

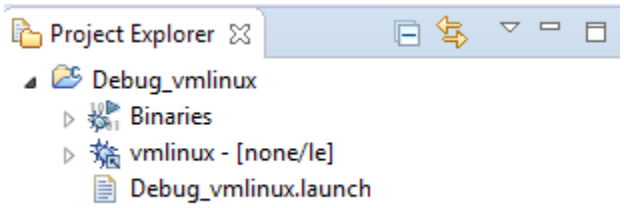


Figure 4. Project creation with CodeWarrior Executable Importer wizard

The launch configuration file is created automatically after the project creation. User only needs to review/change the settings and start the debug session.

2.3 Creating projects

This section explains how to use the ARMv8 New Project wizard to quickly create new projects with default settings, build and launch configurations.

The section explains:

- Creating CodeWarrior Bareboard project
- Creating CodeWarrior Linux Application project

2.3.1 Creating CodeWarrior Bareboard project

You can create a CodeWarrior Bareboard project using the ARMv8 Stationery wizard.

1. From CodeWarrior IDE menu bar, select **File > New > ARMv8 Stationery**
2. From **Available stationaries**, select **ARMv8 > Bare board > Hello World C Project**.
3. In **Project name** text box, enter `FirstProjectTest`.

   NOTE

   The **Location** text box shows the default workspace location. To change this location, uncheck the **Use default location** text box and click **Browse** to select a new location.

4. Click **Finish**.

   The new project appears in the **Project Explorer** view.

   NOTE

   Before you build and debug the project, ensure that the target board is ready. For details, see Preparing Target.

5. Build the bare metal project.
6. Debug the bare metal project. Refer Debugging Bareboard project.

You can create a CodeWarrior Bareboard project for following configurations:

- Assembly Project
- C Project
- C Static Library Project
- C++ Project
- C++ Static Library Project

2.3.2 Creating CodeWarrior Linux Application project

You can create a CodeWarrior Bareboard project using the ARMv8 Stationery wizard.

1. From CodeWarrior IDE menu bar, select **File > New > ARMv8 Stationery**.
2. From **Available stationeries**, select **ARMv8 > Linux Application Debug > Hello World C Project**.
3. In **Project name** text box, enter `FirstLinuxProject`.

   NOTE

   The **Location** text box shows the default workspace location. To change this location, uncheck the **Use default location** text box and click **Browse** to select a new location.

4. Click **Finish**.

   The new project appears in the **Project Explorer** view.

   NOTE

   Before you build and debug the project, ensure that the target board is ready. For details, see Preparing Target.

5. Build the Linux application project.
6. Debug the Linux application project. Refer Debugging projects

You can create a Linux application project for following configurations:

- C Project
- C Static Library Project

- C Shared Library Project
- C++ Project
- C++ Static Library Project

For further details about application debug projects, refer Linux application debug.

2.4 Preprocess/Disassemble files

You can access the Preprocess/Disassemble commands from the **Project Explorer** or **Editor** view.

The Preprocess/Disassemble commands are available to the user:

- from the menu that appears when you right-click on a file in the **Project Explorer** view, or
- from the menu that appears when you open the file in the **Editor** view and right-click inside the **Editor** view.

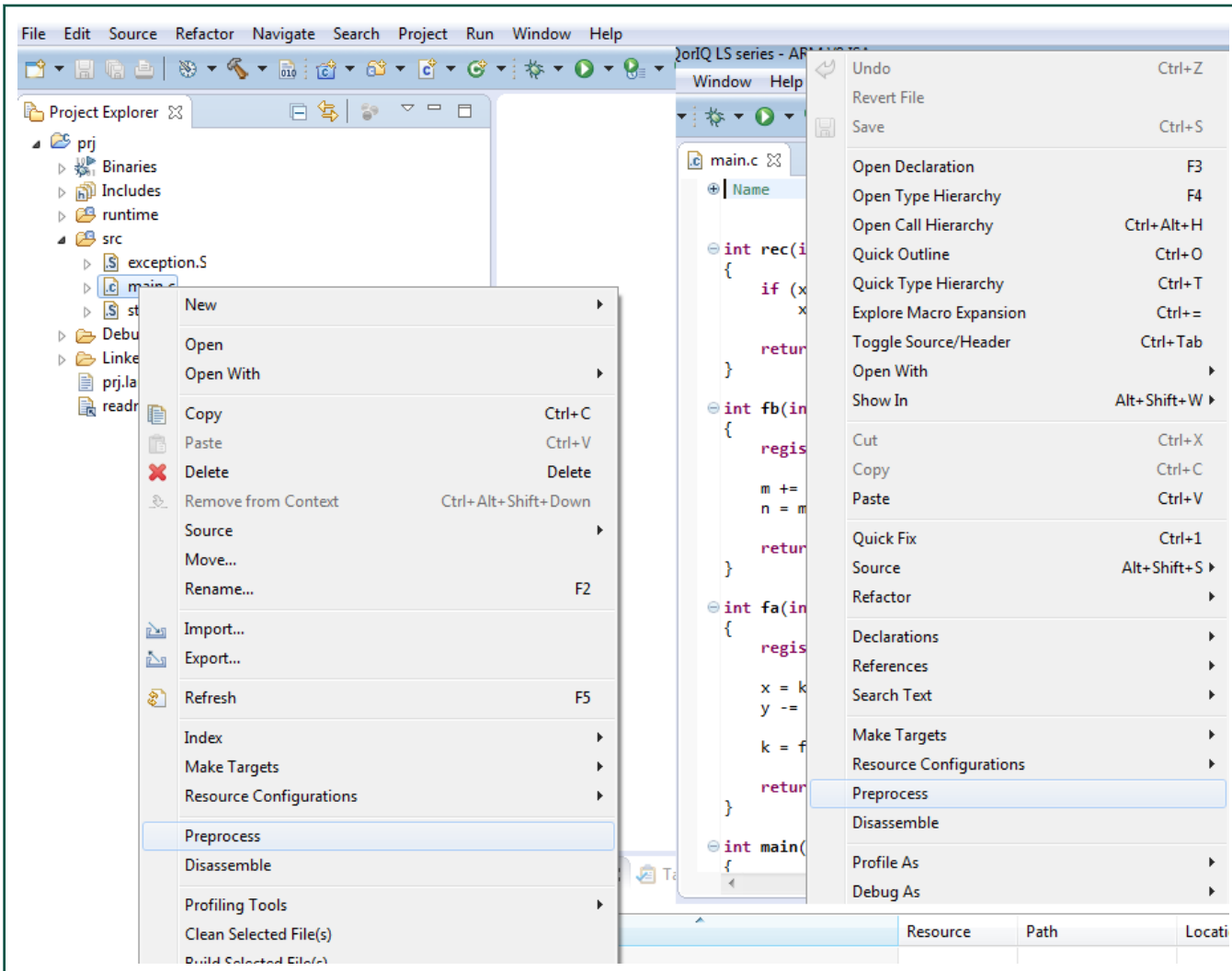


Figure 5. Project Explorer view and Editor view

The result of preprocessing a file or disassembling an object code is provided to the user in the Editor. Upon invocation, the Preprocess command preprocesses the C/C++/ASM file and shows the resulting text in a new file. Similarly, upon invocation, the Disassemble command compiles and disassembles the C/C++/ASM file or directly disassembles the binary file. In all the cases, the resulted files are located in the active configuration directory.

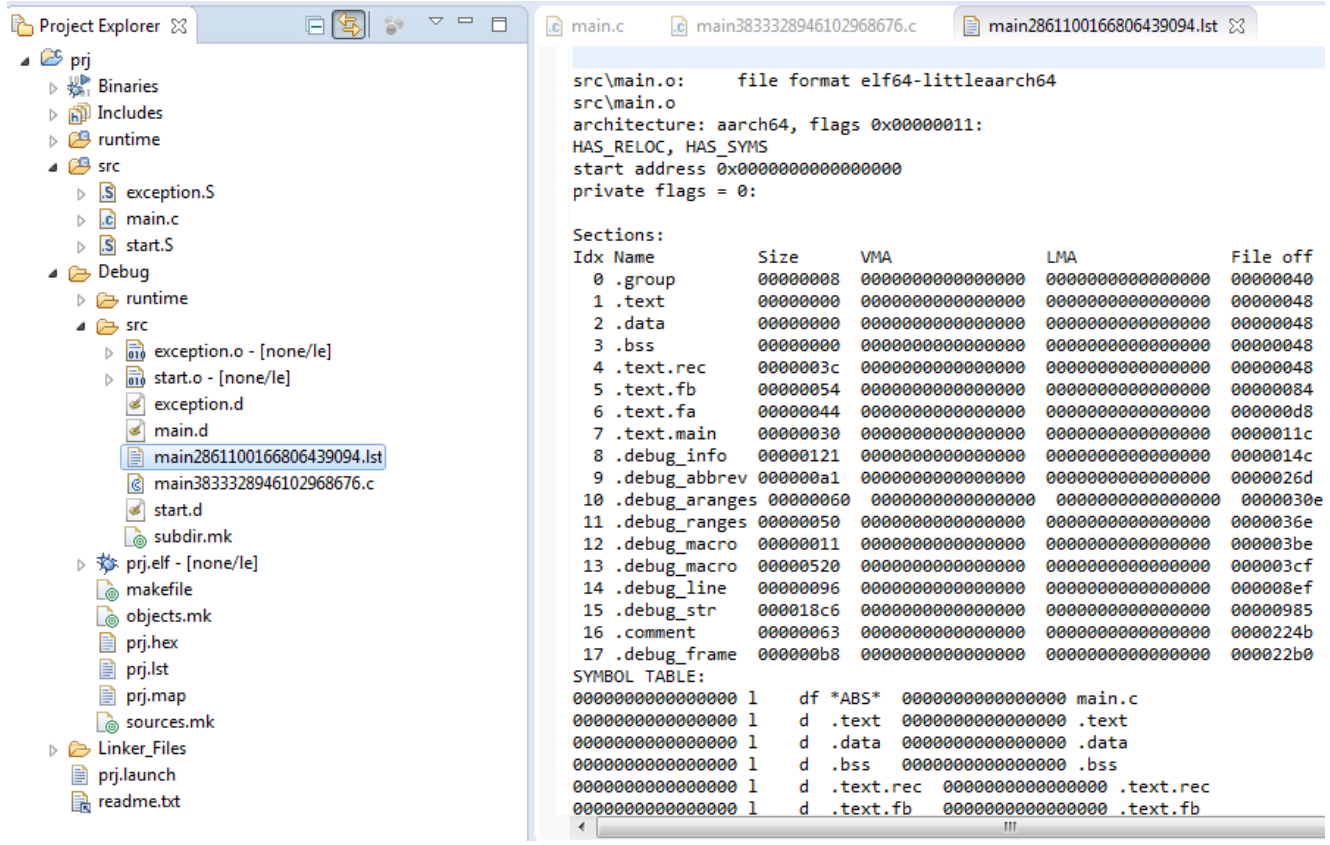


Figure 6. Editor view

NOTE

A new Console is created for each operation.

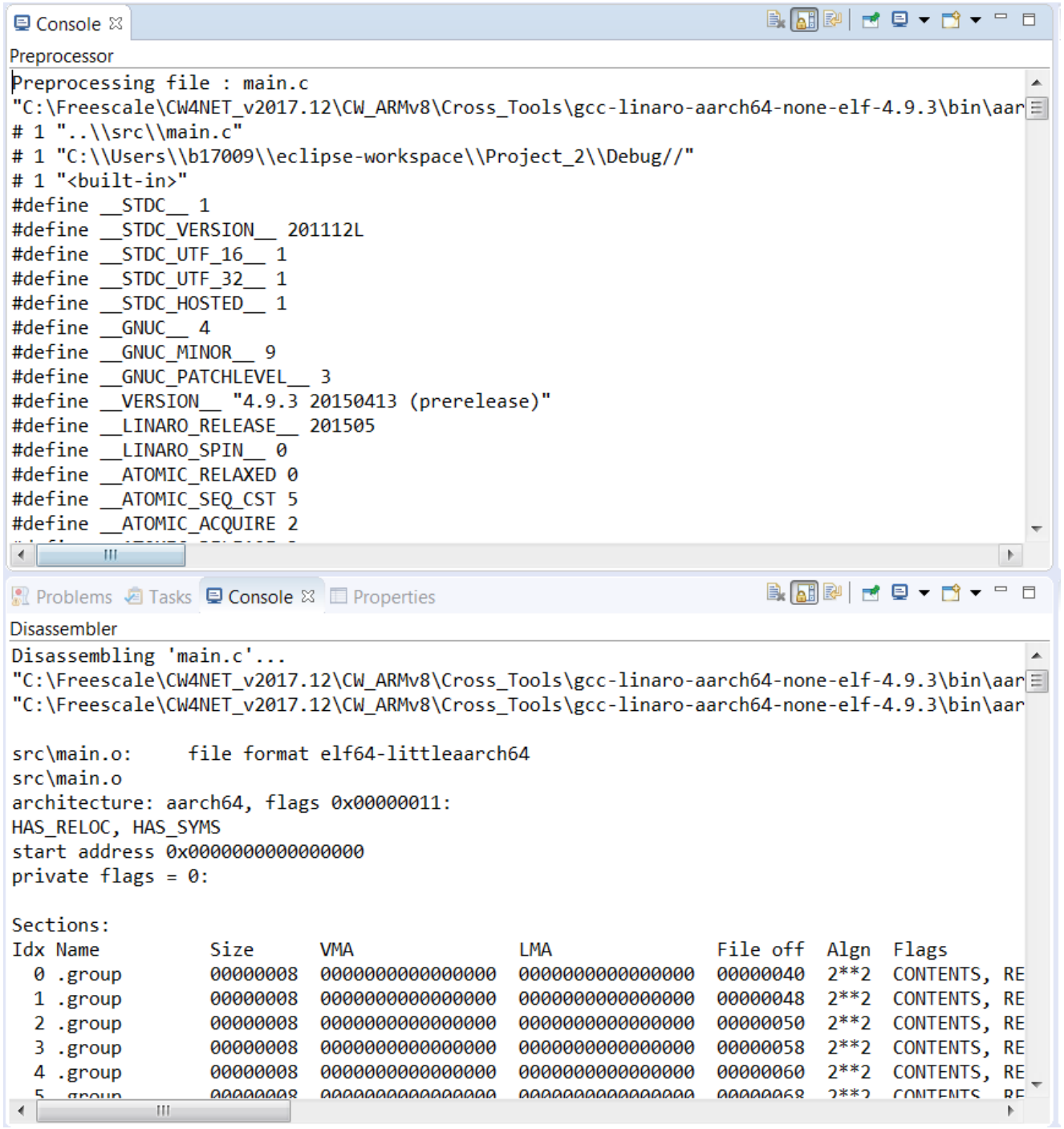


Figure 7. Console view

The user can define or modify preprocessor/disassembler options in the **Project Properties** dialog > **Settings > Tool Settings** page.

2.5 Debugging projects

When you use the ARMv8 Project wizard to create a new project, the wizard sets the debugger settings of the project's launch configurations to default values. You can change these default values based on your requirements.

To debug a project:

1. From the CodeWarrior IDE menu bar, select **Run > Debug Configurations**.

   The **Debug Configurations** dialog appears. The left side of this dialog box has a list of debug configurations that apply to the current application.

The ARMv8 Project wizard adds a default launch configuration in all application sample projects. The debugger settings are mapped to the default values but you can change these values based on your requirements.

2.5.1 Debugging Bareboard project

This topic describes how to debug a bareboard project.

Ensure that the project contains the default launch configuration file of type GDB Hardware Debugging, named as <projectName>.launch. To start debugging a project:

1. In the **Debug Configuration** dialog, select the available launch configuration.
2. Select a Target Connection Configurator. For details on this, refer Target Connection configurator overview and Configure the target configuration using Target Connection Configurator.
3. Click **Apply** in the **Debug Configurations** dialog. The IDE saves your settings.
4. Click **Debug**.

   The IDE switches to the **Debug** perspective. The debugger downloads your program to the target board and halts execution at the first statement of main().

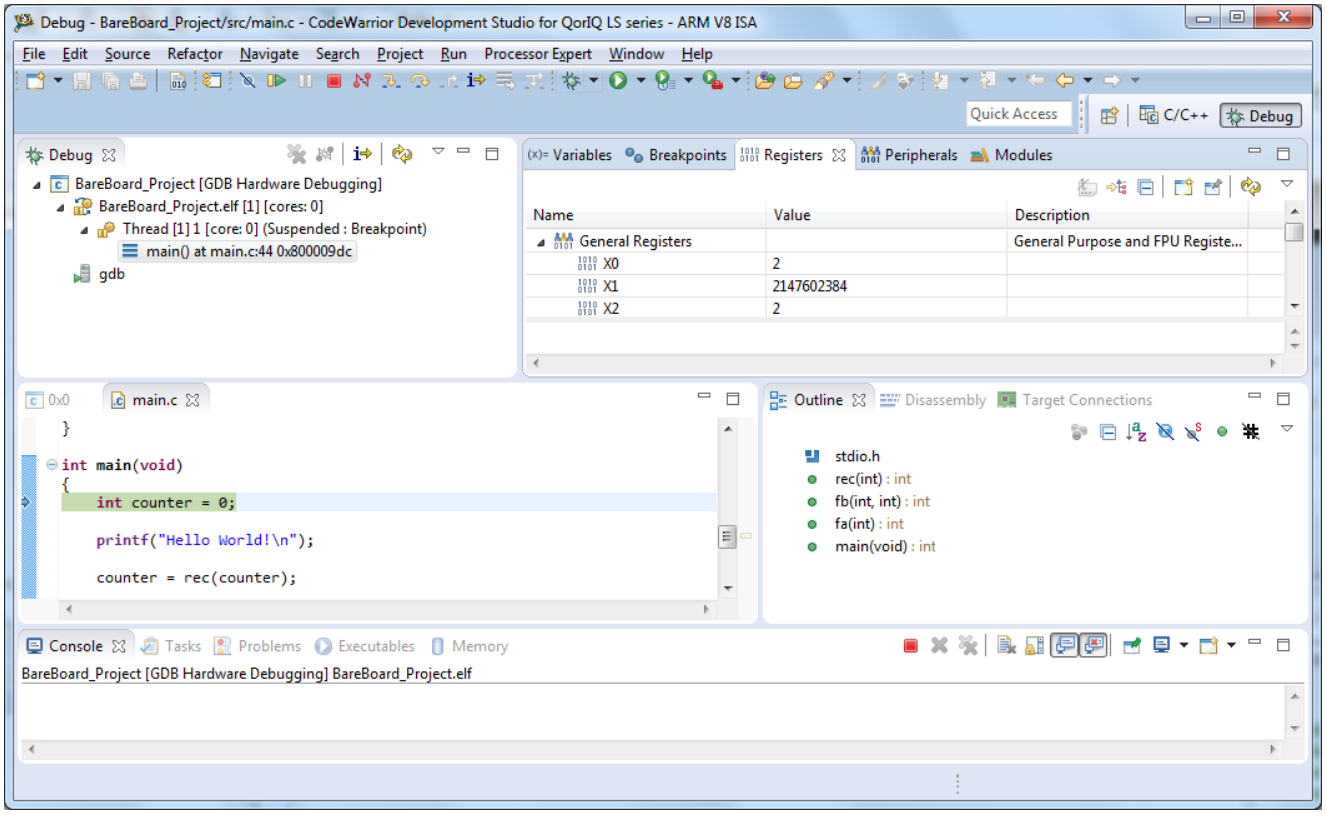


Figure 8. Debugging bareboard project

NOTE

To end the debug session, use the **Terminate** or **Disconnect** buttons in the **Debug** view (or from the **Run** menu). When using Terminate to end a debug session, debugger detaches from the cores that were debugged and keeps them in debug mode.

On the other hand, the Disconnect button resumes in-debug cores before the debugger detaches from the target.

2.5.2 Debugging Linux Application project

This topic describes how to debug a Linux application project.

Ensure that the project contains the default launch configuration file of type C/C++ Remote Application, named as <projectName>.launch.

The CodeWarrior software creates a default **remote ssh** connection, named **Remote Host**, once Linux application debug support is initialized. This connection is available in the **Connections** view. The default launch configuration file used in a Linux Application debug project points to this connection. The user can change the default settings, for example the IP of the Linux target.

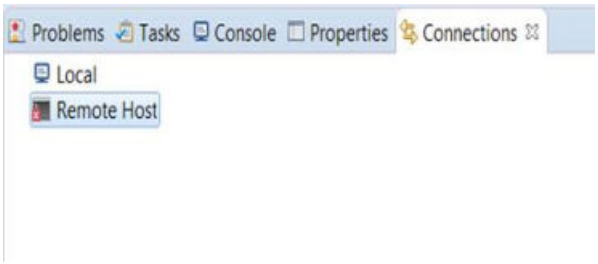


Figure 9. Remote Connections

To start debugging a project:

NOTE

If target is accessible on a port different than the default 22, like in the case of the ssh tunnelling to other port, the tunnelling port should be specified instead.

1. In the **Debug Configuration** dialog, select the available launch configuration.
2. Click **Debug**.

NOTE

For further details, refer Linux application debug.

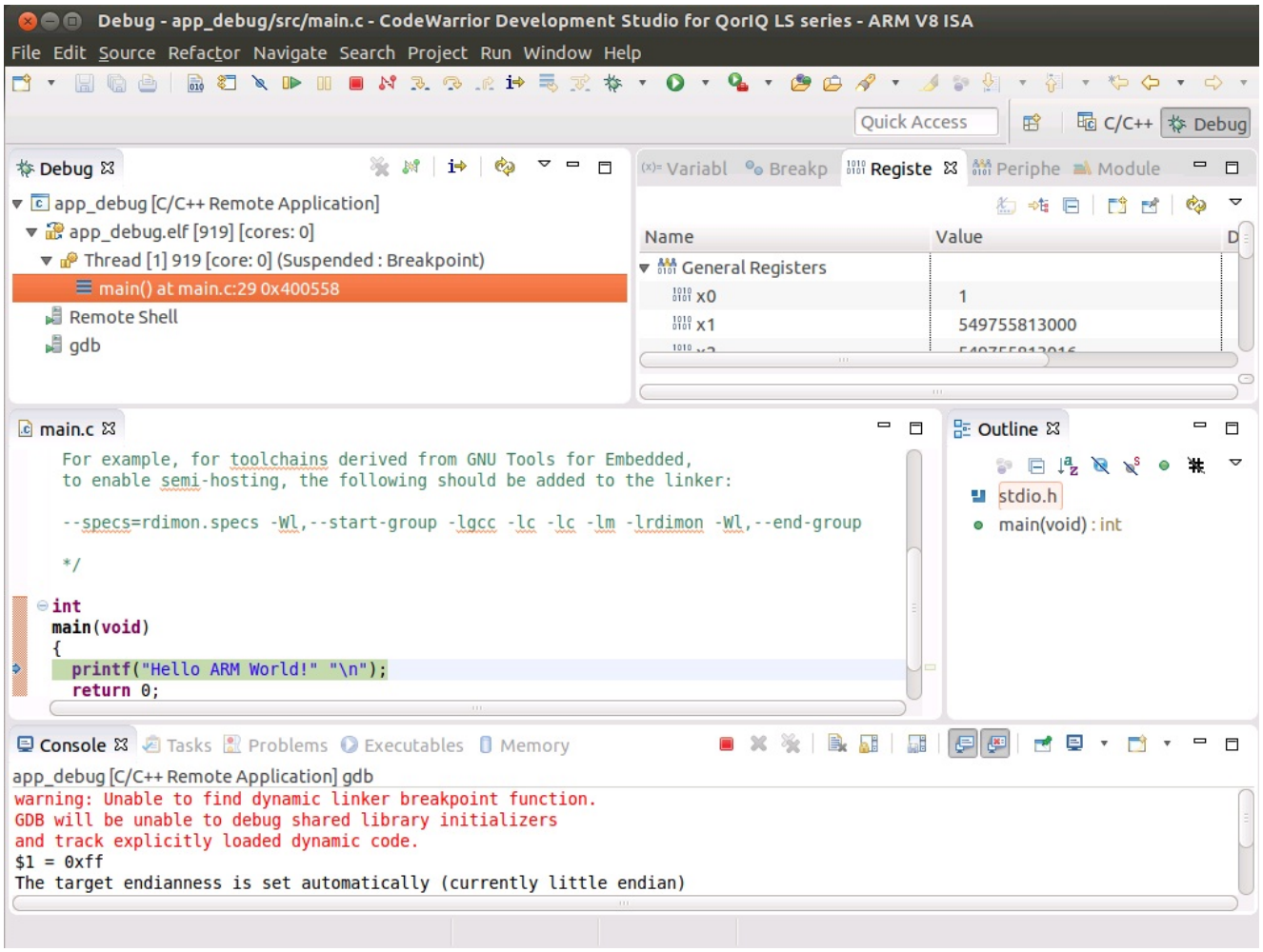


Figure 10. Debugging Linux Application project

Chapter 3
ARMv8 Build Properties

A build configuration is a named collection of build tools options.

The set of options in a given build configuration causes the build tools to generate a final binary with specific characteristics. For example, the binary produced by a "Debug" build configuration might contain symbolic debugging information and have no optimizations, while the binary product by a "Release" build configuration might contain no symbolics and be highly optimized.

For details about how ARMv8 projects are managed and all the available toolchains, refer Arm GNU Eclipse documentation available at: https://community.arm.com/tools/b/blog/posts/gnu-arm-eclipse-open-source-tools-with-experimental-cmsis-pack-support

NOTE

NXP does not own Arm GNU Eclipse documentation. The documents are mentioned solely for the reference purpose.

3.1 Changing Build Properties

The New Bareboard Project wizard creates a set of build properties for the project. You can modify these build properties to better suit your needs.

Perform these steps to change build properties:

1. Start the IDE.
2. In the **CodeWarrior Projects** view, select the project for which you want to modify the build properties.
3. Select **Project > Properties**.

   The **Properties** window appears. The left side of this window has a properties list. This list shows the build properties that apply to the current project.
4. Expand the **C/C++ Build** property.
5. Select **Settings**.

   The **Properties** window shows the corresponding build properties.
6. Use the Configuration drop-down list to specify the launch configuration for which you want to modify the build properties.
7. Click the **Tool Settings** tab.

   The corresponding page appears.
8. From the list of tools on the **Tool Settings** page, select the tool for which you want to modify properties.
9. Change the settings that appear in the page.
10. Click **Apply**.

    The IDE saves your new settings.

You can select other tool pages and modify their settings. When you finish, click OK to save your changes and close the **Properties** window.

3.2 ARMv8 build settings

The **Properties for** *<project>* window shows the corresponding Settings page for a project.

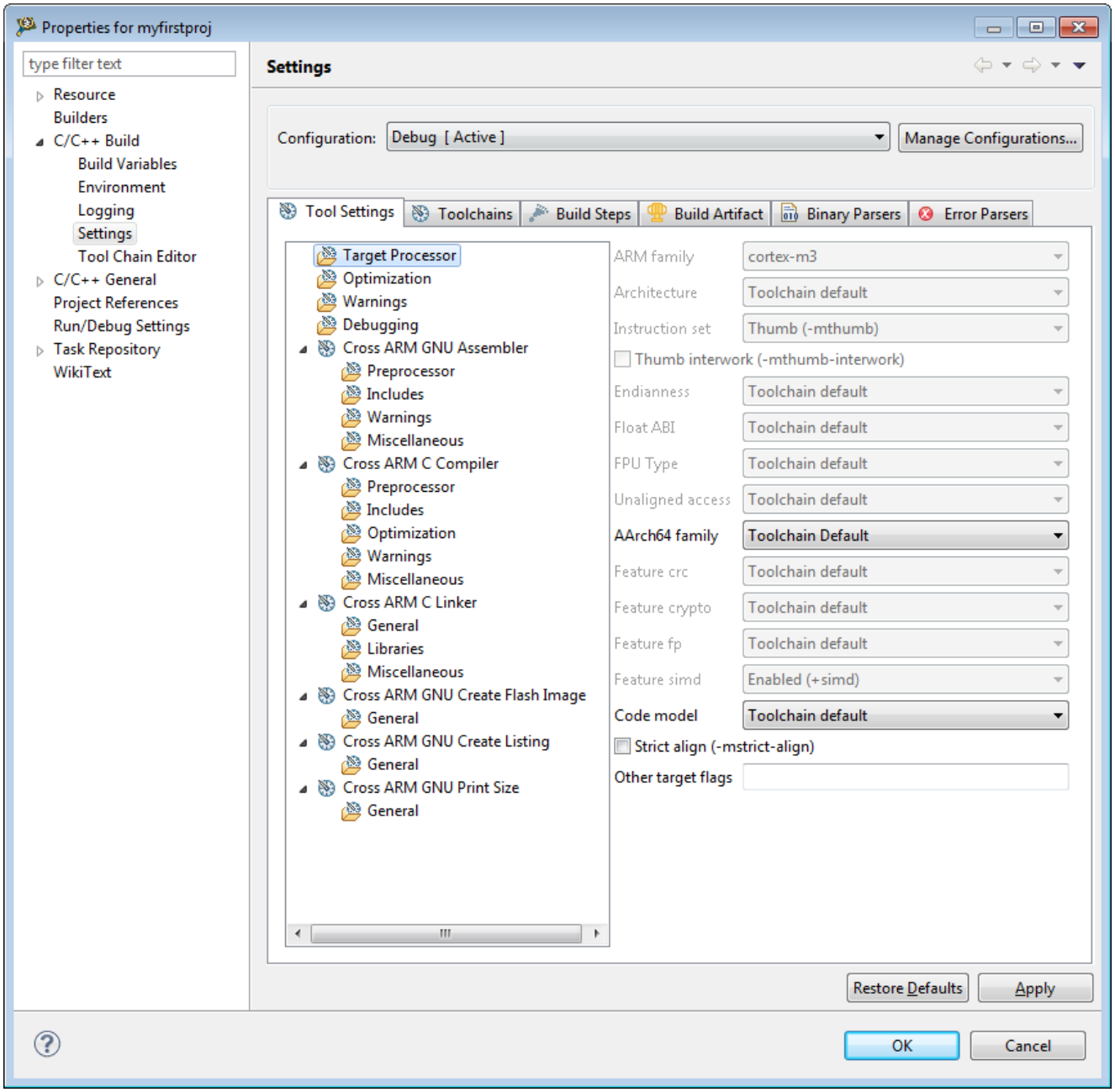


Figure 11. Settings page

The following table lists the build properties specific to developing software for Arm Embedded Processors.

The properties that you specify in the **Tool Settings** panels apply to the selected build tool on the **Tool Settings** page of the **Properties for** *<project>* dialog box.

Table 3. Build Properties for Bare Metal project

Tool Settings	Sub Tool Settings
Target Processor	Target Processor
Optimization	Optimization

Table continues on the next page...

Table 3. Build Properties for Bare Metal project (continued)

Tool Settings	Sub Tool Settings
Warnings	Warnings
Debugging	Debugging
Cross ARM GNU Assembler	Preprocessor
	Includes
	Warnings
	Miscellaneous
Cross ARM C Compiler	Preprocessor
	Includes
	Optimization
	Warnings
	Miscellaneous
Cross ARM C Linker	General
	Libraries
	Miscellaneous
Cross ARM GNU Create Flash Image	General
Cross ARM GNU Create Listing	General
Cross ARM GNU Print Size	General

3.2.1 Target Processor

Use this panel to configure the target processor options.

The following table lists the options in the **Target Processor** panel.

Table 4. Target Processor options

Option	Description
ARM family	Use to specify the Arm family name. Default: cortex-m3
Architecture	Use to specify the target hardware architecture or processor name. The compiler can take advantage of the extra instructions that the selected architecture provides and optimize the code to run on a specific processor. The inline assembler might display error messages or warnings if it assembles some processor-specific instructions for the wrong target architecture. Default: Toolchain default
Instruction set	Use to generate suitable interworking veneers when it links the assembler output. You must enable this option if you write Arm code that you want to interwork with Thumb code or vice versa. The only functions that need to be compiled for interworking are the functions that are called from the other state. You must ensure that your code uses the correct interworking return instructions.

Table continues on the next page...

Table 4. Target Processor options (continued)

Option	Description
	Default: Thumb (-mthumb)
Thumb interwork (-mthumb-interwork)	Check to have the processor generate Thumb code instructions. Clear to prevent the processor from generating Thumb code instructions. The IDE enables this setting only for architectures and processors that support the Thumb instruction set. Default: Clear
Endianness	Use to specify the byte order of the target hardware architecture: • Little-little endian; right-most bytes (those with a higher address) are most significant
Float ABI	Use to specify the float Application Binary Interface (ABI). Default: Toolchain default
FPU Type	Use to specify the type of floating-point unit (FPU) for the target hardware architecture: The assembler might display error messages or warnings if the selected FPU architecture is not compatible with the target architecture. Default: Toolchain default
Unaligned access	Use to specify unaligned access. Default: Toolchain default
AArch64 family	Use to specify the architecture family: • Generic (-mcpu=generic) • Large (-mcpu=large) • Toolchain default Default: Toolchain default
Feature crc	Use to specify Feature crc.
Feature crypto	Use to specify Feature crypts.
Feature fp	Use to specify Feature fp.
Feature simd	Use to specify Feature simd.
Code model	Specifies the addressing mode that the linker uses when resolving references. This setting is equivalent to specifying the -mcmodel keyword command-line option. • Tiny (-mcmdel=tiny) • Small (-mcmodel=small) • Large (-mcmodel=large) • Toolchain default
Strict align (-mstrict-align)	Controls the use of non-standard ISO/IEC 9899-1990 ("C90") language features.
Other target flags	Specify additional command line options; type in custom flags that are not otherwise available in the UI.

3.2.2 Optimization

Use this panel to configure the optimization options.

The following table lists the options in the **Optimization** panel.

Table 5. Optimization options

Option	Description
Optimization level	Specify the optimizations that you want the compiler to apply to the generated object code: • None (-O0)-Disable optimizations. This setting is equivalent to specifying the -O0 command-line option. The compiler generates unoptimized, linear assembly-language code. • Optimize (-O1)-The compiler performs all target-independent (that is, non-parallelized) optimizations, such as function inlining. This setting is equivalent to specifying the -O1 command-line option. The compiler omits all target-specific optimizations and generates linear assembly-language code. • Optimize more (-O2)-The compiler performs all optimizations (both target-independent and target-specific). This setting is equivalent to specifying the -O2 command-line option. The compiler outputs optimized, non-linear, parallelized assembly-language code. • Optimize most (-O3)-The compiler performs all the level 2 optimizations, then the low-level optimizer performs global-algorithm register allocation. This setting is equivalent to specifying the that is usually faster than the code generated from level 2 optimizations. • Optimize size (-Os)-The compiler optimizes object code at the specified Optimization Level such that the resulting binary file has a smaller executable code size, as opposed to a faster execution speed. This setting is equivalent to specifying the -Os command-line option. • Optimize for debugging (-Og)-The compiler optimizes object code at the specified Optimization Level such that the resulting binary file has a faster execution speed, as opposed to a smaller executable code size.
Message length (-fmessage-length=0)	Check if you want to specify the maximum length in bytes for the message.
'char' is signed (-fsigned-char)	Check to treat char declarations as signed char declarations.
Function sections (-ffunction-sections)	Check to enable function sections.
Data sections (-fdata-sections)	Check to enable data sections.
No common unitialized (-fno-common)	Controls the placement of uninitialized global variables.
Do not inline functions (-fno-inline-functions)	Suppresses automatic inlining of subprograms.
Assume freestanding environment (-ffeestanding)	Asserts that compilation takes place in a freestanding environment. This implies -fno-builtin.
Disable builtin (-fno-builtin)	Switches off builtin functions.

Table continues on the next page...

Table 5. Optimization options (continued)

Option	Description
Single precision constants (-fsingle-precision-constant)	Check to enable single precision constants.
Position independent code (-fPIC)	Select to instruct the build tools to generate position independent-code.
Link-time optimizer (-flto)	Runs the standard link-time optimizer. When invoked with source code, it generates GIMPLE (one of GCC's internal representations) and writes it to special ELF sections in the object file. When the object files are linked together, all the functions bodies are read from these ELF sections and instantiated as if they had been part of the same translation unit.
Disable loop invariant move (-fno-move-loop-invariants)	Disables the loop invariant motion pass in the RTL loop optimizer. Enabled at level '-O1'.
Other optimization flags	Specify additional command line options; type in custom optimization flags that are not otherwise available in the UI.

3.2.3 Warnings

Use this panel to configure the warning options.

The following table lists the options in the **Warnings** panel.

Table 6. Warnings options

Option	Description
Check syntax only (-fsyntax-only)	Check this option if you want to check the syntax of commands and throw a syntax error.
Pedantic (-pedantic)	Check if you want warnings like -pedantic, except that errors are produced rather than warnings.
Pedantic warnings as errors (-pedantic-errors)	Check this option if you want to inhibit the display of warning messages.
Inhibit all warnings (-w)	Check this option if you want to enable all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning), even in conjunction with macros.
Warn on various unused elements (-Wunused)	Warn whenever some element (label, parameter, function, etc.) is unused.
Warn on uninitialized variables (-Wuninitialised)	Warn whenever an automatic variable is used without first being initialized.
Enable all common warnings (-Wall)	Check this option if you want to enable all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning), even in conjunction with macros.
Enable extra warnings (-Wextra)	Check this option to enable any extra warnings.
Warn on undeclared global function (-Wmissing-declaration)	Check to warn if an undeclared global function is encountered.
Warn on implicit conversions (-Wconversion)	Check to warn of implicit conversions.

Table continues on the next page...

Table 6. Warnings options (continued)

Option	Description
Warn if pointer arithmetic (-Wpointer-arith)	Check to warn if pointer arithmetic are used.
Warn if padding is not included (-Wpadded)	Check to warn if padding is included in a structure either to align an element of the structure or the whole structure.
Warn if shadowed variable (-Wshadow)	Check to warn if shadowed variable are used.
Warn if suspicious logical ops (-Wlogical-op)	Check to warn in case of suspicious logical operation.
Warn in struct is returned (-Wagreggrate-return)	Check to warn if struct is returned.
Warn if floats are compared as equal (-Wfloat-equal)	Check to warn if floats are compared as equal.
Generate errors instead of warnings (-Werror)	Check to generate errors instead of warnings.
Other warning flags	Specify additional command line options; type in custom warning flags that are not otherwise available in the UI.

3.2.4 Debugging

Use this panel to configure the debugging options.

The following table lists the options in the **Debugging** panel.

Table 7. Debugging options

Option	Description
Debug level	Specify the debug levels: • None - No Debug level. • Minimal (-g1) - The compiler provides minimal debugging support. • Default (-g) - The compiler generates DWARF 1.xconforming debugging information. • Maximum (-g3) - The compiler provides maximum debugging support.
Debug format	Specify the debug formats for the compiler.
Generate prof information (-p)	Generates extra code to write profile information suitable for the analysis program prof. You must use this option when compiling the source files you want data about, and you must also use it when linking.
Generate gprof information (-pg)	Generates extra code to write profile information suitable for the analysis program gprof. You must use this option when compiling the source files you want data about, and you must also use it when linking.
Other debugging flags	Specify additional command line options; type in custom debugging flags that are not otherwise available in the UI.

3.2.5 Cross ARM GNU Assembler

Use this panel to configure the Arm GNU assembler options.

The following table lists the options in the **Cross ARM GNU Assembler** panel.

Table 8. Cross ARM GNU Assembler options

Option	Description
Command	Shows the location of the assembler executable file. Default: `${cross_prefix}${cross_c}${cross_suffix}`
All Options	Shows the actual command line the assembler will be called with. Default: `-x assembler-with-cpp -Xassembler -g`
Expert settings	
Command line pattern	Shows the expert settings command line parameters. Default: `${COMMAND} ${cross_toolchain_flags} ${FLAGS} -c ${OUTPUT_FLAG} ${OUTPUT_PREFIX}${OUTPUT} ${INPUTS}`

3.2.5.1 Preprocessor

Use this panel to configure the Arm GNU assembler preprocessor options.

The following table lists the options in the **Cross Arm GNU Assembler Preprocessor** panel.

Table 9. Cross Arm GNU Assembler Preprocessor options

Option	Description
Use preprocessor	Check this option to use the preprocessor for the assembler.
Do not search system directories (-nostdinc)	Check this option if you do not want the assembler to search the system directories. By default, this checkbox is clear. The assembler performs a full search that includes the system directories.
Preprocess only (-E)	Check this option if you want the assembler to preprocess source files and not to run the compiler. By default, this checkbox is clear and the source files are not preprocessed.
Defined symbols (-D)	Use this option to specify the substitution strings that the assembler applies to all the assembly-language modules in the build target. Enter just the string portion of a substitution string. The IDE prepends the -D token to each string that you enter. For example, entering opt1 x produces this result on the command line: -Dopt1 x. Note: This option is similar to the DEFINE directive, but applies to all assembly-language modules in a build target.
Undefined symbols (-U)	Undefines the substitution strings you specify in this panel.

3.2.5.2 Includes

Use this panel to configure the Arm GNU assembler includes options.

The following table lists the options in the Cross Arm GNU Assembler Includes panel.

Table 10. Cross Arm GNU Assembler Includes options

Option	Description
Include paths (-I)	This option changes the build target's search order of access paths to start with the system paths list. The compiler can search #include files in several different ways. You can also set the search order as follows: For include statements of the form #include"xyz", the compiler first searches user paths, then the system

Table continues on the next page...

Table 10. Cross Arm GNU Assembler Includes options (continued)

Option	Description
	paths For include statements of the form #include<xyz>, the compiler searches only system paths This option is global.
Include files (-include)	Use this option to specify the include file search path.

3.2.5.3 Warnings

Use this panel to configure the Arm GNU assembler warning options.

The following table lists the options in the **Cross Arm GNU Assembler Warnings** panel.

Table 11. Warnings options

Option	Description
Other warning flags	Specify additional command line options; type in custom warning flags that are not otherwise available in the UI.

3.2.5.4 Miscellaneous

Use this panel to configure the Arm GNU assembler miscellaneous options.

The following table lists the options in the **Cross Arm GNU Assembler Miscellaneous** panel.

Table 12. Cross Arm GNU Assembler Miscellaneous options

Option	Description
Assembler flags	Specify the flags that need to be passed with the assembler.
Generates assembler listing (-Wa, -adhlns="$@.lst")	Enables the assembler to create a listing file as it compiles assembly language into object code.
Save temporary files (--save-temps Use with caution!)	Store the usual “temporary” intermediate files permanently.
Verbose (-v)	Check this option if you want the IDE to show each command-line that it passes to the shell, along with all progress, error, warning, and informational messages that the tools emit. This setting is equivalent to specifying the -v command-line option. By default this checkbox is clear. The IDE displays just error messages that the compiler emits. The IDE suppresses warning and informational messages.
Other assembler flags	Specify additional command line options; type in custom flags that are not otherwise available in the UI.

3.2.6 Cross ARM C Compiler

Use this panel to configure the Arm C compiler options.

The following table lists the options in the **Cross ARM C Compiler** panel.

Table 13. Cross ARM C Compiler options

Option	Description
Command	Shows the location of the compiler executable file. Default: `${cross_prefix}${cross_c}${cross_suffix}`
All Options	Shows the actual command line the compiler will be called with.
Expert settings	
Command line patterns	Shows the expert settings command line parameters. Default: `${COMMAND} ${cross_toolchain_flags} ${FLAGS} -c ${OUTPUT_FLAG} ${OUTPUT_PREFIX}${OUTPUT} ${INPUTS}`

3.2.6.1 Preprocessor

Use this panel to configure the Arm C compiler preprocessor options.

The following table lists the options in the **Cross Arm C Compiler Preprocessor** panel.

Table 14. Cross Arm GNU compiler Preprocessor options

Option	Description
Do not search system directories (-nostdinc)	Check this option if you do not want the compiler to search the system directories. By default, this checkbox is clear. The compiler performs a full search that includes the system directories.
Preprocess only (-E)	Check this option if you want the compiler to preprocess source files and not to run the compiler. By default, this checkbox is clear and the source files are not preprocessed.
Defined symbols (-D)	Use this option to specify the substitution strings that the compiler applies modules in the build target. Enter just the string portion of a substitution string. The IDE prepends the -D token to each string that you enter. For example, entering opt1 x produces this result on the command line: -Dopt1 x. Note: This option is similar to the DEFINE directive, but applies to all assembly-language modules in a build target.
Undefined symbols (-U)	Undefines the substitution strings you specify in this panel.

3.2.6.2 Includes

Use this panel to configure the Arm C compiler includes options.

The following table lists the options in the Cross Arm C Compiler Includes panel.

Table 15. Cross Arm C Compiler Includes options

Option	Description
Include paths (-I)	This option changes the build target's search order of access paths to start with the system paths list. The compiler can search #include files in several different ways. You can also set the search order as follows: For include statements of the form #include"xyz", the compiler first searches user paths, then the system paths For include statements of the form #include<xyz>, the compiler searches only system paths This option is global.
Include files (-include)	Use this option to specify the include file search path.

3.2.6.3 Optimization

Use this panel to configure the Arm C compiler optimization options.

The following table lists the options in the **Optimization** panel.

Table 16. Optimization options

Option	Description
Language standard	Select the programming language or standard to which the compiler should conform. • ISO C90 (-ansi) - Select this option to compile code written in ANSI standard C. The compiler does not enforce strict standards. For example, your code can contain some minor extensions, such as C++ style comments (//), and $ characters in identifiers. • ISO C99 (-std=c99) - Select this option to instruct the compiler to enforce stricter adherence to the ANSI/ISO standard. • Compiler Default (ISO C90 with GNU extensions) - Select this option to enforce adherence to ISO C90 with GNU extensions. • ISO C99 with GNU Extensions (-std=gnu99)
Other optimization flags	Specify additional command line options; type in custom optimization flags that are not otherwise available in the UI.

3.2.6.4 Warnings

Use this panel to configure the Arm C compiler warnings options.

The following table lists the options in the **Warnings** panel.

Table 17. Warnings options

Option	Description
Warn if a global function has no prototype (-Wmissing-prototype)	Warn if a global function has no prototype.
Warn if a function has no arg type (-Wstrict-prototypes)	Warn if a function is declared or defined without specifying the argument types.
Warn if a wrong cast (-Wbad-function-cast)	Warn whenever a function call is cast to a non-matching type.
Other warning flags	Specify additional command line options; type in custom warning flags that are not otherwise available in the UI.

3.2.6.5 Miscellaneous

Use this panel to configure the Arm C compiler miscellaneous options.

The following table lists the options in the **Miscellaneous** panel.

Table 18. Miscellaneous options

Option	Description
Generates assembler listing (-	Enables the assembler to create a listing file as it compiles assembly language into object code.

Table continues on the next page...

Table 18. Miscellaneous options (continued)

Option	Description
Wa, -adhlns="$@.lst")	
Save temporary files (--save-temps Use with caution!)	Store the usual “temporary” intermediate files permanently.
Verbose (-v)	Check this option if you want the IDE to show each command-line that it passes to the shell, along with all progress, error, warning, and informational messages that the tools emit. This setting is equivalent to specifying the -v command-line option. By default this checkbox is clear. The IDE displays just error messages that the compiler emits. The IDE suppresses warning and informational messages.
Other compiler flags	Specify additional command line options; type in custom flags that are not otherwise available in the UI.

3.2.7 Cross ARM C Linker

Use this panel to configure the ARM C linker options.

The following table lists the options in the **Cross ARM C Linker** panel.

Table 19. Cross ARM C Linker options

Option	Description
Command	Shows the location of the linker executable file. Default: `${cross_prefix}${cross_c}${cross_suffix}`
All Options	Shows the actual command line the assembler will be called with. Default: `-T "${ProjDirPath}"/Linker_Files/aarch64elf.x -nostartfiles -nodefaultlibs -L"C:\Users\b14174\workspace-15\FirstProjectTest" -Wl,-Map,"FirstProjectTest.map"`
Expert settings	
Command line patterns	Shows the expert settings command line parameters. Default: `${COMMAND} ${cross_toolchain_flags} ${FLAGS} ${OUTPUT_FLAG} ${OUTPUT_PREFIX}${OUTPUT} ${INPUTS}`

3.2.7.1 General

Use this panel to configure the Arm C linker general options.

The following table lists the options in the General panel.

Table 20. General options

Option	Description
Script files (-T)	This option passes the -T argument to the linker file
Do not use standard start files (-nostartfiles)	This option passes the -nostartfiles argument to the linker file. It does not allow the use of the standard start files.
Do not use default libraries (-nodefaultlibs)	This option passes the -nodefaultlibs argument to the linker file. It does not allow the use of the default libraries.
No startup or default libs (-nostdlib)	This option passes the -nostdlib argument to the linker file. It does not allow the use of startup or default libs.

Table continues on the next page...

Table 20. General options (continued)

Option	Description
Remove unused sections (-Xlinker --gc-sections)	This option passes the -Xlinker --gc-sections argument to the linker file. It removes the unused sections.
Print removed sections (-Xlinker --print-gc-sections)	This option passes the -Xlinker --print-gc-sections argument to the linker file. It prints the removed sections.
Omit all symbol information (-s)	This option passes the -s argument to the linker file. This option omits all symbol information.

3.2.7.2 Libraries

Use this panel to configure the Arm C linker libraries options.

The following table lists the options in the Libraries panel.

Table 21. Libraries options

Option	Description
Libraries (-l)	This option changes the build target's search order of access paths to start with the system paths list. The compiler can search #include files in several different ways. You can also set the search order as follows: For include statements of the form #include"xyz", the compiler first searches user paths, then the system paths. For include statements of the form #include<xyz>, the compiler searches only system paths. This option is global.
Library search path (-L)	Use this option to specify the include library search path.

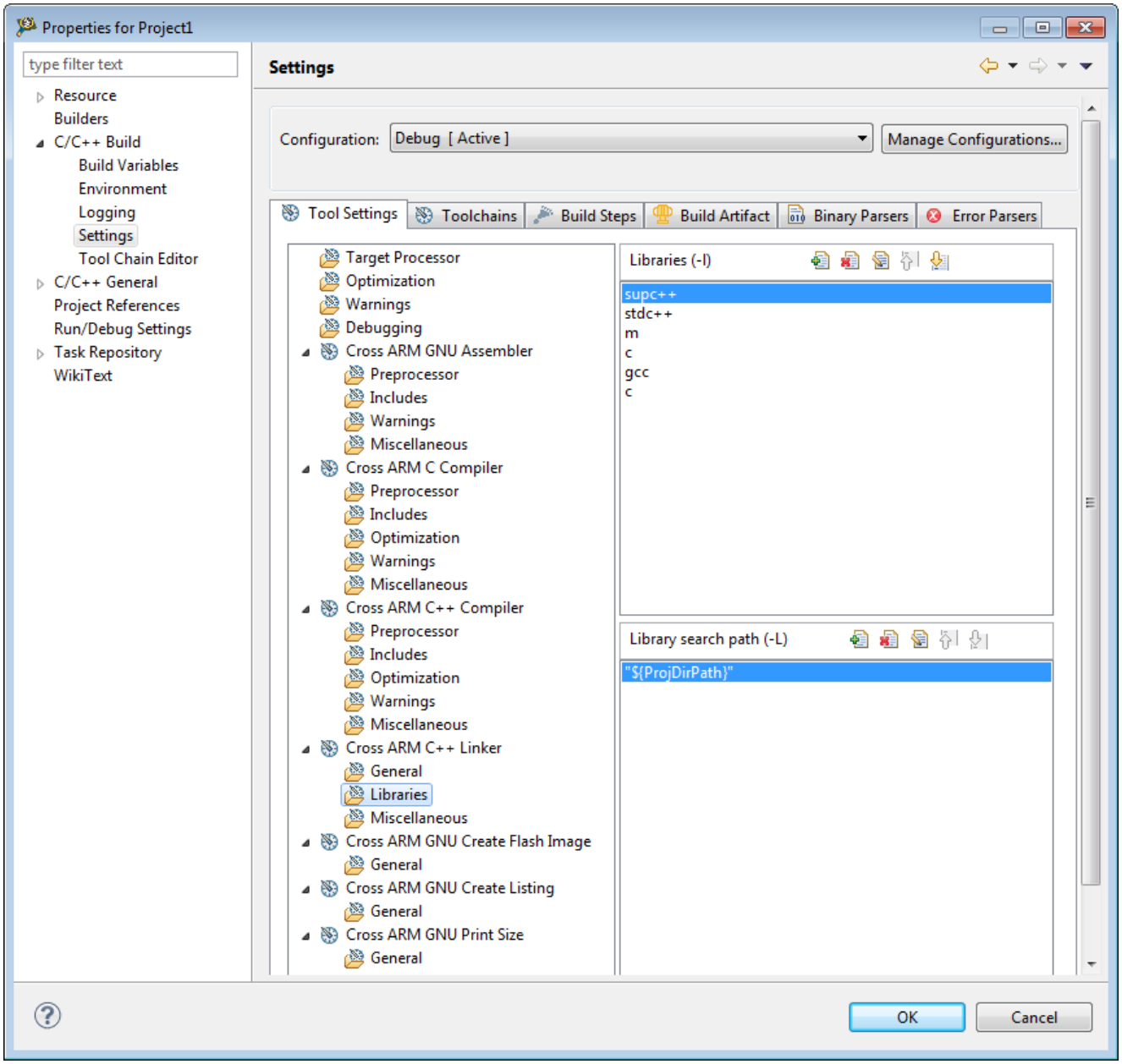


Figure 12. Libraries panel

3.2.7.3 Miscellaneous

Use this panel to configure the Arm C linker miscellaneous options.

The following table lists the options in the **Miscellaneous** panel.

Table 22. Miscellaneous options

Option	Description
Linker flags	This option specifies the flags to be passed with the linker file.

Table continues on the next page...

Table 22. Miscellaneous options (continued)

Option	Description
Other objects	This option lists paths that the VSPA linker searches for objects. The linker searches the paths in the order shown in this list.
Generate Map	This option specifies the map filename. Default: $ {BuildArtifactFileBaseName}.map
Cross Reference (-Xlinker --cref)	Check this option to instruct the linker to list cross-reference information on symbols. This includes where the symbols were defined and where they were used, both inside and outside macros.
Print link map (-Xlinker --printf-map)	Check this option to instruct the linker to print the map file.
Use newlib-nano (--specs=nano.specs)	Check this option to select a version of Newlib focused on code size. Newlib-Nano can help to reduce the size of your application compared to using the standard version of Newlib for both C and C++ projects.
Use float with nano printf (-u_printf_float)	Check this option to handle floating-point value using nano Newlib-Nano printf. By default, Newlib-Nano uses non-floating point variants of the printf and scanf family of functions, to reduce the size if only integer values are used by such functions.
Use float with nano scanf (-u_scanf_float)	Check this option to handle floating-point value using nano Newlib-Nano scanf. By default, Newlib-Nano uses non-floating point variants of the printf and scanf family of functions, to reduce the size if only integer values are used by such functions.
Verbose (-v)	Check this option to show verbose information, including hex dump of program segments in applications; default setting
Other linker flags	Specify additional command line options for the linker; type in custom flags that are not otherwise available in the UI.

3.2.8 Cross ARM GNU Create Flash Image

Use this panel to configure the Cross Arm GNU create flash image options.

The following table lists the options in the **Cross ARM GNU Create Flash Image** panel.

Table 23. Cross ARM GNU Create Flash Image options

Option	Description
Command	Shows the location of the executable file. Default: `${cross_prefix}${cross_objcopy}${cross_suffix}`
All Options	Shows the actual command line the assembler will be called with. Default: `"FirstProjectTest.elf" -O ihex`
Expert settings	
Command line patterns	Shows the expert settings command line parameters. Default: `${COMMAND} ${FLAGS} ${OUTPUT_FLAG} ${OUTPUT_PREFIX}${OUTPUT} ${INPUTS}`

3.2.8.1 General

Use this panel to configure the Cross Arm GNU create flash image general options.

The following table lists the options in the General panel.

Table 24. General options

Option	Description
Output file format	Defines the object file format.
Section: -j .text	Select to define section: -j .text.
Section: -j .data	Select to define section: -j .data.
Other sections (-j)	Add other sections.
Other flags	Specify additional command line options; type in custom flags that are not otherwise available in the UI.

3.2.9 Cross ARM GNU Create Listing

Use this panel to configure the Cross Arm GNU create listing options.

The following table lists the options in the **Cross ARM GNU Create Listing** panel.

Table 25. Cross ARM GNU Create Listing options

Option	Description
Command	Shows the location of the executable file. Default: `${cross_prefix}${cross_objdump}${cross_suffix}}`
All Options	Shows the actual command line the assembler will be called with. Default: `"FirstProjectTest.elf" --source --all-headers --demangle --line-numbers --wide`
Expert settings	
Command line patterns	Shows the expert settings command line parameters. Default: `${COMMAND} ${FLAGS} ${OUTPUT_FLAG} ${OUTPUT_PREFIX}${OUTPUT} ${INPUTS}`

3.2.9.1 General

Use this panel to configure the Cross Arm GNU create listing general options.

The following table lists the options in the General panel.

Table 26. General options

Option	Description
Display source	Check to display source.
Display all headers	Check to display headers in the listing file; disassembler writes listing headers, titles, and subtitles to the listing file
Demangle names	Check to demangle names.
Display debugging info	Check to display debugging information.
Disassemble	Check to disassembles all section content and sends the output to a file. This command is global and case-sensitive.
Display file headers	Check to display the contents of the overall file header.
Display line numbers	Check to display the line numbers.
Display relocation info	Check to displays the relocation entries in the file.

Table continues on the next page...

Table 26. General options (continued)

Option	Description
Display symbols	Check to display the symbols.
Wide lines	Check to display wide lines.
Other flags	Specify additional command line options for the linker; type in custom flags that are not otherwise available in the UI.

3.2.10 Cross ARM GNU Print Size

Use this panel to configure the Cross Arm GNU print size options.

The following table lists the options in the **Cross ARM GNU Print Size** panel.

Table 27. Cross ARM GNU Print Size options

Option	Description
Command	Shows the location of the executable file. Default: `$${cross_prefix}${cross_size}${cross_suffix}`
All Options	Shows the actual command line the assembler will be called with. Default: `--format=berkeley "FirstProjectTest.elf"`
Expert settings	
Command line patterns	Shows the expert settings command line parameters. Default: `${COMMAND} ${INPUTS} ${FLAGS}`

3.2.10.1 General

Use this panel to configure the Cross Arm GNU print size options.

The following table lists the options in the General panel.

Table 28. General options

Option	Description
Size format	Select size format: Berkeley or SysV
Hex	Select to choose Hex.
Show totals	Select to show totals.
Other flags	Specify additional command line options for the linker; type in custom flags that are not otherwise available in the UI.

Chapter 4
Preparing Target

This chapter describes how to prepare for debugging various target types:

- Preparing hardware targets

4.1 Preparing hardware targets

Please refer to the Hardware Board Getting Started Guide for a description on how to prepare the supported hardware targets.

Chapter 5
Configuring Target

The Target Connection Configuration (TCC) feature lets you configure the probe and the target hardware.

TCC eases out the configuration process due to the auto- discovery capabilities and live validation of the configuration. TCC lets you use one configuration for multiple projects by setting it as the active configuration (configure once debug all projects), but if more than one configuration is required, you can add as many configuration as necessary. TCC can be used as an RCP application for eclipse allowing the user to benefit from the full capabilities either way.

This chapter lists:

5.1 Target Connection configurator overview

You can view all existing configuration, manage configurations, and set the active configuration using the Target Connection View.

To access the Target Connections view, select **Window > Show View > Other > Target Connections**.

The view lists a brief information about the current connection.

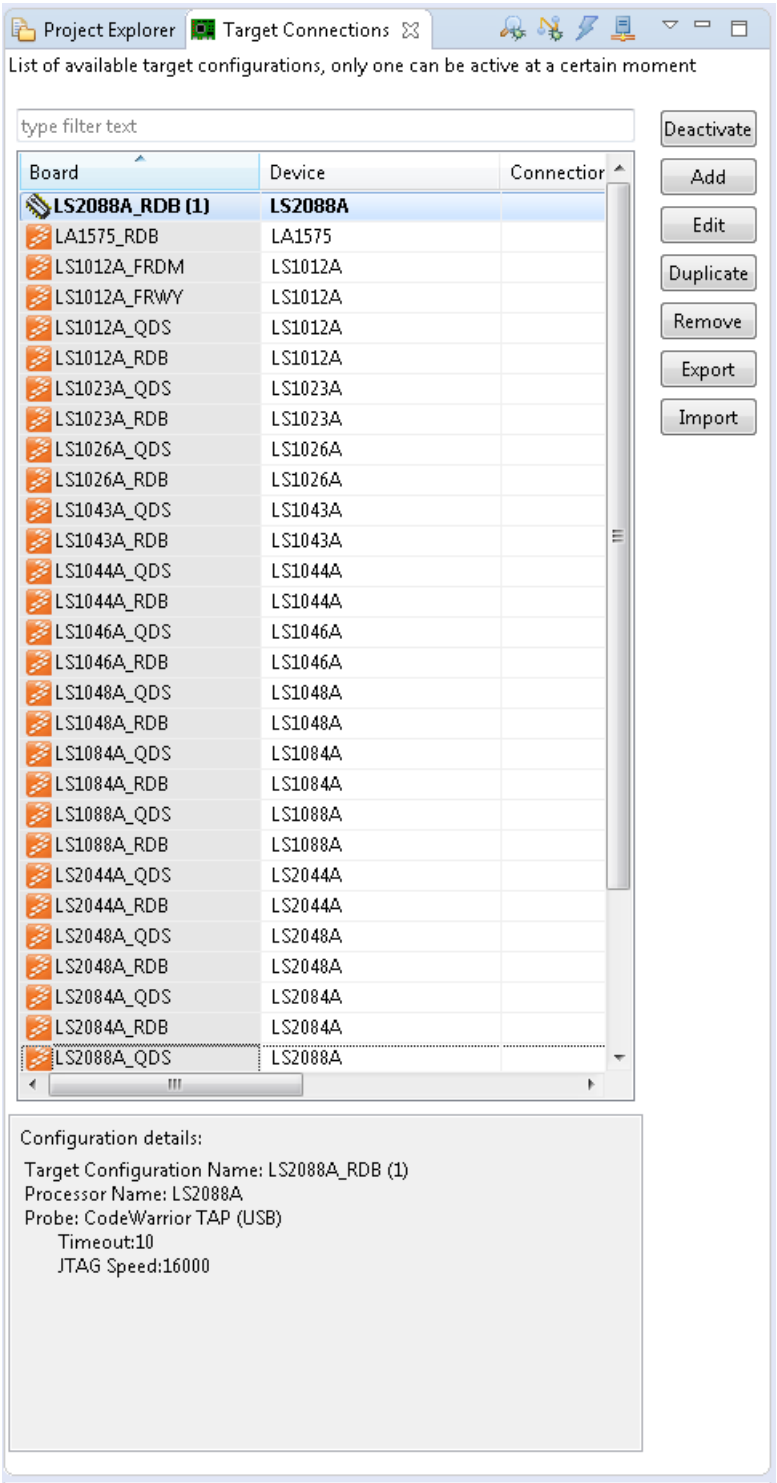


Figure 13. Target Connection view

Target Connections view has the capability to filter the existing configurations. You can filter the list of existing configuration based on board name, device name, or connection details.

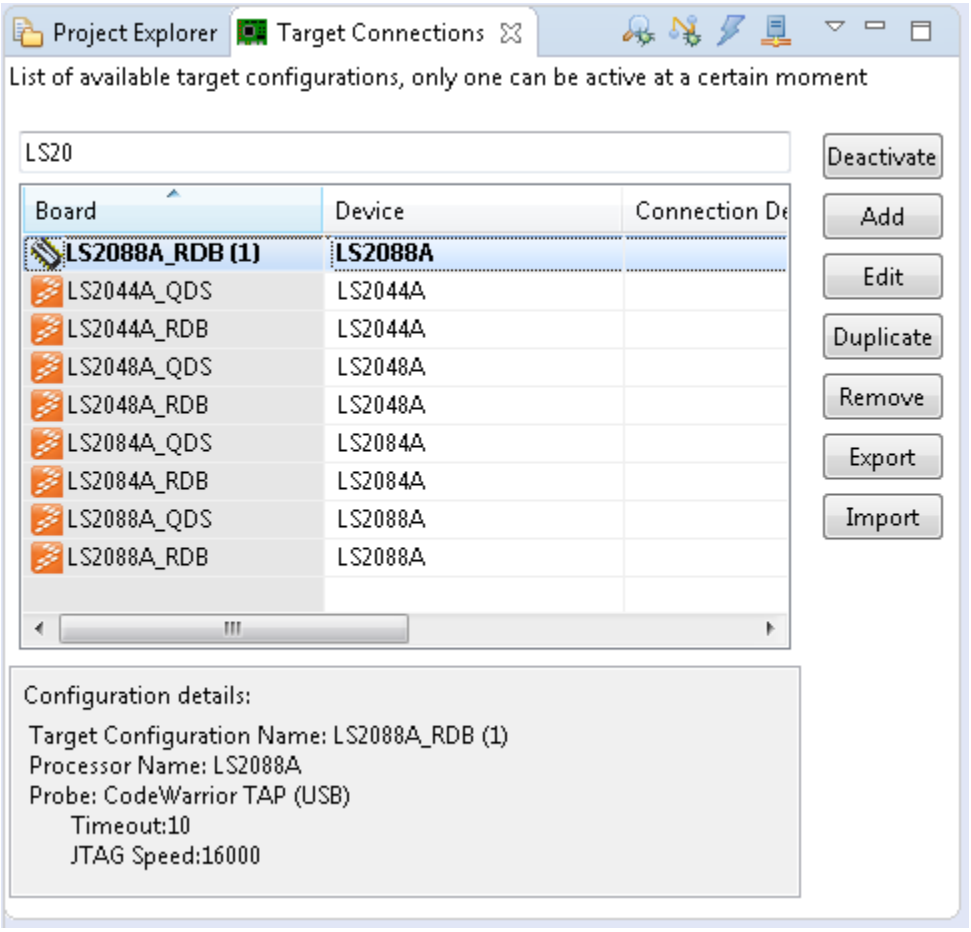


Figure 14. Filter existing target configurations

5.2 Configuration types

There are two type of target connection configuration: user-defined and pre-defined.

The pre-defined configurations are marked with orange icons. Unlike, user-defined configuration, pre-defined configurations cannot be removed. Also, the user doesn't have access to the pre-defined configuration file; therefore the pre-defined configurations cannot be imported or exported.

However, the pre-defined configuration can be duplicated and saved under a different name.

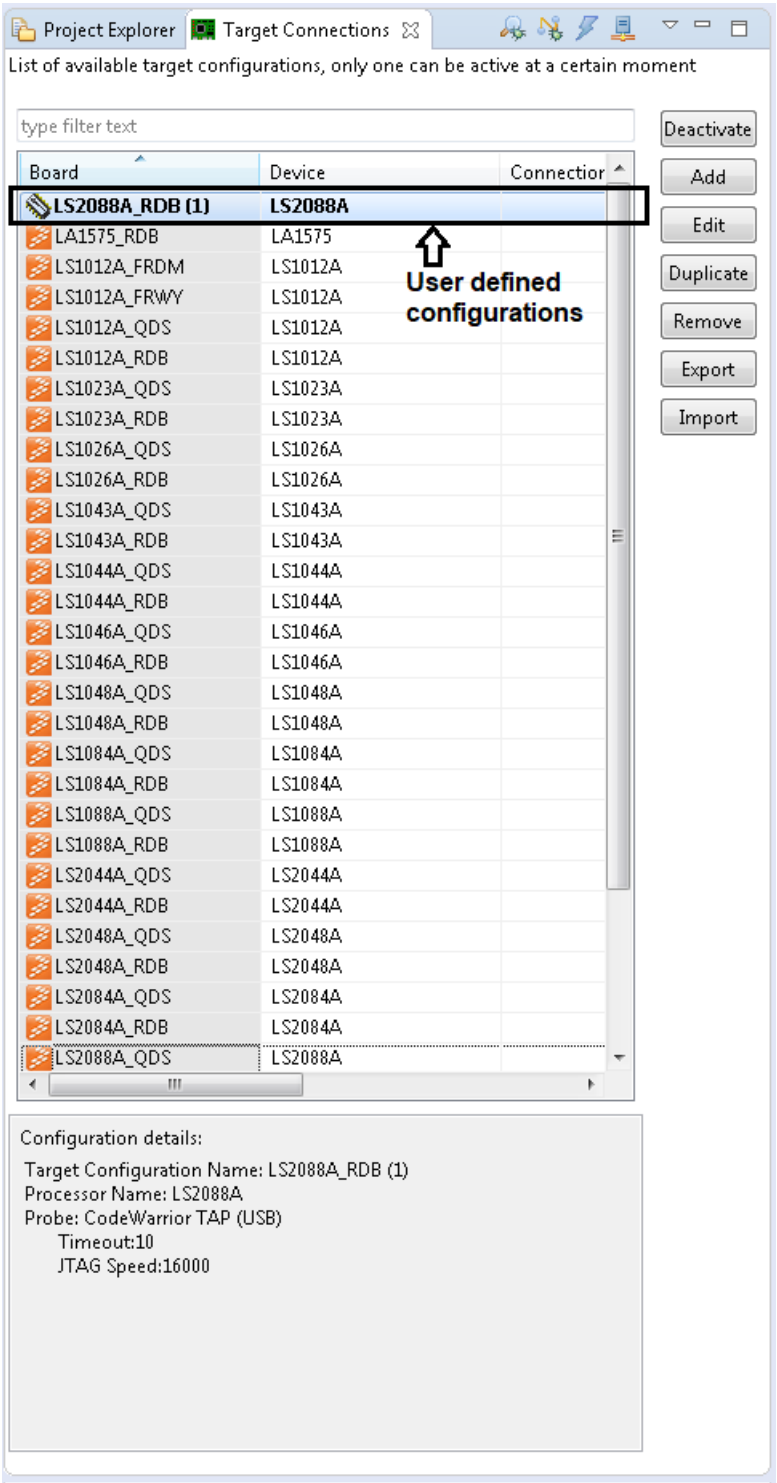


Figure 15. Configuration types

5.3 Operations with configurations

This topic explains the target connection configuration options.

The table below lists the target connection configuration options that you can use to manage configurations, view all existing configuration, and set the active configuration.

Table 29. Target Connections Configurations options

Option	Description
Inspect	Inspect a board without an elf to debug. It attaches to a board without reset, image, or symbols and doesn't run the target initialization file. Inspect command is enabled when a selection exists.
Connect	Connect to a board without an elf to debug. The connect operation is done with reset and running the target initialization file. The connect command is enabled when a selection exists.
Flash Programmer	Start the Flash Programmer UI. Using this button, a new debug session similar to a Connect will be started in the background in order to allow programming of the available flash devices on a board. An existing connection can also be reused.
Diagnose Connection	Use it to perform diagnostics on a selected target connection. When a connection is being diagnosed, a new Connection Diagnostics view will appear, depicting all the tests that are being executed.
Add	Use to create a new configuration.
Edit	Use to edit the selected configuration.
Duplicate	Use to duplicate an existing configuration. You can edit the duplicated configuration.
Remove	Use to remove an existing configuration. Select the configuration you want to delete, and click **Remove**.
Export	Use to export a configuration to the workspace. Select the configuration you want to export. Click **Export**. Select the location in the workspace where you want to export the configuration and click **OK** to finish.
Import	Use to import a configuration from the workspace. Select the configuration you want to import to the internal configuration folder. Click **Import**.
Set Active configuration	Check the checkbox next to the configuration to set it as Active Configuration.
View details about a configuration	TCC panel lets you determine whether a configuration is pre-defined or user-defined by using different color icons; Orange for pre-defined and Green for user-defined. Also, if a configuration is not complete and cannot be used for debug, TCC panel marks it as *(Incomplete)*.

5.4 Configure the target configuration using Target Connection Configurator

To configure the target configuration in Target Connection Configurator, you need to select the debugged processor and the probe.

1. Choose the debugged core from the launch configuration file.
2. In order to connect to the target, select a connection type, such as hardware. And configure the probe options, such as IP, serial number for USB connection.

   You may also choose to apply a reset delay; the value of the delay is specified in milliseconds and it may be needed when the PBL activity takes longer than the default allocated time to complete (2 seconds).

The available probes depend on the selected processor. For example, since there is CMSIS DAP support for LS1012RDB, CMSIS DAP probe is supported additional to CWTAP. In case you select CWTAP, you could use probe discovery capability.

NOTE

In this release, the list of detected CWTAPs also includes the probes connected to other processors in addition to the one selected.

a. Click the **Search for HW probes** button to automatically discover the probes connected to the local machine or network for SoCs that support CWTAP .

NOTE

CMSIS is a vendor-independent hardware abstraction layer provided by Arm. CMSIS Debug Access Port (CMSIS-DAP) is a standardized firmware for a debug unit that connects to the CoreSight Debug Access port.

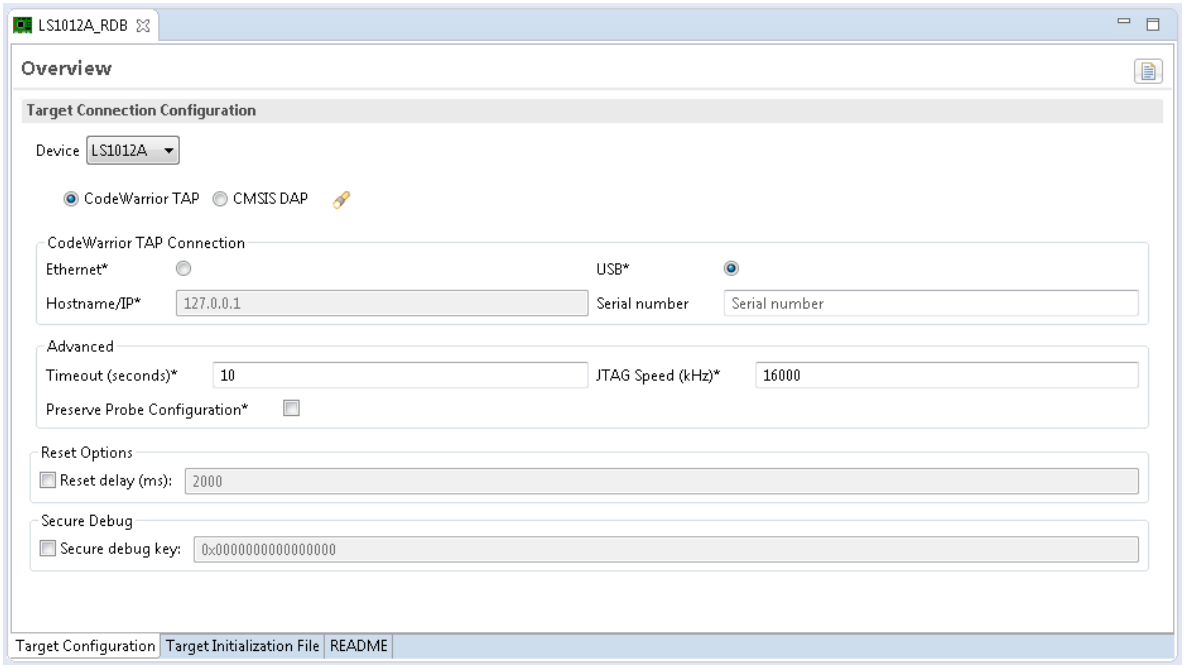


Figure 16. Target Connection Configurator

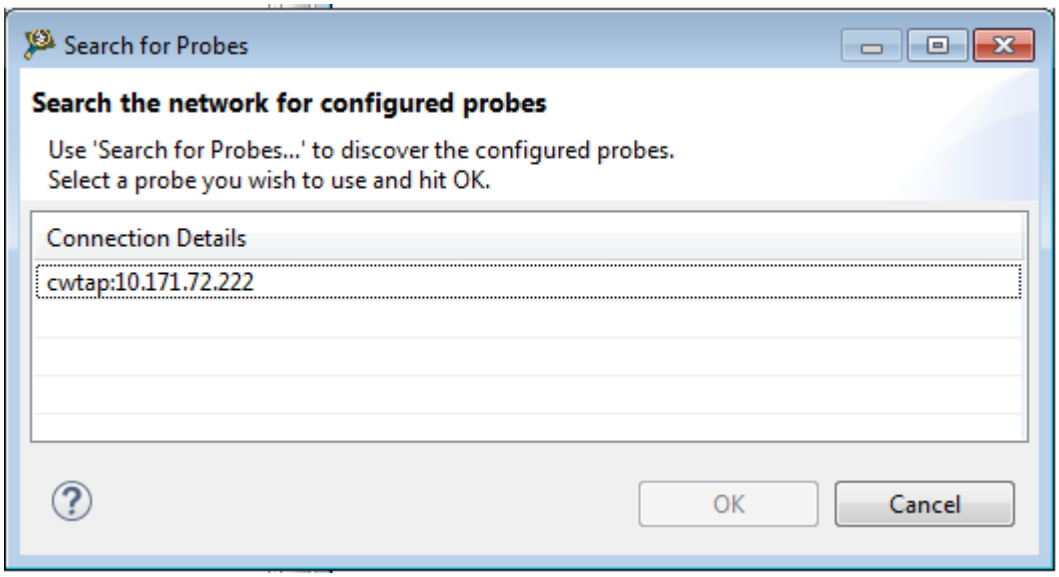


Figure 17. Search for probes

b. When the user selects one of the discovered probes, the target configuration will use the selected CWTAP and the selected probe attributes will be updated accordingly.

3. Select **Preserve Probe Configuration** to make all CWTAP configurations disappear. In this case, you will have to specify only CCS server used to access the CWTAP.
4. Check the **Use Target Init** checkbox to load/edit the gdb file and launch the target init GDB script.
5. You can load/save and edit the gdb script in the **Target Init File** tab page.

   The script is loaded by GDB and it is run automatically before launching a debug session. Initialization script is embedded in the TCC configuration.
6. Click **Apply** to save and set the new target connection as an active configuration. To set an active configuration, select the check box corresponding to the target connection to be set as active configuration. The active configuration acts as source for the target connection data necessary to start a debug session.

NOTE

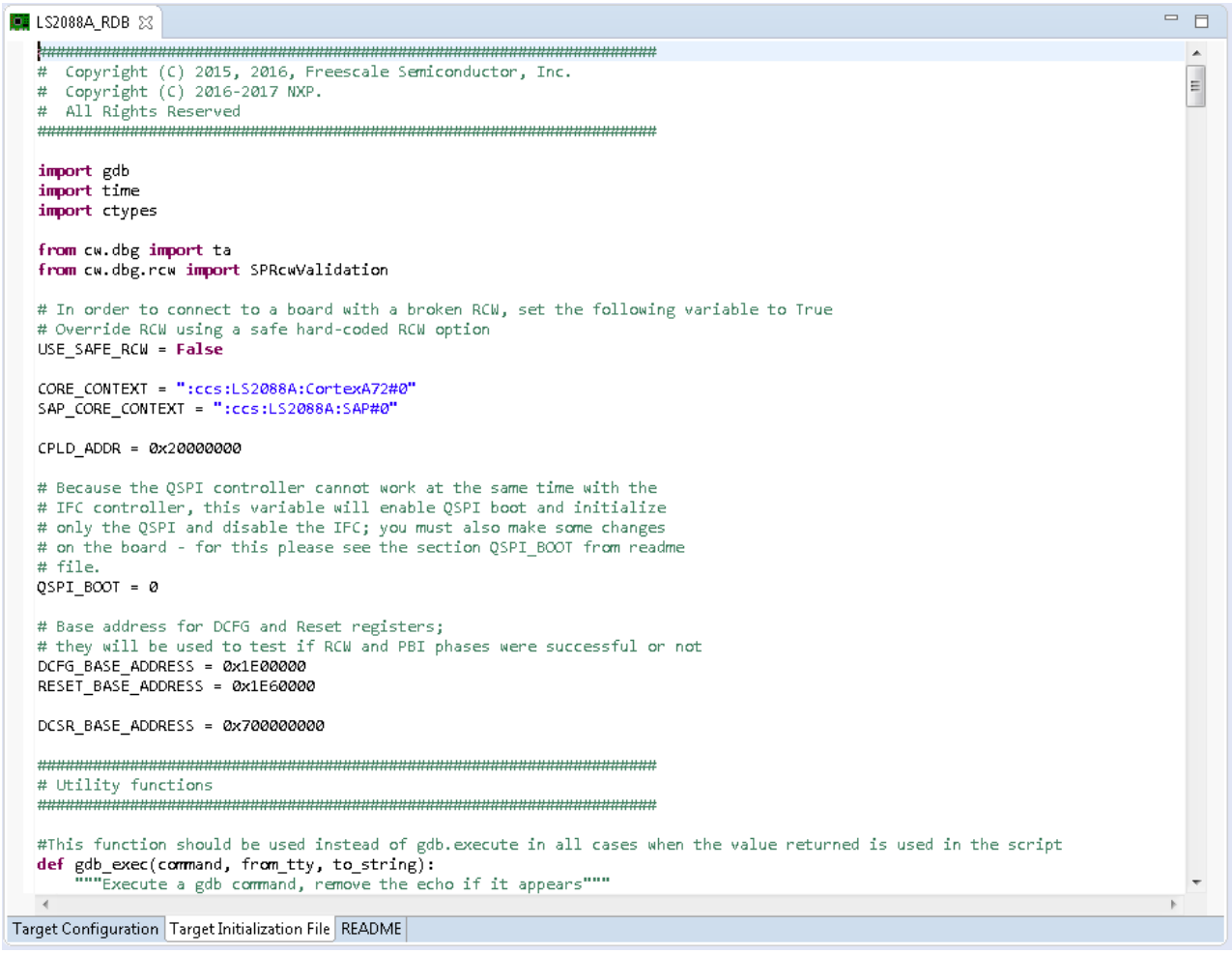
In the **Target Connection** view, the active configuration name is displayed in bold.

5.5 Target Connection editor

To open the Target Connection editor, double-click on a target configuration in the Target Connections view or on a `.tcc` file in a project.

The Target Connection editor consists of three tabs.

- Target Configuration – contains the data about the device and the probe
- Target Initialization File – the initialization file (it's read only for predefined configurations and writeable for user-defined configurations)




```
#################################################################################
#  Copyright (C) 2015, 2016, Freescale Semiconductor, Inc.
#  Copyright (C) 2016-2017 NXP.
#  All Rights Reserved
#################################################################################

import gdb
import time
import ctypes

from cw.dbg import ta
from cw.dbg.rcw import SPRcwValidation

# In order to connect to a board with a broken RCW, set the following variable to True
# Override RCW using a safe hard-coded RCW option
USE_SAFE_RCW = False

CORE_CONTEXT = ":ccs:LS2088A:CortexA72#0"
SAP_CORE_CONTEXT = ":ccs:LS2088A:SAP#0"

CPLD_ADDR = 0x20000000

# Because the QSPI controller cannot work at the same time with the
# IFC controller, this variable will enable QSPI boot and initialize
# only the QSPI and disable the IFC; you must also make some changes
# on the board - for this please see the section QSPI_BOOT from readme
# file.
QSPI_BOOT = 0

# Base address for DCFG and Reset registers;
# they will be used to test if RCW and PBI phases were successful or not
DCFG_BASE_ADDRESS = 0x1E00000
RESET_BASE_ADDRESS = 0x1E60000

DCSR_BASE_ADDRESS = 0x700000000

#################################################################################
# Utility functions
#################################################################################

#This function should be used instead of gdb.execute in all cases when the value returned is used in the script
def gdb_exec(command, from_tty, to_string):
    """Execute a gdb command, remove the echo if it appears"""
```

Figure 18. Target Initialization File

- README – readme information (it's read only for predefined configurations and writeable for user-defined configurations)

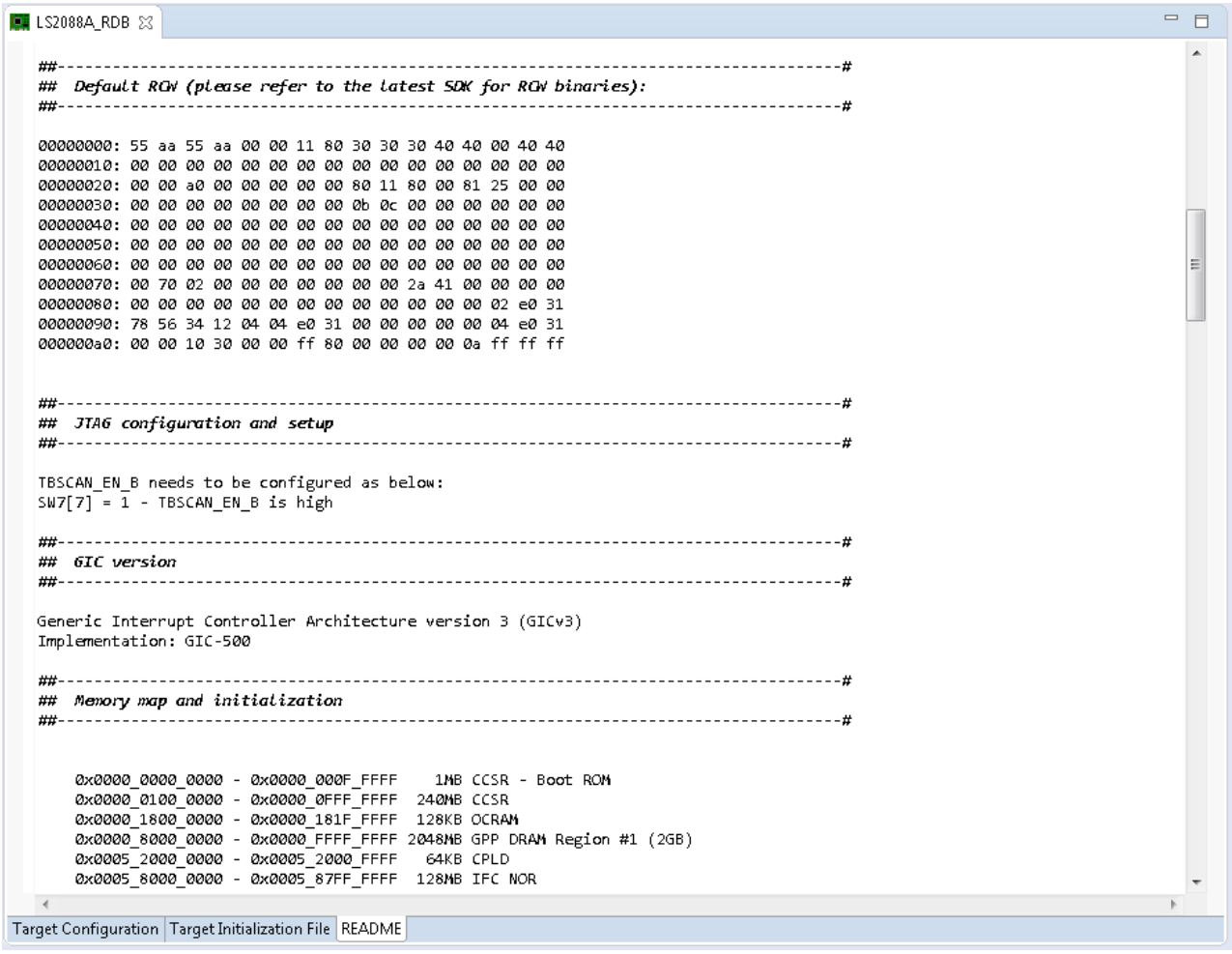


Figure 19. README

5.6 Generating GDB script from a configuration

TCC configures the target by sending a set of commands to the GDB server.

TCC configures the target by sending a set of commands to the GDB server. These commands can be exported and viewed as a `.gdb` script. To export the `.gdb` script:

1. Configure the target configuration using Target Connection Configurator.
2. Click the toolbar command **Generate GDB script**.
3. Select the path where you want to export the `.gdb` script.

   GDB script can be used as it is when starting a debug session from the (GDB) console mode.

5.7 Debugger server connection

Each target connection configuration allows the user to select the type of connection to use with GTA: a local server or a remote connection to an already set up GTA server.

- **Debug Server Connection**
  - **Start local server**: Automatically starts the GTA first time when a certain command is issued to GTA. The GTA will be stopped when the user chooses to use a remote GTA or the CodeWarrior software is closed.

— **Connect to**: User can specify the server address and IP of an already running debug server.

In the **CodeWarrior Connection Server** section the user can specify the connection server parameters.

- **CodeWarrior Connection Server**
  - **Start Local Server**: GTA starts and configures the connection server.
  - **Connect to**: The connection server is already started/configured and the GTA can use it.

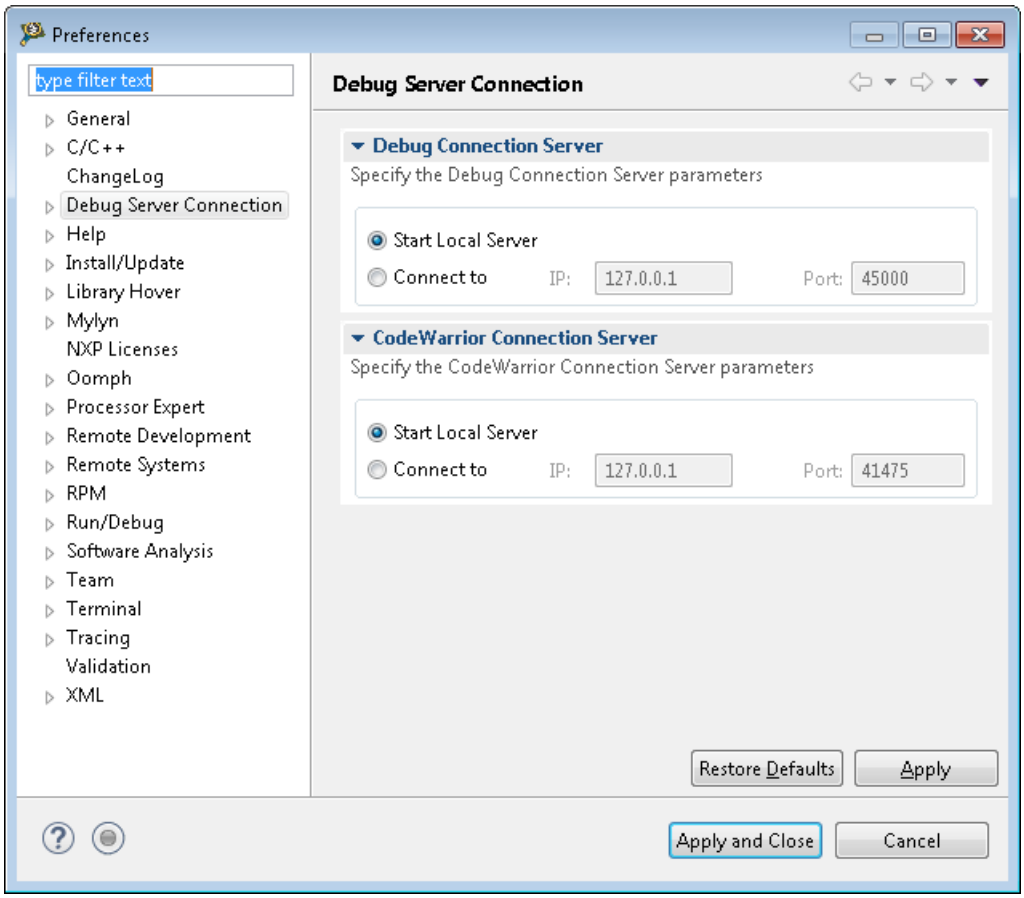


Figure 20. Debug Server Connection

5.8 Logging Configuration

The **Logging Configuration** preference panel can be used to enable the protocol logging (ccs).

Using this panel, the user can configure the logging level and choose the destination for the collected info:

- an Eclipse console, PROTOCOL Logging Console. The console is visible only when **Enable logging to Eclipse console** is selected
- a file

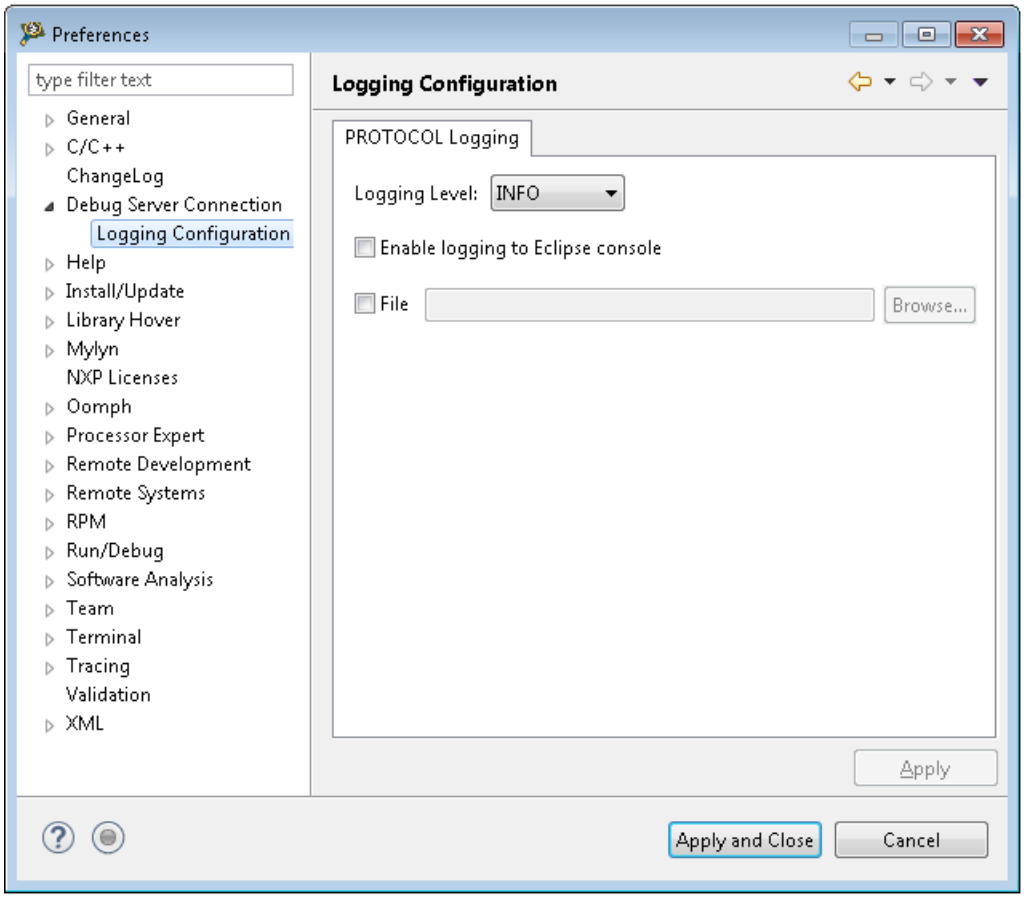


Figure 21. Logging Configuration panel

The INFO level for logging adds more information to the output, for example register IDs, memory addresses, memory spaces. But it does not output the contents for register values, memory, and JTAG chain expansion (for get_config_chain() command).

For details about monitor log commands, refer Logging.

Chapter 6
FSL Debugger References

This chapter explains how to customize debug configurations.

This chapter lists:

6.1 Customizing debug configuration

You can use the **Debug Configurations** dialog to customize various aspects of a debug configuration.

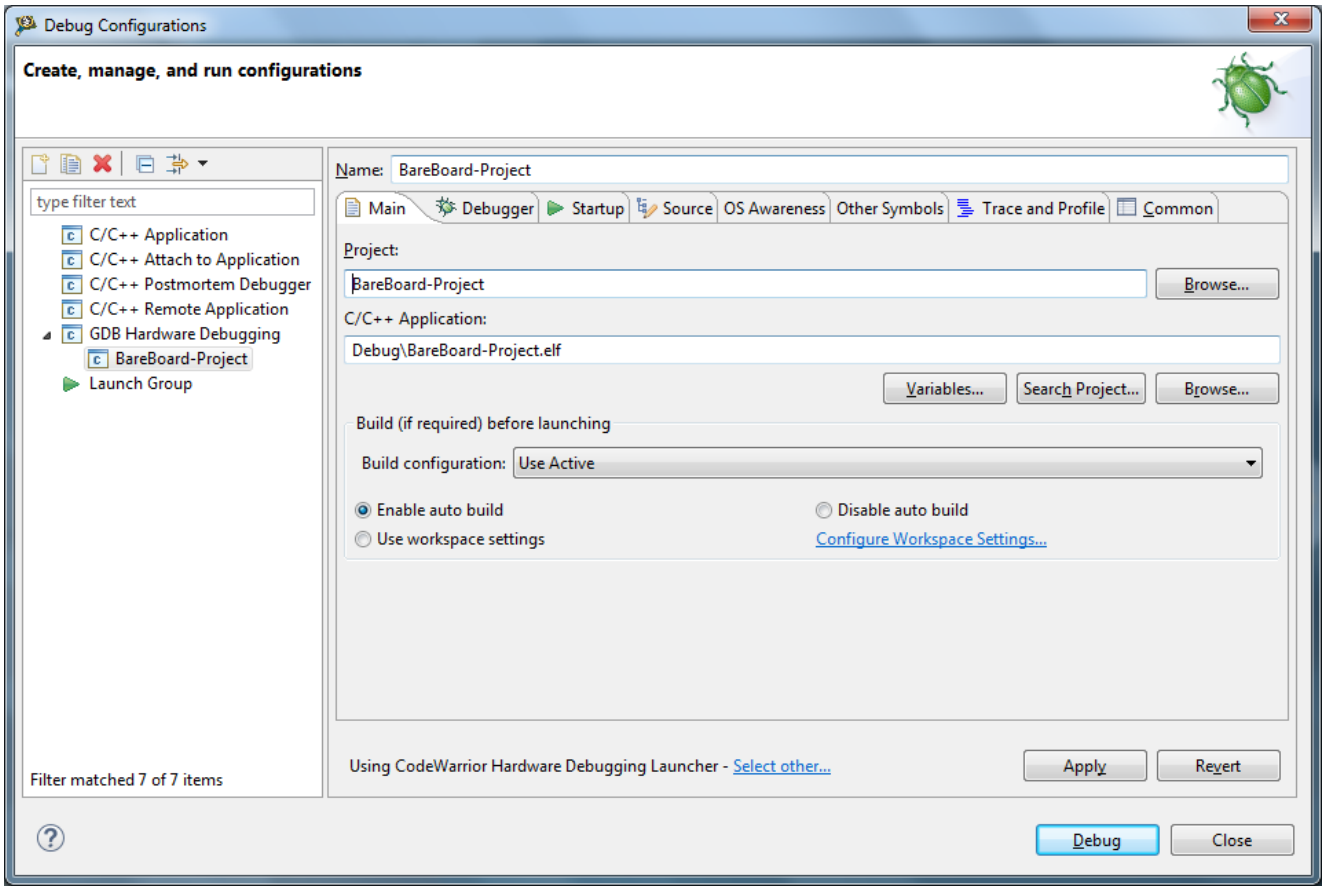


Figure 22. Debug Configurations dialog

NOTE

The CodeWarrior debugger shares some pages, such as Connection and Download. The settings that you specify in these pages also apply to the selected debugger.

To modify a debug configuration:

1. Click **Run > Debug Configurations** in the CodeWarrior IDE.

   The **Debug Configurations** dialog appears.
2. Make the required changes, and click the **Apply** button to save the pending changes.
3. To undo the pending changes, click the **Revert** button.

   The IDE restores the last set of saved settings to all pages of the Debug Configurations dialog. The **Revert** button appears disabled until you make new pending changes.
4. A debug configuration can be saved within the project by setting its location relative to the project loaded in the current workspace. For this, click the Common tab, and in the **Shared file** option, select a project directory where you want to save the debug configuration. Now, you can import the project into another workspace without loosing the debug configuration file.
5. Click the **Close** button to close the **Debug Configurations** dialog.

The tabs in the **Debug Configurations** dialog are:

- Main

- Debugger
- Startup
- Source
- OS Awareness
- Other Symbols
- Common
- Trace and Profile

6.1.1 Main

Use this page to specify the project and the application you want to run or debug.

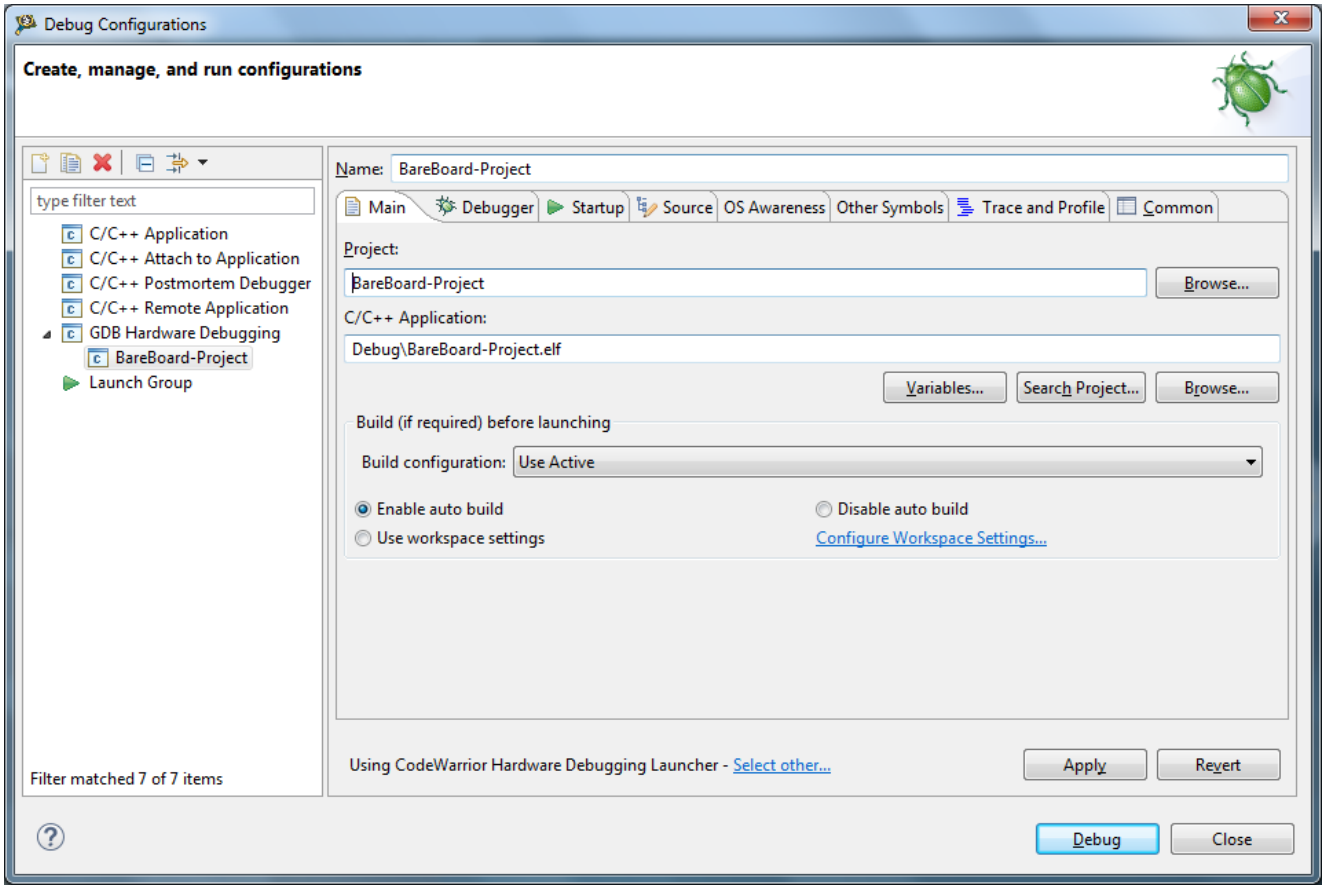


Figure 23. Main tab

The Main tab options are explained in the following table.

Table 30. Main tab options

Option	Description
C/C++ Application	Specifies the name of the C or C++ application.

Table continues on the next page...

Table 30. Main tab options (continued)

Option	Description
Variables	Click to open the Select build variable dialog box and select the build variables to be associated with the program. **Note:** The dialog box displays an aggregation of multiple variable databases and not all these variables are suitable to be used from a build environment.
Search Project	Click to open the **Program Selection** dialog box and select a binary.
Browse	Click **Browse** to select a different C/C++ application.
Project	Specifies the project to associate with the selected debug launch configuration. Click **Browse** to select a different project.
Build (if required) before launching	Controls how auto build is configured for the launch configuration. Changing this setting overrides the global workspace setting and can provide some speed improvements. NOTE: These options are set to default and collapsed when Connect debug session type is selected.
Build configuration	Specifies the build configuration either explicitly or use the current active configuration.
Select configuration using 'C/C++ Application'	Select/clear to enable/disable automatic selection of the configuration to be built, based on the path to the program.
Enable auto build	Enables auto build for the debug configuration which can slow down launch performance.
Disable auto build	Disables auto build for the debug configuration which may improve launch performance. No build action will be performed before starting the debug session. You have to rebuild the project manually.
Use Active (default)	Uses the global auto build settings.
Configure Workspace Settings	Opens the Launching preference panel where you can change the workspace settings. It will affect all projects that do not have project specific settings.

6.1.2 Debugger

Use this page to select a debugger to use when debugging an application.

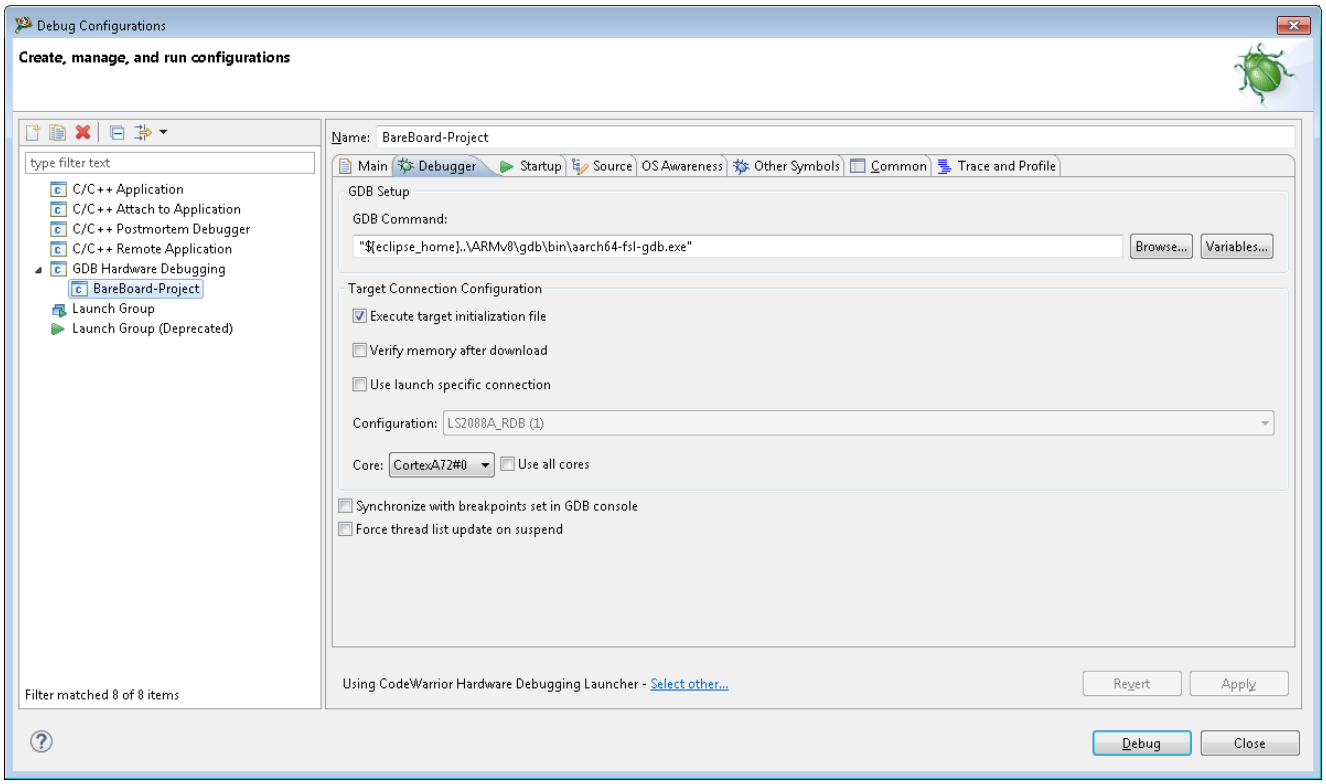


Figure 24. Debugger tab

The Debugger tab options are explained in the following table.

Table 31. Debugger tab options

Option	Description
GDB Setup	
GDB Command	Specifies the GDB command. For example: ${eclipse_home}..\ARMv8\gdb\bin\aarch64-fsl-gdb.exe.
Browse	Click to navigate.
Variables	Click to select variables.
Target Connection Configuration	
Verify memory download	If selected, download validation is performed after binary is downloaded to target. The console displays the validation result in an output similar to the one presented below. *Section .note.gnu.build-id, range 0x400000 -- 0x400024: matched.* Section .text, range 0x400040 -- 0x400568: matched. Section .rodata, range 0x400568 -- 0x400578: matched. *Section .data, range 0x410578 -- 0x410cd0: matched.* If checkbox is deselected, validation is not performed and the above output is not displayed.

Table continues on the next page...

Table 31. Debugger tab options (continued)

Option	Description

	In GDB command line, a user can execute the `compare-sections` command after the executable is loaded to target (using the `load` command), and same output will be displayed. A typical GDB session with download validation is presented in the example below.
	`target extended-remote host:port`
	`mon ctx set current :ccs:LS2080A:A57#0`
	`attach 1`
	`load elf_file`
	`file elf_file`
	`compare-sections`

Use launch specific connection	Select to specify the target connection configuration in this launch. This will override the configuration specified globally in **Window->Preferences** dialog.
Configuration	Enabled when Use launch specific connection is checked. Use to select the required configuration.
Core	Select the core to debug.
Use all cores	Select if your application uses all cores (SMP).
Synchronize with breakpoints set in GDB console	When activated, all breakpoints set from the GDB console are synchronized with the CodeWarrior UI. Breakpoints created in the GDB console appear in the **Breakpoints** view and preserved from the current Debug session to the next.
Force thread list update on suspend	Click if you want to force thread list update on suspend.

NOTE

When trying read I/O operations (scanf, fget etc.) using semihosting, you have to check **Use separate console for target output and input**, otherwise, if you check the other option from **Application Console**, **Use GDB console for target output**, the read I/O operations will be unreliable.

6.1.3 Startup

Use this page to specify the startup options and values to use when an application runs.

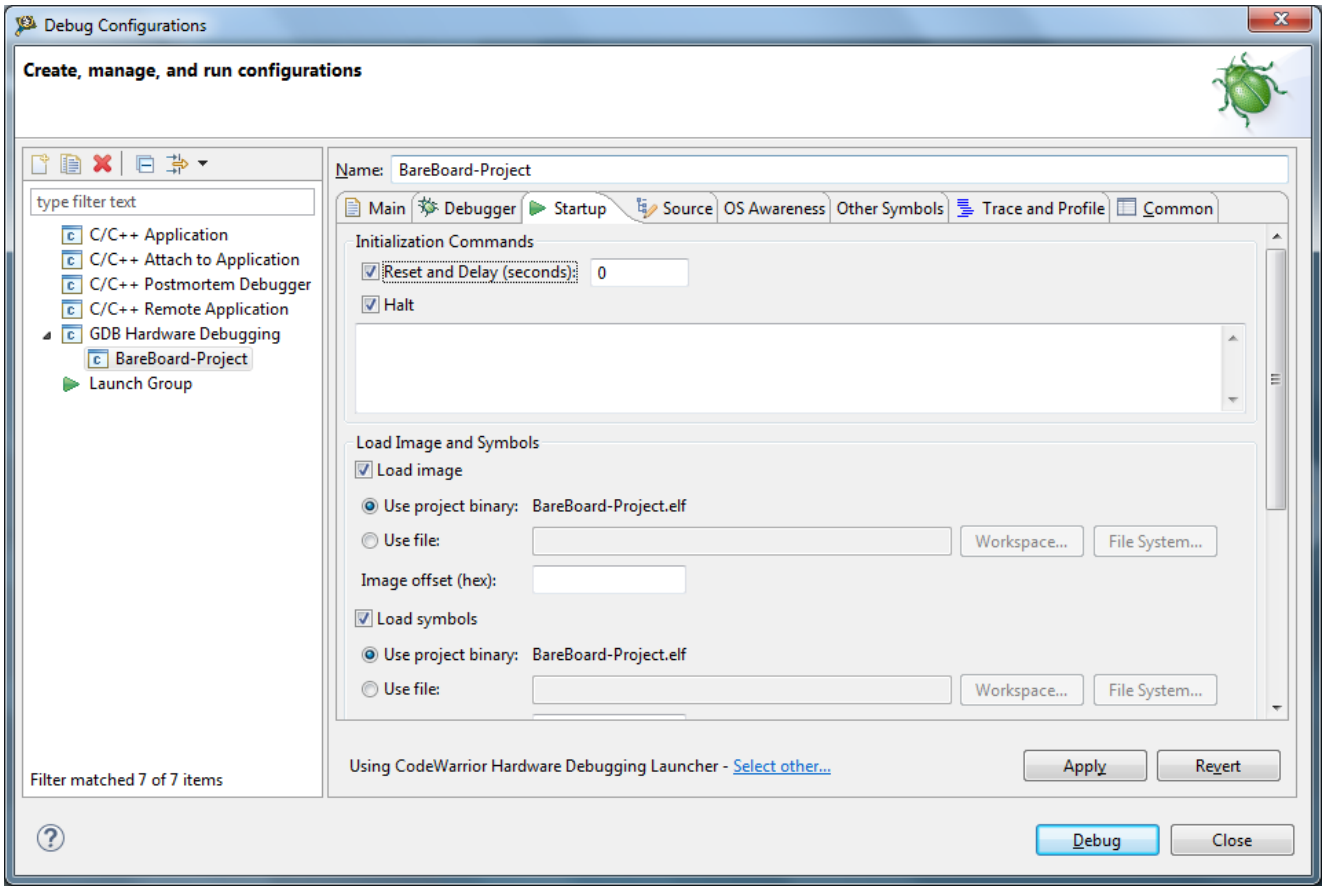


Figure 25. Startup tab

The following table list the **Startup** tab options.

Table 32. Startup tab options

Option	Description
Reset and Delay (seconds)	Select to reset the target at startup and delay the initialization for the specified amount of seconds
Halt	Select to halt the target at startup
Load image	Select to specify that an image should be loaded to the target
Use project binary	Select to load the binary of the current project
Use file	Select to load a different file
Workspace	Click to select a file to load from the workspace
File System	Click to select a file to load from the file system
Image offset (hex)	Specify the offset on the target from where to load the image
Load symbols	Select to specify that symbols should be loaded in the debugger
Use project binary	Select to load symbols from the binary of the current project
Use file	Select to load symbols from a different file

Table continues on the next page...

Table 32. Startup tab options (continued)

Option	Description
Workspace	Click to select a file with symbols to load from the workspace
File System	Click to select a file with symbols to load from the file system
Symbol offset (hex)	Specify an offset for the symbols
Set program counter at (hex)	Select to set the PC at startup to a specified value
Set breakpoint at	Select to set a breakpoint at a specified location
Resume	Select to indicate the execution should resume
Run commands	Specify commands to be run in the debugger after loading image / symbols

6.1.4 Source

Use this page to specify the location of source files used when debugging a C or C++ application.

By default, this information is taken from the build path of your project.

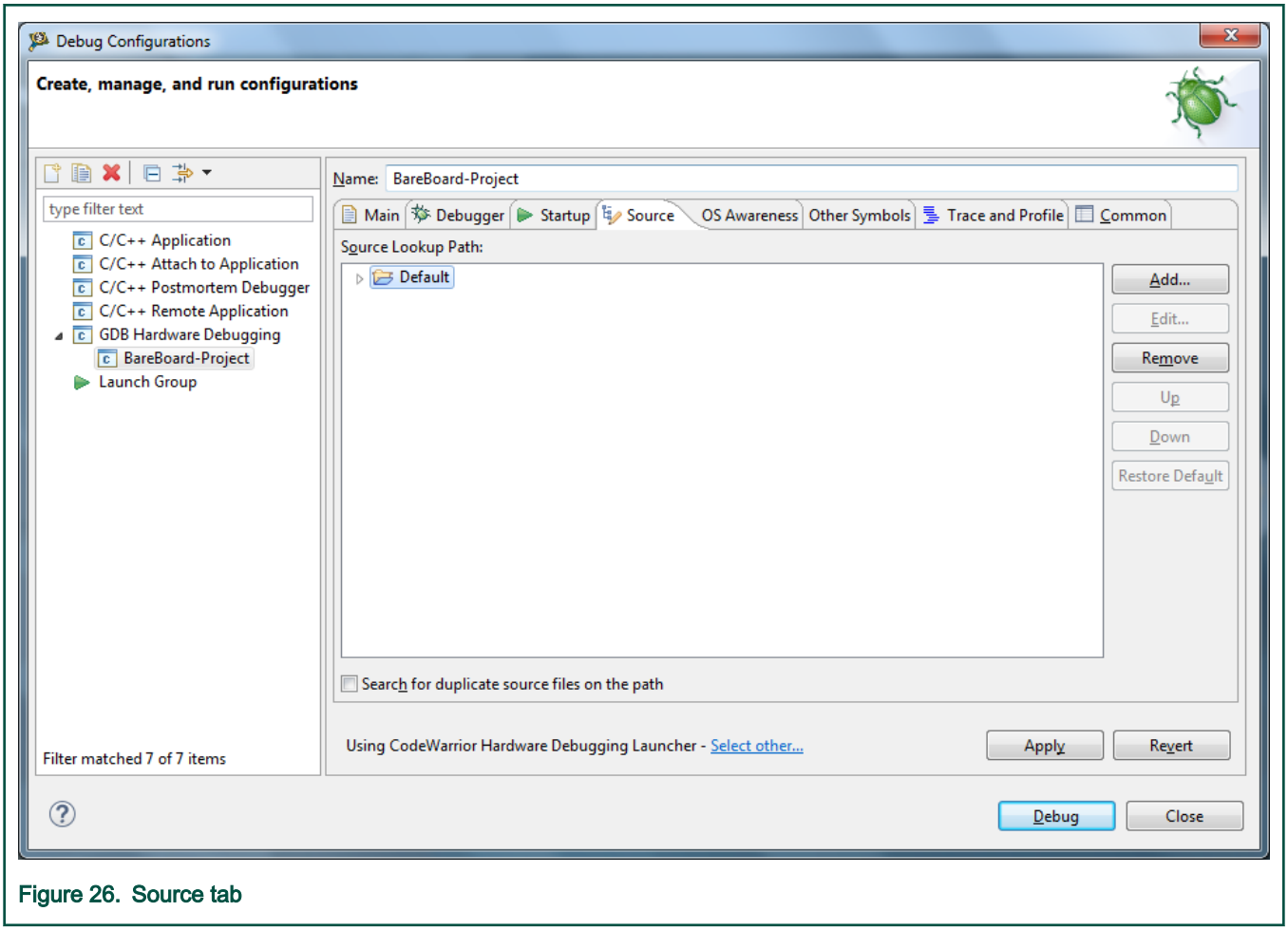


Figure 26. Source tab

The Source tab options are explained in the following table.

Table 33. Source tab options

Option	Description
Source Lookup Path	Lists the source paths used to load an image after connecting the debugger to the target.
Add	Click to add new source containers to the Source Lookup Path search list.
Edit	Click to modify the content of the selected source container.
Remove	Click to remove selected items from the **Source Lookup Path** list.
Up	Click to move selected items up the **Source Lookup Path** list.
Down	Click to move selected items down the **Source Lookup Path** list.
Restore Default	Click to restore the default source search list.
Search for duplicate source files on the path	Select to search for files with the same name on a selected path.

6.1.5 OS Awareness

Use this page to specify whether the OS Awareness should be enabled.

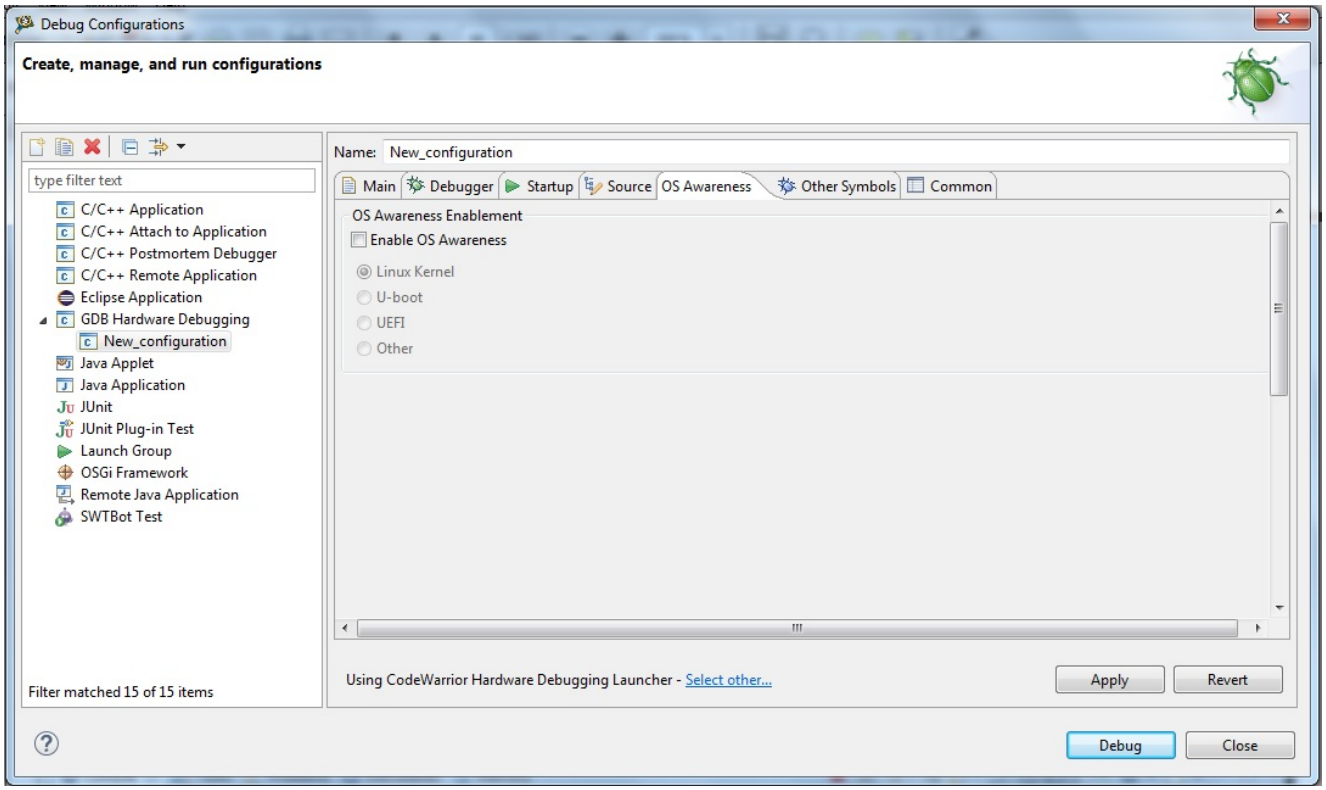


Figure 27. OS Awareness tab

The following table list the **OS Awareness** tab options.

Table 34. OS Awareness tab options

Option	Description
Enable OS Awareness	Select to enable OS Awareness (and activate the other tab options).
Linux Kernel	Select to enable OS awareness for Linux Kernel.
U-boot	Select to enable OS awareness for U-Boot.
UEFI	Select to enable OS Awareness for UEFI.
Other	Select to enable user-defined types of OS awareness.
Use CodeWarrior script for Linux Kernel Awareness	Select to enable OS Awareness for Linux Kernel using CodeWarrior specific script.
Use CodeWarrior script for U-boot Awareness	Select to enable OS Awareness for U-boot using CodeWarrior specific script.
Use CodeWarrior script for UEFI Awareness	Select to enable OS Awareness for UEFI using CodeWarrior specific script.
Use script	Select to specify a custom script to enable OS Awareness.
Add SPL U-Boot ELF	Select to specify an SPL U-Boot ELF in order to debug an SPL-based U-Boot (e.g. NAND/SD type). By using this option, debugger automatically activates all the required processing needed to debug the SPL part followed by the main/DDR U-Boot elf debug.
Workspace	Click to select a custom script from the workspace.
File System	Click to select a custom script from the file system.
Suspend target when module insert or removal is detected	Select to suspend target when module insert or removal is detected.
Automatically load configured symbolic file at module init detection	Select to automatically load symbolic files.
Auto-load module symbolics files list	Lists automatically loaded symbolic files.
UEFI root layout	Select the Build folder inside the local path to the UEFI root layout
Add symbols for EFI images loaded at runtime	Select to automatically add symbols at attach for EFI images loaded at runtime

6.1.6 Other Symbols

Use this page to specify other symbols settings.

The **Other Symbols** tab allows reading additional symbol table information from one or more `elf` files given by the user.

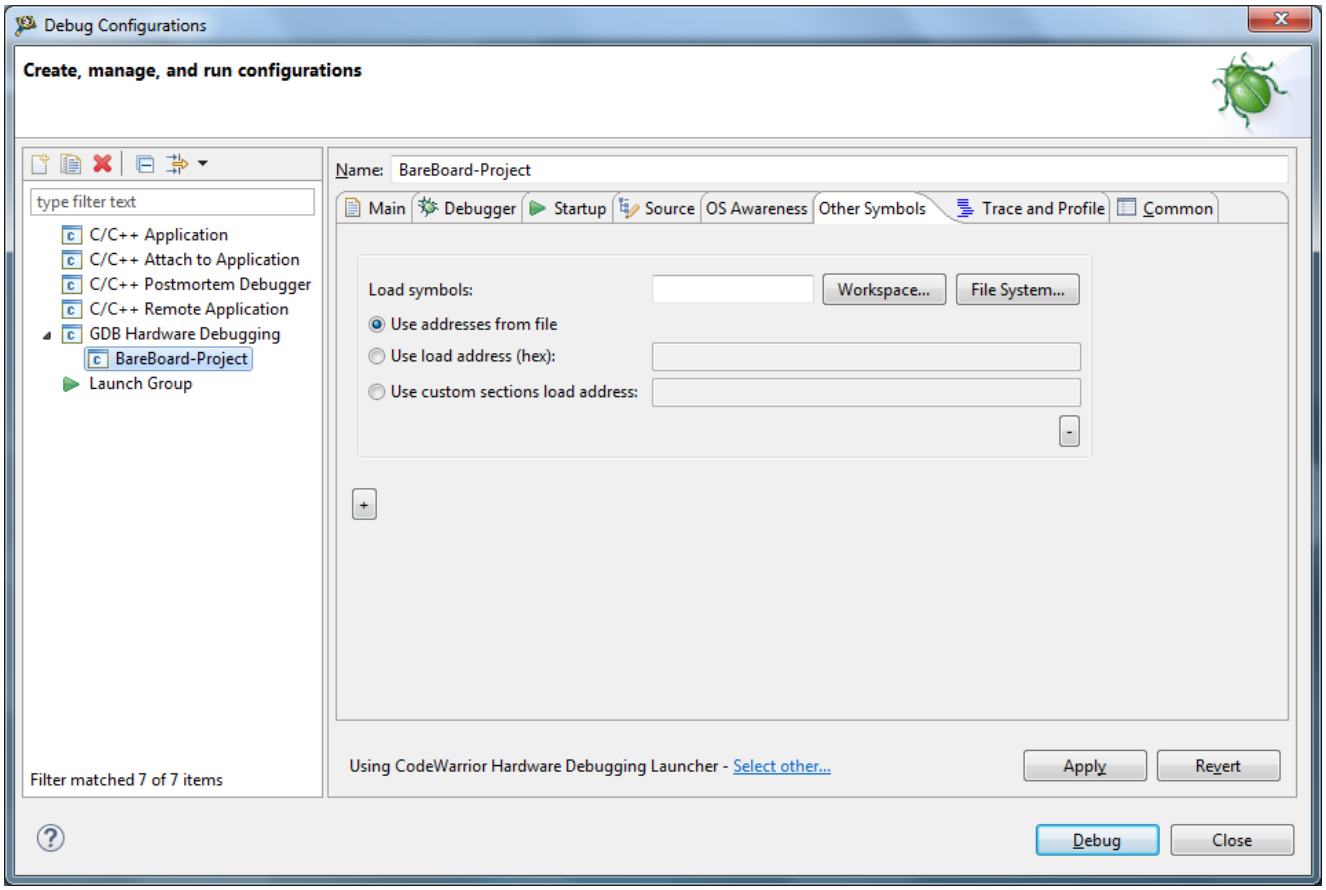


Figure 28. Other Symbols tab

The symbol table information is read by using the `add-symbols` command; this command is similar to the GDB `add-symbol-file` command. However, unlike the `add-symbol-file` command, the `add-symbols` doesn't require the user to provide the load address for the file. The symbols from the `elf` file are loaded using the compile-time addresses for all loadable sections in case this address is not given as a parameter.

If an address is given as a parameter, then the `add-symbols` command loads symbols for all loadable sections based on the specified memory load address. Similar to the GDB command `add-symbol-file`, the `add-symbols` may load symbols for only specific sections at the given load addresses. In order to add symbols from more than one `elf` file, you only need to add a new **Load Symbols** group specifying the new `elf` file and the load options. To remove an `elf` file, press the **Remove** button corresponding to the **Load Symbols** group you want to eliminate.

Option	Description
Load Symbols	Choose the `elf` ffile you want to use from either the file system or the workspace.
Use addresses from file	Select to load the symbols from the `elf` file using compile-time addresses for all loadable sections.
Use load address (hex)	Select to load symbols for all loadable sections based on the specified memory load address. The input from the user is a hex address, without the `0x` prefix and it represents the load address in target memory (address of first loadable section).

Table continues on the next page...

Table continued from the previous page...

Use custom sections load address	Select to load symbols for explicitly provided sections at the specified load addresses. Here the user must specify the section and the load address. Example: `-s .text 0x80000000`

6.1.7 Common

Use this page to specify the location to store your run configuration, standard input and output, and background launch options.

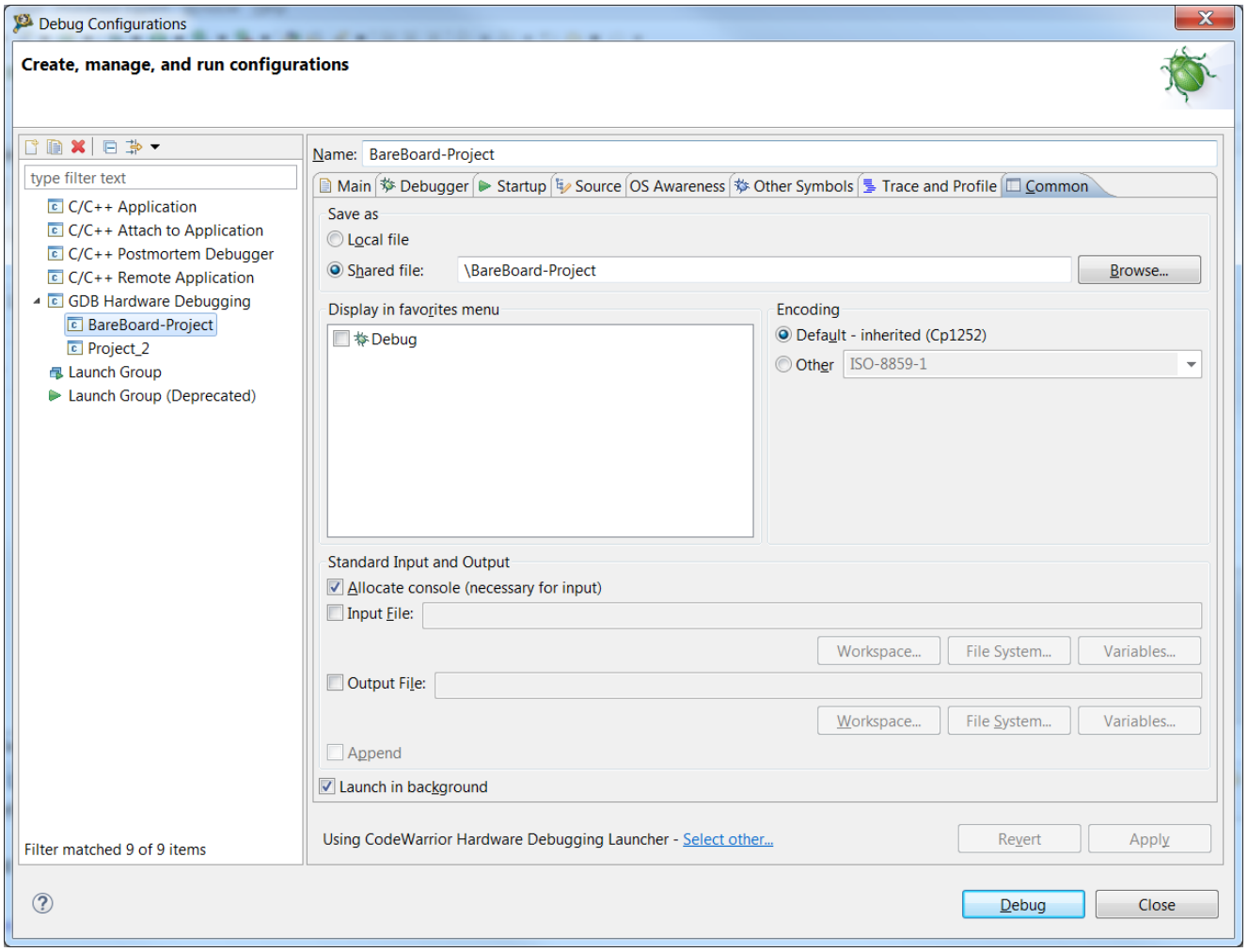


Figure 29. Common tab

The following table lists and explains the **Common** tab options.

Table 35. Common tab options

Option	Description
Save as	
Local file	Select to save the launch configuration locally.

Table continues on the next page...

Table 35. Common tab options (continued)

Option	Description
Shared file	Select to specifies the path of, or browse to, a workspace to store the launch configuration file, and be able to commit it to a repository.
Display in favorites menu	Check to add the configuration name to Run or Debug menus for easy selection.
Encoding	Select an encoding scheme to use for console output.
Standard Input and Output	
Allocate Console (necessary for input)	Select to assign a console view to receive the output.
Input File	Specify the file name to save input.
Output File	Specify the file name to save output.
Browse Workspace	Specifies the path of, or browse to, a workspace to store the output file.
Browse File System	Specifies the path of, or browse to, a file system directory to store the output file.
Variables	Select variables by name to include in the output file.
Append	Check to append output. Uncheck to recreate file each time.
Port	Check to redirect standard output (`stdout`, `stderr`) of a process being debugged to a user specified socket. **Note:** You can also use the `redirect` command in debugger shell to redirect standard output streams to a socket.
Act as Server	Select to redirect the output from the current process to a local server socket bound to the specified port.
Hostname/IP Address	Select to redirect the output from the current process to a server socket located on the specified host and bound to the specified port. The debugger will connect and write to this server socket via a client socket created on an ephemeral port
Launch in background	Check to launch configuration in background mode.

6.1.8 Trace and Profile

Use this page to specify trace and profile settings.

For any new project, go to **Debug configuration**, select a launch configuration from **GDB Hardware Debugging** from the left panel and select the **Trace and Profile** tab from the right panel.

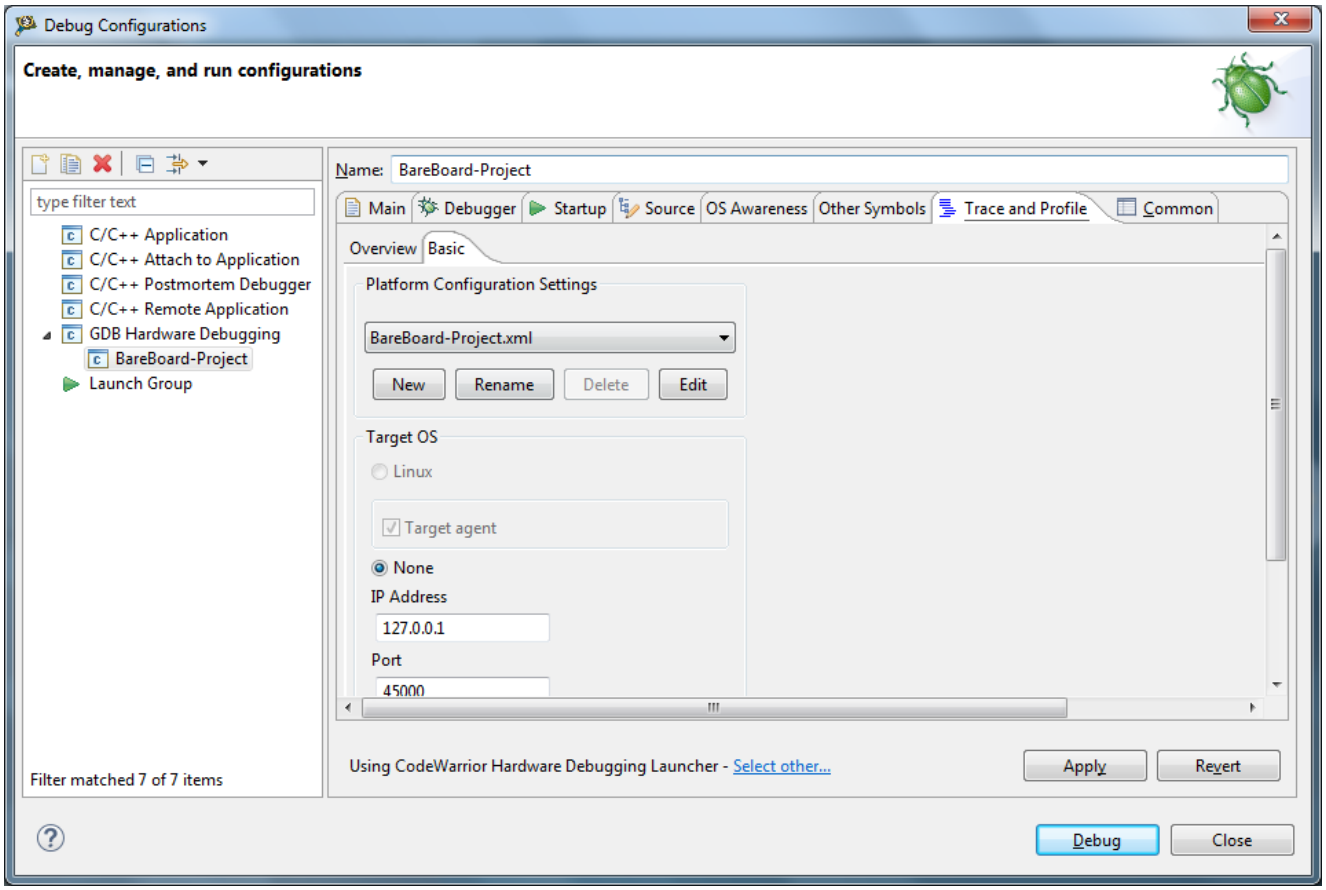


Figure 30. Trace and Profile tab

6.2 Registers features

This topic explains Peripherals view and GDB customer register commands.

This section lists:

- Peripherals view
- GDB custom register commands

6.2.1 Peripherals view

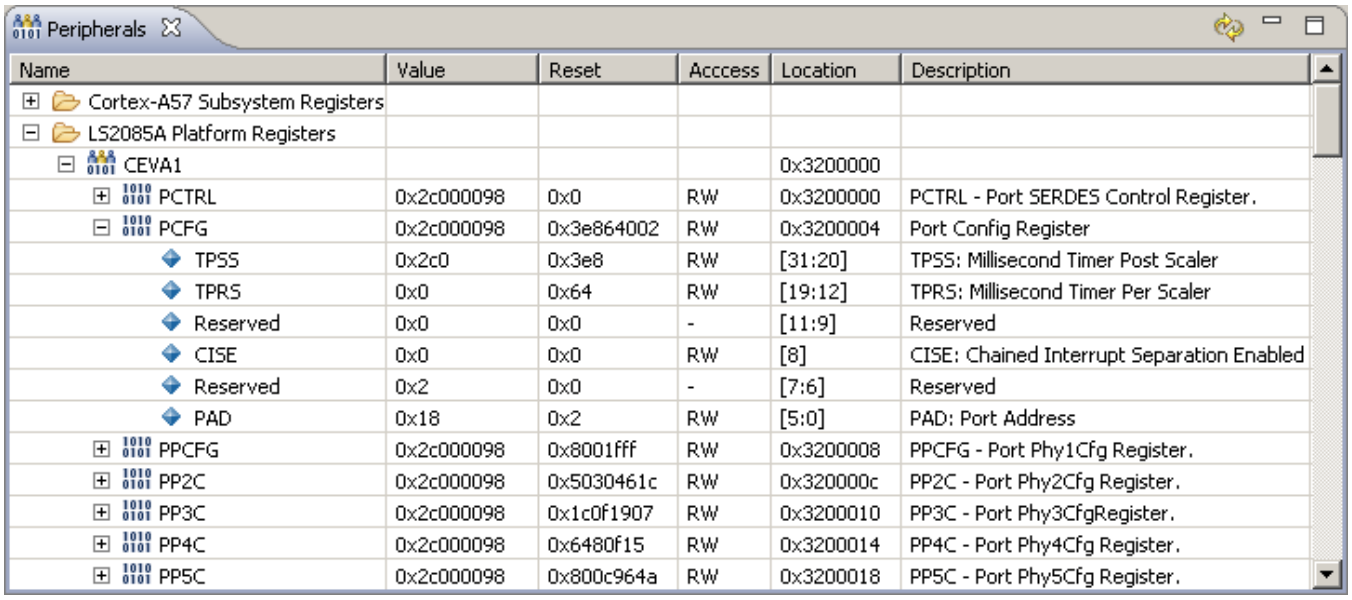
The Peripherals view lists information about the processor system and platform ip-blocks organized in the form of register groups and memory mapped register groups.

The registers are displayed in a tree view with three columns:

- Name – the name of the register or group
- Value – the value from of the register read from target
- Location – the address of the register or the address of the first register for groups (applicable only for platform groups - MMR).
- Access – the access mode: R=read-only, RW=read-write, W=write-only
- Reset – the register reset value
- Description – the register description

Register and bit field values can be modified on target by clicking in the value cell and entering a new value.

The view is automatically opened when a debug session is started and it is populated with registers when the target is first suspended at program entry point. The view can also be opened manually from the menu: **Window > ShowView> Other > Peripherals** or by using the shortcut: **Alt+Shift+Q, R**.



Name	Value	Reset	Acccess	Location	Description
Cortex-A57 Subsystem Registers					
LS2085A Platform Registers					
CEVA1				0x3200000	
PCTRL	0x2c000098	0x0	RW	0x3200000	PCTRL - Port SERDES Control Register.
PCFG	0x2c000098	0x3e864002	RW	0x3200004	Port Config Register
TPSS	0x2c0	0x3e8	RW	[31:20]	TPSS: Millisecond Timer Post Scaler
TPRS	0x0	0x64	RW	[19:12]	TPRS: Millisecond Timer Per Scaler
Reserved	0x0	0x0	-	[11:9]	Reserved
CISE	0x0	0x0	RW	[8]	CISE: Chained Interrupt Separation Enabled
Reserved	0x2	0x0	-	[7:6]	Reserved
PAD	0x18	0x2	RW	[5:0]	PAD: Port Address
PPCFG	0x2c000098	0x8001fff	RW	0x3200008	PPCFG - Port Phy1Cfg Register.
PP2C	0x2c000098	0x5030461c	RW	0x320000c	PP2C - Port Phy2Cfg Register.
PP3C	0x2c000098	0x1c0f1907	RW	0x3200010	PP3C - Port Phy3CfgRegister.
PP4C	0x2c000098	0x6480f15	RW	0x3200014	PP4C - Port Phy4Cfg Register.
PP5C	0x2c000098	0x800c964a	RW	0x3200018	PP5C - Port Phy5Cfg Register.

6.2.2 GDB custom register commands

There are several GDB commands for manipulating system and platform registers. The commands are querying into an SQLite DB associated with the target that is currently debugged in order to fetch register information based on its name.

6.2.2.1 reg_write command

Write register value.

Usage

```
reg_write [context] REG_GROUP.REG_NAME VALUE
```

Table 36. Mandatory arguments

Name	Address
REG_GROUP.REG_NAME	REG_GROUP is the name of the register group or IP block. REG_NAME is the name of the register from REG_GROUP.
VALUE	Value to be written

Table 37. Optional arguments

Name	Address
context	GTA (GDB server) debug context :ccs:<soc>[:core] e.g. :ccs:LS2088A or :ccs:LS2088A:CortexA72#0. If present, the context can be a core context; if not, the current context is used.

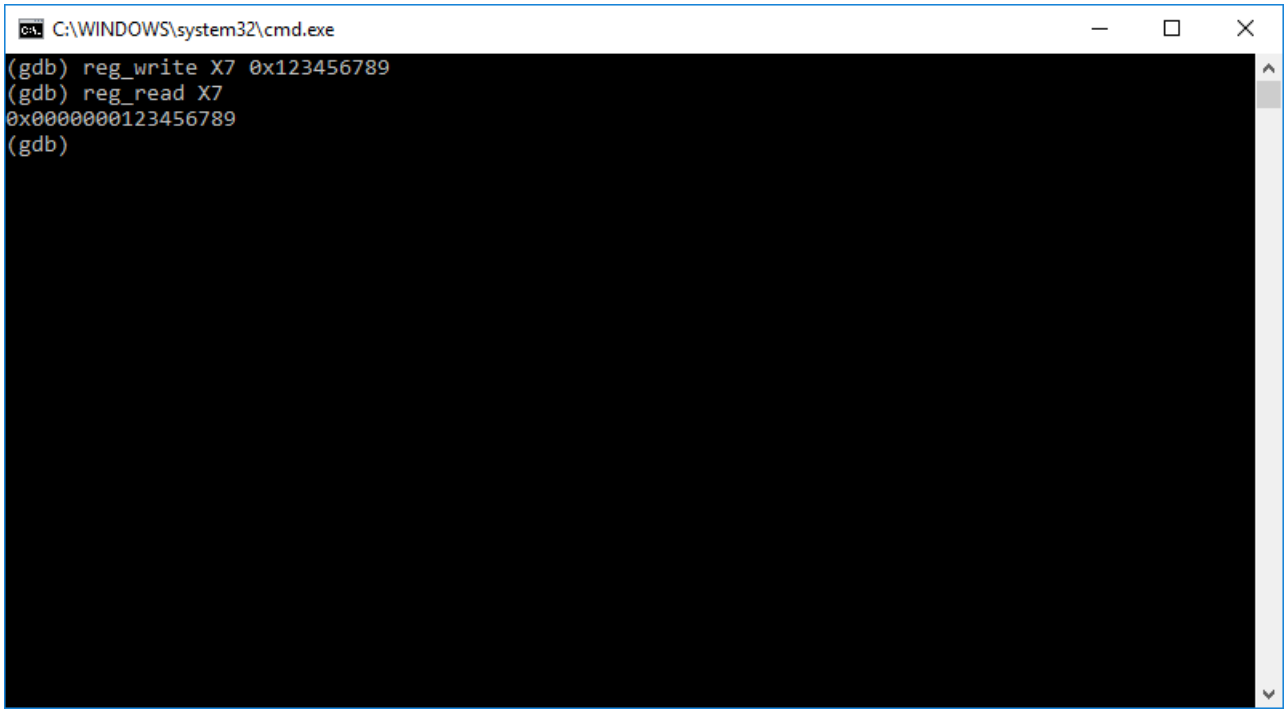


Figure 31. Writing a register using `reg_write` command

6.2.2.2 reg_read command

Read register value.

Usage

```
reg_read [context] REG_GROUP.REG_NAME
```

Table 38. Mandatory arguments

Name	Address
REG_GROUP.REG_NAME	REG_GROUP is the name of the register group or IP block. REG_NAME is the name of the register from REG_GROUP.

Table 39. Optional arguments

Name	Address
context	GTA (GDB server) debug context :ccs:<soc>[:core] e.g. :ccs:LS2088A or :ccs:LS2088A:CortexA72#0. If present, the context can be a core context; if not, the current context is used.

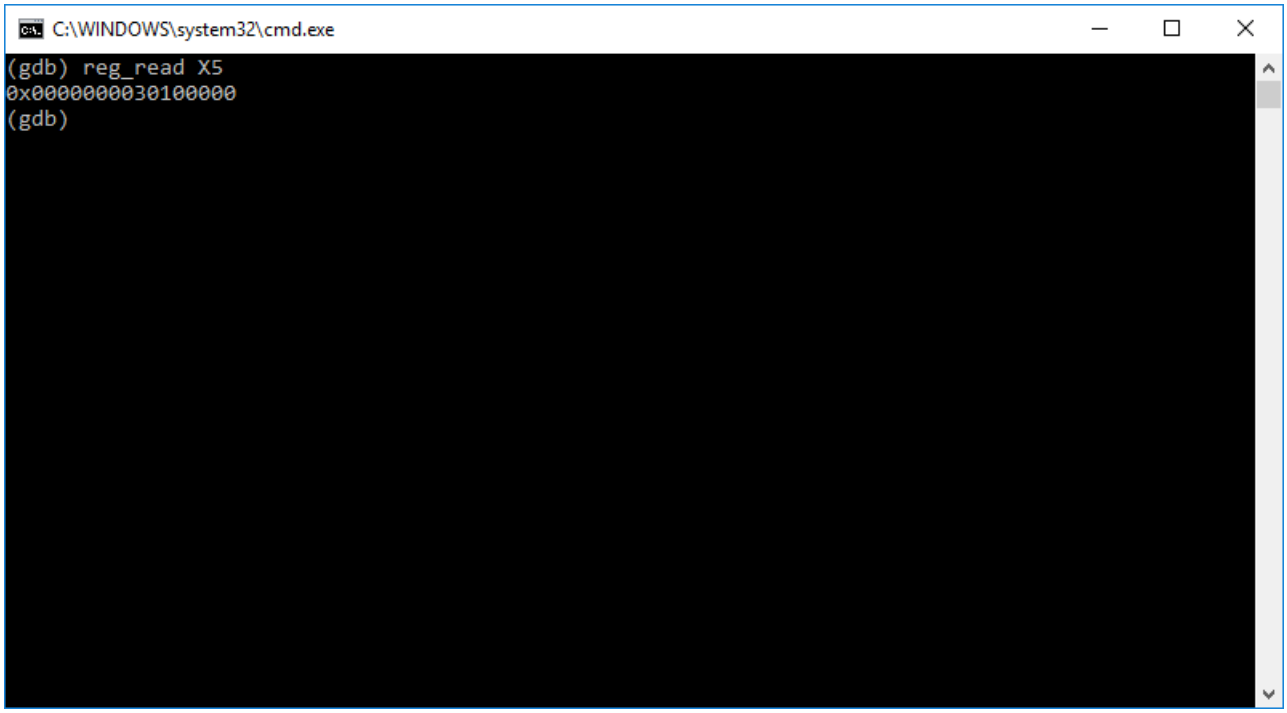


Figure 32. Reading a register using `reg_read` command

6.2.2.3 reg_print command

Print a list with all group registers registers / bifields details.

Usage

```
reg_print [-h] [context] name [-s SELECT]
```

Table 40. Mandatory arguments

Name	Address
name	REG_GROUP[.REG_NAME[.BIT_FIELD]] REG_GROUP - name of the register group (IP block) REG_NAME - name of a register from REG_GROUP BIT_FIELD - name of a bit field from the REG_NAME.

Table 41. Optional arguments

Name	Address
context	GTA (GDB server) debug context :ccs:<soc>[:core] e.g. :ccs:LS2088A or :ccs:LS2088A:CortexA72#0. If present, the context can be a core context; if not, the current context is used.
-s SELECT, --select SELECT	specify what information will be printed; possible values: a - access, b - bit range, d - description, l - location, r - reset value, v - value

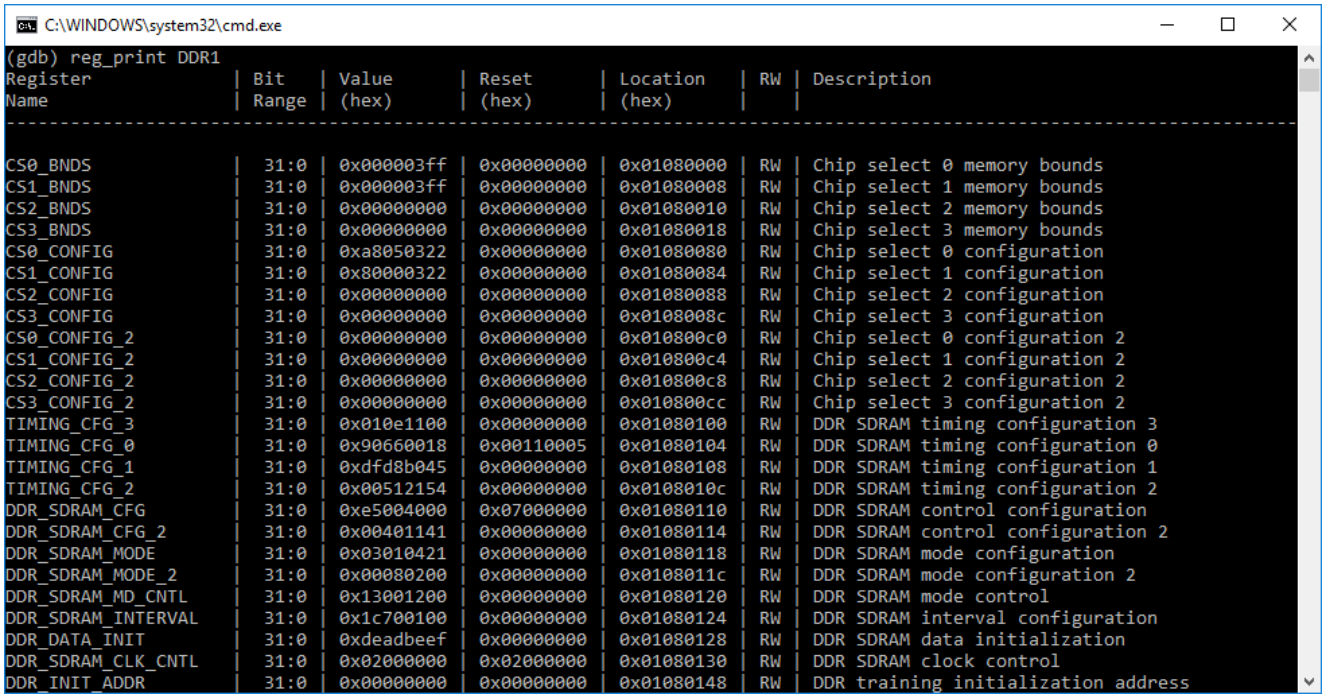


Figure 33. Reading an entire registers group using `reg_print` command

6.2.2.4 reg_export command

Export register groups to a file.

Usage

```
reg_export FILE <ALL | GROUP_NAME1,GROUP_NAME2...> [FORMAT=regs]
```

Table 42. Mandatory arguments

Name	Address
FILE	Path for the exported result.
<ALL | GROUP_NAME1,GROUP_NAME2...>	What to export. All register groups (IP blocks) or a subset of them.

Table 43. Optional arguments

Name	Address
FORMAT	Format of the exported file. Supported option: regs

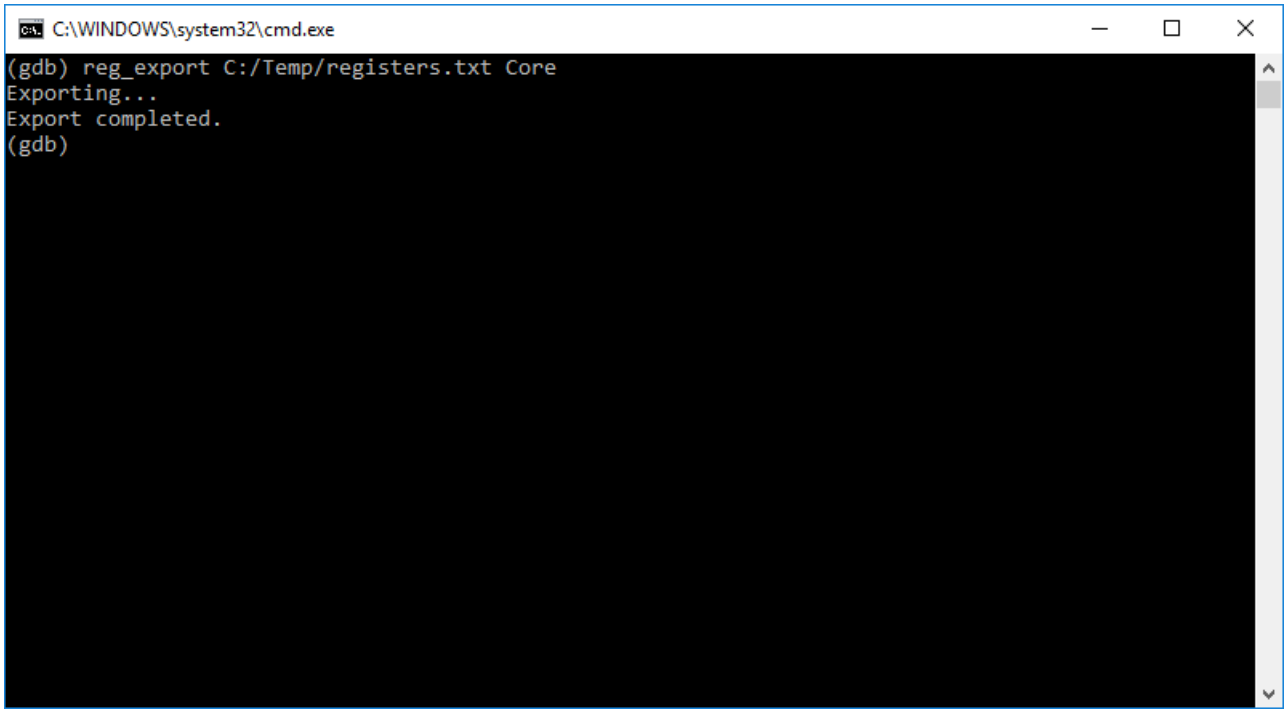


Figure 34. Exporting a registers group using `reg_export` command

6.3 OS awareness

OS awareness support in the CodeWarrior software is a group of features that simplify and improve the user experience while debugging the OS-specific projects.

The OS awareness features are enabled from the **OS Awareness** tab in the **Debug Configurations** dialog.

Currently, predefined support exists for the following types of OS Awareness: Linux Kernel, U-boot, and UEFI. When importing an executable image for a Linux kernel, U-Boot, or UEFI project using the CodeWarrior Executable Importer wizard, the image type is auto-detected and the configuration of the options in the **OS Awareness** tab is done automatically. The user can manually change the options in the **OS Awareness** tab at any time. Advanced users can also use custom scripts to add the OS awareness support for their specific projects.

6.3.1 Linux kernel awareness

This topic explains how to enable Linux kernel awareness.

To enable Linux kernel awareness, select the checkboxes **Enable OS Awareness**, **Linux Kernel** , and **Use CodeWarrior script for Linux Kernel Awareness** in the **OS Awareness** tab.

For details on how to create a Linux kernel project and start a debug session, see Linux kernel debug.

6.3.1.1 List Linux kernel information

Linux kernel awareness allow users to see relevant Linux kernel operating system information.

- Build time and kernel version
- Kernel module list
- Kernel thread list

The Linux kernel information is available in the command line and in the Eclipse view.

6.3.1.1.1 GDB commands

Once a debug session is started and debug is suspended, go to the gdb console and run `<gdb_command>`.

The following GDB commands are available:

- `- ka-show-info`: Prints Linux kernel general information

```
(gdb) ka-show-info
Build Time = #7 Mon Mar 31 11:44:09 EEST 2014
Linux Version = 3.12.0+
```

- `- ka-show-thread-list`: Prints the kernel threads

```
(gdb) ka-show-thread-list
```

Name	Pid	State	Address	Core
Swapper	0	running	0xffffffc0004de430	0
init	1	interruptible	0xffffffc079c50000	0
kthreadd	2	interruptible	0xffffffc079c50880	0

- `- ka-reset`: Resets the board when an OS Awareness is enabled (Linux Kernel, U-Boot, or UEFI)

```
(gdb) ka-reset
```

6.3.1.1.2 Eclipse view

When Linux kernel awareness is enabled from the tab, the OS Resources view displays information about:

- Linux system information
- modules list
- kernel thread information

6.3.1.2 Linux kernel debug

Linux kernel module debugging is enabled by default when kernel awareness extensions are enabled.

The following gdb commands are implemented for Linux kernel module debug.

6.3.1.2.1 GDB commands

This topic explains the GDB commands.

GDB commands:

- - ka-show-module-list :

  Description: Prints Linux kernel modules

  (gdb) ka-show-module-list

Name	Address
krng	0xffffffbffc010000
rng	0xffffffbffc00c000

- ka-module-load:

  Description: Loads symbolics file for a kernel module.

  The commands has the following parameters:

  — (required) the kernel module symbolics file

  — (optional) the module name, necessary when the symbolics file name and the kernel module name are different

  Example:

```
(gdb) ka-module-load /data/ARM_DEVEL/linux/ls2-linux/crypto/krng.o
              Symbol file /data/ARM_DEVEL/linux/ls2-linux/crypto/krng.o loaded successfully
```

- ka-module-unload:

  Description: Unloads symbolics file for a kernel module.

  The commands has one required parameter: the module name

  Example:

```
(gdb) ka-module-unload rng
       Symbol file /data/ARM_DEVEL/linux/ls2-linux/crypto/rng.o unloaded successfully
```

- ka-module-files:

  Description: Shows the loaded symbolics file for a kernel modules.

  The command has an optional argument (integer > 0) representing the maximum number of files

  Example:

```
(gdb) ka-module-files
     Name           Loaded file
     rng            /data/ARM_DEVEL/linux/ls2-linux/crypto/rng.o
     krng           /data/ARM_DEVEL/linux/ls2-linux/crypto/krng.o
```

- ka-module-config-suspend:

  Description: Configures module detect suspend action:

  The command has one optional argument (boolean):

  — True: suspend target when module insert/removal is detected

  — False: do not suspend target when module insert/removal is detected

  If no parameter is passed, the command returns the configuration value

  Example:

```
(gdb) ka-module-config-suspend True
(gdb) ka-module-config-suspend True
```

- ka-module-config-auto-load:

  Description: Configures module detect auto-load action:

  The command has one optional argument (boolean):

  — True: automatically load configured symbolic files at module init detection

  — False: no not automatically load module symbolics at module init detection

If no parameter is passed, the command returns the configuration value.

Example:

```
(gdb) ka-module-config-auto-load True
(gdb) ka-module-config-auto-load True
```

- ka-module-config-map-load:

  Description: Adds the module symbolics file in the module configuration map.

  If the auto-load is enabled, this symbolics file is automatically loaded when the corresponding module is inserted.

  The commands has the following parameters:

  — (required) the kernel module symbolics file

  — (optional) the module name, necessary when the symbolics file name and the kernel module name are different

  Example:

```
(gdb) ka-module-config-map-load /data/linux/crypto/krng.o
```

- ka-module-config-map-unload:

  Unloads symbolics file from the module configuration map. The commands has one required parameter: - the module name

  Example:

```
(gdb) ka-module-config-map-unload krng
```

- ka-module-config-show:

  Description: Shows the module configuration parameters. The command has an optional argument (integer > 0) representing the maximum number of files shown from the configuration map

  Example:

```
(gdb) ka-module-config-show
Name                    Loaded file
rng                     /data/linux/crypto/rng.o
krng                    /data/linux/crypto/krng.o
```

6.3.1.3 Linux kernel image version verification

When Kernel awareness is enabled, CodeWarrior performs a Linux Kernel image version verification to validate that the binary image on the target (uImage) matches the ELF symbolics file (vmlinux) in the debugger.

When access to target version is available (after the u-boot copies the Linux kernel image into DDRAM), the debugger performs the version verification. In case of mismatch, the debugger prints the following message in the gdb console: `Warning: Kernel image running on the target is different than the vmlinux image in debugger.`

In addition, the user can trigger at any time the version verification in the following way:

- From CLI using the `ka-show-info` commands. For example:

```
(gdb) ka-show-info

Build Time = #2 SMP PREEMPT Thu Nov 13 10:09:26 EET 2014
Linux Version = 3.16.0-Layerscape2-SDK+gec37efe
Target version check : ELF image version matches target image version
```

In case of version mismatch, the Target version check message is `Warning: Kernel image running on the target is different than the vmlinux image in debugger`. If the access to target version is not available yet, the Target version check message is `not available yet`. The user should check again after the u-boot copies the Linux kernel image into the DDRAM.

- From Eclipse, OS Resources window, select **System Info**. The same information as for the CLI command is shown here.

6.3.2 U-Boot awareness

This topic explains how to enable U-Boot awareness.

The U-Boot awareness features enhance and improve the usability for U-Boot debugging by simplifying the debugging interface. The U-Boot awareness feature provides:

- a single debug session for all U-Boot booting phases that allows user to debug from the first instruction after reset to relocation in DDRAM
- possibility to debug an SPL-based U-Boot (e.g. for NAND/SD) by specifying the SPL U-Boot and covering all booting stages
- U-Boot command line prompt for booting the Linux kernel

With U-Boot awareness, the debugger automatically detects each U-Boot stage using debugger eventpoints and performs specific actions, such as setting the relocation offset for DDRAM.

To enable the U-Boot awareness features, select the checkboxes **Enable OS Awareness**, **U-boot**, and **Use CodeWarrior script for U-boot Awareness** in the **OS Awareness** tab. For details on how to create a U-Boot project and how to start a debug session, see U-Boot debug.

6.3.2.1 List U-Boot information

When U-Boot awareness is enabled from the **OS Awareness** tab, the **OS Resources** view displays information about:

- U-Boot version, configuration, and build time
- Memory, that is RAM size, RAM top, relocation address, and relocation offset. However, this information is displayed only when the data is available after relocation.

6.3.2.2 U-Boot image version verification

For U-Boot, CodeWarrior performs the same kind of checking as for Linux kernel image.

In the same way, the mismatch warning is shown in the gdb console when the U-Boot version is available and the user can check the version at any time from Eclipse, OS Resource window, selecting “Version”.

NOTE

Note that versions verification when doing SPL U-Boot debug will be available once the main/DDR U-Boot gets loaded in DDR. Until this happens, target version check will be reported as being unavailable.

6.3.3 UEFI awareness

This topic explains how to enable UEFI awareness.

To enable UEFI awareness, select the checkboxes **Enable OS Awareness**, **UEFI**, and **Use CodeWarrior script for UEFI Awareness** in the **OS Awareness** tab.

6.3.3.1 Load debug data for all loaded EFI images

The UEFI awareness support in the CodeWarrior software allows users to load the debug data for all the EFI images loaded in memory (that are instances of the Loaded Image protocol).

When UEFI awareness is enabled, a GDB command is available in the CLI for this capability. The feature provides meaningful results only after the EFI core has created the EFI Debug Support table. The command has no effect if it is invoked for example during the Platform Initialization phase.

6.3.3.1.1 GDB command

The following GDB command loads symbols for all the EFI images loaded in memory.

```
uefi-add-symbols
```

The command has the following parameters:

- (optional) --mstart: The base of the physical memory (RAM) used by UEFI
- (optional) --msize: The size of the physical memory (RAM) used by UEFI
- (optional) -s SAVE_COMMANDS_TO_FILE, --save_commands_to_file SAVE_COMMANDS_TO_FILE: A file where to store all the GDB commands that load the debug data
- (optional) -r UEFI_LAYOUT_ROOT, --root_uefi_layout UEFI_LAYOUT_ROOT: The directory path to the root of the UEFI layout used for debug information
- (optional) -v, --verbose: Increase output verbosity

The command provides meaningful results only after the EFI core has created the EFI Debug support table. It will display an error message if it is invoked, for example during the SEC phase. The command will display "Done", after it finishes the execution successfully.

Example:

```
(gdb) uefi-add-symbols
```

6.3.3.2 Show information for all loaded EFI images

UEFI awareness allows users to view information about all the EFI images loaded in memory (that are instances of the Loaded Image protocol).

The feature provides meaningful results only after the EFI core has created the EFI Debug Support table. It has no effect if it is invoked, for example during the SEC phase.

6.3.3.2.1 GDB command

When UEFI awareness is enabled, the following GDB command displays information about all the EFI images loaded in memory.

```
uefi-show-images
```

The command has the following parameters:

- (optional) --mstart: The base of the physical memory (RAM) used by UEFI
- (optional) --msize: The size of the physical memory (RAM) used by UEFI
- (optional) -n: Display the name of the EFI image (the file name of the symbol file, without extension)
- (optional) -b: Display the EFI image base address
- (optional) -c: Display the EFI image BaseOfCode address
- (optional) -e: Display the EFI image EntryPoint address
- (optional) -a: Display whether the symbol file for the EFI image was added to the debug information (yes / no)
- (optional) -s: Display the compile time file path of the symbol file for the EFI image

The order of the information displayed for each image will correspond to the order of the options given as parameters. When invoked with no parameters, the command will display by default the image name and base address.

Example:

```
(gdb) uefi-show-images
```

6.3.3.2.2 Eclipse view

When UEFI awareness is enabled from the **OS Awareness** tab, the OS Resources view displays information about Loaded EFI Images.

By default the following columns are displayed:

- Image name
- Image base address
- Image BaseOfCode address
- Image EntryPoint address
- Whether the symbol file for the EFI image was added to the debug information (yes / no)

The user can customize the columns that are displayed.

6.4 Launch a hardware GDB debug session where no configuration is available

This topic explains how to launch a hardware GDB debug session.

Before you proceed, ensure that you have an ARMv8 project in your workspace, which is compiled, and the binary elf file is available.

To launch the debug session, you need to:

1. Create a debug configuration
2. Configure the target configuration using Target Connection Configurator

6.4.1 Create a debug configuration

This topics explains how to create a debug configuration.

To create a debug configuration:

1. Select the ARMv8 project in the **Project Explorer** view.
2. Select **Debug > Debug Configurations**. The **Debug Configurations** dialog appears.
3. Right-click **GDB Hardware Debugging** and select **New**.
4. Select the **Main** tab.
5. Make sure that the text box under the **C/C++ Application** option specifies the elf file path of the project you want to use. For example, `Debug/<project name>.elf`
6. Select the **Debugger** tab.
7. In the text box under the **GDB Command** option set the path to gdb. For example, `$ {eclipse_home}..\ARMv8\gdb\bin\aarch64-fsl-gdb.exe`
8. Click the **Debug** button.

NOTE

For details about configuring target connection, refer Configure the target configuration using Target Connection Configurator

6.5 Memory tools GDB extensions

This topic explains memory tools GDB extensions.

For details about other GDB debug commands that can be run in GDB console from console view, refer the GDB documentation available at: https://sourceware.org/gdb/current/onlinedocs/gdb/

NOTE

Note that NXP does not own GDB documentation, and is mentioned solely for reference purpose.

6.5.1 mem_spaces command

List the available memory spaces.

Usage

```
mem_spaces
```

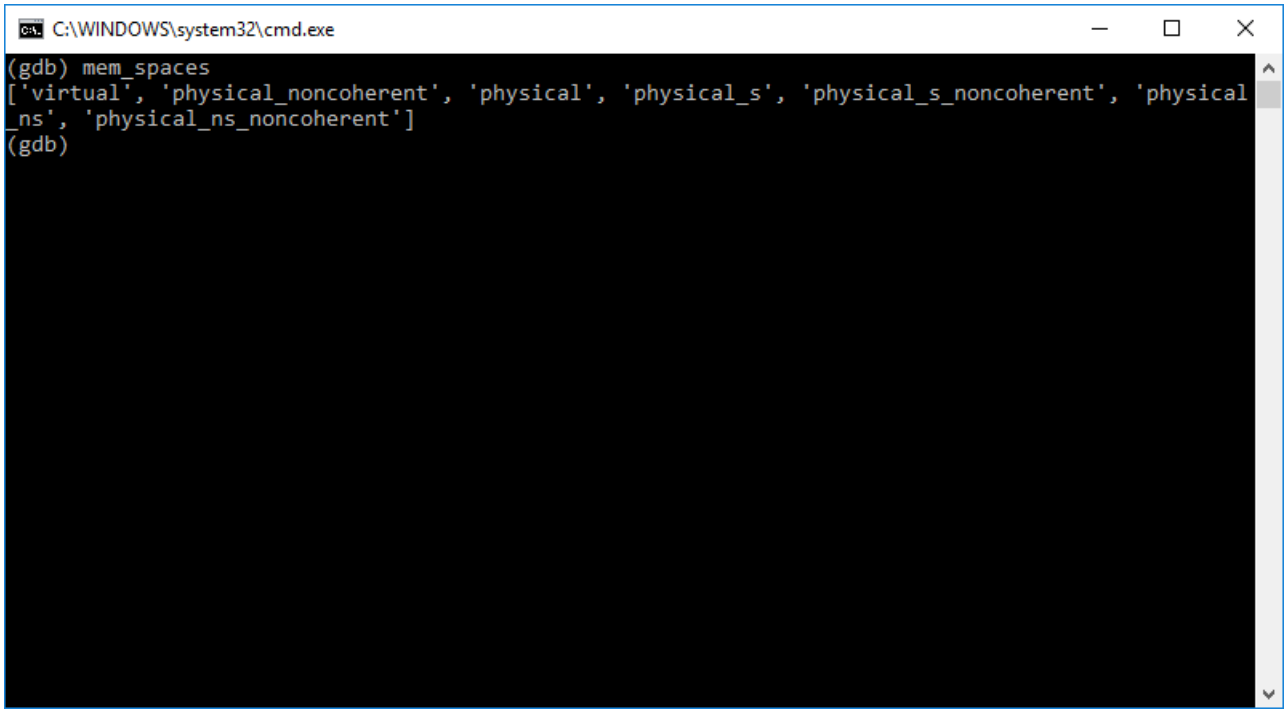


Figure 35. Displaying the available memory spaces using `mem_spaces` command

6.5.2 mem_read command

Read memory from an address using the provided access size, memory space and count. The result is displayed as a hexadecimal encoded byte stream.

Usage

```
mem_read [context] <address> <access_size> <space> <count>
```

Table 44. Mandatory arguments

Name	Address
<address>	Start address.
<access_size>	Access size.
<space>	Memory space.
<count>	Number of elements, each element having `access_size` bytes.

Table 45. Optional arguments

Name	Address
context	GTA (GDB server) debug context :ccs:<soc>[:core] e.g. :ccs:LS2088A or :ccs:LS2088A:CortexA72#0. If present, the context can be a core context; otherwise, the current context is used.

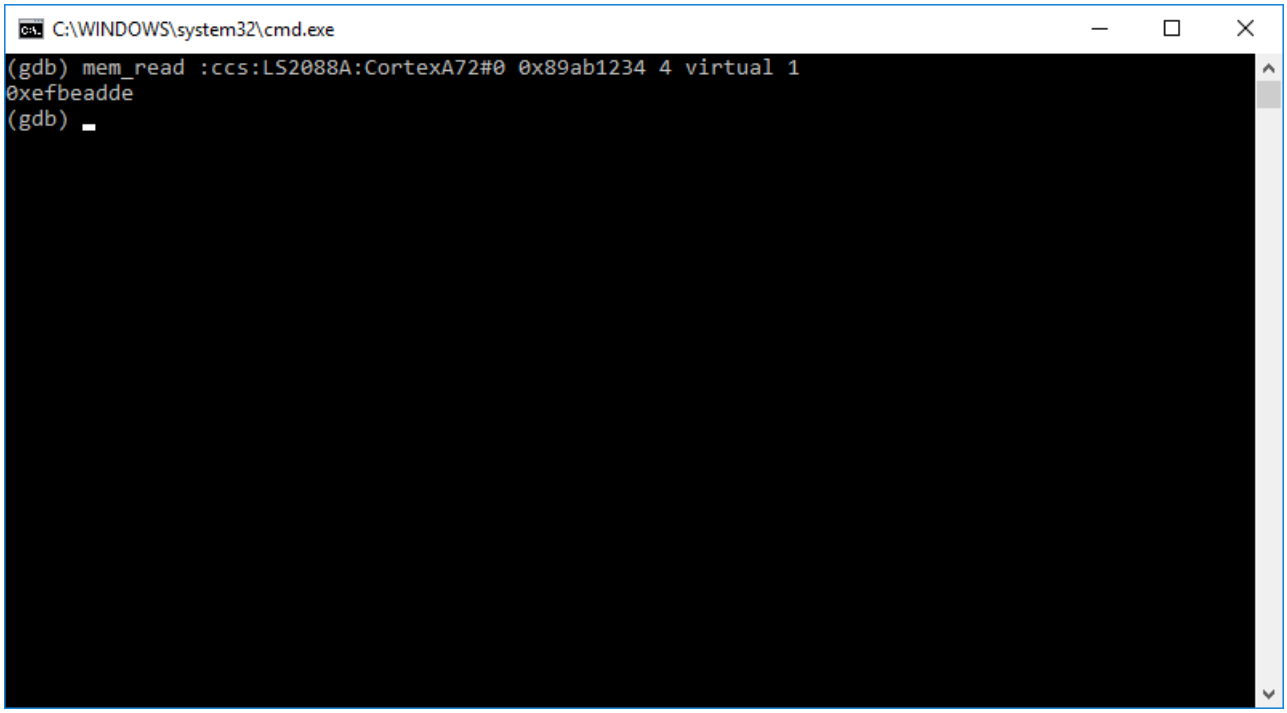


Figure 36. Reading memory using `mem_read` command

6.5.3 mem_write command

Write memory to address using the provided access size and memory space.

Usage

```
mem_write [context] <address> <access_size> <space> <data>
```

Table 46. Mandatory arguments

Name	Address
<address>	Start address.
<access_size>	Access size.
<space>	Memory space.
<data>	Data to be written as a sequence of hexadecimal byte values.

Table 47. Optional arguments

Name	Address
context	GTA (GDB server) debug context :ccs:<soc>[:core] e.g. :ccs:LS2088A or :ccs:LS2088A:CortexA72#0. If present, the context can be a core context; if not, the current context is used.

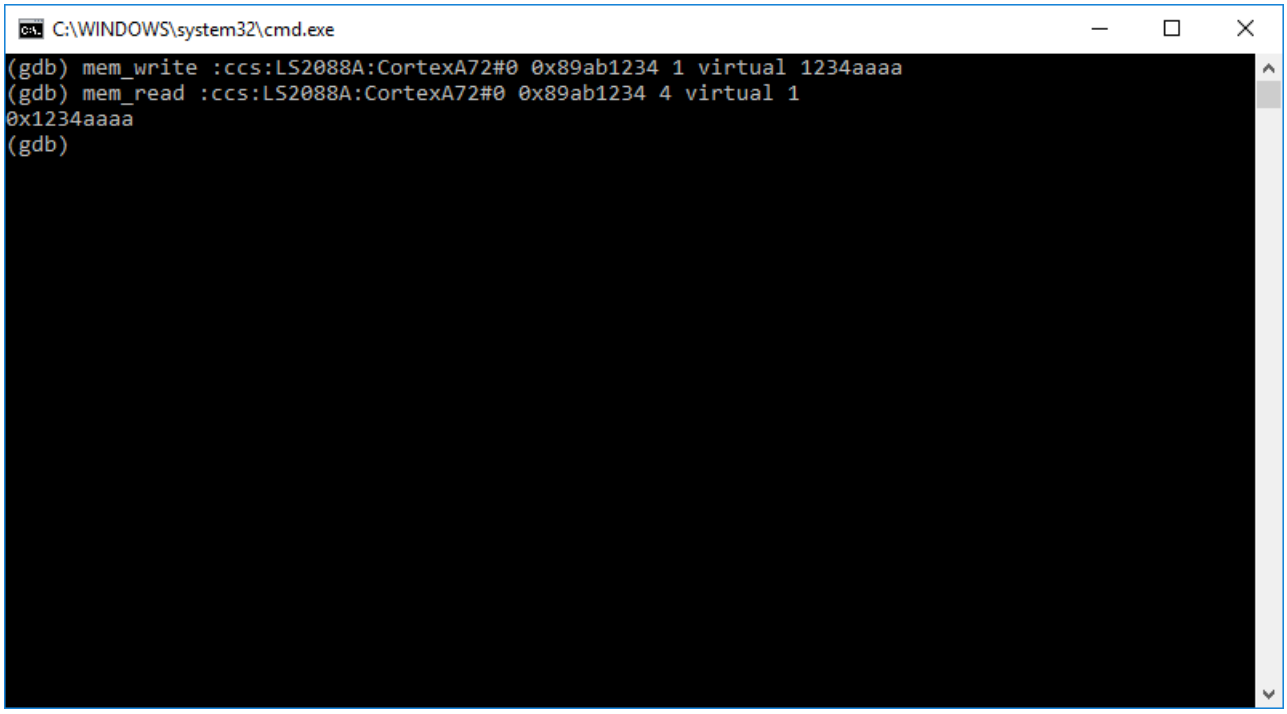


Figure 37. Writing memory using `mem_write` command

6.5.4 mem_fill command

Fill a memory range with the specified byte value.

Usage

```
mem_fill <start_address> <finish_address> <value>
```

Table 48. Mandatory arguments

Name	Address
<start_address>	Range start address.
<finish_address>	Range end address.
<value>	Value used to fill the memory range.

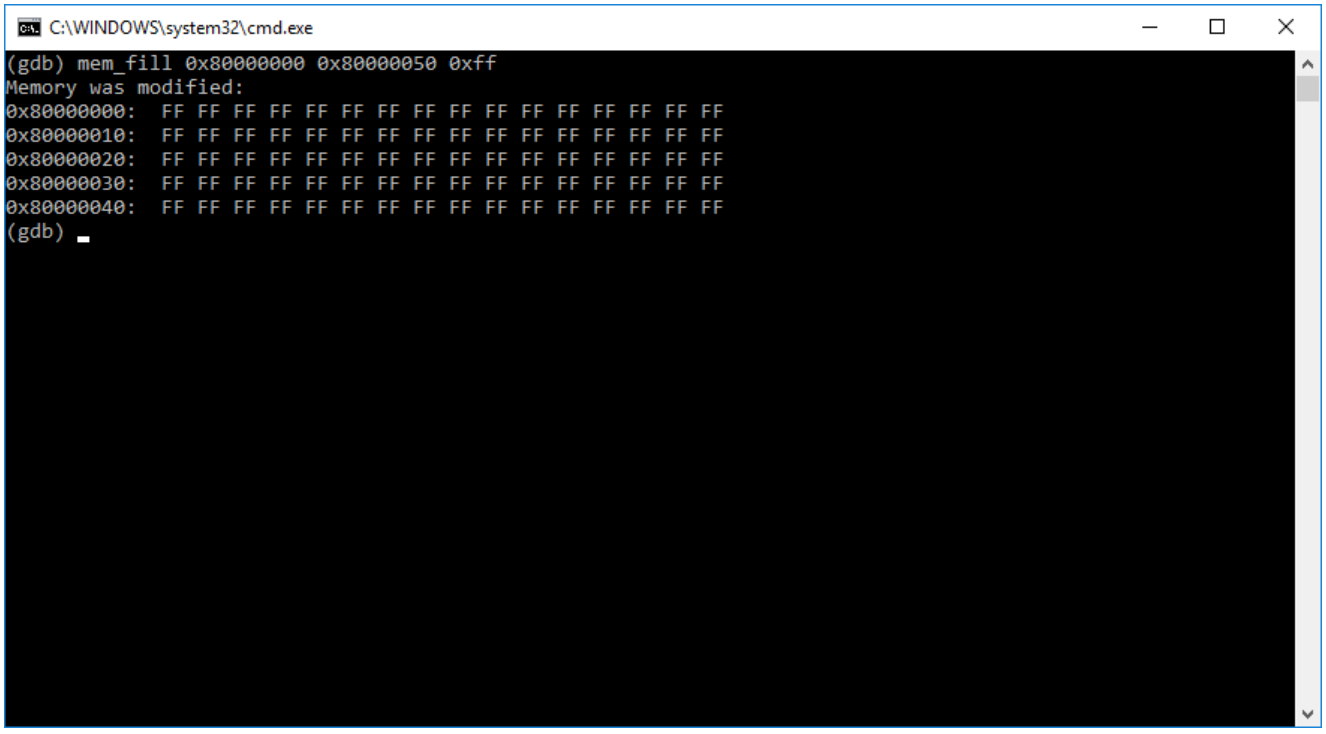


Figure 38. Filling an entire memory area with a given value using `mem_fill` command

6.5.5 mem_compare command

Compare contents of two memory ranges.

Usage

```
mem_compare <address1> <address2> <count>
```

Table 49. Mandatory arguments

Name	Address
<address1>	Start address of the first range.
<address2>	Start address of the second range.
<count>	Number of bytes to be compared.

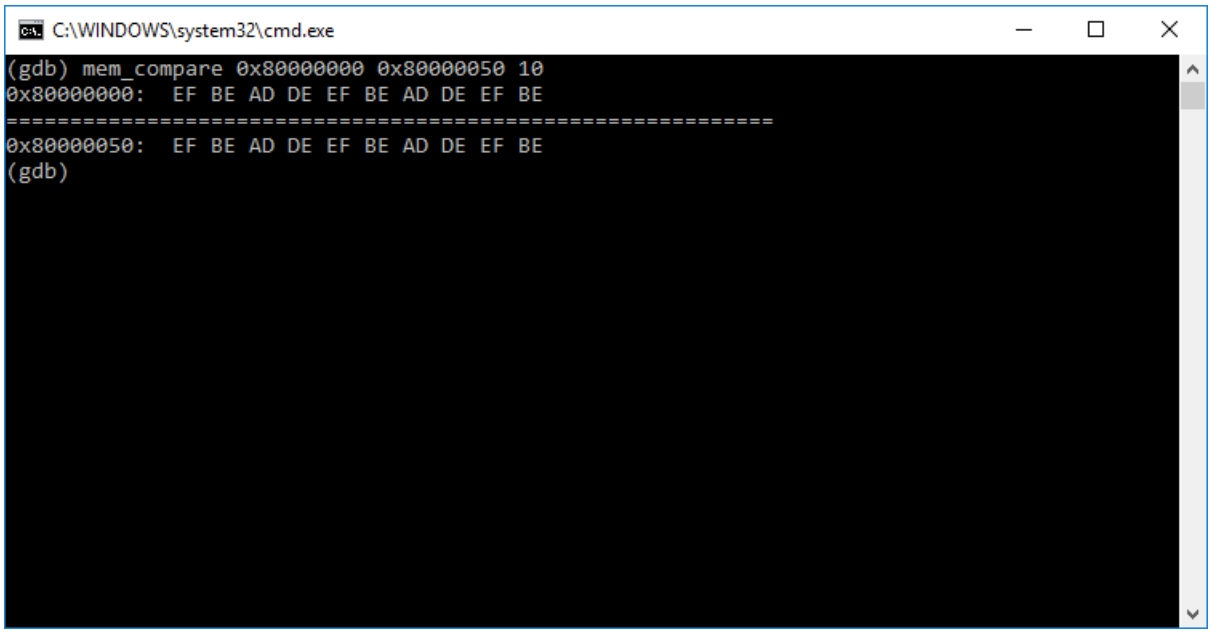


Figure 39. Comparing two memory areas using `mem_compare` command

6.5.6 mmu command

Display MMU (Memory Management Unit) state in a user readable format.

Usage

```
mmu [-h] [<arch-specific-options>]
```

Table 50. Optional arguments

Name	Address
-h, --help	Show this help message and exit
<arch-specific-opts>	Architecture specific options.

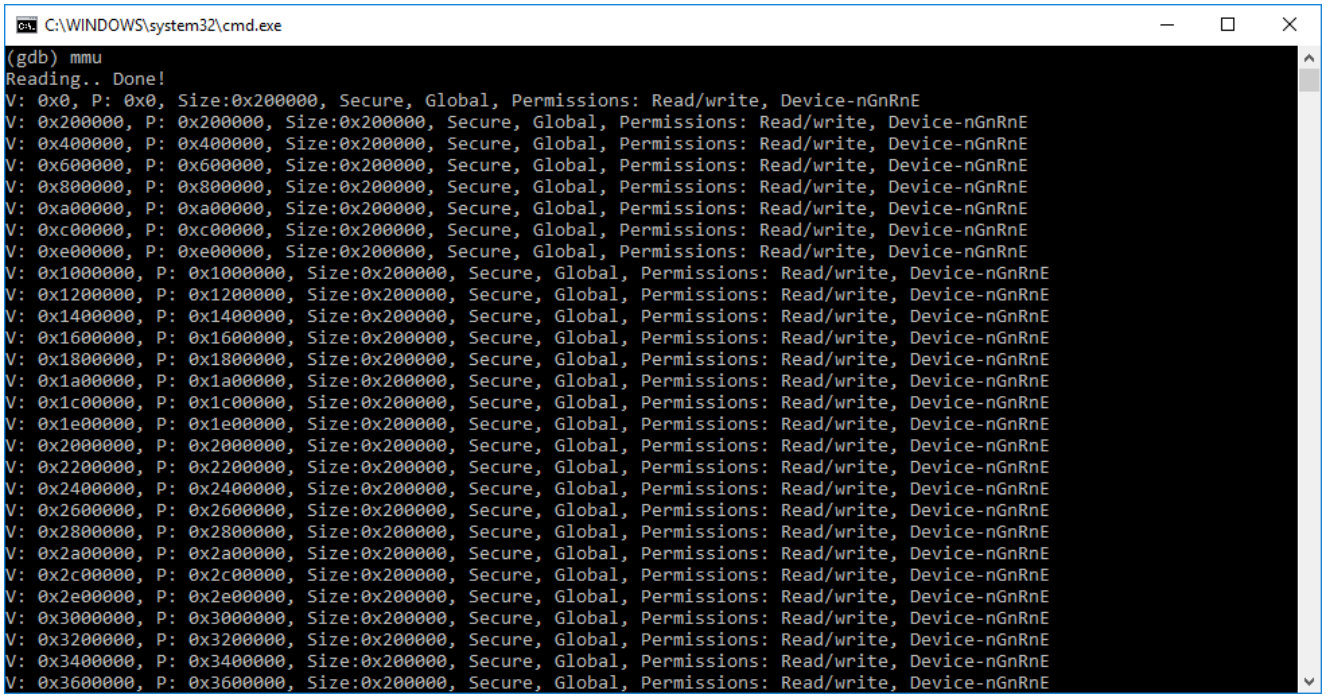


Figure 40. Obtaining current MMU entries

- Issuing `mmu` command without any parameters will list all the MMU valid entries for the current exception level.
- Issuing `mmu -el 3` command will list all the valid MMU entries for the EL3 exception level.
- Issuing `mmu -t 0x2000000` will translate the virtual address 0x2000000 to the corresponding physical address using MMU state for the current exception level.

6.6 Connection tools GDB extensions

This topic explains connection tools GDB extensions. They provide support for connecting to a target and diagnosing an existing connection.

For details about other GDB debug commands that can be run in GDB console from console view, refer the GDB documentation available at: https://sourceware.org/gdb/current/onlinedocs/gdb/

NOTE

Note that NXP does not own GDB documentation, and is mentioned solely for reference purpose.

6.6.1 cw-launch command

Usage:

```
cw-launch --config CONFIG [--probe PROBE] [--serial-number SERIAL_NUMBER] [--init-script [INIT_SCRIPT]]
[--ccs-address CCS_ADDRESS] [--secure-debug-key SECURE_DEBUG_KEY] [--gta-address GTA_ADDRESS] [--gta-
context GTA_CONTEXT] [--reset] [--elf-file FILE] [--other-symbols SYMBOLS_FILE] [--download] [--os-
awareness [OS_TYPE]]
```

Table 51. Mandatory arguments

Name	Address
-c CONFIG [CONFIG ...], --config CONFIG [CONFIG ...]	Specify the path to a valid tcc file; template tcc files from `{CW Folder}/CW_ARMv8/Config/boards` folder can be used.

Table 52. Optional arguments

Name	Address
-h, --help	Show this help message and exit
-p PROBE, --probe PROBE	Probe details; it overrides what is set in tcc file. For example: cwtap:192.168.0.1 - CodeWarrior TAP ethernet connection using 192.168.0.1 as IP address
-s SERIAL_NUMBER, --serial-number SERIAL_NUMBER	USB serial number of the probe.
-i [INIT_SCRIPT], --init-script [INIT_SCRIPT]	A full path to a .py file that is a valid target initialization script; if no path specified, no initialization file will be executed; if parameter not present, initialization file from tcc will be used.
-ca CCS_ADDRESS, --ccs-address CCS_ADDRESS	CCS address; "<host>:<port>" For example, 127.0.0.1:41475 or "auto"; default is auto.
-sk SECURE_DEBUG_KEY, --secure-debug-key SECURE_DEBUG_KEY	Secure debug key used to unlock debug support.
-ga GTA_ADDRESS, --gta-address GTA_ADDRESS	GTA (GDB server) address; "<host>:<port>" e.g. 127.0.0.1:45000 or "auto". Default is 127.0.0.1:45000.
-gc GTA_CONTEXT, --gta-context GTA_CONTEXT	GTA (GDB server) debug context :ccs:<soc>[:core] e.g. :ccs:LS2088A or :ccs:LS2088A:CortexA72#0.
-r, --reset	Execute reset; no reset is performed if this parameter is not provided.
-ng, --dont-start-gta	Do not start the GTA (GDB server) process.
-ps, --preserve	Preserve existing CCS connection details.
-elf ELF_FILE, --elf-file ELF_FILE	ELF file containing debug symbols. File will also be loaded into target memory if --download is specified.
-sym OTHER_SYMBOLS, --other-symbols OTHER_SYMBOLS	Load additional debug symbols from the specified file. When performing U-Boot debug with an SPL file involved, first ELF file specified via --other-symbols is the actual SPL ELF.
-d, --download	Load the specified ELF file into target memory.
-os [OS_AWARENESS], --os-awareness [OS_AWARENESS]	Control OS awareness enablement. By default, OS awareness is auto-detected based on the specified ELF file.

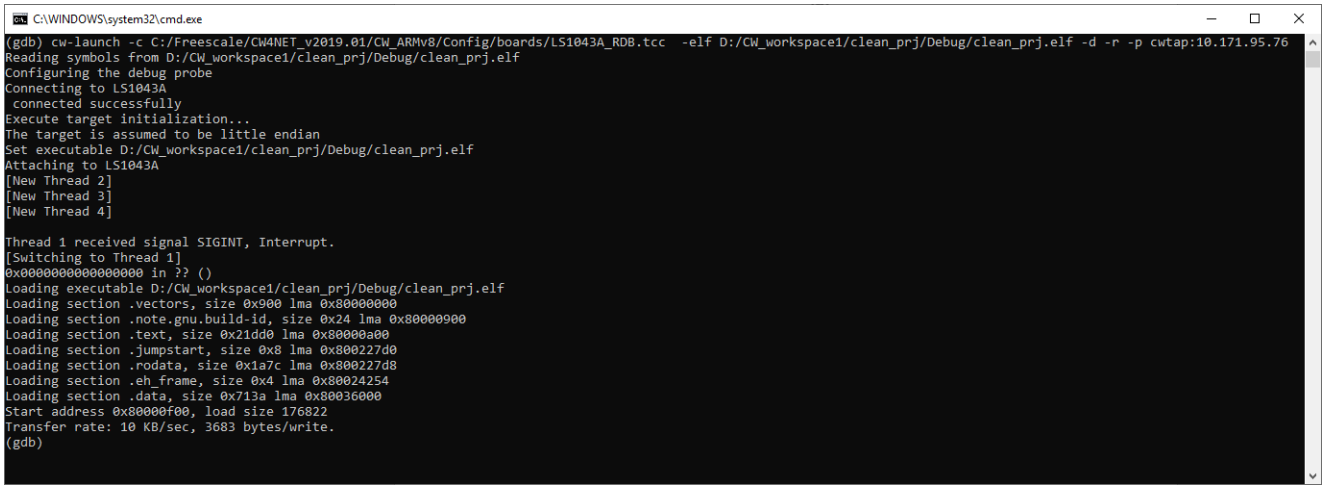


Figure 41. Launching a debug session using `cw-launch` command

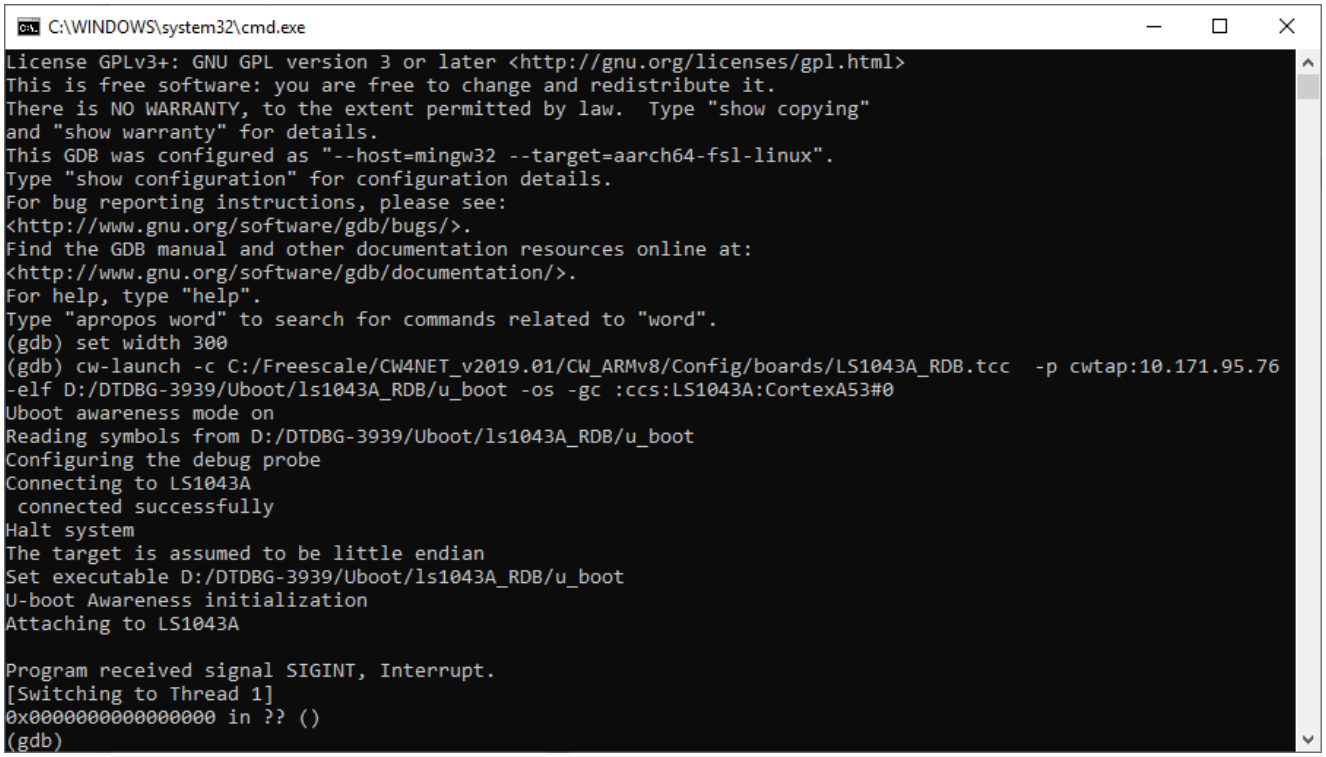


Figure 42. Example with launching a debug session with OS awareness support

NOTE

In order to start a debug session from command line, launch GDB from the following folder: `{CW Folder}\CW_ARMv8\ARMv8\gdb\bin\` and use `cw-launch` command from the GDB console to connect to the target.

`cw-launch` requires a corresponding `.tcc` file from the folder `{CW folder}/CW_ARMv8/Config/boards`.

6.6.2 cw-diag command

This command is used to diagnose a connection.

Usage

```
cw-diag -p <probe> -gc <GTA context> [-i <init script>] [-ca <ccs address>] [-sk <secure debug key>] [-
ga <gta address>] [-j <jtag speed>] [-ct <ccs timeout>] [-sc] [-ps]
```

Table 53. Mandatory arguments

Name	Address
-p PROBE, --probe PROBE	Probe details; For example: cwtap:192.168.0.1 - CodeWarrior TAP ethernet connection using 192.168.0.1 as IP address
-gc GTA_CONTEXT, --gta-context GTA_CONTEXT	GTA (GDB server) debug context :ccs:<soc>[:core] e.g. :ccs:LS2088A or :ccs:LS2088A:CortexA72#0

Table 54. Optional arguments

Name	Address
-h, --help	Show this help message and exit
-s SERIAL_NUMBER, --serial-number SERIAL_NUMBER	USB serial number of the probe
-i [INIT_SCRIPT], --init-script [INIT_SCRIPT]	Full path to a .py file that is a valid target initialization script; if no path specified, no initialization file will be executed
-ca CCS_ADDRESS, --ccs-address CCS_ADDRESS	CCS address; "<host>:<port>" e.g. 127.0.0.1:41475 or "auto"; default is auto
-sk SECURE_DEBUG_KEY, --secure-debug-key SECURE_DEBUG_KEY	Secure debug key used to unlock debug support
-ga GTA_ADDRESS, --gta-address GTA_ADDRESS	GTA (GDB server) address; "<host>:<port>" e.g. 127.0.0.1:45000 or "auto". Default is 127.0.0.1:45000
-sg, --start-gta	Do not start the GTA process
-j JTAG_SPEED, --jtag-speed JTAG_SPEED	JTAG speed (kHz)
-ct CCS_TIMEOUT, --ccs-timeout CCS_TIMEOUT	CCS timeout (seconds)
-sc, --start-ccs	Start CCS process; by default it starts CCS
-ps, --preserve-settings	Preserve settings already configured in the running CCS instance

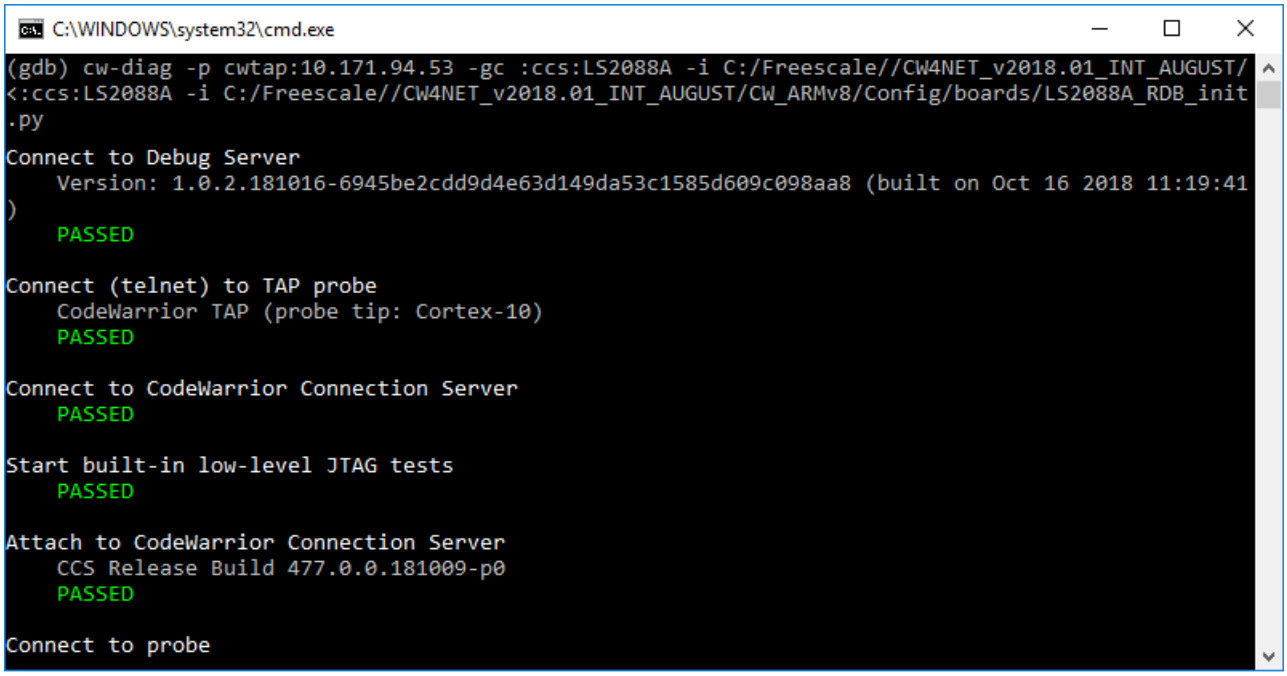


Figure 43. Running diagnostics on a given target board using `cw-diag` command

NOTE

In order to start a debug session from command line, launch GDB from the following folder: `{CW Folder}\CW_ARMv8\ARMv8\gdb\bin\` and use `cw-diag` command from the GDB console to diagnose the connection.

6.7 Miscellaneous tools GDB extensions

This topic presents other GDB extensions such as the ones for: SPD support, RCW override support, etc..

For details about other GDB debug commands that can be run in GDB console from console view, refer the GDB documentation available at: https://sourceware.org/gdb/current/onlinedocs/gdb/

NOTE

Note that NXP does not own GDB documentation, and is mentioned solely for reference purpose.

6.7.1 template command

This command is used to configure a CCS template specific option. Each template may provide a custom set of configuration options. Examples include configuring endian-ness of data for memory read/write or turning on or off memory write verification. This function provides a generic interface for those options.

Usage

```
template [-h] [-c CONTEXT] register [data]
```

Table 55. Mandatory arguments

Name	Description
register	Index of the register (template) to configure.

Table 56. Optional arguments

Name	Description
-h, --help	Show this help message and exit
-c [CONTEXT], --context [CONTEXT]	GTA (GDB server) debug context :ccs:soc[:core] e.g. :ccs:LS2088A:CortexA72#0 or :ccs:LS2088A:SoC#0
data	If no value is specified, it lists the available template for the provided config_reg. If a value is specified, it configures config_reg with provided config_data value.

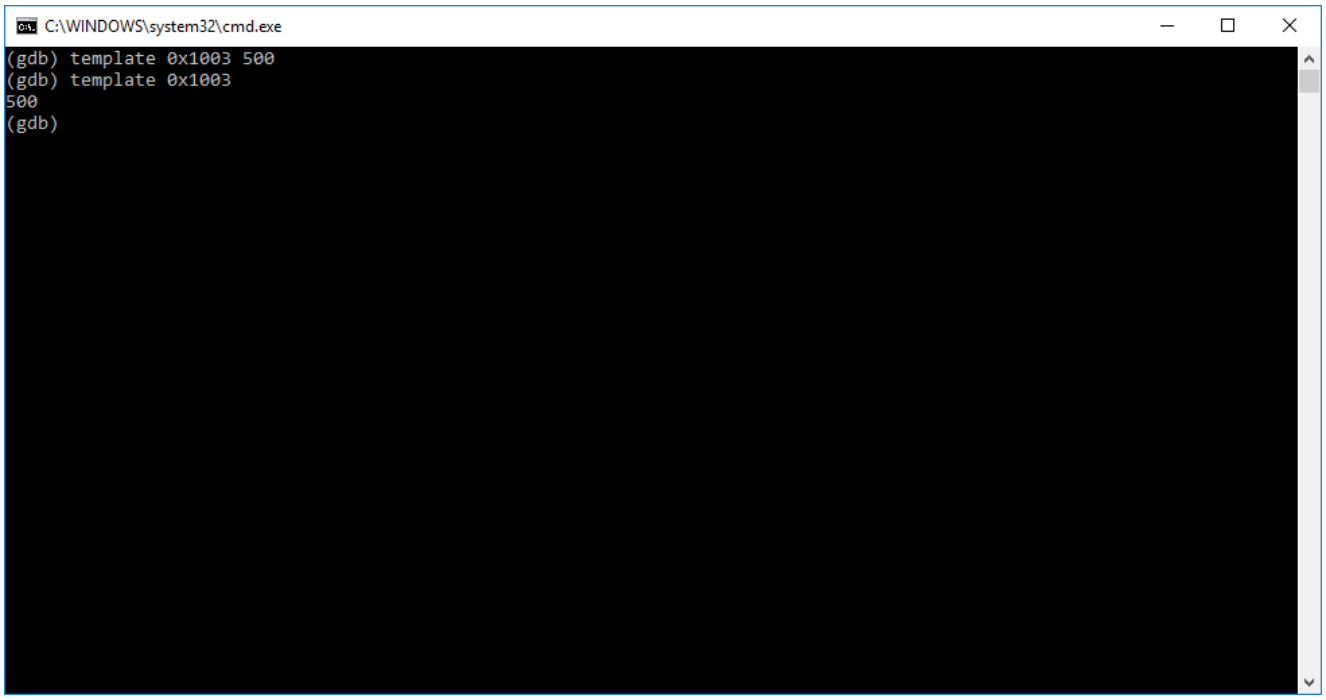


Figure 44. Configuring the CCS option `0x1003` using `template` command

NOTE

In order to execute this command, GDB has to be connected to the target board.

6.7.2 spd command

This command is used to read SPD (Serial Presence Detect) from the target and display it in a human-readable format.

Usage

```
spd [-h] [-d DEVICE] [-a ADDRESS] [-wa WORKSPACE_ADDRESS]
```

Table 57. Optional arguments

Name	Description
-h, --help	Show this help message and exit

Table continues on the next page...

Table 57. Optional arguments (continued)

Name	Description
-d DEVICE, --device DEVICE	SPD eeprom address
-a ADDRESS, --address ADDRESS	Address of the I2C controller
-wa WORKSPACE_ADDRESS, --workspace-address WORKSPACE_ADDRESS	Address of the workspace

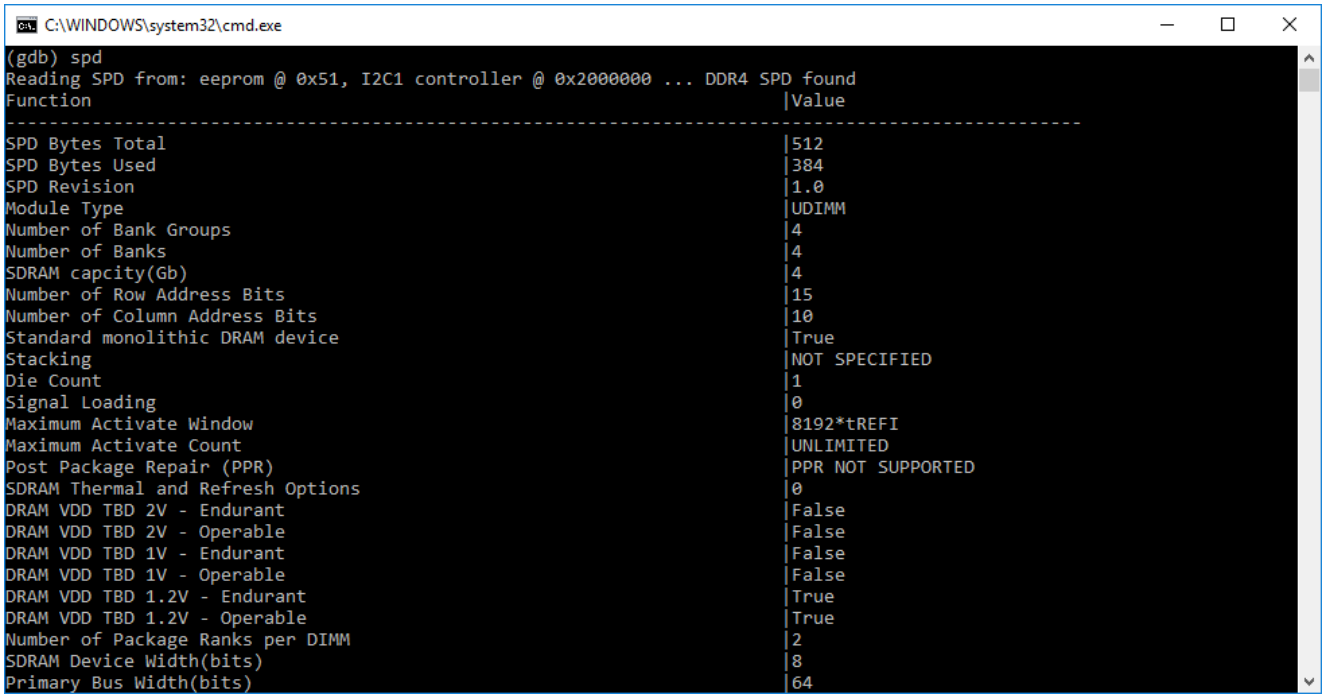


Figure 45. Detected parameters after running `spd` command on a board with DDR4 DIMMs

NOTE

In order to execute this command, GDB has to be connected to the target board.

6.7.3 rcw command

This command is used to read/override the RCW (Reset Configuration Word).

Usage

```
rcw [-h] [-r] [-s [SOURCE]] [-d DATA [DATA ...]]
```

Table 58. Optional arguments

Name	Description
-h, --help	Show this help message and exit
-r, --reset	Discard any previously configured RCW source and register values

Table continues on the next page...

Table 58. Optional arguments (continued)

Name	Description
-s [SOURCE], --source [SOURCE]	If no value is specified, it lists the available RCW sources defined for currently used processor. If a value is specified, it configures the RCW source to be used for RCW override functionality.
-d DATA [DATA ...], --data DATA [DATA ...]	Configure the values (hexadecimal) for ranges of RCW register(s) to be used for RCW override functionality; e.g.: rcw -d 1:0x12345678 10:0xabcdef12

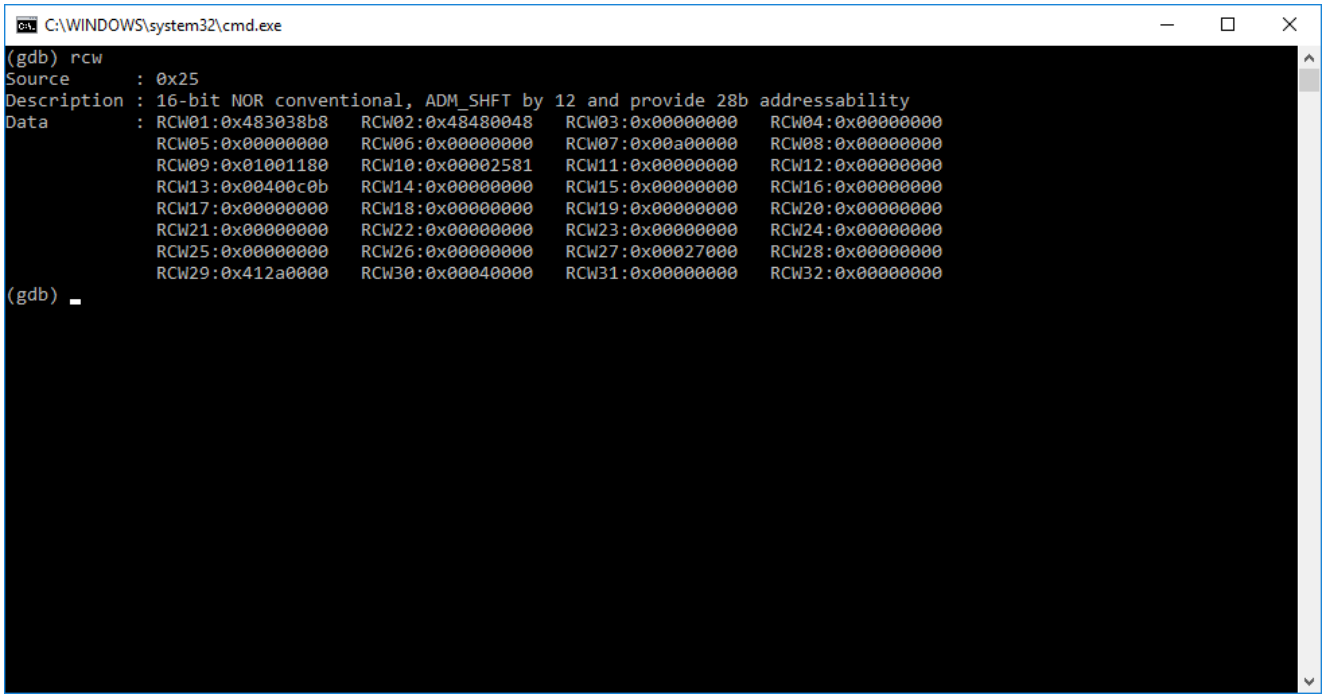


Figure 46. Displaying current RCW configuration using the `rcw` command

NOTE

In order to execute this command, GDB has to be connected to the target board.

6.7.4 discover command

Discover available debug probes and connected devices.

Usage

```
discover [-h] <type> [-p PROBE]
```

Table 59. Mandatory arguments

Name	Address
<type>	Can be one of the following options: probes, idcode, soc. "probes" - discover available debug probes. "idcode" - list IDCODE of devices

Table continues on the next page...

Table 59. Mandatory arguments

Name	Address
	connected to the specified probe. "soc" - detect SoC connected to the specified probe

Table 60. Optional arguments

Name	Address
-p PROBE, --probe PROBE	Probe specification (eg. cwtap:<ip/name>)
-h, --help	Show this help message and exit

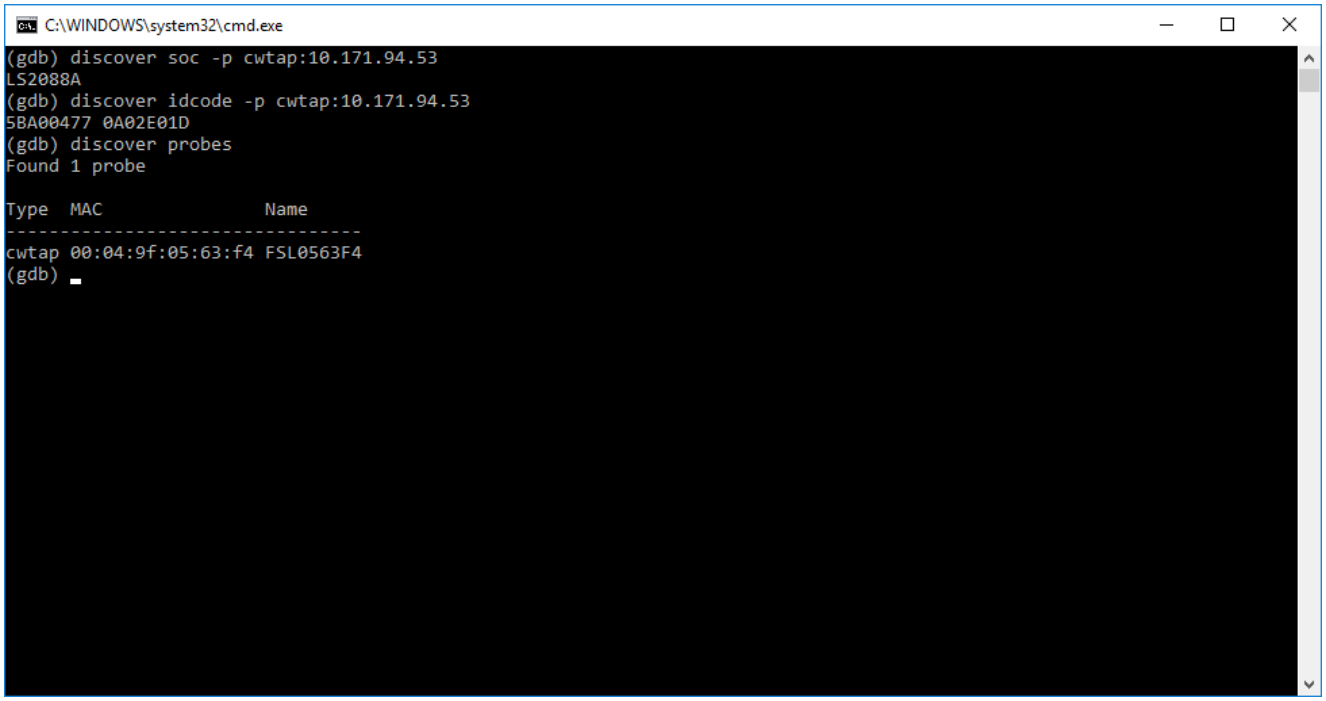


Figure 47. Discovering probes and devices using `discover` command

6.7.5 log command

Configure logging of protocol-level communication.

Usage

```
log file [-h] [-l {CRITICAL,ERROR,WARNING,INFO,DEBUG}] filepath
```

```
log console [-h] [-l {CRITICAL,ERROR,WARNING,INFO,DEBUG}] {out,err,log}
```

```
log socket [-h] [-l {CRITICAL,ERROR,WARNING,INFO,DEBUG}] hostname:port
```

```
log list [-h]
```

```
log remove [-h] [name]
```

Table 61. Positional arguments

Name	Address
filepath	Specify logging to a file identified by filepath
{out,err,log}	Specify console output using cerr, cout or clog respectively
hostname:port	Specify logging to a socket identified by hostname (default: localhost) and port
name	Remove the logging configuration identified by name or all configurations if no name is provided

Table 62. Optional arguments

Name	Address
-h, --help	Show this help message and exit
-l, --level	Logging level (default: INFO). Possible values: CRITICAL,ERROR,WARNING,INFO,DEBUG

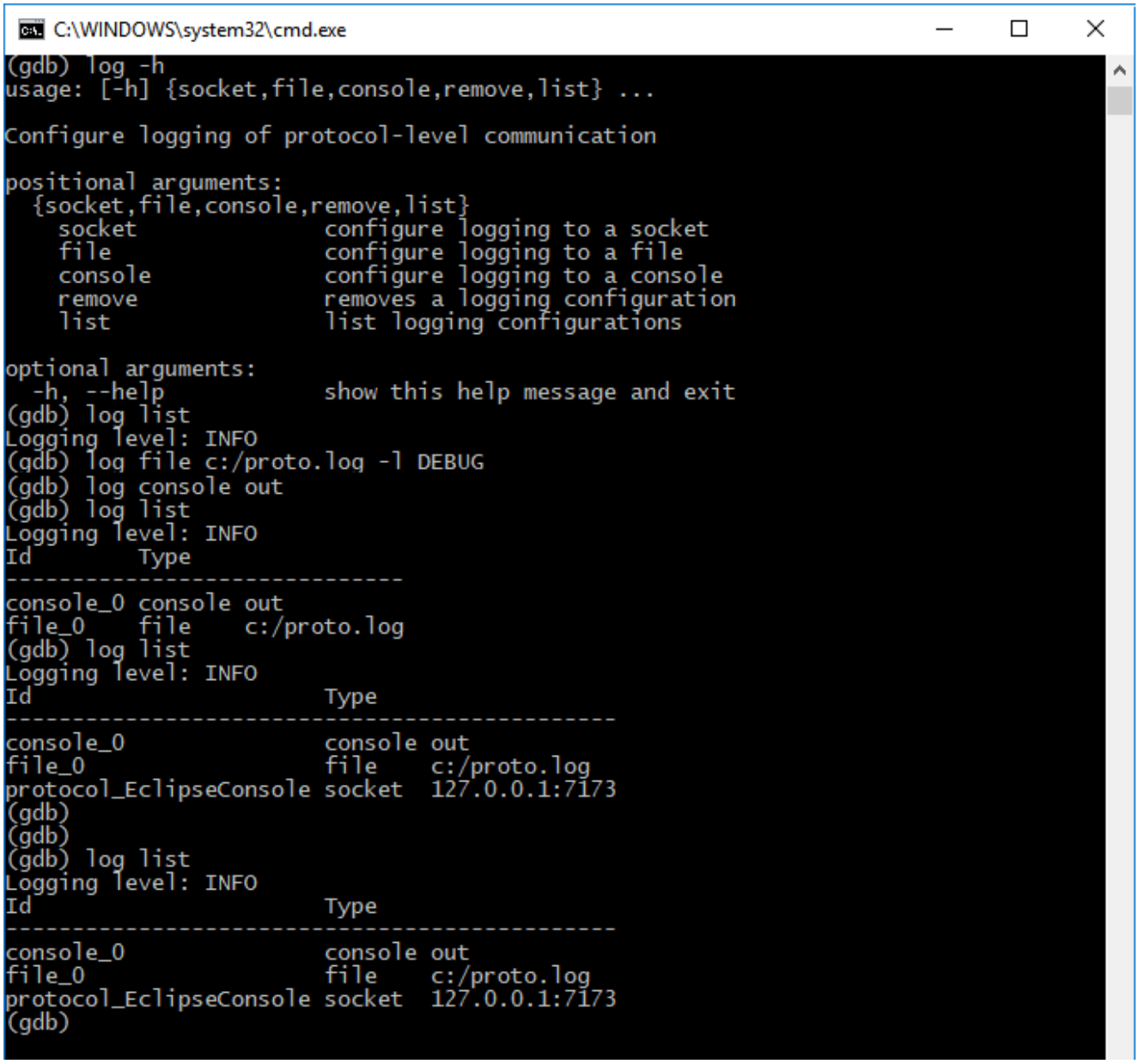
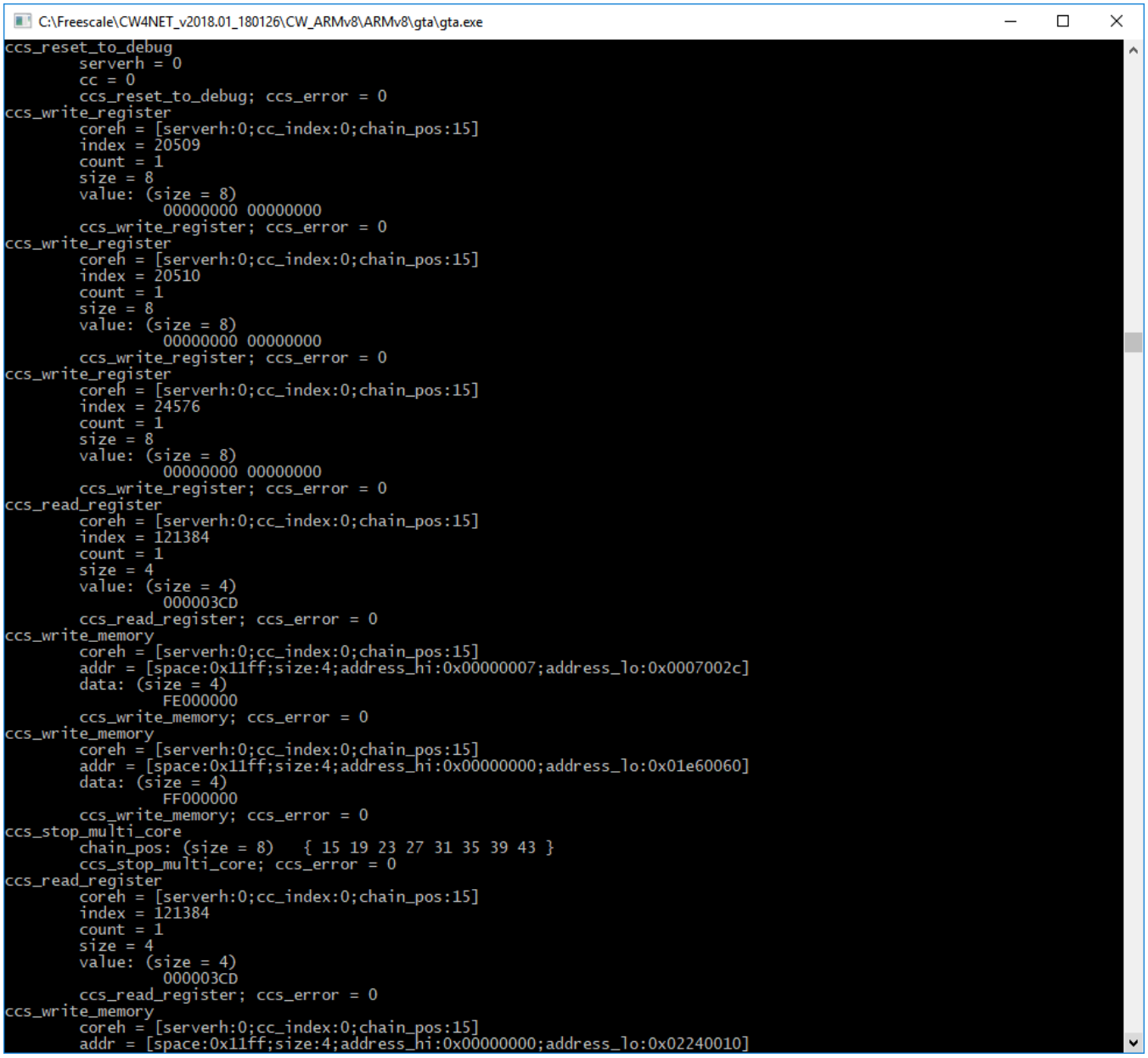


Figure 48. Usage example for `log` command

Figure 49. Output example for `log` command



6.8 Monitor commands

This topic explains monitor commands.

The following table lists the available monitor commands.

Command	Syntax	Description
Display contexts tree	mon ctx id <ctx-id> list	Displays the debug contexts tree having as root the specified context. The context has the format: <connection>:<soc>:<core#no>

Table continues on the next page...

Table continued from the previous page...

Set current context	mon ctx set current ctx_id	Set the context for the debug session. This should be set after target extended-remote and before attach. For a single core application the context should look like: <connection>:<soc>:<core#no>. For a multicore application (SMP) the context should be: <connection>:<soc>
	mon ctx get current	Show the current context
	mon ctx id <ctx-id> info	List all properties of the specified context
	mon ctx id <ctx-id> set <prop-name> <value>	Set a property for the specified context
Reset	monitor reset debug	Performs reset and keeps cores in debug mode.

6.9 I/O support

Librarian I/O model is divided into 2 modes.

Librarian I/O model is divided into 2 modes:

- UART_C_Static_Lib_Bare: `printf` support through UART port.
- simrdimon: I/O operations through debugger console.

NOTE

These libraries are compiled by using the highest optimize level for speed (-O3) and no debug data (no DWARF information). The user can recompile these libraries to change the compiler options and use the new libraries in their projects. Projects for these library are located at *{CW_ARMv8}\ARMv8\CodeWarrior_Examples*

There are two examples in ARMv8 Stationery wizard:

- C (HelloWorld_C_Base)
- C++ (HelloWorld_CPP_Bare)

The default I/O mode is debugger console; in other words the simrdimon library is used. The user can verify the status by looking at the **Other linker flag** text box, which contains `--specs="${ProjDirPath}/lib/simrdimon.specs"`. Navigate to **Cross ARM C** (or C++) **Linker** > **Miscellaneous** from the left pane under **Tool Settings** tab, to see **Other linker flag** text box.

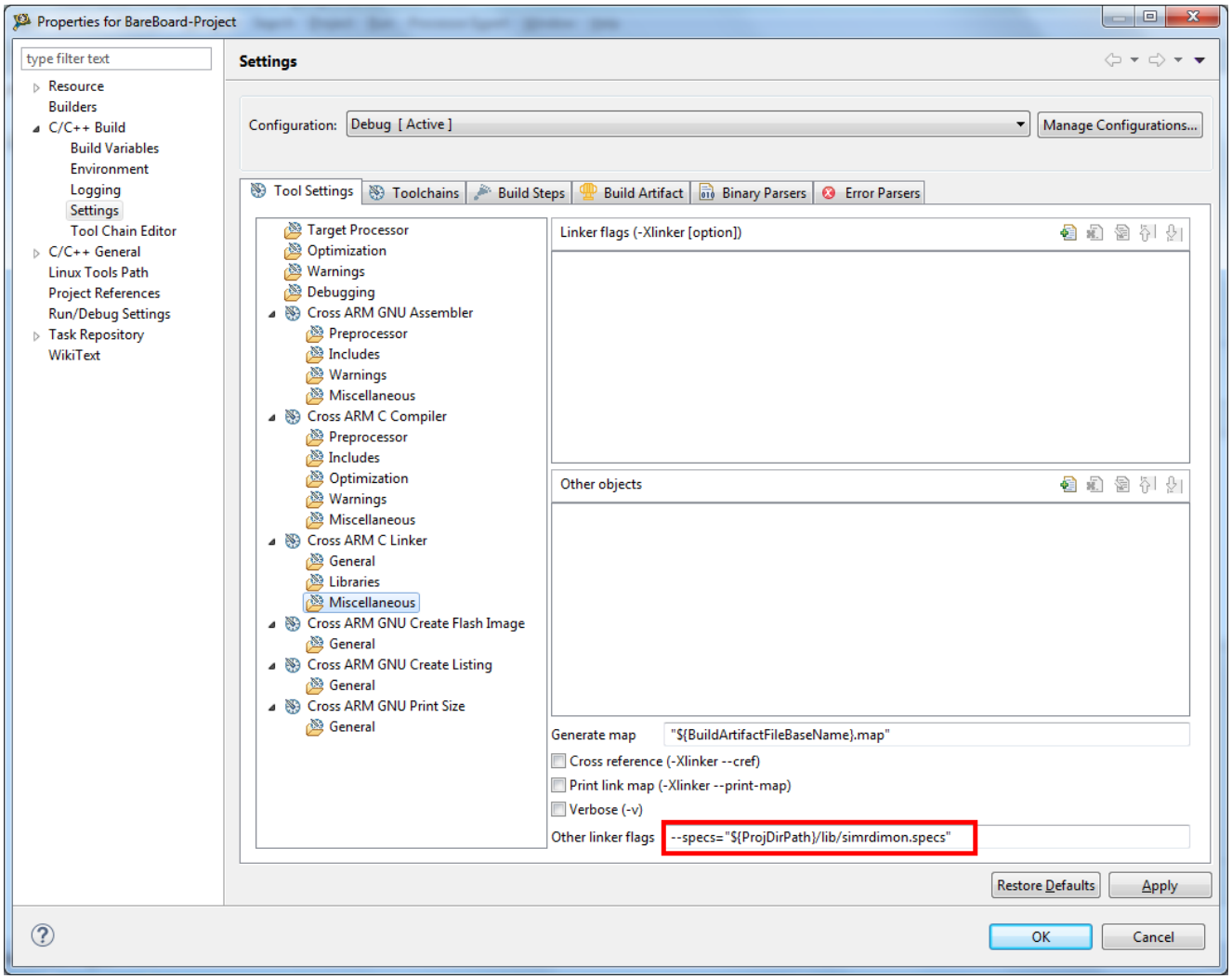


Figure 50. Properties dialog - simrdimon.specs

The user can switch to the I/O UART model by changing the file spec for UART model. The user should replace the *simrdimon.specs* with *uart.specs* in the **Other linker flags** text box from **Cross ARM C (or C++) Linker-- > Miscellaneous**.

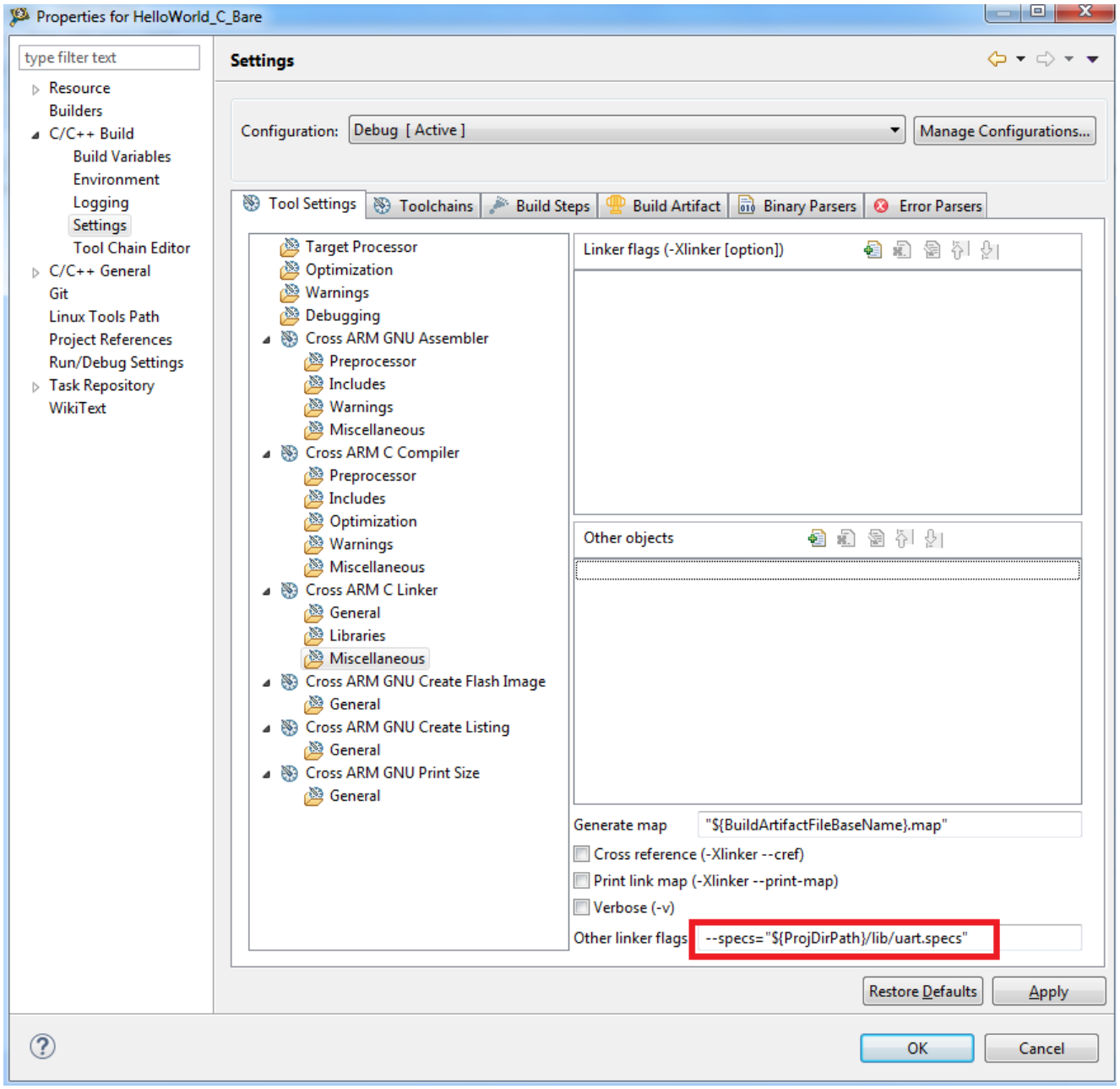


Figure 51. Properties dialog - uart.specs

Chapter 7
Flash Programmer

Flash programming is done using python script.

```
{CW Install Dir}\CW_ARMv8\ARMv8\gdb_extensions\flash\cwflash.py
```

7.1 Configuring flash programmer

To configure the flash programmer, open the `cwflash.py` script in an editor and modify the connection parameters in accordance to your setup.

- BOARD_TYPE – supported options are “QDS” and “RDB” for the corresponding board types.
- FLASH_TYPE – supported options are “nor” and “nand”. Please take into account that some device types may not be supported for the selected board.

  First two options should be sufficient for most of the use cases (CodeWarrior TAP connected through USB to GDB host machine). However, if additional configuration is required, please update the next parameters too.
- PROBE_CONNECTION – If empty, it assumes that probe used is CWTAP connected through USB. For Ethernet connection, use an IP address or a hostname, for example `PROBE_CONNECTION = “192.168.0.1”`. For CMSIS-DAP connection, set to `cmsisdap`, for example `PROBE_CONNECTION = “cmsisdap”`.
- SOC_NAME – name of the SoC. For example: LS2080A.
- JTAG_SPEED – JTAG frequency used by debugger to communicate with the target.
- CCS_CONNECTION – IP and port of the CCS instance. If empty and no connection IP and port is available, debugger will automatically start a CCS instance on default connection IP and port, that is `127.0.0.1:41475`.

  If some connection is explicitly specified, for example `127.0.0.1:41476`, debugger will use that already running ccs session.
- GTA_CONNECTION – IP and port of the GTA (GDB server) instance. If empty, debugger will automatically start a GTA instance. If some connection is explicitly specified, for example `127.0.0.1:45000`, flash programmer will use that already running GTA instance.
- GDB_TIMEOUT - Number of seconds to wait for the remote target responses.

7.2 Starting flash programmer

This topic explains steps to start the flash programmer.

To start the flash programmer, perform the following steps:

1. Open a terminal and switch to the following location:

   ```
   {CW Install Dir}\CW_ARMv8\ARMv8\gdb\bin
   ```
2. Start GDB from this location:
   - Windows: Run `aarch64-fsl-gdb.exe`
   - Linux: Run `./aarch64-fsl-gdb`
3. Execute `cwflash.py` script.

   ```
   source ../../gdb_extensions/flash/cwflash.py
   ```

   If the connection is successful, the output is shown as follows:

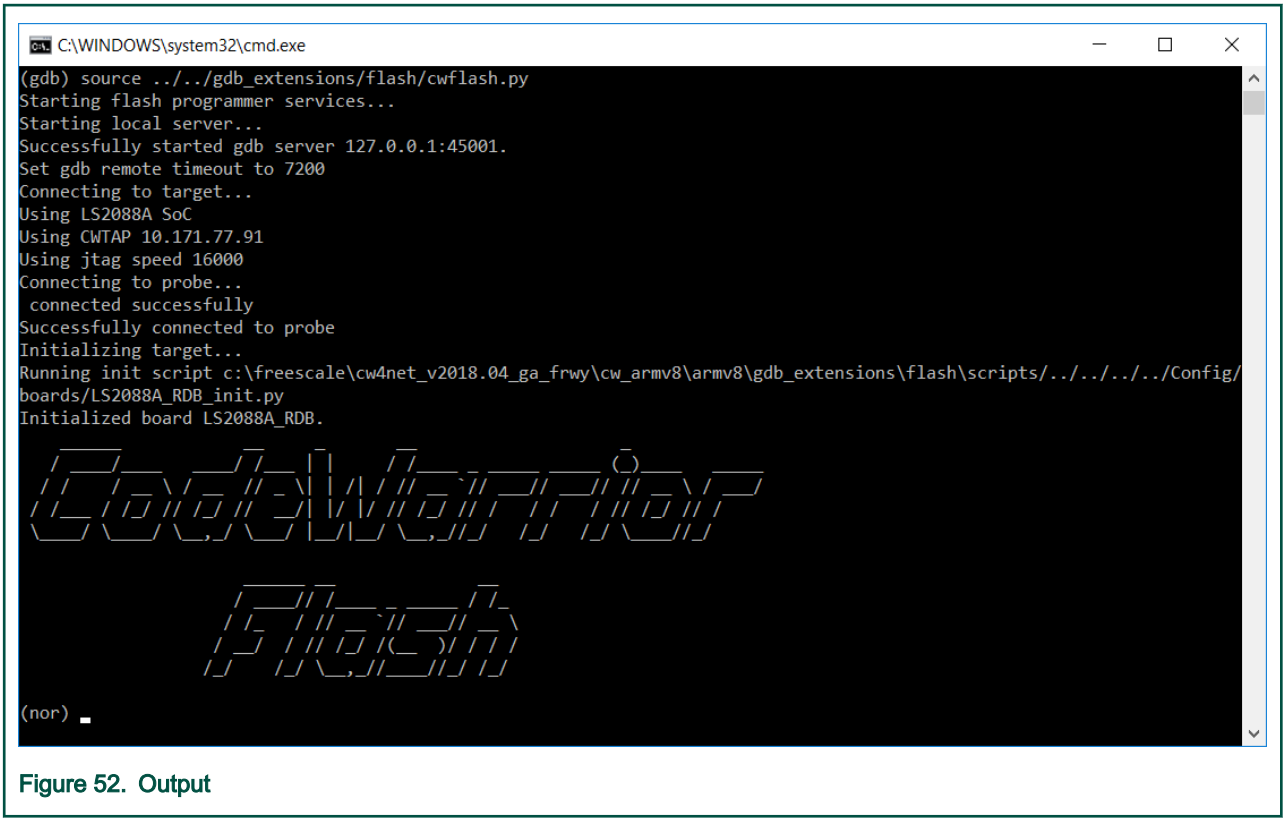


Figure 52. Output

7.3 Using flash programmer

This section explains the operations supported by flash programmer.

- Erase flash memory
- Write binary file in flash memory
- Dump flash memory content
- Protect memory content
- Unprotect memory content
- List supported flash devices
- Associate flash device with board
- Read manufacturer and device ID
- Verify flash memory content

7.3.1 Erase flash memory

This topic explains command to erase an area of flash device.

To erase an area of the flash device, use the following command:

```
fl_erase offset size
```

where:

- <offset>: Specifies the offset inside the device.
- <size>: Specifies the size of the area that will be erased.

For example:

```
fl_erase 0x40000 0x100
```

Type `fl_erase -h` for command help.

7.3.2 Write binary file in flash memory

This topic explains command to write binary file in the flash memory.

To write binary file in the flash memory, use the following command:

```
fl_write offset data  [size]  [--erase]  [--verify   [-n NUMBER] / [-a] ]
```

where:

- <verify>: Performs a verify of the content written in flash.
- <n>: Specifies the number of mismatches shown if verify option is used.
- <a>: Shows all the mismatches if verify option is used.

For example:

```
fl_write 0x40000 u-boot.bin --erase --verify -n 4
```

Type `fl_write -h` for command help.

NOTE

The path to binary file must not contain spaces.

7.3.3 Dump flash memory content

This topic explains command to dump the contents of the flash memory.

To dump the contents of the flash memory, use the following command:

```
fl_dump offset size [-f FILE] [-c {1, 2, 4, 8, 16}]
```

where:

- <offset>: Specifies the offset inside the device.
- <size>: Specifies the size of data to be read.
- <-f>: Specifies the path to the file where the data will be saved.
- <-c>: Specifies the number of bytes per cell. It is incompatible with –f options since it applies only when the output is shown directly in the console.

For example to dump the content in a binary file:

```
fl_dump 0x40000 0x20000 -f dump.bin
```

To dump the content in the console:

```
fl_dump 0x40000 0x20000
```

If [–f FILE] file option is not present the content will be displayed in the console.

NOTE

The path to binary file must not contain spaces.

7.3.4 Protect memory content

This topic explains command to protect an area of flash device.

To protect an area of the flash device, use the following command:

```
fl_protect offset size
```

where:

- <offset>: Specifies the offset inside the device
- <size>: Specifies the size of the area that will be protected

For example:

```
 fl_protect 0x100000 0x100
```

Type `fl_protect -h` for command help.

7.3.5 Unprotect memory content

This topic explains command to unprotect an area of flash device.

To unprotect an area of the flash device, use the following command:

```
fl_unprotect offset size
```

where:

- <offset>: Specifies the offset inside the device
- <size>: Specifies the size of the area that will be unprotected.

For example:

```
 fl_unprotect 0x100000 0x100
```

Type `fl_unprotect -h` for command help.

7.3.6 List supported flash devices

This topic explains command to list all supported flash devices.

To list the devices, use the following command:

```
fl_list
```

7.3.7 Associate flash device with board

This topic explains command to specify that a device is available on current board.

To specify that a device is available on the current board, use the following command:

```
fl_device [-h] [-al ALIAS] -n NAME -a ADDRESS -wa WADDRESS -ws WSIZE -g GEOMETRY -c CONTROLLER [-d DIE]
```

positional arguments:

- -n, --name - name of the device; must match one of the supported devices (see "fl_list" command)
- -a, --address - address of the device
- -wa, --waddress - address where the workspace will be located; the workspace is the area where the flash programmer algorithm will be downloaded

- -ws, --wsize - size of the workspace
- -g, --geometry - number of words per row
- -c, --controller - controller used to interact with the device
- -d, --die - index of the die in case the device has multiple stacked dies

optional arguments:

- -al, --alias - alias that will be used as context name; if an alias is not specified, the name will be used instead
- -h, --help - show this help message and exit

7.3.8 Read manufacturer and device ID

This topic explains command to read the manufacturer and device ID for the current device.

To read the manufacturer and device ID for the current device, use the following command:

```
fl_id
```

7.3.9 Verify flash memory content

This topic explains command to compare flash memory content against a file on the disk or data provided by the user.

To compare flash memory content with a file on the disk or data provided by the user.

```
fl_verify [-h] [-s [SIZE]] [-n NUMBER] [-a] offset data [data ...]
```

positional arguments:

- offset - offset in device address range
- data - data to be verified in flash; can be a hex sequence or a binary file

optional arguments:

- -h, --help - show this help message and exit
- -s [SIZE], --size - [SIZE] how much to verify
- -n [NUMBER], --number [NUMBER] - number of mismatches shown
- -a, --all - show all the mismatches

7.4 Switch current device used for flash programming

This topic explains command to switch current device used for flash programming.

To switch the current device used for flash programming, use the following command:

```
fl_current flash_type
```

For example:

```
fl_current nor
```

NOTE

If the command succeeds, the output appears as shown in Figure. Output.

7.5 SD/eMMC flash programmer

The SD/eMMC card uses the notion of blocks, not addresses.

One block has 512 bits. When you want to write something, for example, from block 4, you need to compute the address by multiplying the number of the block with 512 and converting the result into hexadecimal (for example, sector 4 x 512 = 2048 = 0x00000800).

By default, the algorithm is set to run from DDR, because it uses direct memory access (DMA) for data transfer. If the DDR memory is not functional:

- In function `Config_Flash_Devices` in the initialization file, change the value for `ws_address` in the `gdb` command, which adds your SD/eMMC device, with the address of the OCRAM.
- Also, comment the call of the DDR initialization function.

This will enable the use of the internal buffer of the eSDHC controller, which will reduce the performance, as DMA is not used in this mode.

7.6 Viewing details about flash device

This topic explains command to view details about currently selected flash memory device.

To view details about currently selected flash memory device, use the following command:

```
fl_info
```

7.7 Using flash programmer from eclipse IDE

You can also use the Flash Programmer features from the CodeWarrior IDE.

The Flash Programmer GUI provides the option to view all devices that can be used for flash programming and select such a device. Information about the devices are also available and can be displayed via a tooltip.

The flash programmer commands, such as erase, write binary, dump memory content, protect memory content, unprotect memory content, can be defined and added into a sequence.

Any command in the sequence or the command order may be later modified as needed, and these command sequences can be executed. The sequence can also be imported and exported thus allowing sharing between users.

For viewing the output of the flash programmer commands, an output console area exists at the bottom of the dialog window.

7.7.1 How to open CodeWarrior flash Programmer window

Flash Programmer can be started from the *Target Connection Configuration* view or from the *Debug* view.

The following figure shows the **Flash Programmer** icons in the **Debug** view and the **Target Connection Configuration** view.

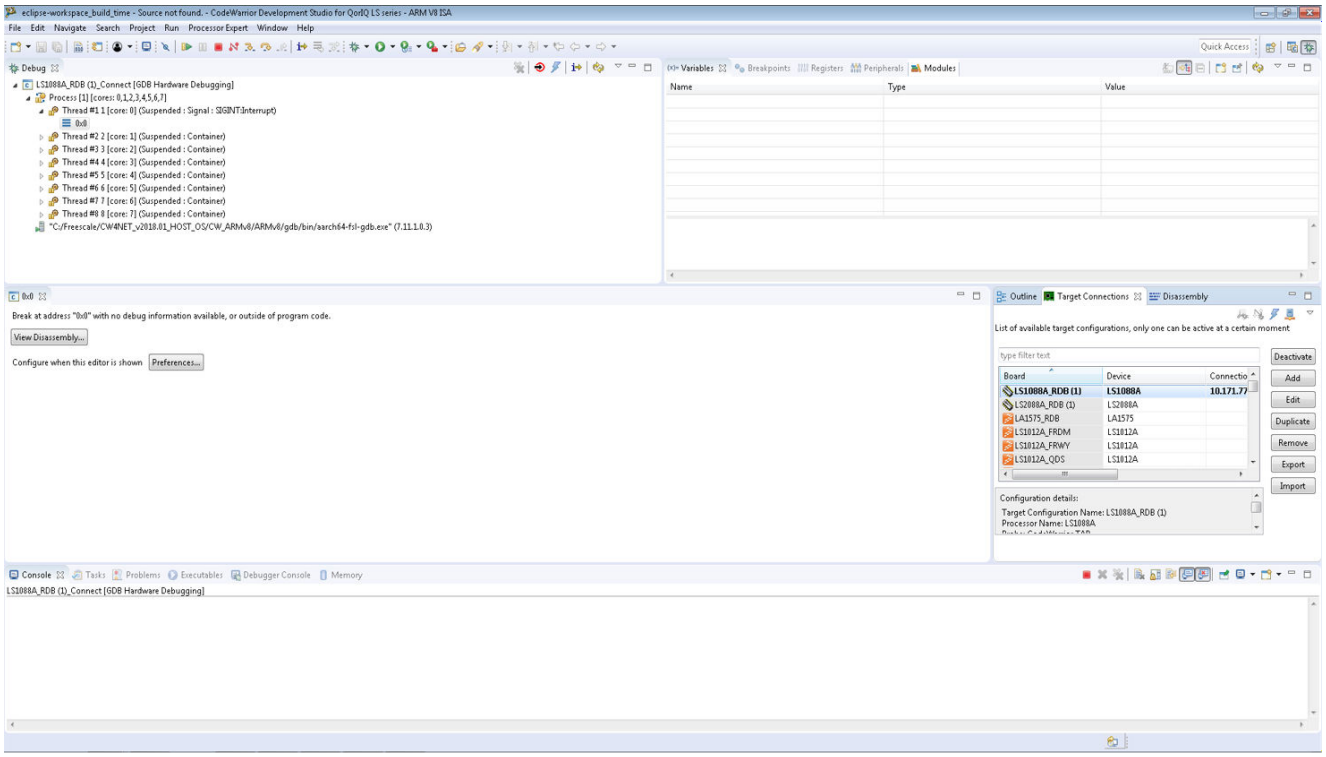


Figure 53. Open Flash Programmer window

The “Flash Programmer” icon in the **Debug** view toolbar is enabled only when a debug session is active and selected. In this case, the debug session connection is used by Flash Programmer.

On clicking the **Flash Programmer** icon, a pop-up appears asking if you wish to continue. Click **OK** if you want the selected debug session to be used by Flash Programmer. The debug sessions terminates when the Flash Programmer is closed.

You can also open the Flash Programmer by clicking the **Flash Programmer** icon from the **Target Connection Configuration** view toolbar. This option has the advantage that it doesn’t require to have an existing debug session and the connection to the target is handled automatically.

This toolbar option is always enabled and uses the selected target configuration (or active target configuration if none is selected) to establish a connection to the target for using Flash Programmer. In most of the cases, a new debug session is started with the settings from the used target configuration. However, if a debug session using that target configuration already exists, the existing debug session is re-used for the Flash Programmer.

7.7.2 Device selection and information

The following figure shows the CodeWarrior Flash Programmer dialog.

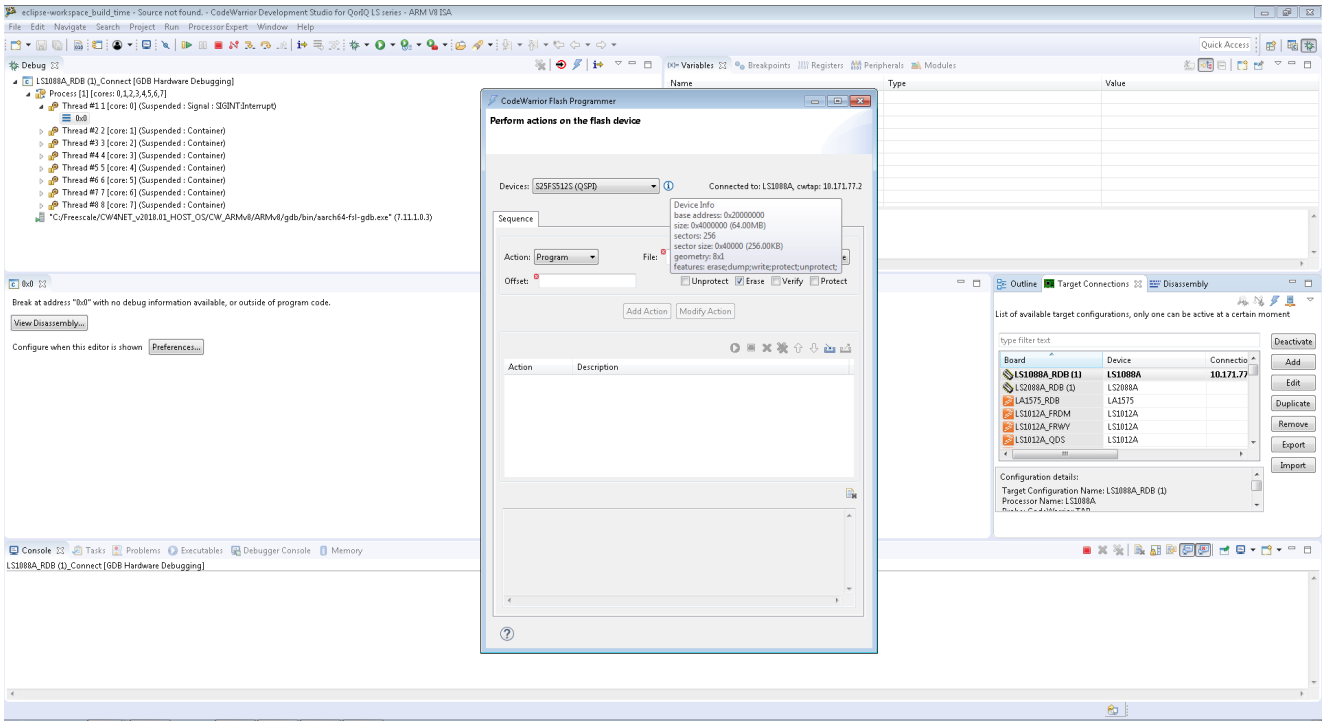


Figure 54. Flash programmer dialog

The **Devices** drop-down list in the **Flash Programmer** dialog lists all the available flash programmer devices. The used device can be changed by selecting a different option from the list.

For more information about the device, place the mouse over the icon at the right side of the **Devices** list. Additional information about the connection to the target is displayed at the right side of the list.

7.7.3 Manage a flash programmer sequence

A sequence is formed of one or more actions (flash programmer commands).

To create such a sequence, you must define correct actions and add them by clicking **Add Action**.

The **Add Action** button is enabled when the action is valid – that is all the mandatory fields are correctly set.

To modify an action from the sequence, select it. Its values are populated in the Action area, modify the values, and press **Modify Action** to save the changes. Now the action selected from the sequence is modified.

To change the order of the actions in the sequence, use the up and down toolbar items in the sequence table. In the same toolbar, there are also buttons for deleting the selected action in the entire sequence. Also, an action may be enabled or disabled by double clicking it.

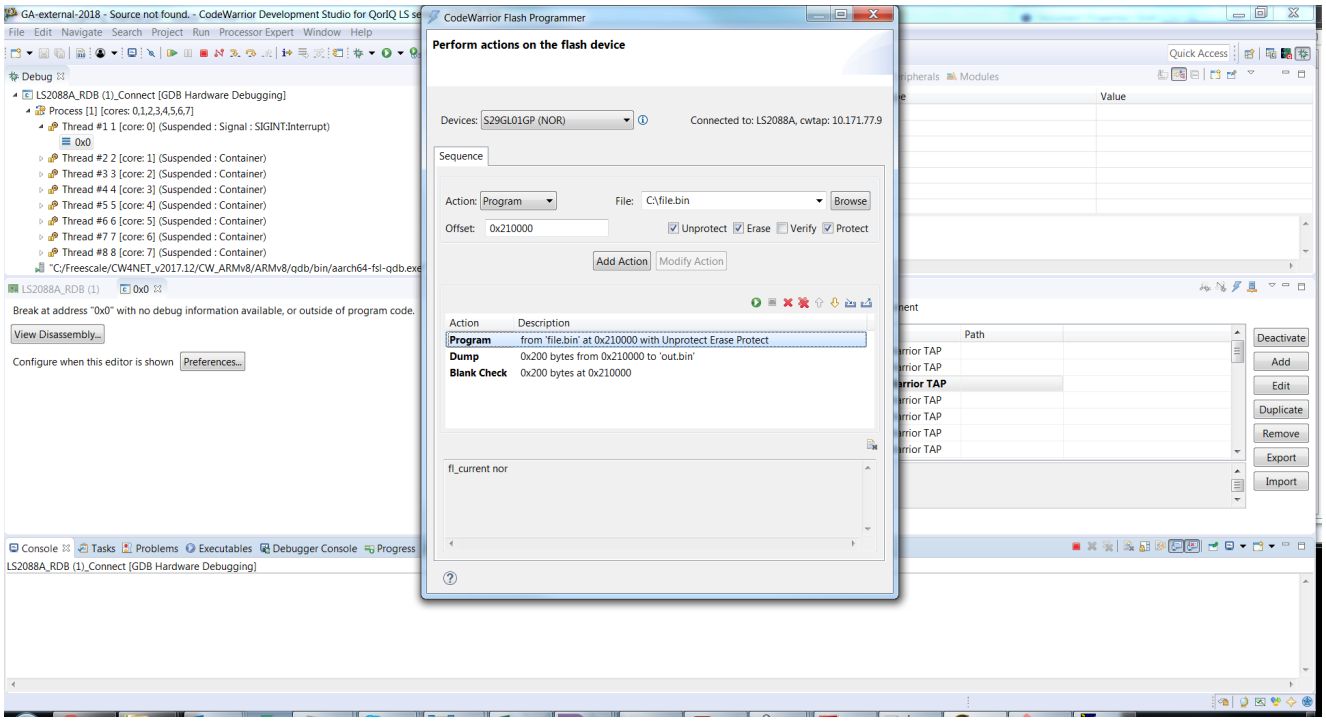


Figure 55. Flash programmer sequence

7.7.4 Launch a flash programmer command sequence

After a sequence has been created it can be executed by clicking the **Execute sequence** comand in the toolbar at top of the sequence table.

Information or errors from the flash programmer execution are displayed in the output console in the Flash Programmer dialog. Actions that are executed successfully are marked with green checkmarks after the execution is completed.

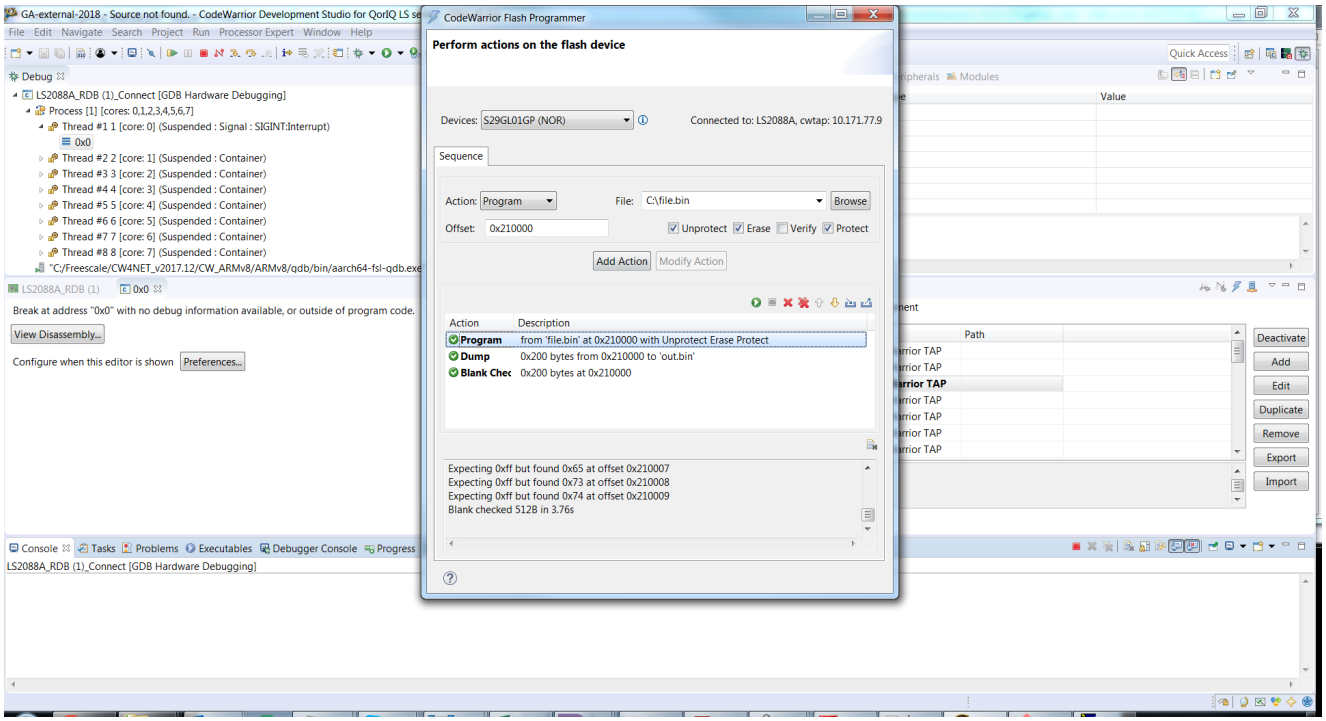


Figure 56. Executing command sequence

If an action fails, the execution of the sequence stops, the failed actions is marked and error details are displayed in the output console.

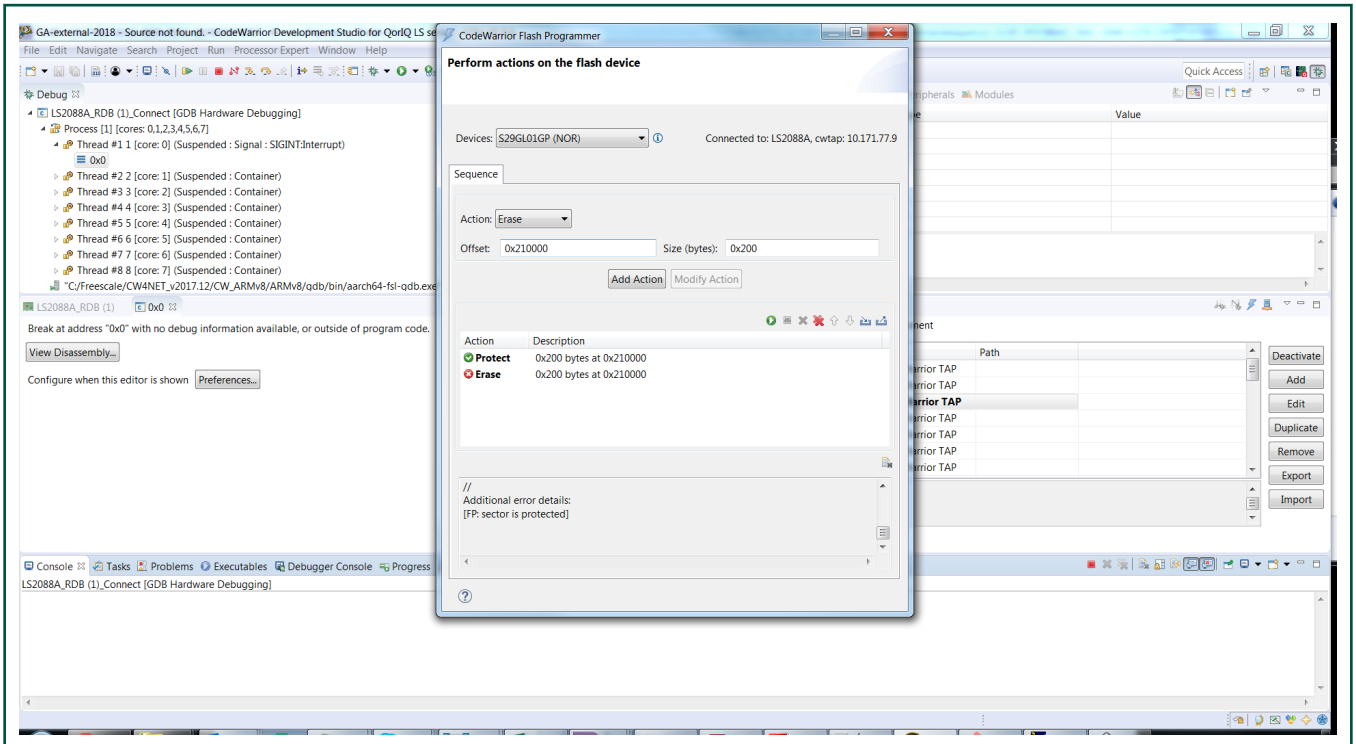


Figure 57. Errors in command sequence

7.7.5 Import export sequence

Users can share a flash programming sequence by using the **Import** and **Export** options in the toolbar.

On clicking **Export**, the sequence and the selected device name is exported in an .xml file.

Importing a sequence restores the sequence in the Flash Programmer GUI. If the device from the imported file exists, that device becomes the selected device, otherwise a warning is displayed to the user. If the user chooses to proceed, then the attempt to launch the imported sequence may result in flash programmer execution errors.

Chapter 8
Use Cases

This chapter explains U-Boot debug, Linux application debug, Linux kernel debug, UEFI debug, and AMP example projects.

This chapter lists:

- U-Boot debug
- Linux application debug
- Linux kernel debug
- UEFI debug
- Import and configure AMP example projects

8.1 U-Boot debug

This topic describes the steps required to perform a U-Boot debug using CodeWarrior Development Studio for QorIQ LS series - ARM V8 ISA.

This topic lists the steps to:

- Build the U-Boot sources and the auxiliary tools.
- Perform U-Boot debug in CodeWarrior Development Studio for QorIQ LS series - ARM V8 ISA.

NOTE

For details on how to prepare the target, see Preparing Target.

8.1.1 U-Boot setup

This topic explains U-Boot build.

For details on U-Boot build, refer to the SDK documentation or the LSDK documentation as per the build system used. Yocto-based SDK uses bitbake commands to build various packages, whereas Dash/LSDK is based on flex-builder and flex-installer toolset.

8.1.2 Create an ARMv8 project for U-Boot debug

This topic explains steps to create an ARMv8 bare metal project for U-Boot debug.

To create an ARMv8 bare metal project for U-Boot debug, perform these steps:

1. Open CodeWarrior for ARMv8.
2. Import a U-Boot image as described in CodeWarrior Executable Importer wizard. In case an SPL-based U-Boot is being debugged, make sure the main/DDR U-Boot elf file is imported at this step.
3. Select **Run > Debug Configurations** to open the Debug Configurations dialog.
4. Click on the **Startup** tab.
    a. Set breakpoint at: _start.
    b. Select the **Resume** checkbox.

NOTE

Step (b) should be done only if nothing is running yet on the target board, or in case you have just started the target board, but have not started U-Boot. However, in case you simply attach it to a running U-Boot session the above step should be skipped. PC will reflect the current PC while U-Boot is running.

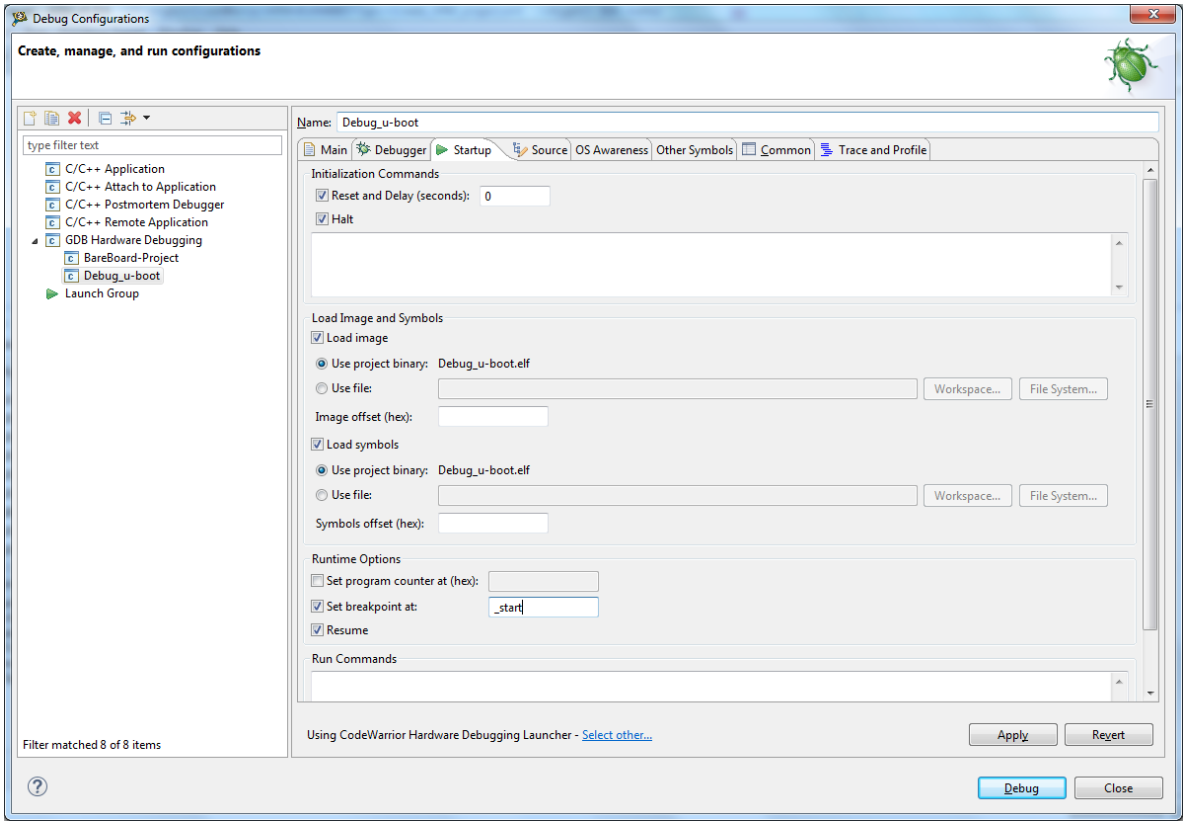


Figure 58. Startup tab

5. If a NAND/SD U-Boot is debugged, the SPL U-Boot ELF file should be specified in the **OS Awareness** tab, as shown in the following figure.

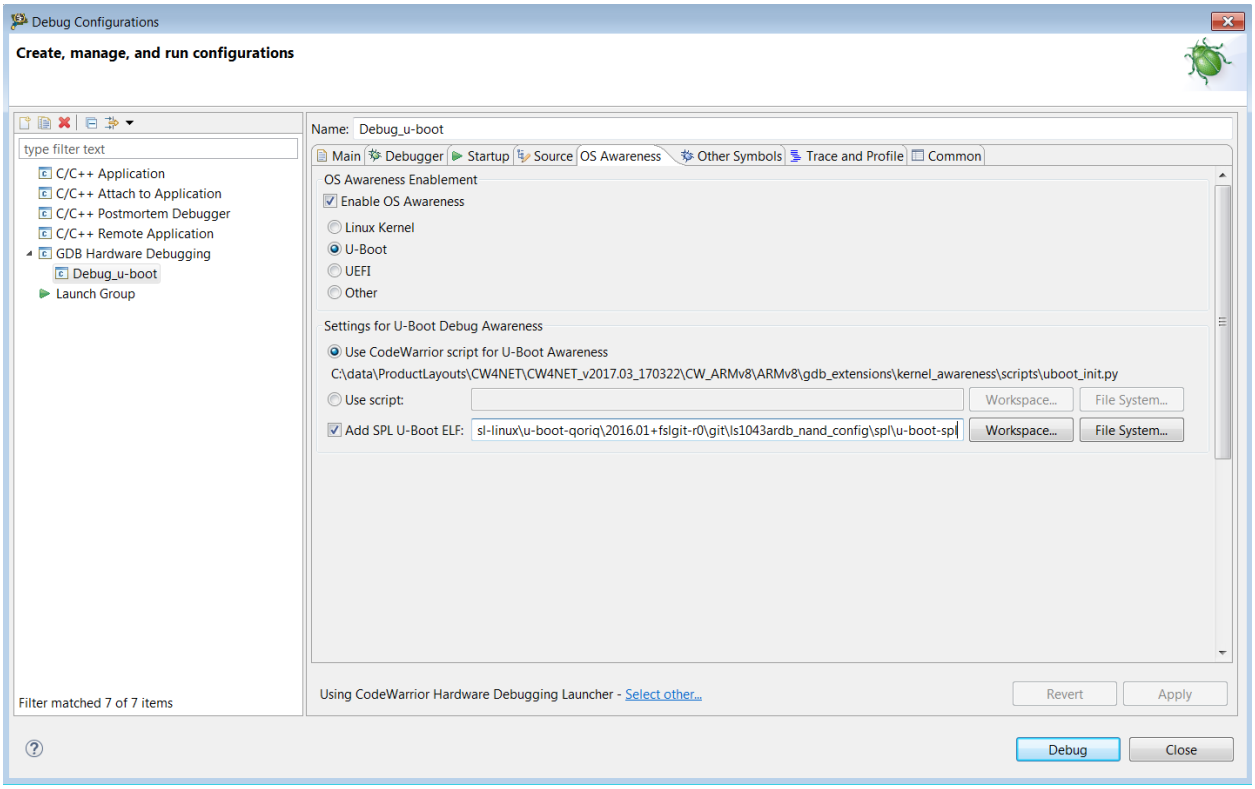


Figure 59. Add SPL U-Boot ELF

6. Set up the target connection configuration, as explained in Configuring Target.
7. Click the **Debug** button to initiate the debug session. The debugger should stop at _start symbol.

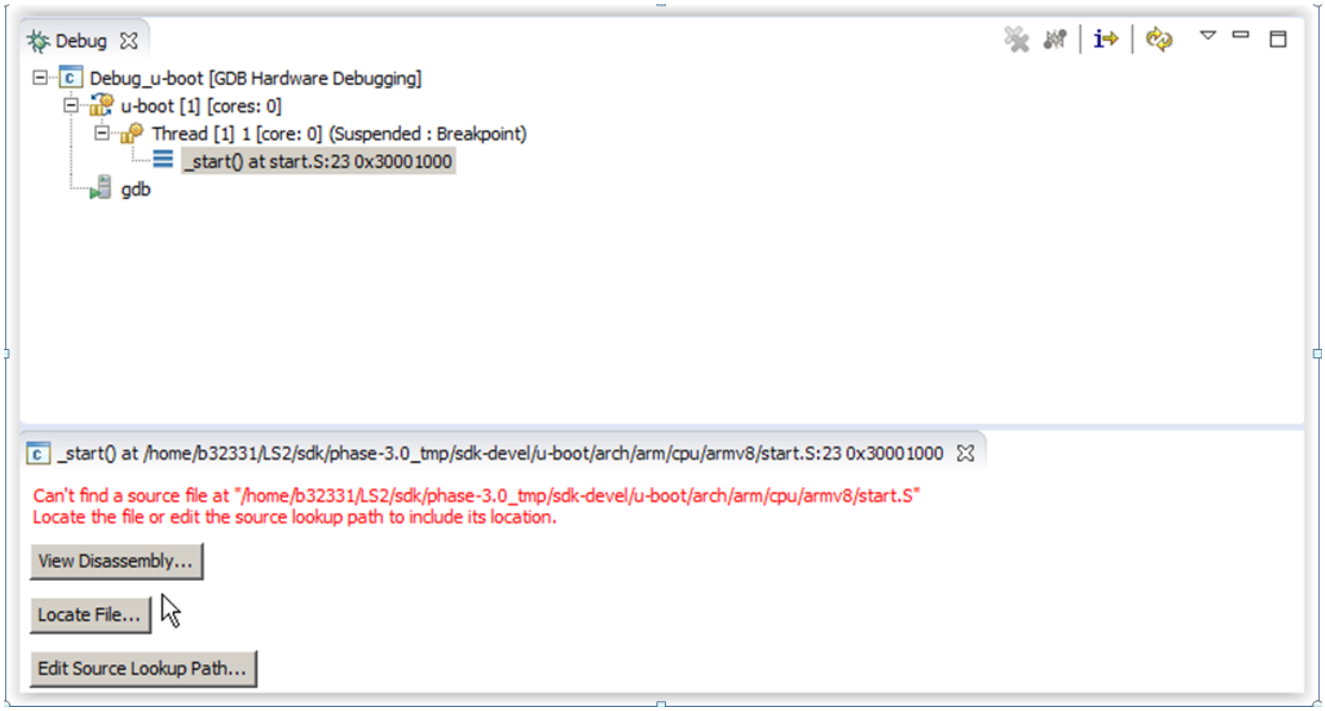


Figure 60. Debugger stops at _start symbol

8.1.3 U-Boot debug support

This section explains steps to perform U-Boot debug in CodeWarrior Development Studio for QorIQ LS series - ARM V8 ISA.

8.1.3.1 Setting the source path mapping

This topic explains steps to load symbols and set source path mapping.

To load symbols and set source path mapping, perform these steps:

1. Locate the file suggested by the debugger.

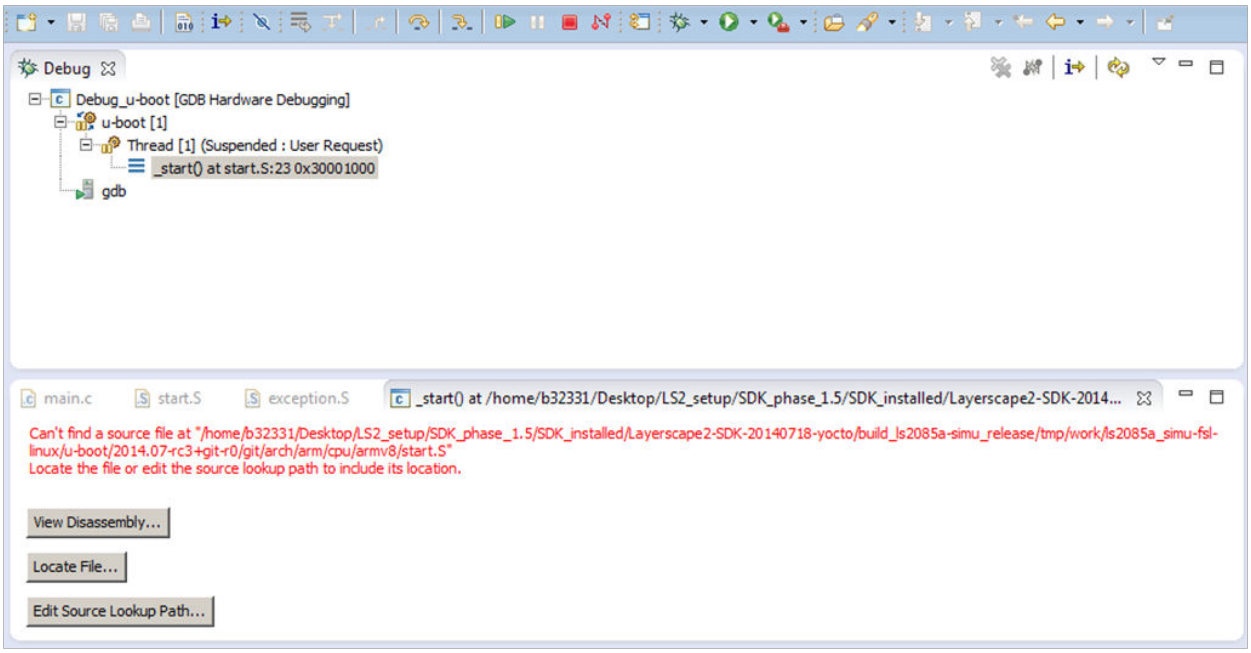


Figure 61. Locate source

2. The stack and the source views appears as in the following figure.

NOTE

You can add a static map entry using the Edit Source Lookup Path button to avoid locating file using the Locate File button, whenever a new file is requested.

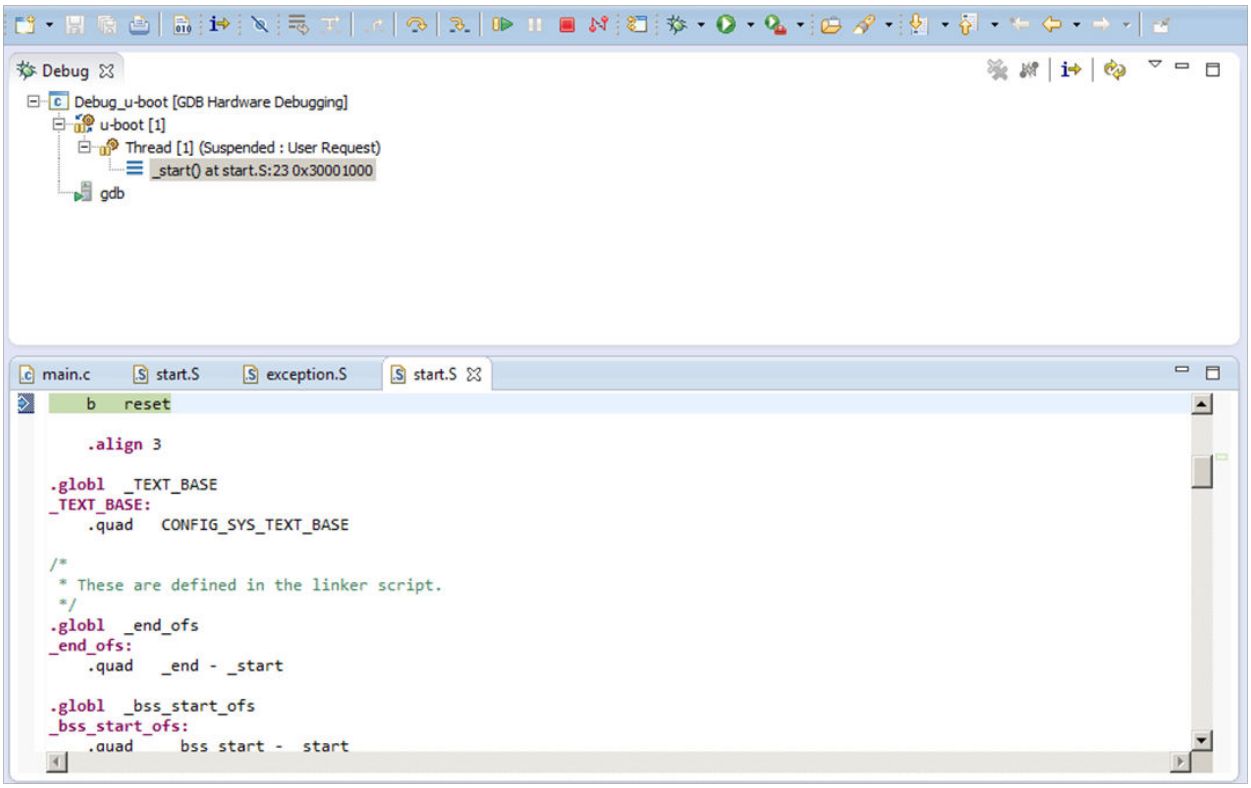


Figure 62. Stack and sources

3. Click the **Resume** button. Alternatively, press the F8 key.
4. If a new U-Boot debug session is required, close/terminate the actual connection or use the Reset button in the Debug view to reset the target.

NOTE

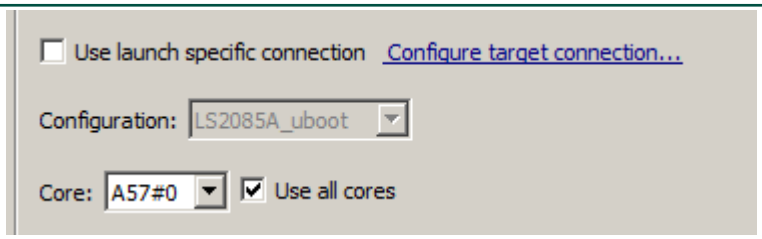
If you want to attach to the same U-Boot session, disconnect the CodeWarrior software and reconnect again. You will not need to set the PC and the path mapping is correct.

8.1.3.2 Debug capabilities

This topic explains steps to bring-up the U-Boot.

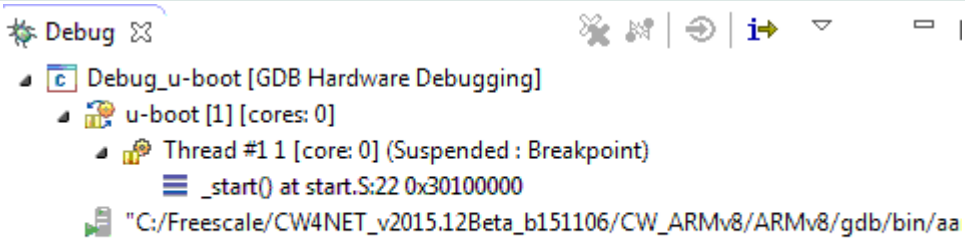
1. The multicore debug is also supported if you want to inspect the secondary cores after release in the last stage.
   a. Select the **Use all cores** checkbox in the **Debugger** tab.

Figure 63. Debugger tab



b. When the debugging starts, you can see stack/registers for every core. Note that the run control is per SoC and not per core.

Figure 64. Debug view



2. If an SPL U-Boot is debugged, make sure the SPL U-Boot ELF is specified in the **OS Awareness** tab, as shown in the following figure.

Figure 65. Add SPL U-Boot ELF

3. Double-click a line to inspect breakpoints. You can inspect these using:
   - **Breakpoints** view
   - `info breakpoints` command from the GDB shell.
4. You can perform the step operations until the U-Boot starts.

8.2 Linux application debug

This document describes the steps required to perform Linux Application Debug using CodeWarrior Development Studio for QorIQ LS series - ARM V8 ISA.

This document lists the steps to:

- Build the Linux sources and auxiliary tools
- Networking support
- Perform Linux application debug in CodeWarrior Development Studio for QorIQ LS series - ARM V8 ISA
- Attach to a Linux application in CodeWarrior Development Studio for QorIQ LS series - ARM V8 ISA

8.2.1 Linux setup

This topic explains Linux setup.

For details on Linux setup, refer to the SDK documentation or the LSDK documentation as per the build system used. Yocto-based SDK and Dash/LSDK use different build systems and different approaches for deploying images on target.

8.2.2 Debugging simple Linux application

This topic explains how to create a simple Linux application project, update remote connection, enable full debug support, and debug the Linux application project.

- Creating simple Linux application project
- Updating remote connection
- Using sysroot
- Debugging Linux application project

8.2.2.1 Creating simple Linux application project

This topic explains steps to create a ARMv8 Linux application project.

To create a new ARMv8 Linux application project:

1. Open CodeWarrior Development Studio for QorIQ LS series - ARM V8 ISA.
2. Select **File > New > ARMv8 Stationery > Linux Application Debug > Hello World C Project**.
3. Specify a project name.
4. Click **Finish**.
5. Select the newly created Linux application project in the **Project Explorer** view.
6. Select **Project > Build project**.

To create a ARMv8 Linux application project using an existing Linux application image, see CodeWarrior Executable Importer wizard.

8.2.2.2 Updating remote connection

This topic explains how to change the settings in a default remote connection.

The IP/hostname and the SSH port of the Linux target must be set to the correct values.

To change the default values perform the following steps:

1. Click **Run > Debug Configurations**.

   The **Debug Configurations** dialog appears.
2. Select the Linux application project in the left-hand panel, under **C/C++ Remote Application**.

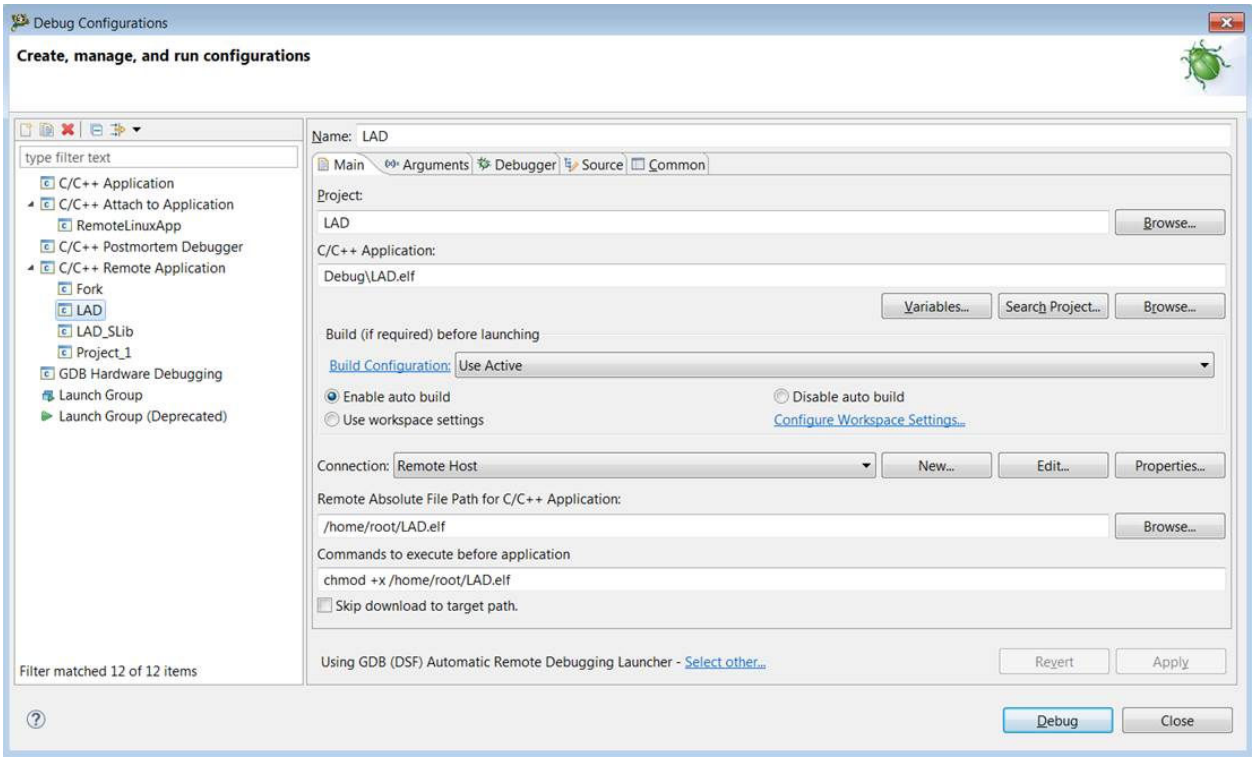


Figure 66. Select Linux application project

3. Click the **Edit** button right next to the default **Remote Host connection**. The **Edit Connection** dialog appears.

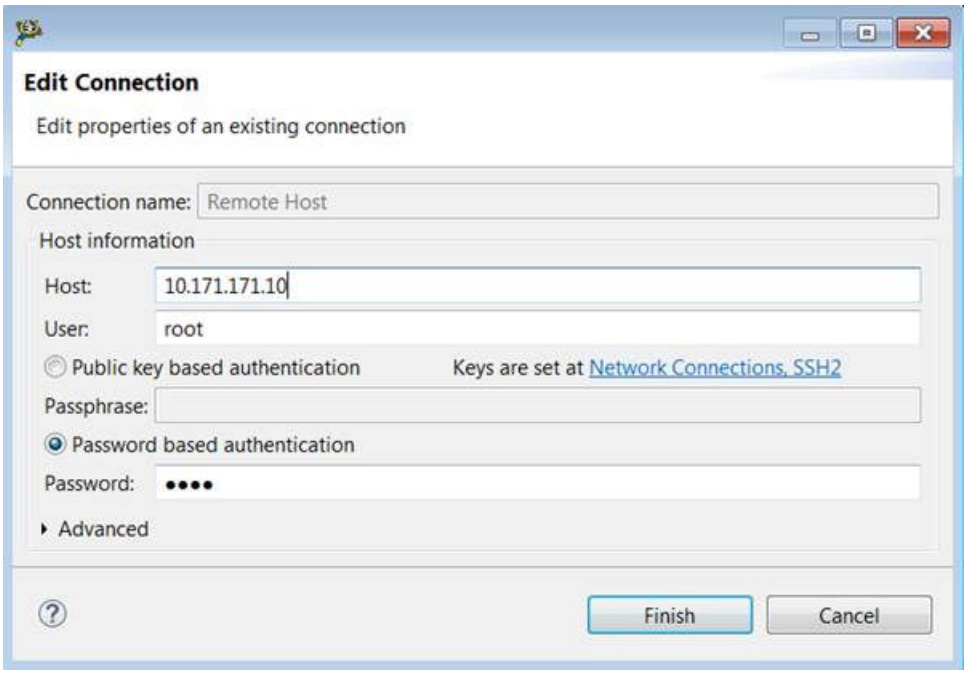


Figure 67. Edit default remote host connection

4. Specify the IP of the Linux target in the **Host** text box, specify the user name in the **User** text box and specify the password in the **Password** text box.

5. Expand the **Advanced** section and configure the SSH port and other connection settings, as needed.

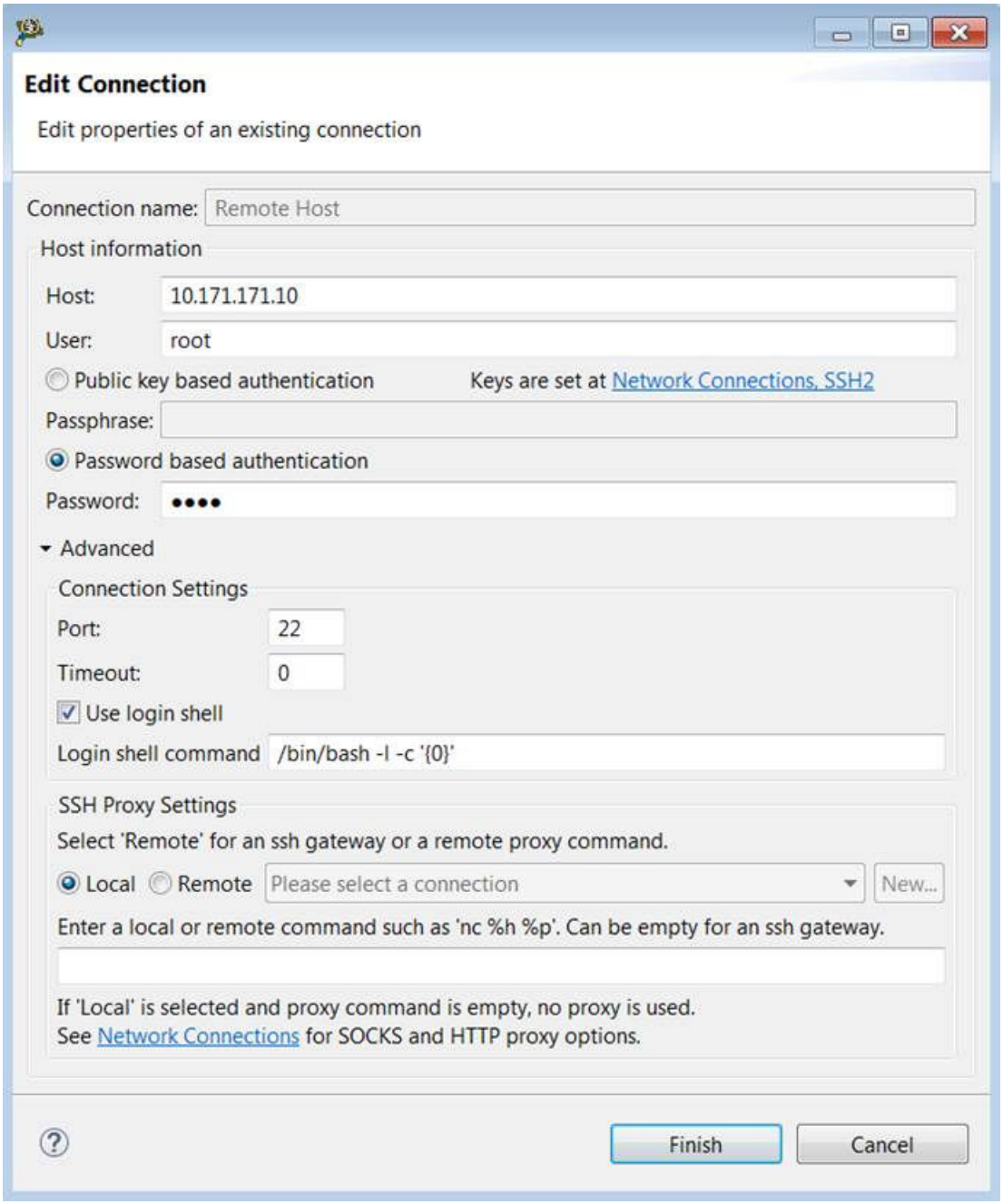


Figure 68. Edit advanced connection settings

6. Click **Finish**.

8.2.2.3 Using sysroot

This section is required only if you want to enable full debug support (inside target system libraries) for the Linux application project.

NOTE

Before you proceed, ensure that you have completed all the steps in Updating remote connection.

To enable full debug support for a Linux application project, perform these steps:

1. GDB should be configured to use the target system libraries.
    a. On the host PC, create a folder `rootfs` and a sub-directory `lib`.
    b. Copy the following libraries: `libc`, `ld`, `libpthread into` to the `rootfs/lib/` folder. Use the full library name as you see it on the target, for example `libpthread.so.0`, `ld-linux-aarch64.so.1`, `libc.so.6`.
    c. Create a `*.gdbinit` file on the file system. For example, `test.gdbinit`
    d. Add following content in the `.gdbinit` file:

    ```
    set sysroot <host_path_to_rootfs>
    ```

    For example, `set sysroot C:\Users\u12345\Desktop\rootfs`

    NOTE

    If you are running the CodeWarrior software on the same Linux machine where you have compiled the SDK package, you can directly set up the sysroot from that location in the gdbinit file. For example, the path to sysroot on a yocto-based sdk would be:

    ```
    set sysroot  /home/u12345/Desktop/SDK_Setup/QorIQ-SDK-V2.0-20160527-yocto/
    build_ls2088ardb/tmp/sysroots/ls2088ardb/
    ```

2. Add missing settings in launch configuration file.
    a. Right-click the project and select **Debug As > Debug Configurations**.

    The **Debug Configurations** dialog appears.
    b. Expand **C/C++ Remote Application**, select the launch configuration for the Linux application project you want to debug.
    c. Click the **Main** sub tab in the **Debugger** tab.
    d. Browse to `*.gdbinit` path in **GDB command file** field.

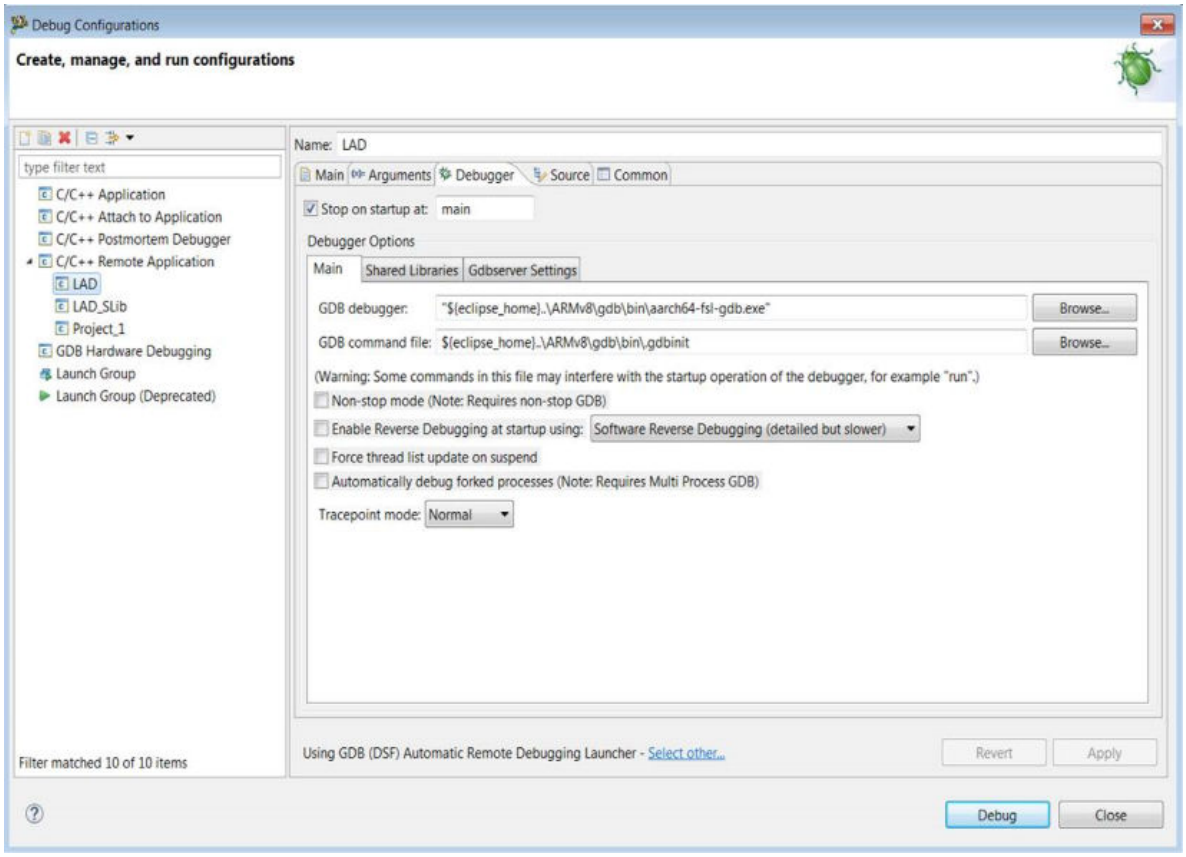


Figure 69. Debugger tab - Main

e. Click **Apply**.

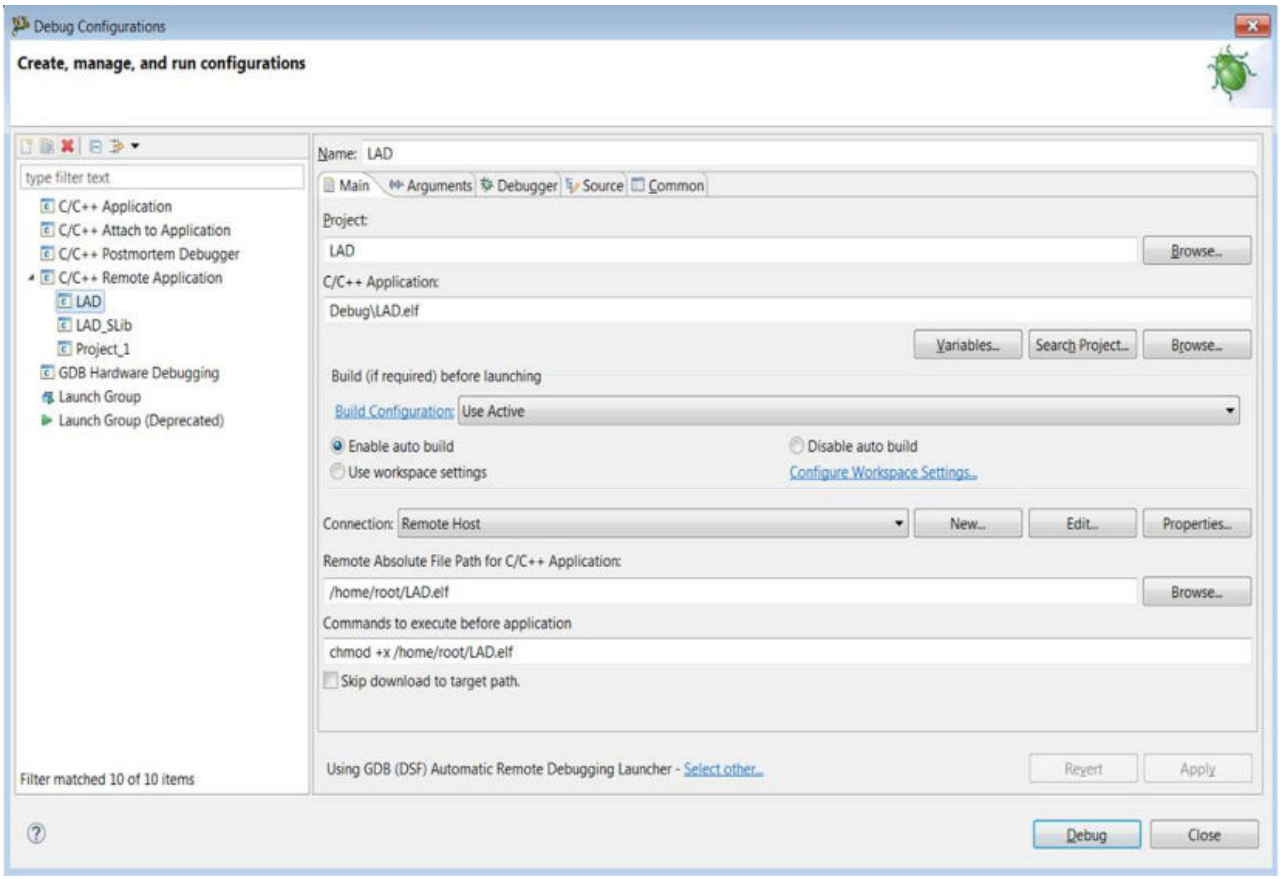
8.2.2.4 Debugging Linux application project

This topic explains steps to debug a Linux application project.

To debug a Linux application project:

1. From the CodeWarrior IDE menu bar, select **Run > Debug Configurations**.

2. In the **Debug Configuration** dialog, expand **C/C++ Remote Application** and select the launch configuration for the Linux application project you want to debug.



3. Click **Debug**.

8.2.3 Debugging a Linux application using a shared library

This topic explains how to debug a Linux application using a shared library.

This topic explains:

- Creating Linux shared library project
- Updating remote connection
- Updating launch configuration for Linux application using shared library
- Debugging Linux shared library project

8.2.3.1 Creating Linux shared library project

This topic explains steps to create an ARMv8 Linux application project using a shared library.

To create an ARMv8 Linux application project using a shared library, perform these steps:

1. Open CodeWarrior Development Studio for QorIQ LS series - ARM V8 ISA.
2. Select **File > New > ARMv8 Stationery > Linux Application Debug > Hello World C Shared Library Project**.
3. Provide a project name.
4. Click **Finish**.
5. Select the project node in the **Project Explorer** view.

6. Build all configurations:

   The project has two build configurations:

   - LibExample - Builds the shared library
   - SharedLibTest (the active configuration) - Uses the shared library

   a. Right-click the project and select **Build Configurations > Set Active > LibExample**.

   b. Build project. The `lib<project_name>.so` library is created.

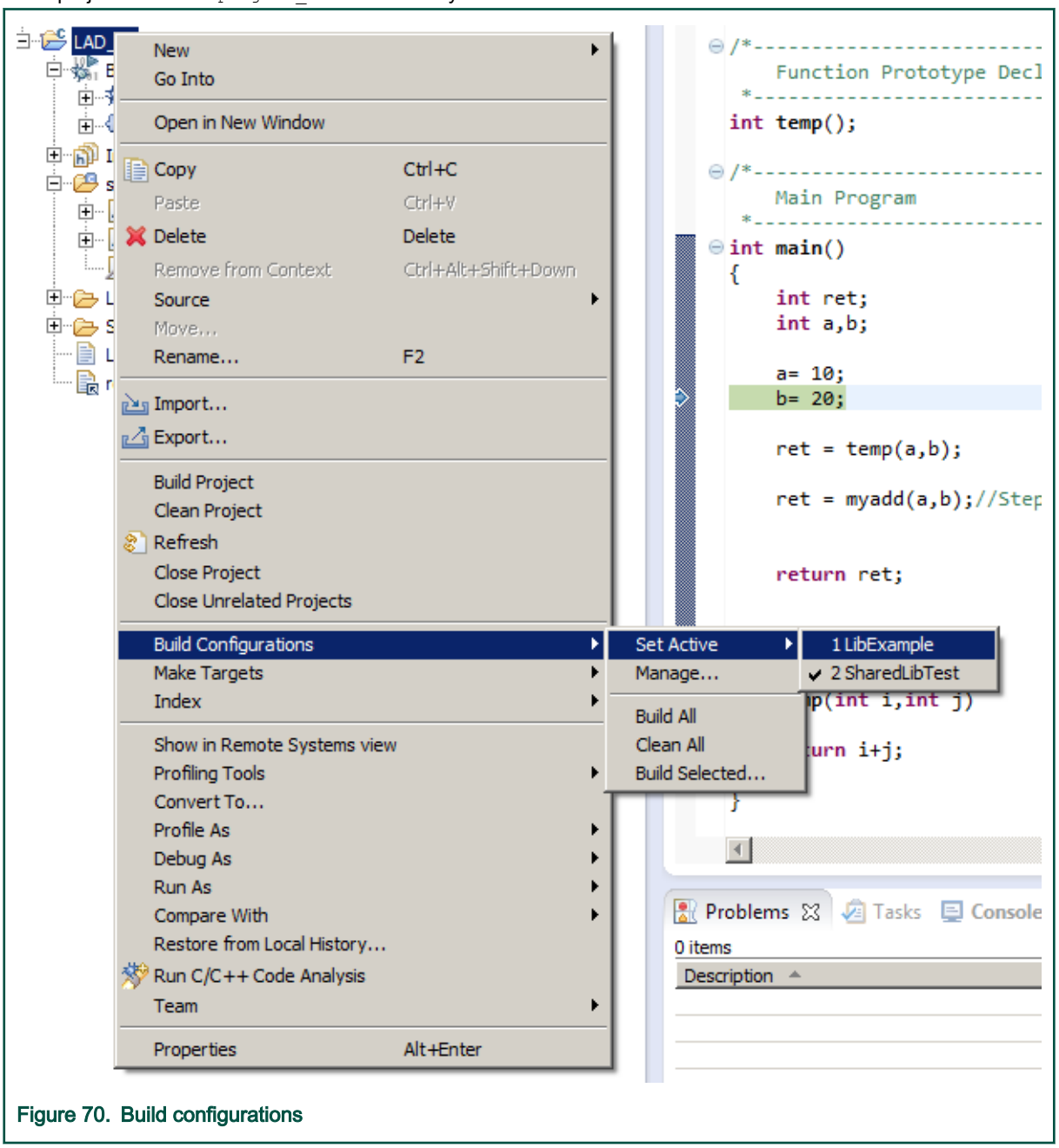


Figure 70. Build configurations

   c. Select the project, right-click and select **Build Configurations > Set Active > SharedLibTest**.

   d. Build project. The `<project_name>.elf` library is created.

8.2.3.2 Updating remote connection

This topic explains how to change the settings in a default remote connection.

Refer to the steps in Updating remote connection.

8.2.3.3 Updating launch configuration for Linux application using shared library

This topic explains steps to set the launch configuration for a Linux application project that uses a shared library

To set the launch configuration for a Linux application project that uses a shared library, perform the following steps:

1. Perform all the steps in Using sysroot.
2. Manually download the `.so` shared library to the Linux target (to the `/lib` path).
3. Copy the `.so` shared library to the `sysroot` location. (Refer Using sysroot, step 1d)

   The location can be:

   a. The `rootfs/lib/` folder you created on your host PC (Refer Using sysroot, step 1a)

   b. The `lib` from the `sysroot` location from SDK, if you are using the CodeWarrior software on the same Linux machine where you have compiled the SDK package and you are using the `sysroot` from SDK.

   Example for yocto-based sdk:

   ```
    /home/u12345/Desktop/SDK_Setup/QorIQ-SDK-V2.0-20160527-yocto/build_ls2088ardb/tmp/
   sysroots/ls2088ardb/lib
   ```

4. Add missing settings in launch configuration file.

   a. Right-click the project and select **Debug As > Debug Configurations**. The **Debug Configurations** dialog appears.

   b. Expand **C/C++ Remote Application**, select the Linux shared library project you want to debug.

   c. Click the **Shared Libraries** sub tab and click **Add** to add the path to the `*.so` library you created in Creating Linux shared library project. The path is

   ```
   ${ProjDirPath}/LibExample
   ```

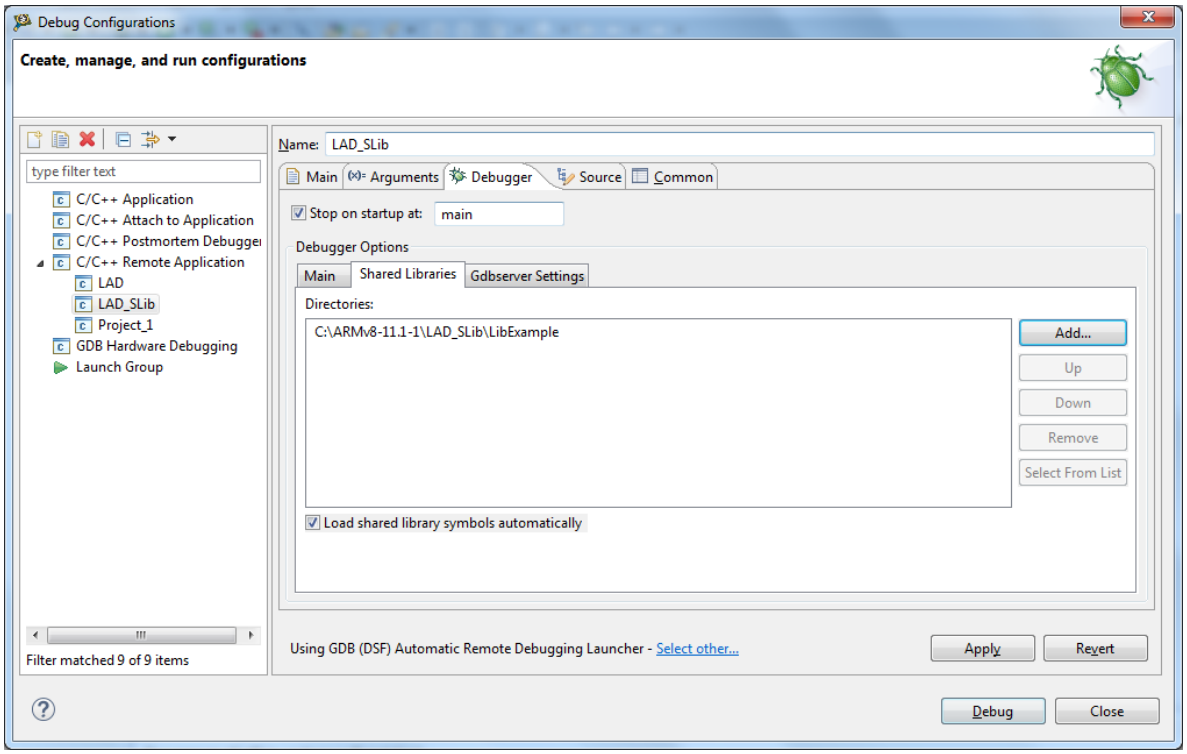


Figure 71. Debugger tab - Shared Libraries

d. Click **Apply**.

8.2.3.4 Debugging Linux shared library project

This topic explains steps to debug Linux shared library project.

To debug Linux shared library project, refer to the steps in Debugging Linux application project

8.2.4 Attaching to a Linux application

This topic explains how to attach to a Linux user space application running on a Linux target and how to debug it using the CodeWarrior software.

This topic also explains how to create a Linux application project, how to update the debug configuration, and how to attach to the Linux application.

NOTE

This topic only applies for attaching to an application that is already running on the target. To debug an application from its entry point, see Debugging simple Linux application.

To attach and perform debug operations on the Linux application, an elf file with debug symbols of the application is required on the machine running the CodeWarrior software. To create an ARMv8 project for attach to a Linux application:

1. Launch CodeWarrior for ARMv8.
2. Import the binary image for the application as described in CodeWarrior Executable Importer wizard.
3. The **Debug Configurations** dialog appears after you perform the last step in the ELF importer wizard. The dialog will have a launch configuration of type **C/C++ Remote Application** selected, with the same name as the newly created project.
4. Select the type **C/C++ Attach to Application** and press the **New launch configuration** button.
5. Name the new launch configuration.

6. In the **Main** tab, click **Browse** and select the newly created project.
7. In the **Debugger** tab, select gdbserver from the **Debugger** drop-down list.
8. In the **Debugger Options** section, select the **Connection** tab.
9. Specify **Type** as **TCP**, specify the host IP of the Linux target and a port number for the gdbserver.
10. Start the gdbserver on the Linux target with the port number specified: `gdbserver --multi :<port_number>`
11. Click **Debug**. The gdb client connects to the gdbserver, and can now attach to any application from Linux target by right-clicking the debug configuration name and selecting **Connect**.

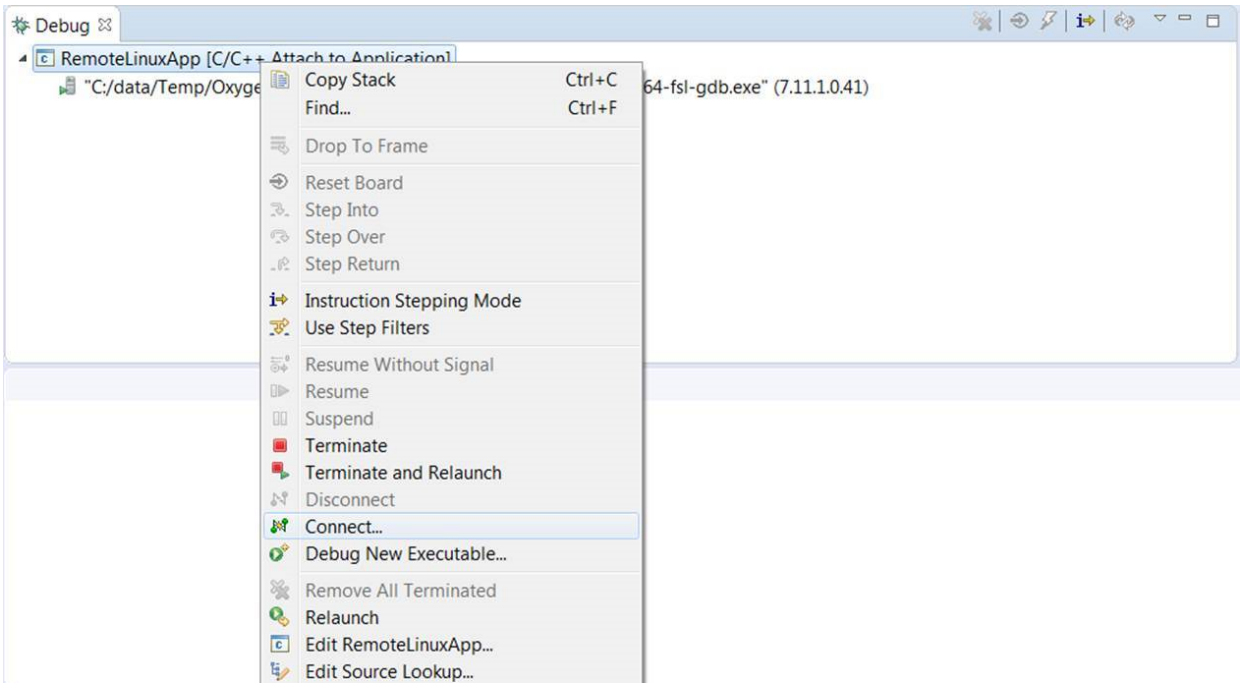
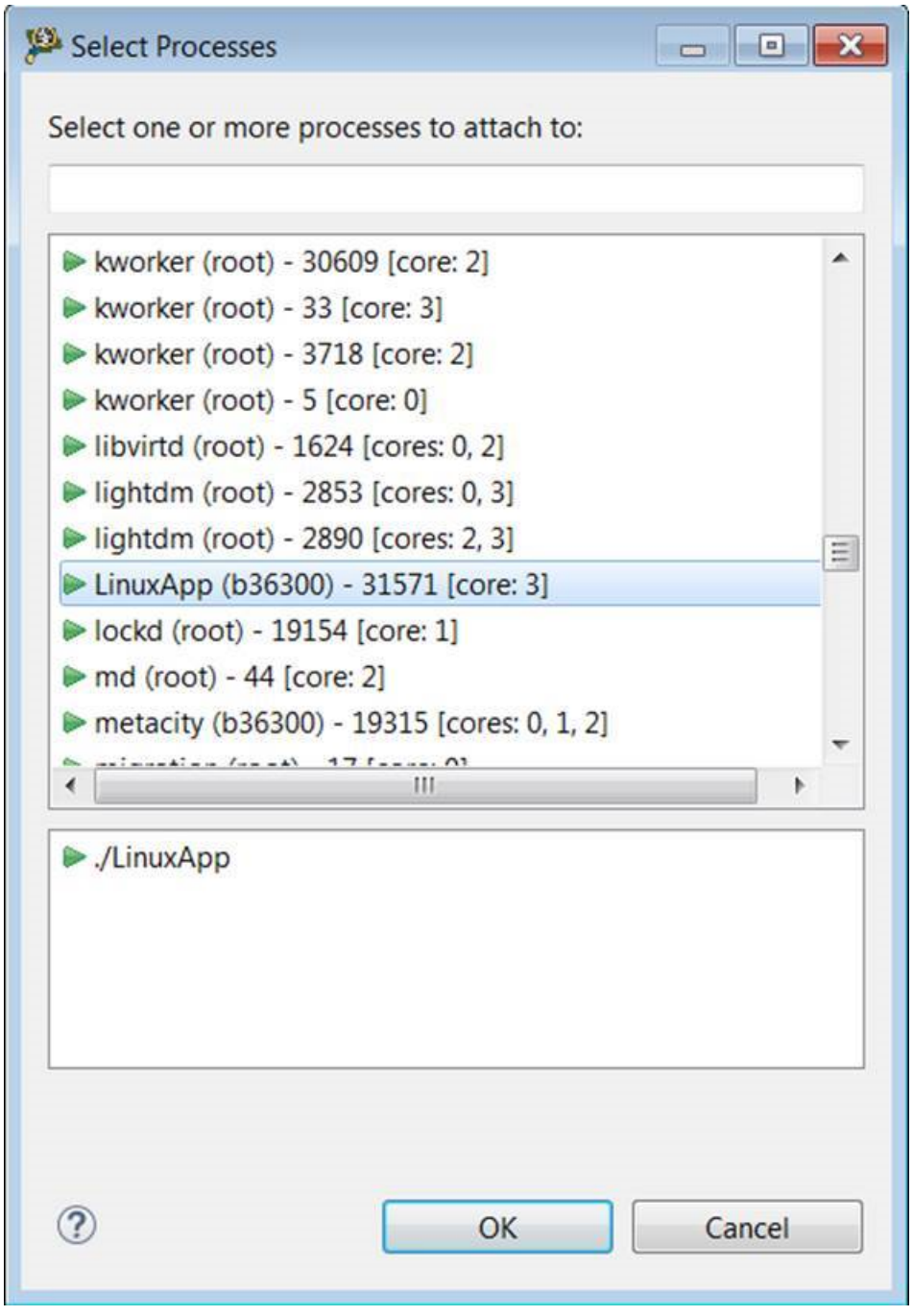


Figure 72. Attach to Linux application



12. Click **OK** to attach to the running application. At this moment, all debug capabilities will be enabled.
13. For setting up breakpoints in global symbols (for example, `main`), you should load the application debug symbols using the `file <path_to_application>` command in the GDB console in the CodeWarrior software.

8.2.5 Debugging multi-process remote applications

To debug a multi-process application:

1. From the CodeWarrior IDE menu bar, select **Run > Debug Configurations**.

2. In the **Debug Configuration** dialog, expand **C/C++ Remote Application** and select the launch configuration for the multi-process application project you want to debug.
3. Click the **Debugger > Main** tab.
4. Check the **Automatically debug forked processes (Note: Requires Multi Process GDB)** checkbox.

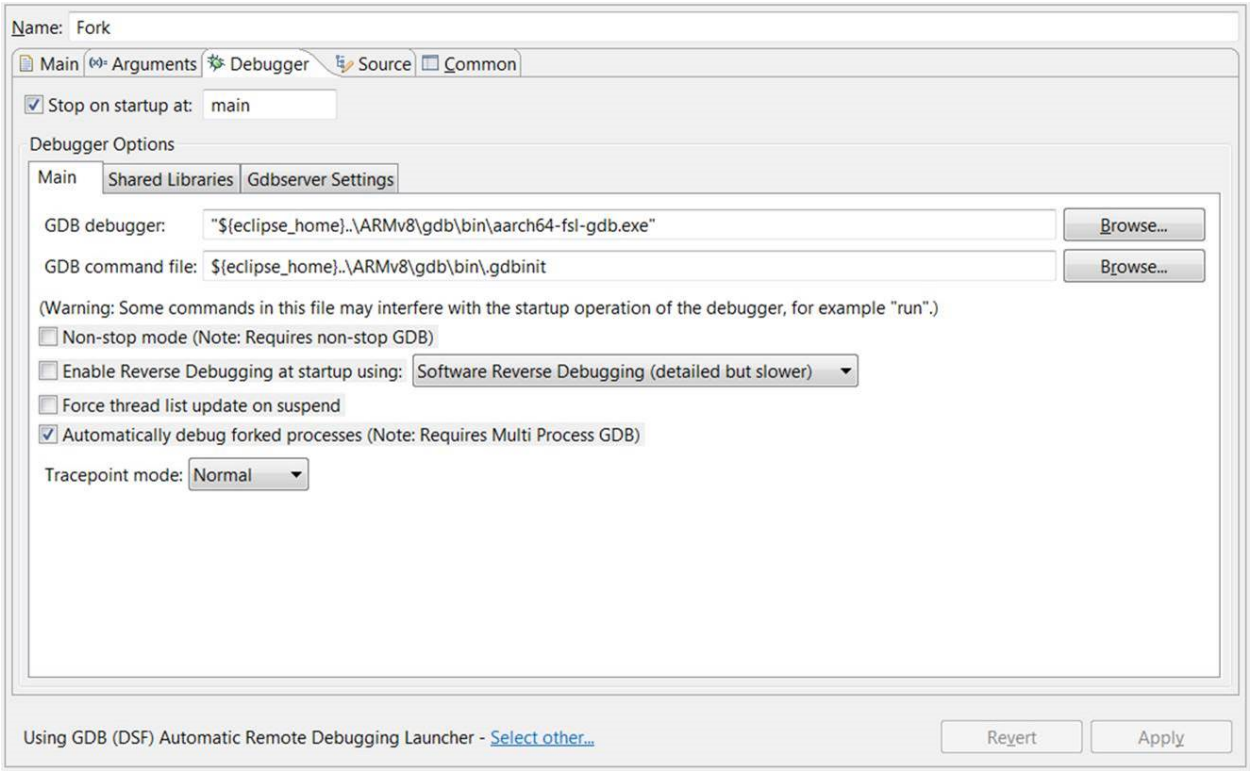


Figure 73. Debug a multi-process application

5. Click **Debug**.

8.3 Linux kernel debug

This document describes the steps required to perform Linux kernel debug.

This document describes the steps required to perform Linux kernel debug using CodeWarrior Development Studio for QorIQ LS series - ARM V8 ISA. This document explains:

- Building the U-Boot, Linux sources, and the auxiliary tools.
- Performing Linux Kernel debug in CodeWarrior Development Studio for QorIQ LS series - ARM V8 ISA.

8.3.1 Linux Kernel setup

This topic explains Linux kernel build.

For details on Linux kernel build, refer to the SDK documentation or the LSDK documentation as per the build system used. Yocto-based SDK uses bitbake commands to build various packages, whereas Dash/LSDK is based on flex-builder and flex-installer toolset.

NOTE

In order to perform Linux kernel debug, please ensure that the kernel image is build with debug symbols. For enabling the debug symbols:

1. Start `menuconfig`, the menu-driven configuration step for the Linux kernel.
2. Go to **General Setup**, disable the option **Compile also drivers which will not load**. Note that without performing this step the option below will not appear in menuconfig.
3. Select **Kernel Hacking -> Compile-time checks and compiler options**, enable option “Compile the kernel with debug info”.

8.3.2 Create an ARMv8 project for Linux kernel debug

This topic explains steps to create an ARMv8 bare metal project for U-Boot debug.

To create an ARMv8 bare metal project for U-Boot debug, perform these steps:

1. Open CodeWarrior for ARMv8.
2. Import a Linux Kernel image as described in CodeWarrior Executable Importer wizard.
3. Select **Run > Debug Configurations** to open the Debug Configurations dialog.
4. Click the **Startup** tab.
    a. Set breakpoint at: 0x80080000.
    b. Check the **Resume** button.

NOTE

Step (b) should be done only if nothing is running yet on the target board, or in case you have just started the target board but have not started the Linux Kernel. However, in case you simply attach it to a running the Linux Kernel session the above step should be skipped. PC will reflect the current PC while the Linux Kernel is running.

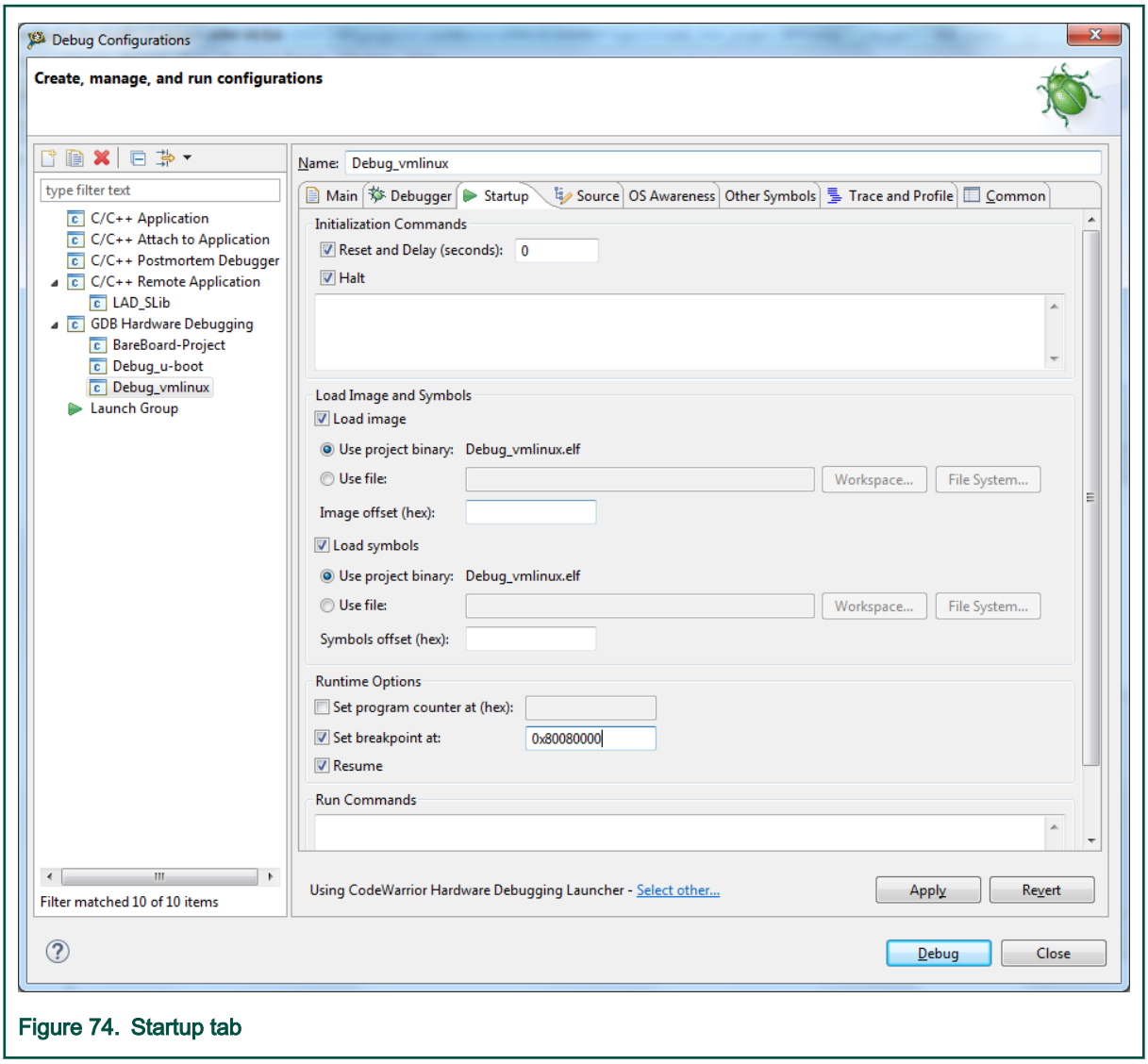


Figure 74. Startup tab

5. Set up the target connection configuration, as explained in Configuring Target.
6. Click the **Debug** button to initiate the debug session. The debugger should stop at 0x80080000 – kernel entry point address.

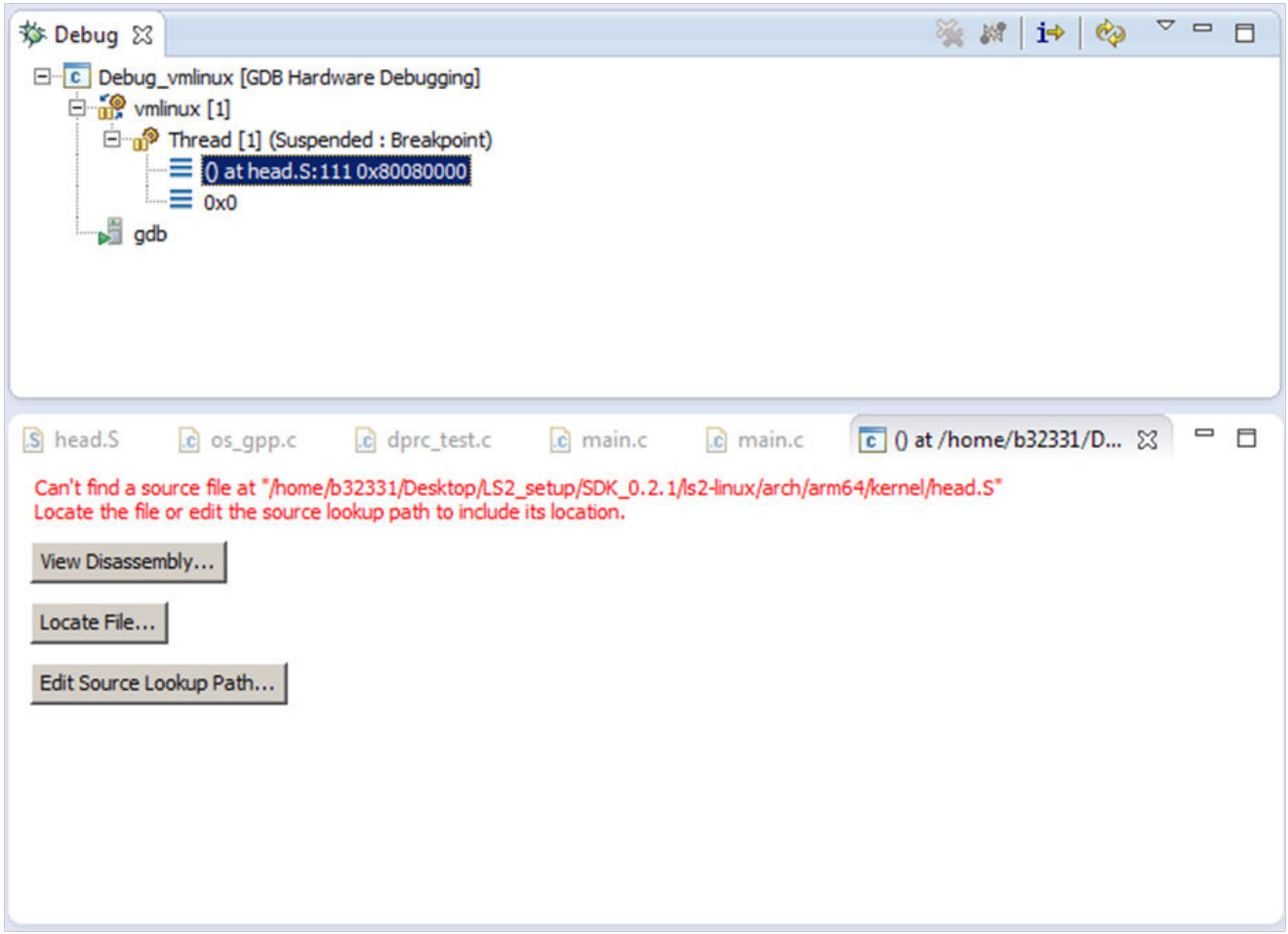


Figure 75. Debug Session Window

8.3.3 Linux Kernel debug support

This section explains the steps required to perform Linux kernel debug in CodeWarrior Development Studio for QorIQ LS series - ARM V8 ISA.

This section includes:

- Setting the source path mapping
- Debug and Kernel Awareness capabilities

8.3.3.1 Setting the source path mapping

This section explains the steps required to load symbols and set source path mapping.

Perform the following steps:

1. Click the **Refresh Debug Views** button to refresh the debug views updated with the new stack and the registers view.
2. Close the **Source not found** window.
3. Double- click the stack for triggering the source-level mapping request.
4. Locate the file suggested by the debugger.

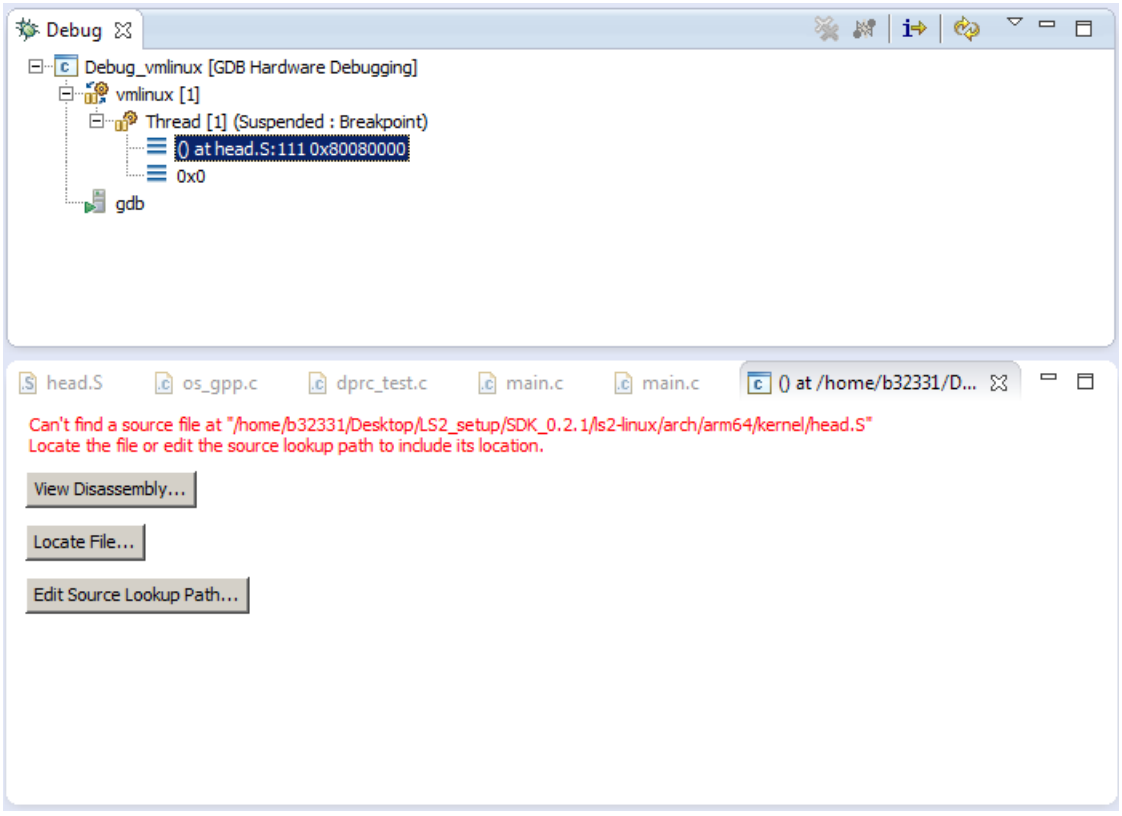


Figure 76. Locate source window

The following figure shows stack and the source views added.

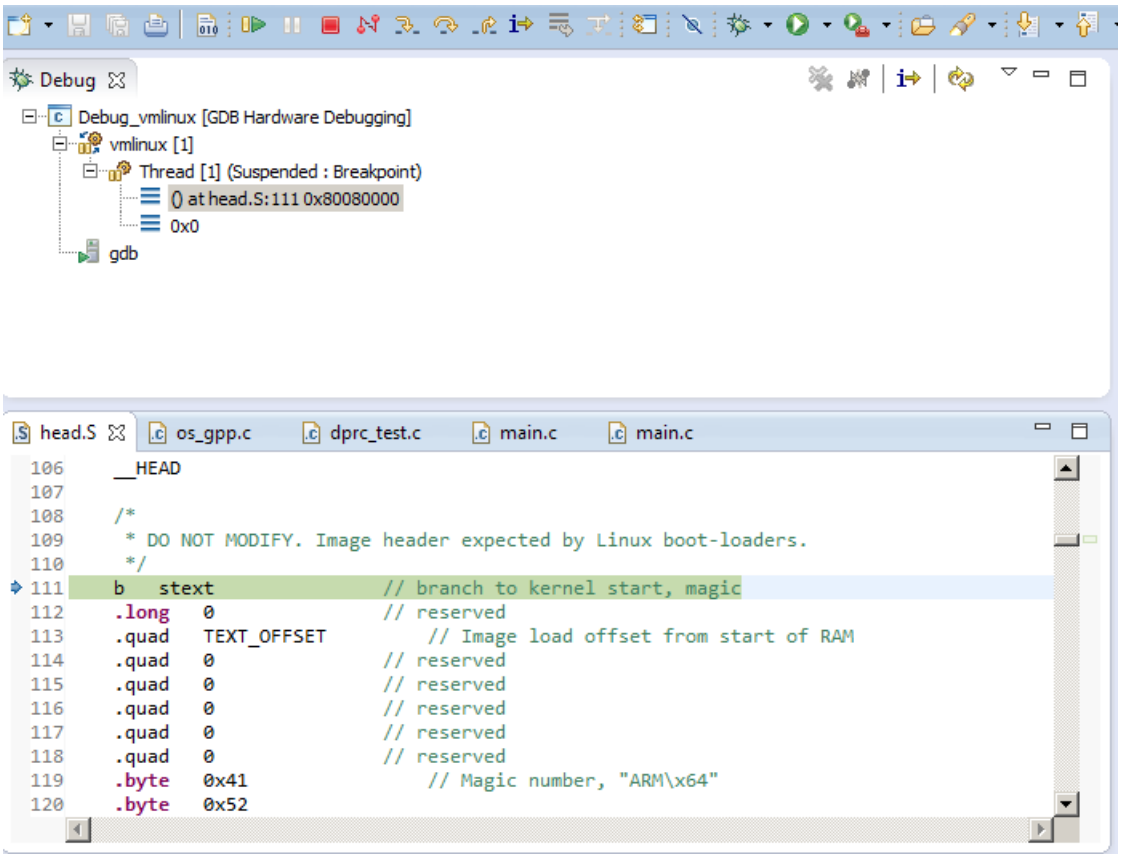


Figure 77. Stack and sources added

NOTE

You can add a static map entry using the Edit Source Lookup Path button to avoid locating file using the Locate File button, whenever a new file is requested

5. To go ahead with next important step in Linux kernel debug (start_kernel), you need to set up a breakpoint there using this command: *break start_kernel* in the same gdb console.
6. Click the **Resume** button. Alternatively, press the F8 key. The breakpoint will be hit.
7. Click the **Refresh Debug Views** button to refresh the debug views updated with the new stack and the registers view.
8. Close the **Source not found** window.
9. Double-click the stack for triggering the source-level mapping request.
10. Locate the file suggested by the debugger.

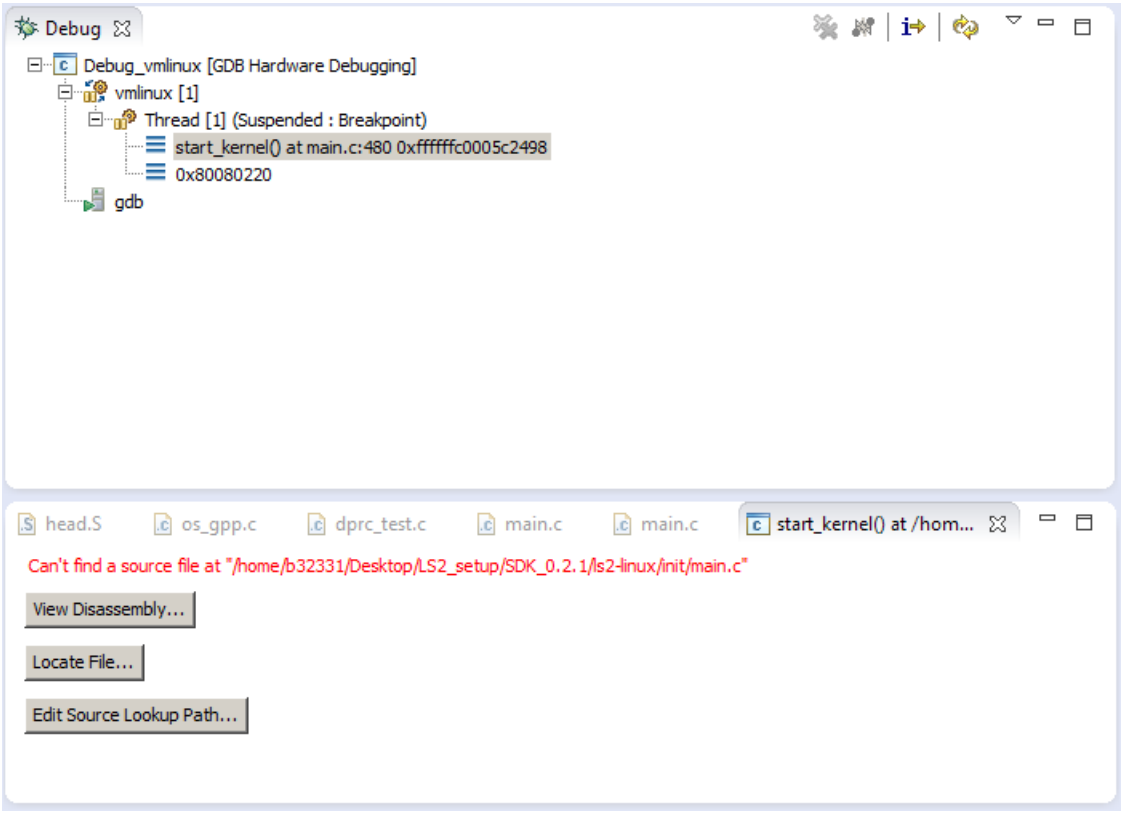


Figure 78. Debug console

11. For details about debug and kernel awareness capabilities, see Debug and Kernel Awareness capabilities.
12. Click the **Resume** button to run the vmlinux. Alternatively, press the F8 key.
13. To start the Linux Kernel debug again, close/terminate the actual connection.

8.3.3.2 Debug and Kernel Awareness capabilities

This section explains various Debug and Kernel Awareness capabilities.

Perform the following steps:

1. Select **Window > Show view > Disassembly** to enable the Disassembly view.
2. Double-click a line to inspect breakpoints. You can inspect them using:
   - Breakpoints view
   - `info breakpoints` command from GDB shell
3. Set up hardware breakpoints using `hbreak` command from GDB console.
4. You can also perform step in, step over, and step return functions from the GUI.

5. Add watchpoints (data breakpoints) using the **Toogle Watchpoint** option from the context menu.

NOTE

A watchpoint only makes sense for a global variable (or to a global memory address).

The watchpoint is then listed in the Breakpoints view.

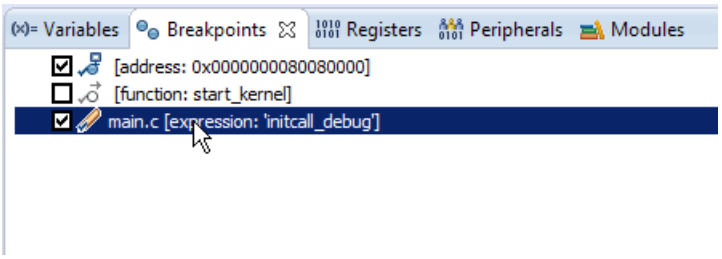
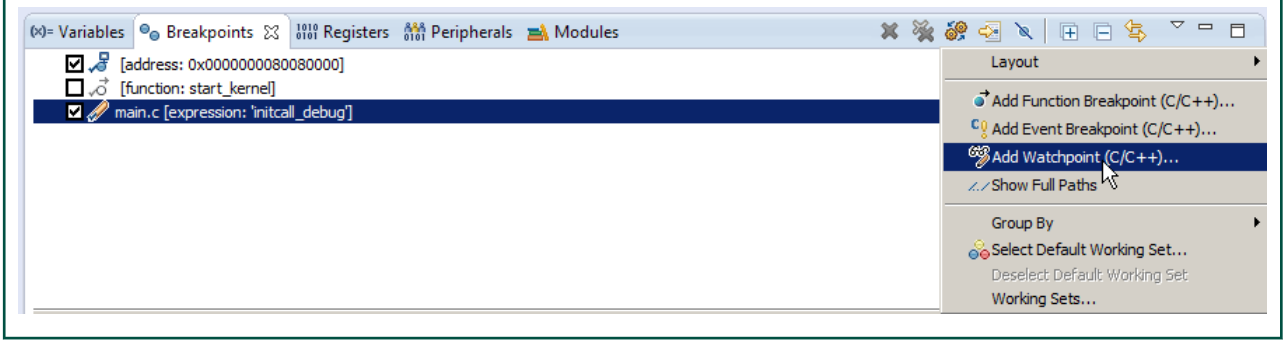


Figure 79. Breakpoints view

You can also add watchpoints using the drop-down menu in the Breakpoint view.



Use CodeWarrior software to see some important information about the Linux kernel, for example general information, build time, modules list, threads list and so on. To see the full Kernel Awareness capabilities, refer Linux kernel awareness.

8.3.4 Module debugging

This topic explains module debugging use cases and module debugging from Eclipse GUI.

This topic explains:

- Module debugging use cases
- Module debugging from Eclipse GUI

8.3.4.1 Module debugging use cases

This topic explains module debugging use cases.

1. Loading and unloading module's symbols file

   The runtime address when the kernel relocates the kernel module it is known only at runtime after the module is loaded while the module's symbols file contains only the compile time address information. Therefore module symbols file can be loaded only when the module is loaded into the kernel (e.g. using insmod or modprobe command).

   Two symbols files are generated after a module compilation: <module_name>.ko and <module_name>.o. The .ko file should be copied to the target and loaded using insmod/modprobe Linux command. The .o file it is the symbols file to be loaded into the debugger.

   The kernel module's symbols file can be loaded in two ways:

   a. Manually, using the ka-module-load command. The command should be executed after the module is loaded. The typical use cases are:

      - Configure debugger to suspend when the module insert is detected. When the debugger suspend, load the corresponding module's symbols file.

- The module being inserted, suspend the target execution and load the corresponding symbols file.

b. Automatically (ka-module-config-auto-load=True.), using ka-module-config-map-loadWhen automatic loading mode is enabled, the debugger detects when a module is inserted (insmod or modprobe) and automatically searches the configured symbols file mapping and loads the symbols file. Before inserting the module, the user should add the corresponding symbols file into the symbols file mapping using ka-module-config-map-load command. This command can be run at any time (before and after module loading), but the symbols file is loaded only when the debugger detects that the corresponding module has been inserted.

The user can unload the module's symbol file if the symbols file is already loaded. When the module is removed (rmmod), the debugger automatically unloads the module symbols file, independent of the value of ka-module-config-auto-load. This is done because the module relocation addresses are not valid anymore and even on a new module insertion there will be different relocation addresses.

2. Setting breakpoints in module

Breakpoints in module's source code or at a specific module function can be set at any time, even the module symbols file is not loaded into the debugger.If the module's symbols file is loaded, the breakpoint is set/enabled and the module relocation address is displayed in the breakpoint properties.

```
(gdb)  break krng_mod_initBreakpoint 3
        at 0xffffffbffc03a000: file crypto/krng.c, line 50.(gdb) info
       breakpoints Num     Type           Disp Enb Address            What3       breakpoint
keep y
      0xffffffbffc03a000 in krng_mod_init at crypto/krng.c:50
```

If the module's symbols file is not loaded, the debugger could not resolve the corresponding breakpoint relocation address, but will set the breakpoint as "pending". When the module is inserted and the module's symbols file is loaded, the debugger will refresh the "pending" breakpoints resolving the relocation address.

The debugger behavior for "pending" breakpoint is configurable using "set breakpoint pending" command with the following values:

- "auto": this is the default value. When the breakpoint is set from command line interface, the debugger asks the user to select one of the following values. From Eclipse/gdb-MI, the "auto" value will make the breakpoint pending "on"
- "on" breakpoint "pending" is enabled
- "off" breakpoint "pending" is disabled. With this setting, the breakpoint can not be set when the module's symbols file is not loaded

3. Debug Linux kernel module from the module_init function

There are several ways of doing kernel module debug from the module_init function:

a. Without suspend at module insertion

- Add the symbols file to the configured map using the command ka-module-config-map-load.
- Enable module auto-load
- Set a breakpoint to the module's init function. The breakpoint will be "pending", as the module is not loaded yet.
- Insert the module (insmod). The debugger will stop at the module's init function

b. With suspend at module insertion

- Enable suspend at module insertion
- Insert the module. The debugger will suspend the target
- Load the symbols file using ka-module-load command
- Set a breakpoint to the module's init function. The breakpoint will be resolved as the module and the symbols file are loaded
- Run. The debugger will stop at the module's init function

4. Module insertion and removal detection

   Module insertion and removal detection is implemented by setting a special breakpoint (named eventpoint) in Linux Kernel code (not module code).

   - When the module is prepared to be executed, but before running the module's init function
   - And when the module is prepared to be remove, after running the module's delete function

   These debugger specific breakpoints are not visible to the user. The command " info breakpoints " displays no information about these breakpoints.

   The eventpoints information can be displayed using the command " maintenance info breakpoints ":

```
(gdb)
       maintenance info breakpoints Num     Type           Disp Enb Address            What-1
breakpoint      keep y    0xffffffc0000ef8fc in
       load_module at kernel/module.c:3020 inf 1-2     breakpoint     keep y    0xffffffc0000eddd4
in
        free_module at kernel/module.c:1840 inf 1(gdb)
```

   The eventpoints have negative breakpoint numbers and the user can not modify the breakpoint properties (e.g. delete breakpoint).

8.3.4.2 Module debugging from Eclipse GUI

Before launching the module debugging session, set the following options in the **OS Awareness** tab.

- Check **Suspend target when module insert or removal is detected**.
- Check **Automatically load configured symbolic files at module init detection**.

If the option, **Automatically load configured symbolic files at module init detection** is enabled, the debugger loads the user defined list of module symbolics files, used to configure the gdb, in the *Auto-load module symbolics files list* section . The module symbolics file name signifies the module name, for example the symbolics file *rng.o* will refer to module *rng*.

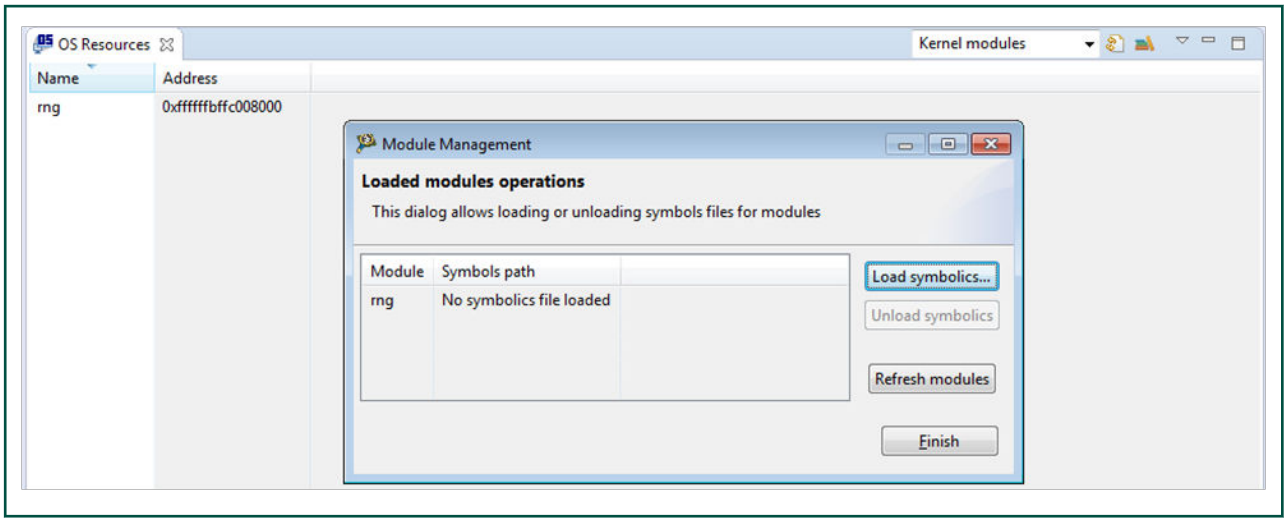
To load a different symbolics file, the Module Management dialog, which is available at runtime from OS resources View, should be used.

1. In the *OS resources View* , click

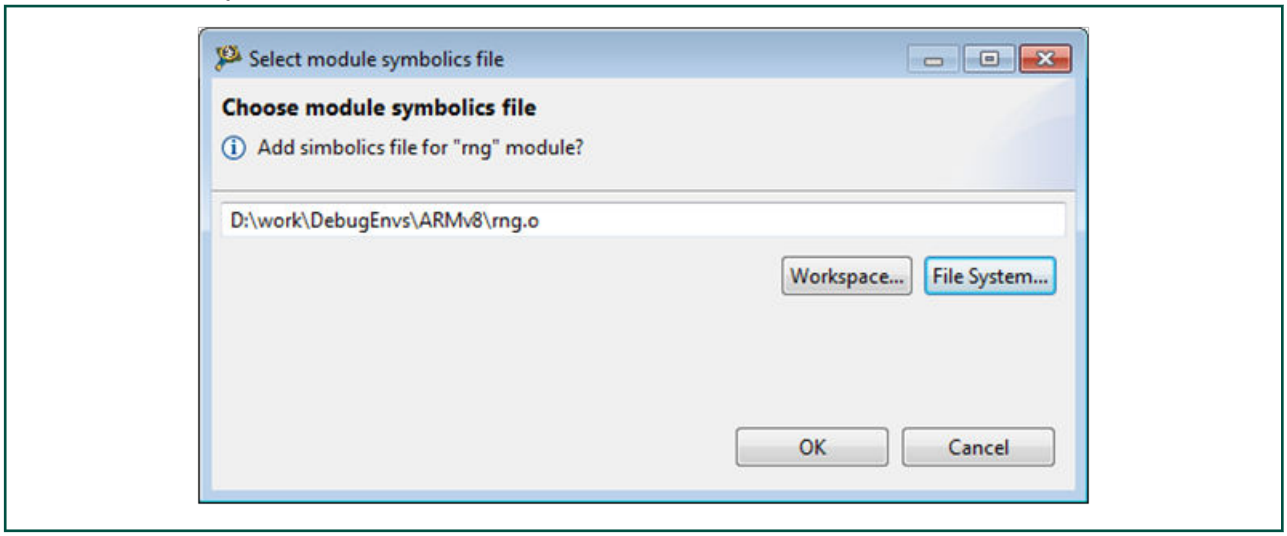
   .

   The **Module Management** dialog appears with the currently available modules. The dialog will also show if the symbolics files for a module is loaded.



2. Click *Load symbolics* to load a symbolics file for a module.

   The **Select module symbolics file** dialog appears.



3. The user can choose a different symbolics file for a module if before opening the **Select module symbolics file** dialog the module was selected from the list. In this case, the dialog will ask to confirm the mapping between the current symbolics file and the module.
4. Click **OK**.

8.4 UEFI debug

This topic describes the steps required to perform a UEFI debug using CodeWarrior Development Studio for QorIQ LS series - ARM V8 ISA.

This topic lists the steps to:

- Build the UEFI sources and the auxiliary tools.
- Perform UEFI debug in CodeWarrior Development Studio for QorIQ LS series - ARM V8 ISA.

NOTE

For details on how to prepare the target, see Preparing Target.

8.4.1 UEFI setup

This topic explains UEFI build.

For details on UEFI build, refer to the SDK documentation or the LSDK documentation as per the build system used. Yocto-based SDK uses bitbake commands to build various packages, whereas Dash/LSDK is based on flex-builder and flex-installer toolset.

8.4.2 Create an ARMv8 project for UEFI debug

This topic explains steps to create an ARMv8 bare metal project for UEFI debug.

NOTE

If you are located on a different machine than where UEFI was built, you need to do one of the following:

- copy the UEFI build layout to a local path
- map the network address where the UEFI build layout is located to a local drive

To create an ARMv8 bare metal project for UEFI debug, perform these steps:

1. Open CodeWarrior for ARMv8.
2. Import a UEFI image as described in CodeWarrior Executable Importer wizard.
3. A **Debug Configurations** dialog is opened automatically, with the debug configuration for the newly created UEFI debug project already selected.
4. Set up the target connection configuration, as explained in Configuring Target.
5. If you want the debugger to automatically load symbols at attach for EFI images loaded during the DXE phase:
    a. Open the **OS Awareness** tab page
    b. Select the **Add symbols for EFI images loaded at runtime** checkbox.

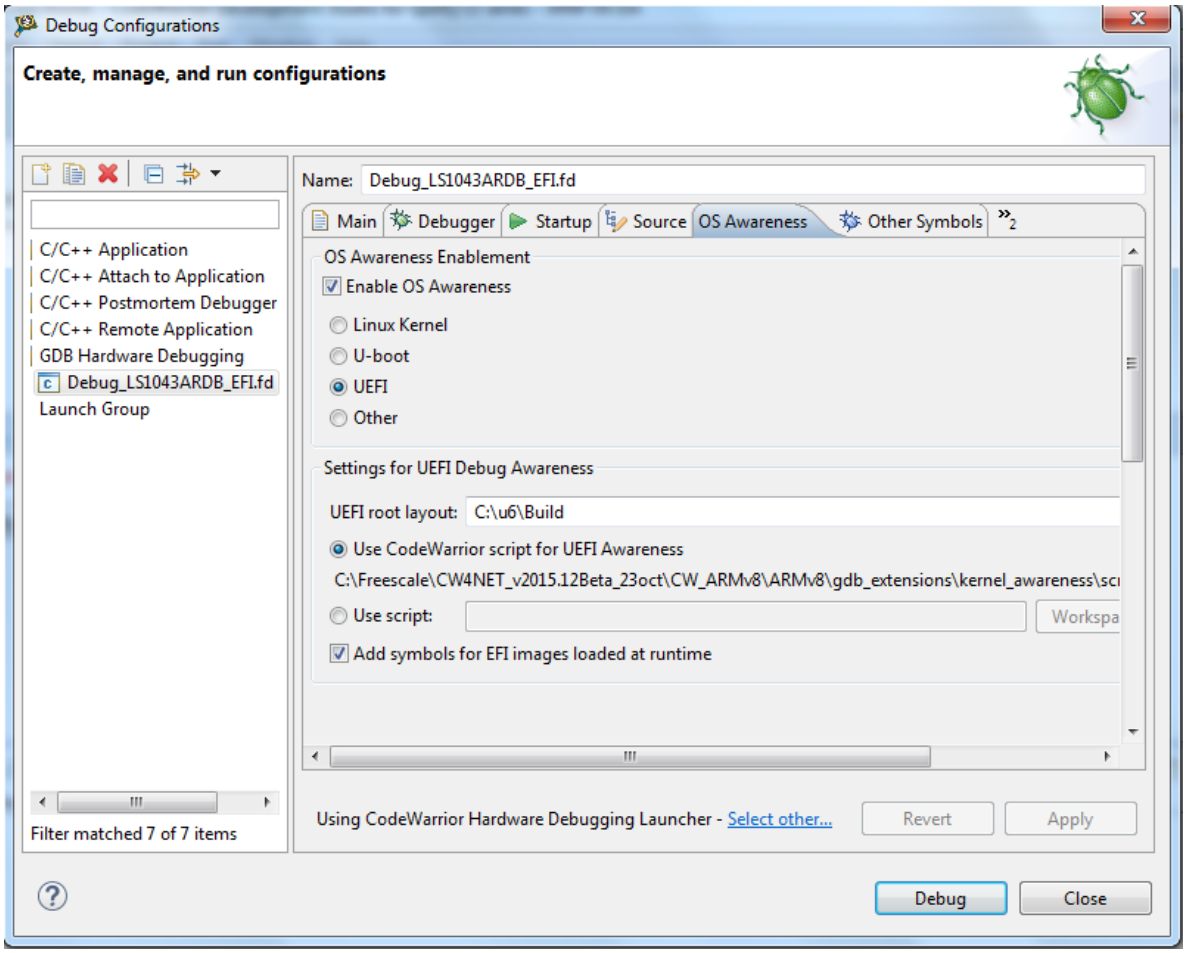


Figure 80. Add symbols for EFI images

6. Click the **Debug** button to initiate the debug session. The debugger should show source information for the current PC:
   - If the current PC is after UEFI entry point, but before DXE phase
   - if the current PC is in a EFI image loaded at runtime, and the **Add symbols for EFI images loaded at runtime** checkbox is selected

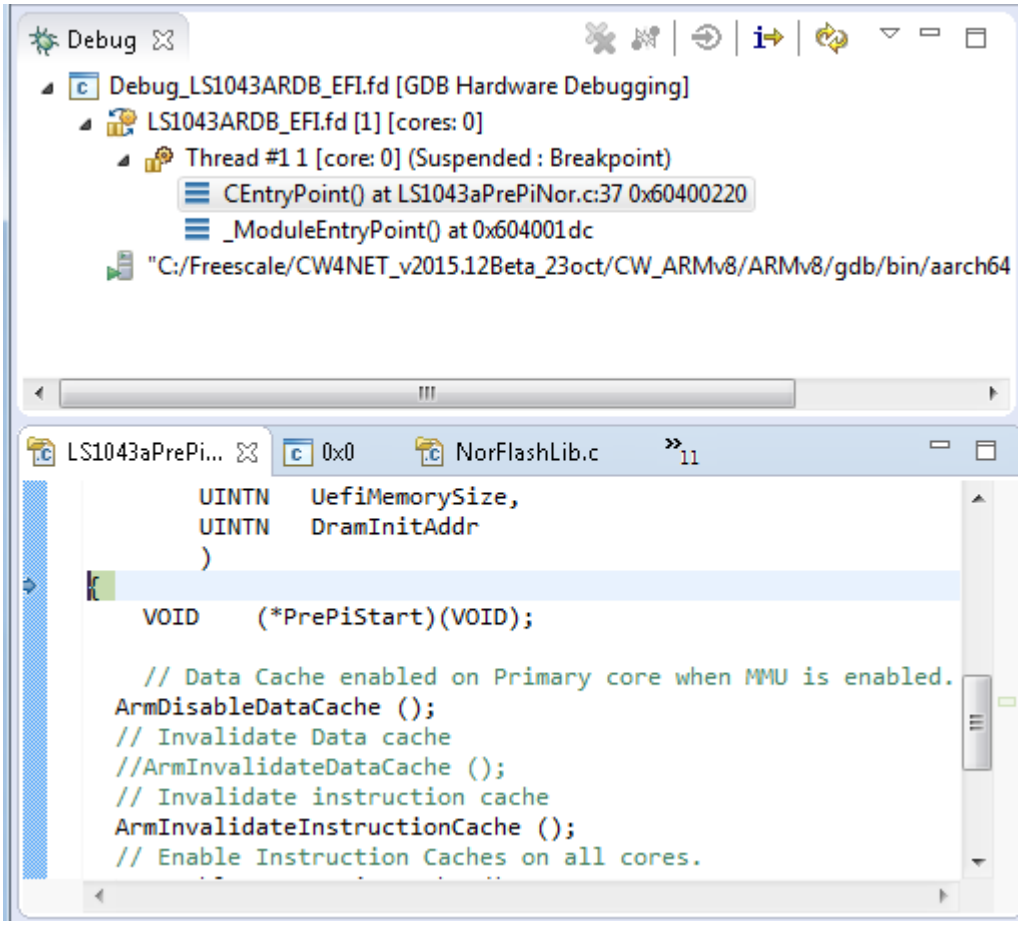


Figure 81. UEFI debug session

8.4.3 UEFI debug support

This section explains steps to perform UEFI debug in CodeWarrior Development Studio for QorIQ LS series - ARM V8 ISA.

8.4.3.1 Starting from the Reset Point

This topic explains steps how to start UEFI debug from the Reset Point (0x0).

1. Go to the **Startup** tab.
2. Select **Reset and Delay (seconds)** and enter 0 in the associated text field.
3. Click the **Debug** button to initiate the debug session. The debugger should stop at address 0x0.
4. The user can now set breakpoints at symbols in code either before relocation to DDRAM, or after relocation to DDRAM, but before starting the DXE phase.

8.4.3.2 Adding debug information for EFI images loaded at runtime

When execution is suspended during DXE or BDS phases and no symbols are displayed, the user can run the `uefi-add-symbols` command at the GDB command line in order to add the symbol files for all the EFI images loaded at the runtime.

The command will display **Done** after it finishes the execution successfully.

See section Load debug data for all loaded EFI images for details.

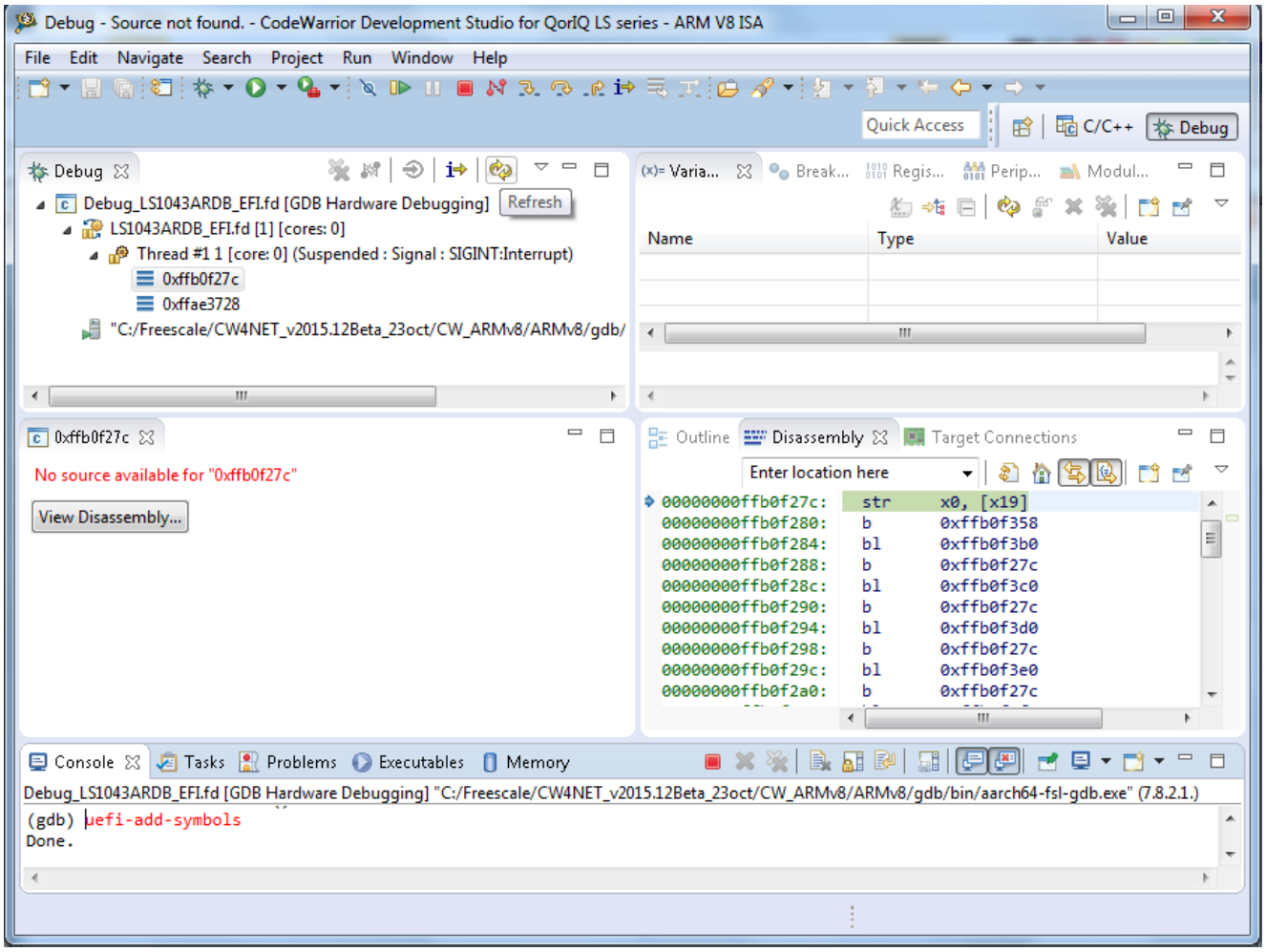


Figure 82. Add debug information for EFI images

The user needs to press the **Refresh** button in the **Debug** view in order for the call stack to be updated with the debug information, and then double-click a stack frame to display the source file.

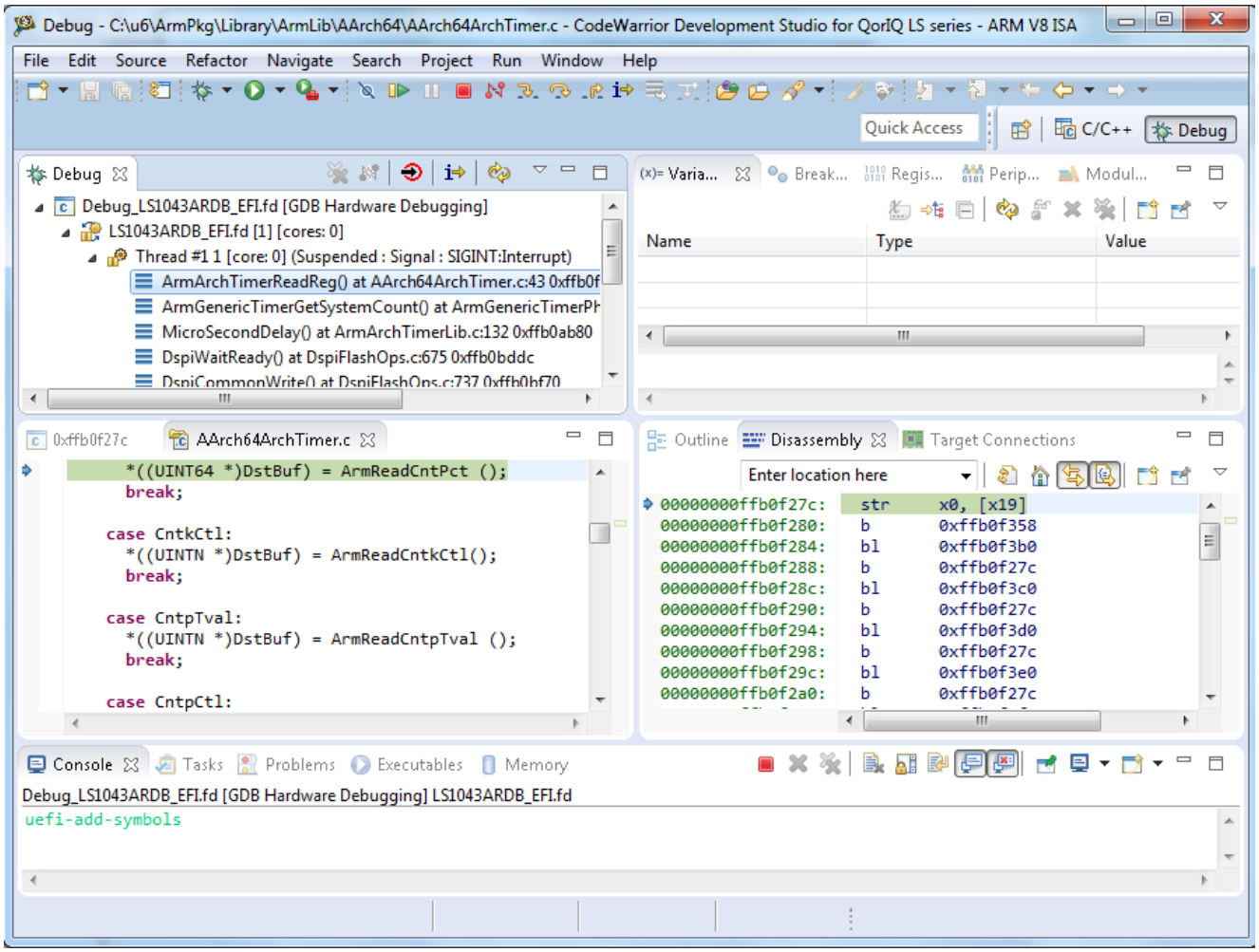


Figure 83. Display source file

8.4.3.3 Viewing information about EFI image loaded at runtime

When execution is suspended during DXE or BDS phases, the user can view details about all the EFI images that have been loaded at runtime in the **OS Resources** view.

See Show information for all loaded EFI images for details.

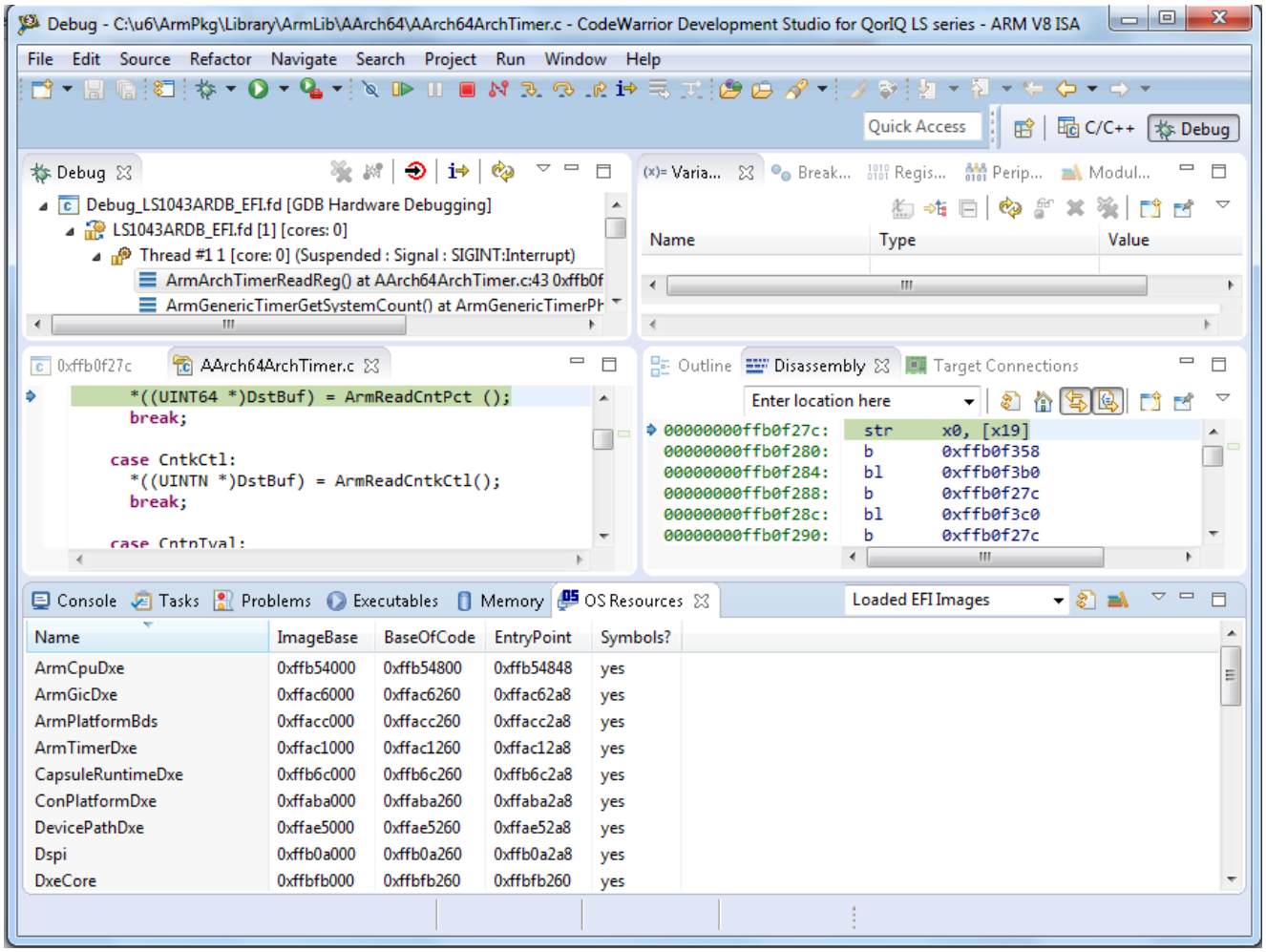


Figure 84. View EFI image information

8.5 Import and configure AMP example projects

This section explains steps to import and configure an AMP configuration.

1. Import the AMP example project available at the following location in the product layout, to your workspace:

   ```
   <CWInstallDir>\CW_ARMv8\ARMv8\CodeWarrior_Examples\HelloWorld_C_AMP_Bare
   ```

2. Build the project.
3. Open the **Debug Configurations** dialog.
4. Select `HelloWorld_AMP_Core0` and select a Target Connection Configurator. For details on this, refer Target Connection configurator overview and Configure the target configuration using Target Connection Configurator.
5. In the **Debugger** tab, select the core you want to debug from the **Core** drop-down list.
6. Repeat steps 3 and 4 for `HelloWorld_AMP_Core1` and make sure that at you have selected another core than the one selected at step 4.
7. Click **Debug**.

8.6 Board Recovery

This topic describes the steps required to perform board recovery using CodeWarrior Development Studio for QorIQ LS series – ARM V8 ISA, when the flash is blank or the image is corrupted.

This topic lists the steps to:

- Use the RCW override feature in the initialization file to connect to a board when RCW is missing or corrupted
- Use the Flash Programmer to load a valid RCW in the flash device
- Use the Flash Programmer to load the U-Boot in the flash device

8.6.1 RCW Override

When RCW is missing or corrupted and switches for setting the board in the hard-coded RCW mode are not available, you can use the RCW override feature available with the CodeWarrior software for board recovery. For using this feature, perform these steps:

1. Open CodeWarrior for ARMv8 and define a Target connection configuration for the board using a pre-defined configuration.

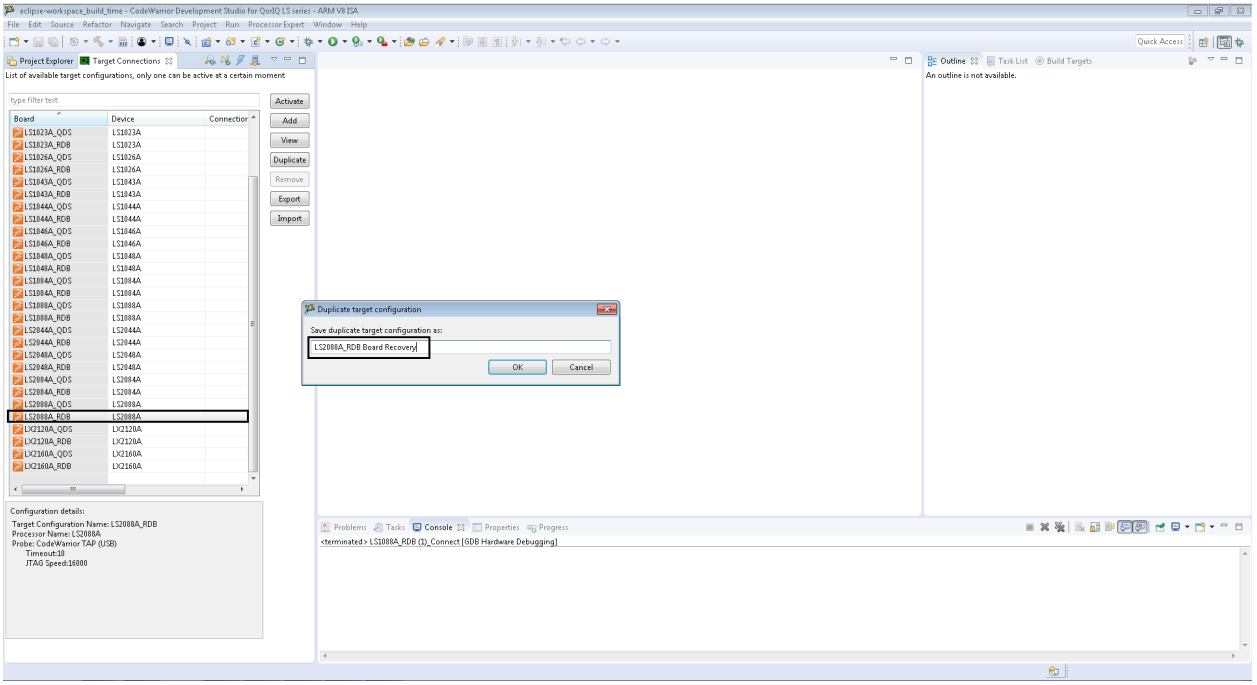
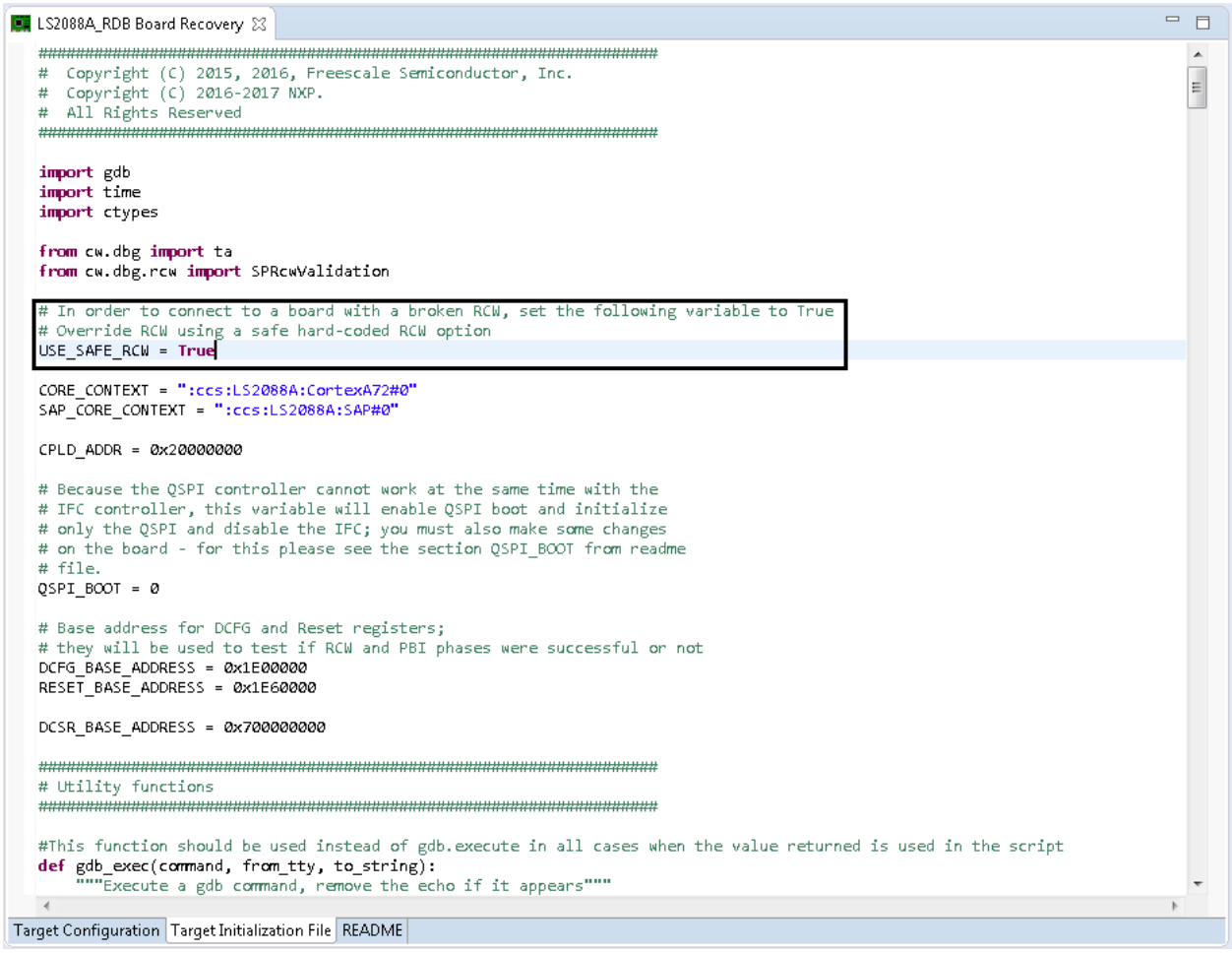


Figure 85. Target Connections

2. Select the newly created configuration and click the **Edit** button.
3. In **Target Configuration** tab, select the probe type.
4. Click the **Target Init File** tab and search for `USE_SAFE_RCW`. To connect to the board with a missing or corrupted RCW, set the `USE_SAFE_RCW` variable to *True*.




```
##################################################################
#  Copyright (C) 2015, 2016, Freescale Semiconductor, Inc.
#  Copyright (C) 2016-2017 NXP.
#  All Rights Reserved
##################################################################

import gdb
import time
import ctypes

from cw.dbg import ta
from cw.dbg.rcw import SPRcwValidation

# In order to connect to a board with a broken RCW, set the following variable to True
# Override RCW using a safe hard-coded RCW option
USE_SAFE_RCW = True

CORE_CONTEXT = ":ccs:LS2088A:CortexA72#0"
SAP_CORE_CONTEXT = ":ccs:LS2088A:SAP#0"

CPLD_ADDR = 0x20000000

# Because the QSPI controller cannot work at the same time with the
# IFC controller, this variable will enable QSPI boot and initialize
# only the QSPI and disable the IFC; you must also make some changes
# on the board - for this please see the section QSPI_BOOT from readme
# file.
QSPI_BOOT = 0

# Base address for DCFG and Reset registers;
# they will be used to test if RCW and PBI phases were successful or not
DCFG_BASE_ADDRESS = 0x1E00000
RESET_BASE_ADDRESS = 0x1E60000

DCSR_BASE_ADDRESS = 0x700000000

##################################################################
# Utility functions
##################################################################

#This function should be used instead of gdb.execute in all cases when the value returned is used in the script
def gdb_exec(command, from_tty, to_string):
    """Execute a gdb command, remove the echo if it appears"""
```

Figure 86. Target Init File

5. Click **OK** to save the configuration.

NOTE

Select one of the available hard-coded RCW options based on the board reset settings,(such as SYSCLK, DDRCLK). For the hard-coded RCW options, see QorIQ LS1012A Reference Manual. In the **Target Init File** tab, go to def Reset() procedure and set the values of the hard-coded RCW in `gdb.execute("monitor rcw source set <hard-coded RCW>")`.

NOTE

In case the CodeWarrior software does not support the RCW override for a specific board or SoC, you can configurie the board for the hard-coded boot source from the DIP switches.

8.6.2 Program valid RCW in flash device using Flash Programmer

To program valid RCW in the flash device using Flash Programmer:

1. Click the **Flash Programmer** button to connect to the board.

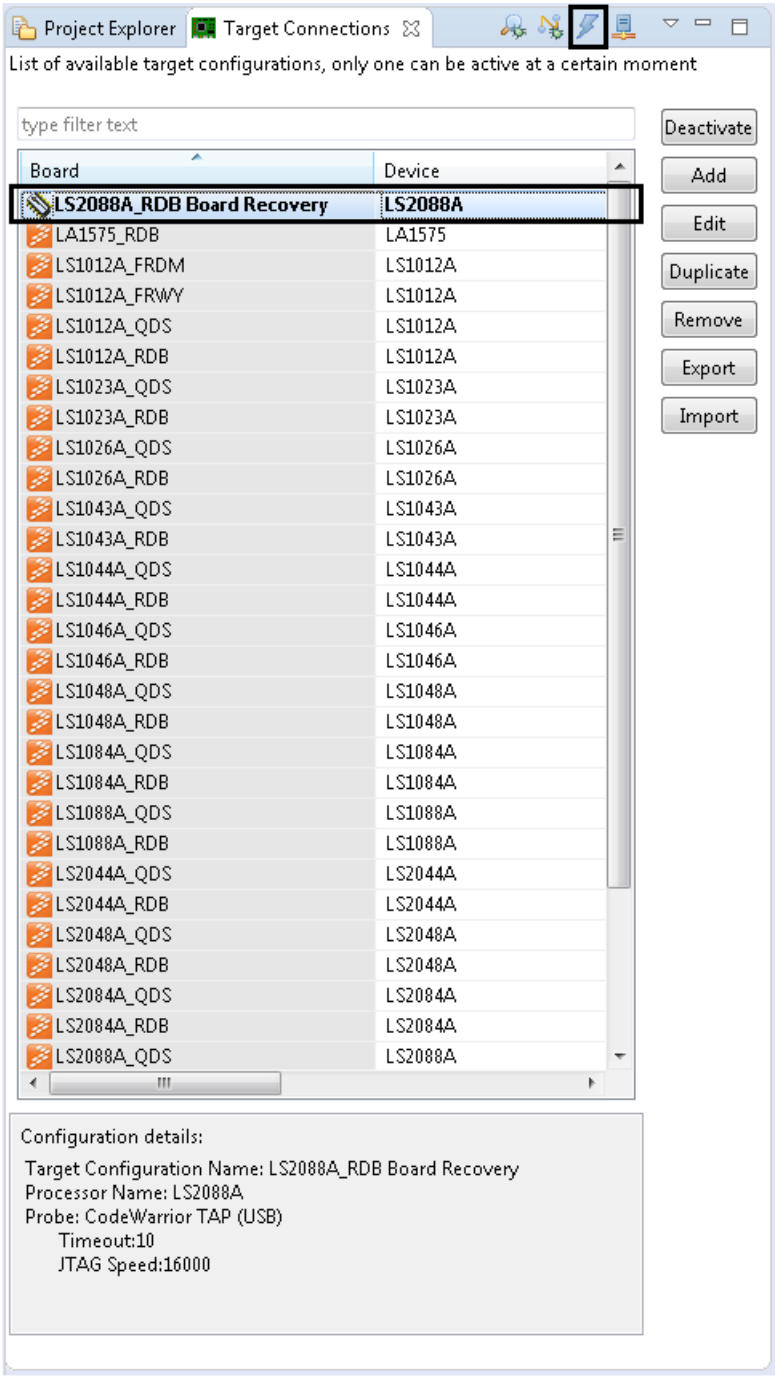


Figure 87. Flash Programmer button

2. When the CodeWarrior software connects to the board, the **CodeWarrior Flash Programmer** window appears.
3. Select the appropriate flash device, the Program action, the RCW file to be programmed and the Offset.
4. Click **Add Action** and execute the flash programmer sequence.

NOTE

When using the CMSIS-DAP probe, it is recommended to verify operation parameters and RCW correctness before continuing. Failure to program a compatible RCW may result in board being unable to boot. Recovery is not possible using CSMSIS-DAP and will require use of the external CodeWarrior TAP unit

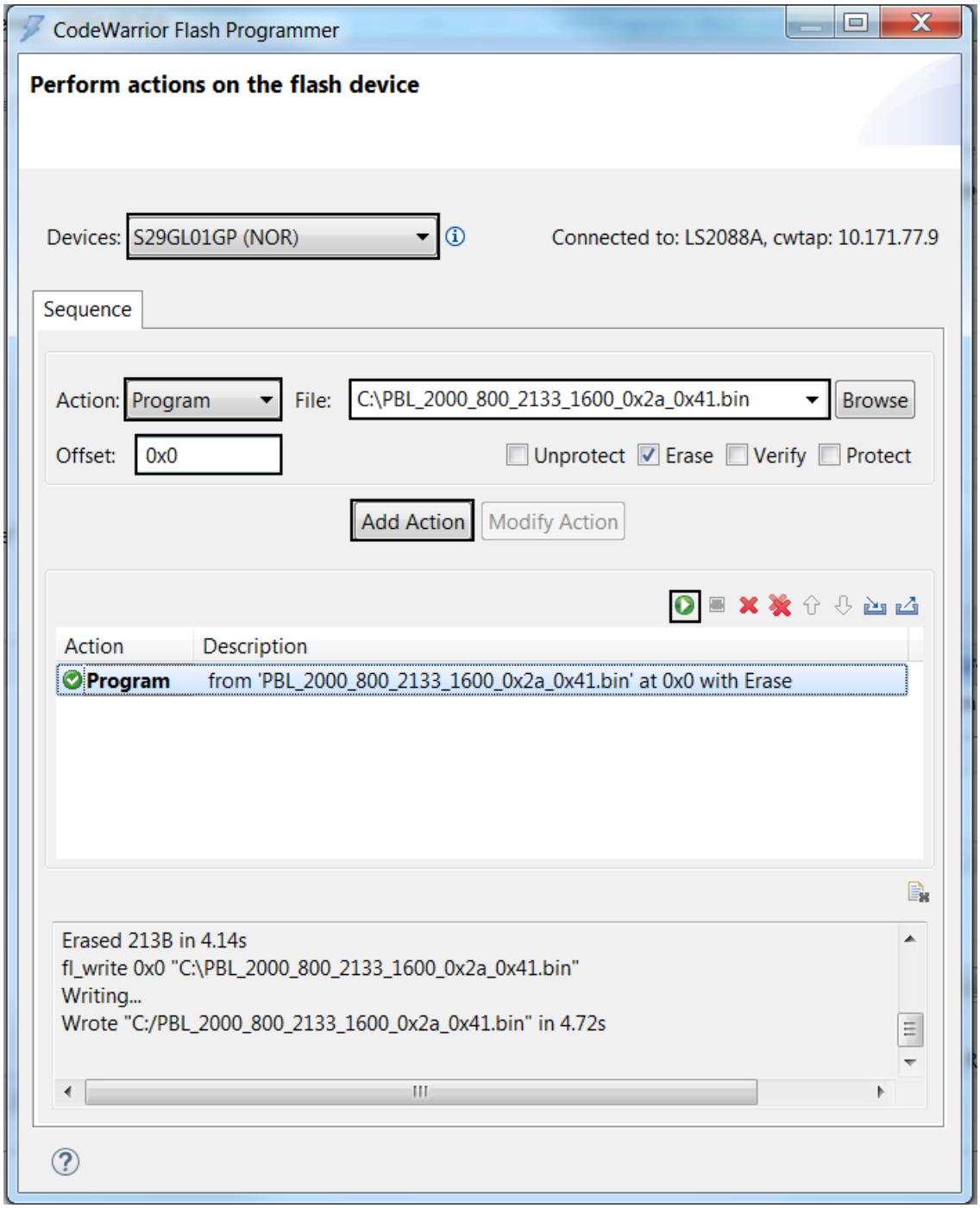


Figure 88. Program RCW file

5. Close the **CodeWarrior Flash Programmer** window and reset the board to load the new RCW.
6. For the next step, programming U-Boot, the changes done in step 4 from RCW Override need to be undone.

NOTE

If the flash device is QSPI, make sure the swapped image of RCW is used.

8.6.3 Program U-Boot in flash device using Flash Programmer

To program valid RCW in the flash device using Flash Programmer:

1. Click the **Flash Programmer** button to connect to the board.

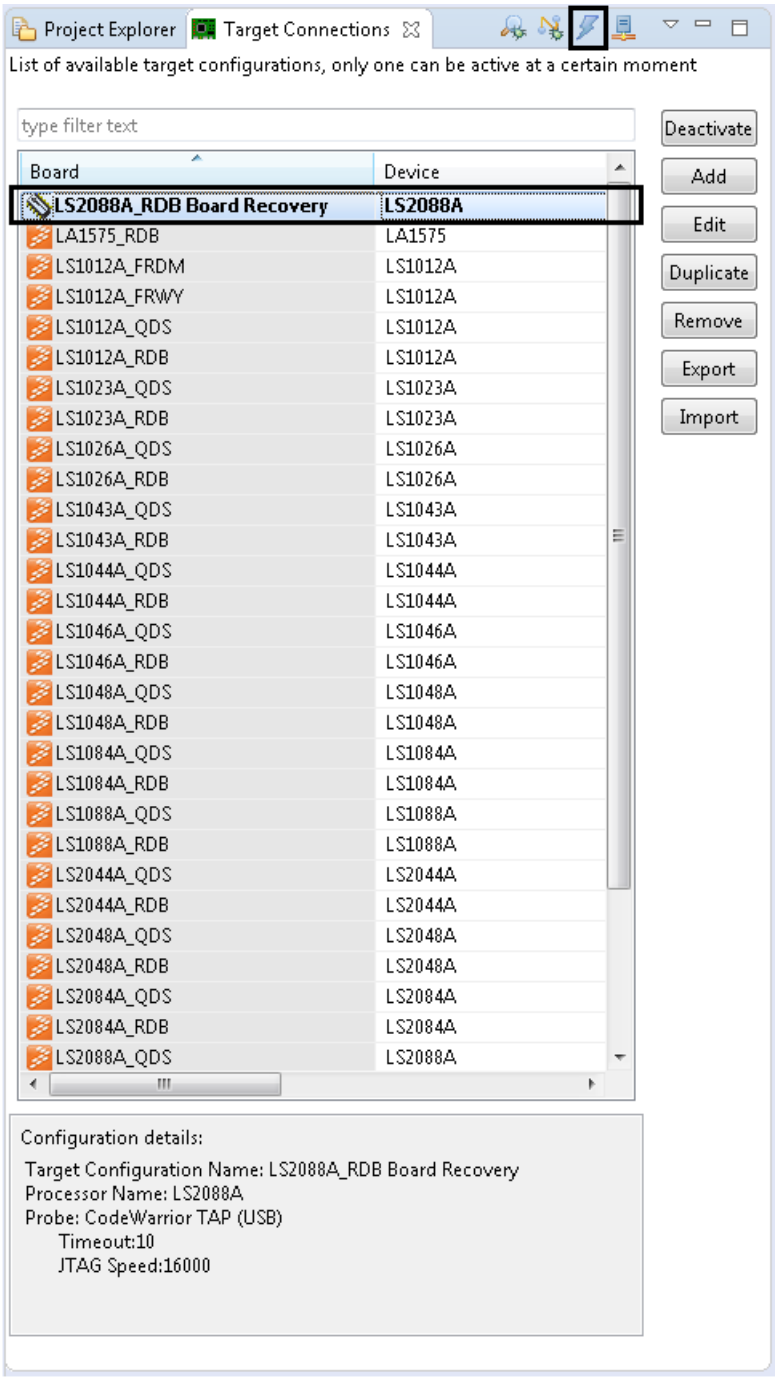


Figure 89. Flash Programmer button

2. When the CodeWarrior software connects to the board, the **CodeWarrior Flash Programmer** window appears.

3. Select the appropriate flash device, the Program action, the U-Boot file to be programmed, and the Offset.
4. Click **Add Action** and execute the flash programmer sequence.

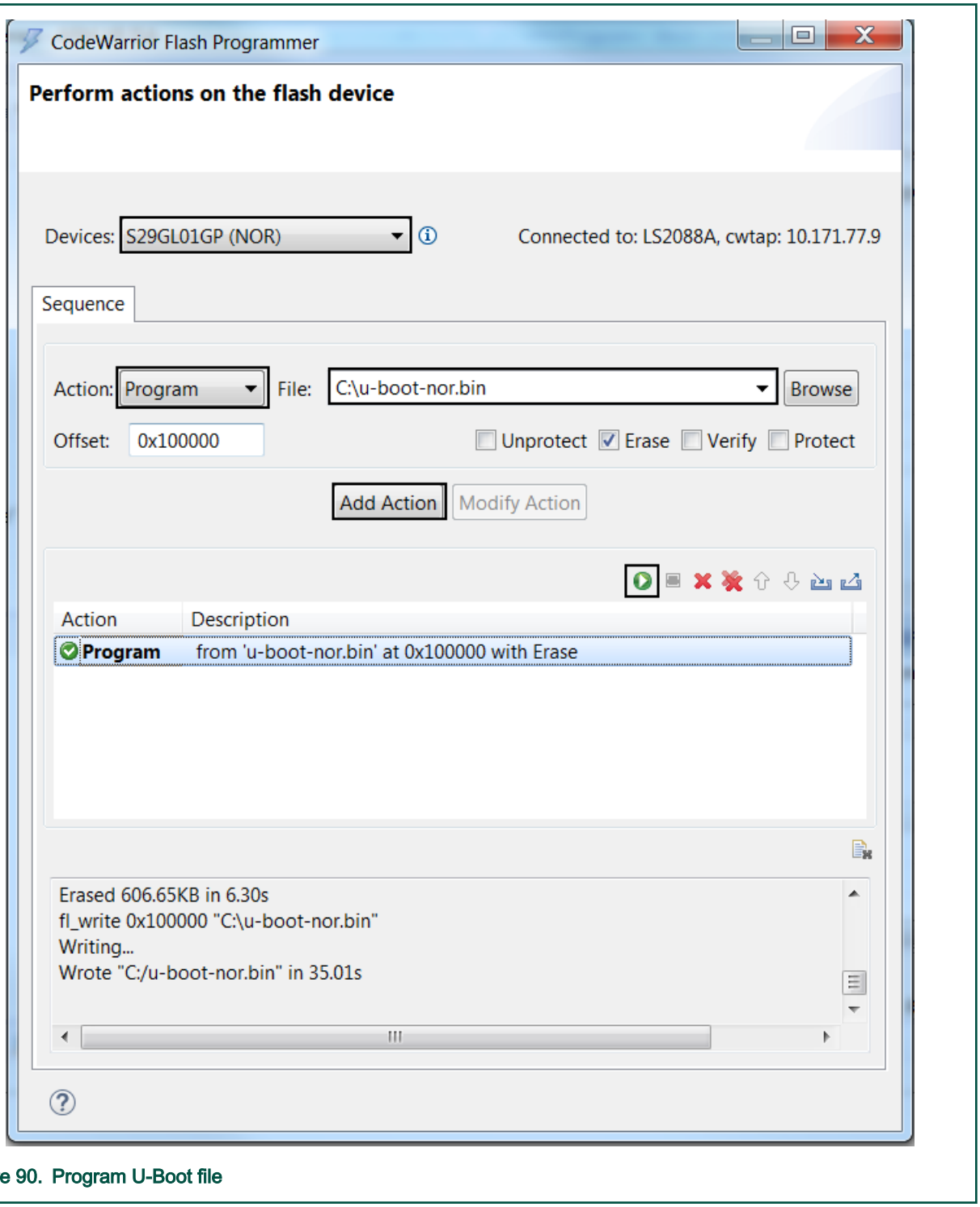


Figure 90. Program U-Boot file

5. Close the **CodeWarrior Flash Programmer** window and power-cycle the board.
6. Open a serial console to check the U-Boot prompt.

8.7 Secure Debug

The QorIQ LS parts can be secured with keys for debugging. This means that users will need a secure debug key that matches a challenge key associated with a given target in order to unlock it and perform regular debugging. Secure debug key can be specified in the **Target Connection** view by enabling the **Secure debug key** field and providing the needed key, as shown below.

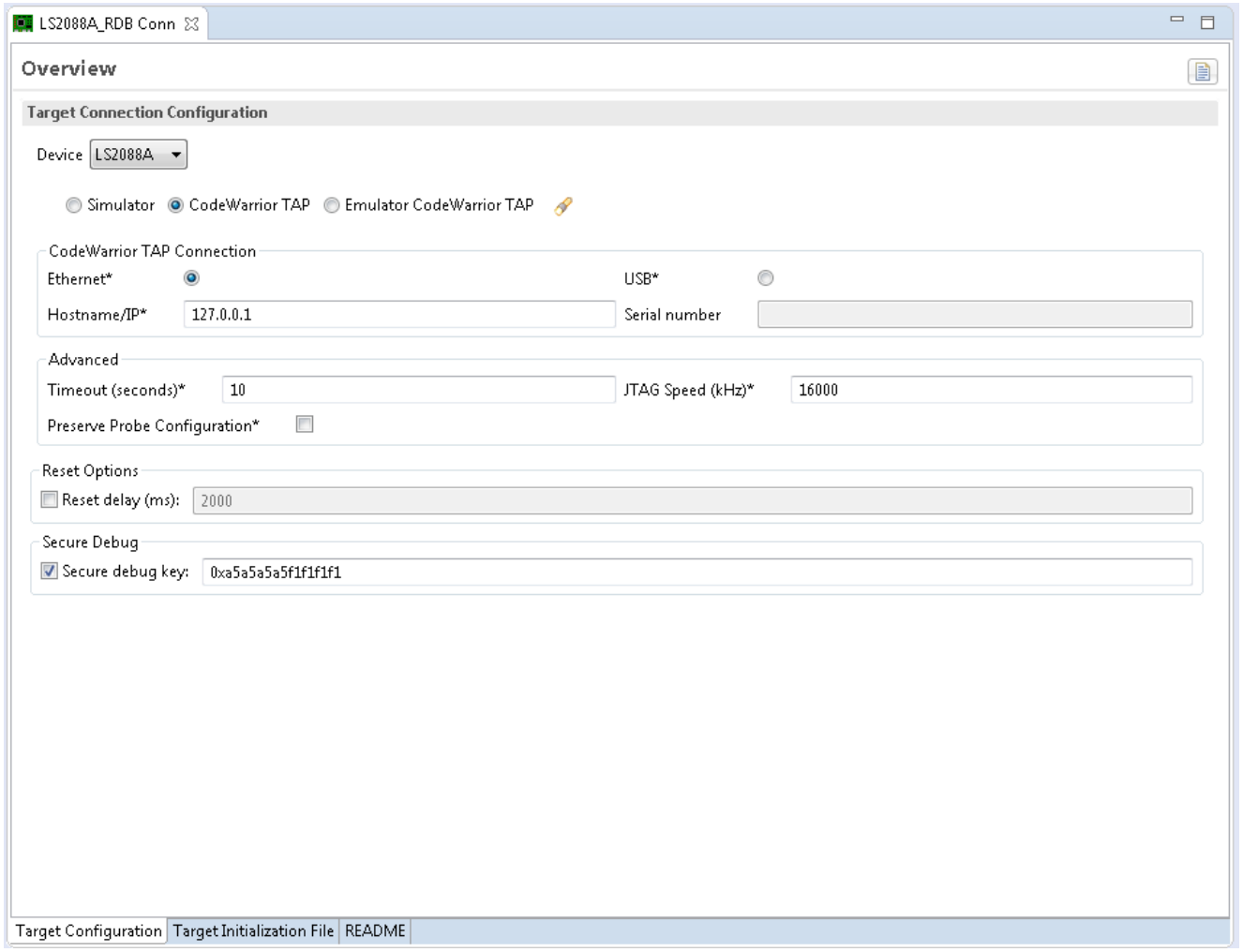


Figure 91.

If Secure debug key is not specified/enabled or an incorrect key is used, a "secure debug violation" error message appears when a debug session is attempted and unlocking fails. A challenge key providing a hint with respect to what secure debug key is required is mentioned in the error message, as shown below.

Figure 92.

After several failed attempts to unlock the target with a incorrect secure debug key, target is locked until a reset occurs. Note that debugger will attempt to perform the reset behind the scene but if the operation fails, a manual reset is required before being able to unlock the target with the provided secure debug key.

Chapter 9
Troubleshooting

This chapter provide troubleshooting details.

This section lists:

9.1 Diagnostic Information Export

The Diagnostic Information Wizard feature allows you to export error log information to NXP support group to diagnose the issue you have encountered while working on the CodeWarrior product.

You can export diagnostic information in the following two ways:

- Whenever an error dialog invokes to inform some exception has occurred, the dialog displays an option to open the Export wizard. You can then choose the files you want to send to NXP support.
- You can manually open the Export wizard to generate an archive of logs and files to report any issue that you have encountered.

9.1.1 General settings for Diagnostic Information

You can specify general settings for diagnostic information using the **Preferences** dialog.

To set general settings for diagnostic information, follow the steps given below:

1. Choose **Windows** > **Preferences** from the IDE menu bar.

   The **Preferences** dialog appears.

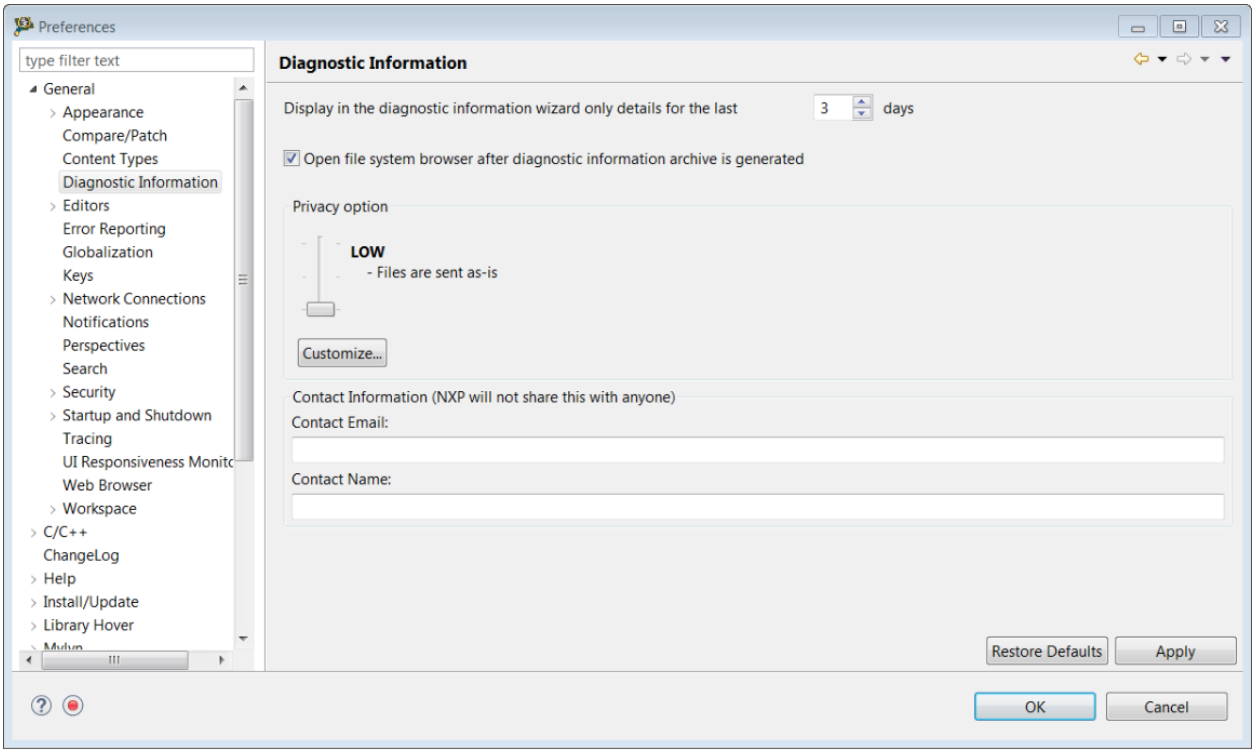


Figure 93. Preferences dialog - Diagnostic Information

2. Expand the **General** group and choose **Diagnostic Information** .

   The **Diagnostic Information** page appears.
3. Enter the number of days for which you want to display the diagnostic information details in the export wizard.
4. Select the **Privacy** option by dragging the bar to low, medium and high.

   Privacy level setting is used to filter the content of the logs.

   - Low: The file is sent as is.
   - Medium: The personal information is obfuscated. You can click on the customize option to view or modify filter.
   - High: The personal information is removed. Filters are used in the rest of the content.
5. Click **Customize** to set privacy filters.

   The **Customize Filters** dialog appears. You can add, remove, and modify filters.
6. Click **OK** .

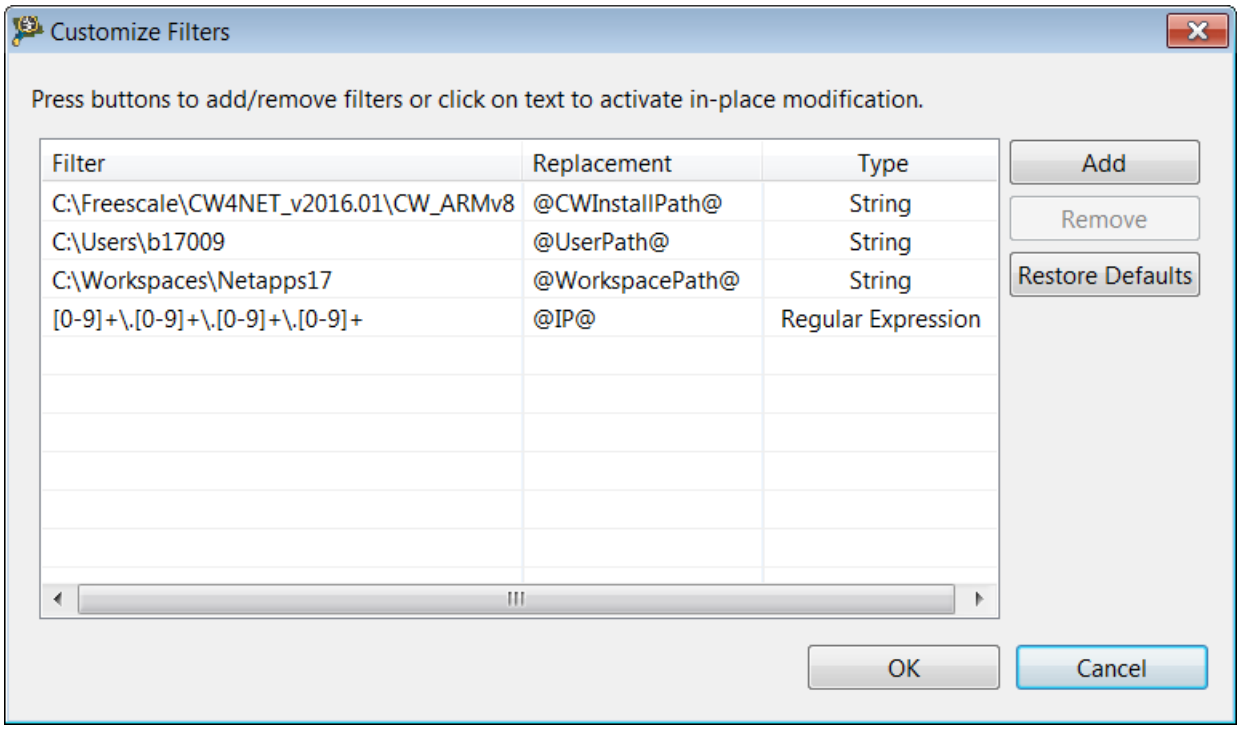


Figure 94. Diagnostic Information - Customize Filters

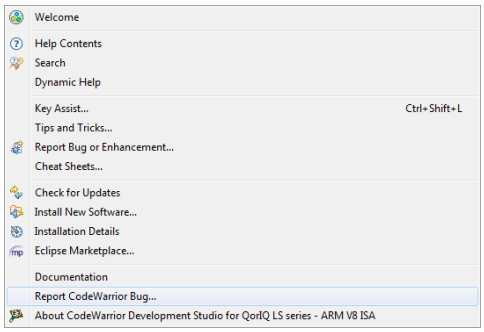
7. Enter **Contact Name** and **Contact Email** in the contact information textbox. This information is optional though NXP will not share this information with anyone.
8. Click **Restore Defaults** to apply default factory settings.
9. Click **OK** .

9.1.2 Export Diagnostic Information

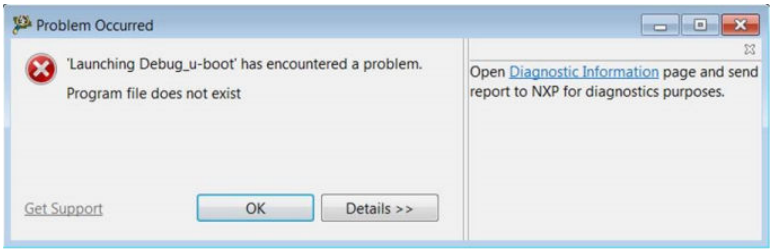
You can export diagnostic information into an archive file in workspace.

Follow the steps given below to export diagnostic information into an archive.

1. Open **Diagnostic Information** wizard, either by:
   - Selecting **Help > Report CodeWarrior Bug**, or



   - Through an error reporting dialog such as below. Click the **Diagnostic Information** link in the error dialog.



The **Diagnostic Information Wizard** appears.

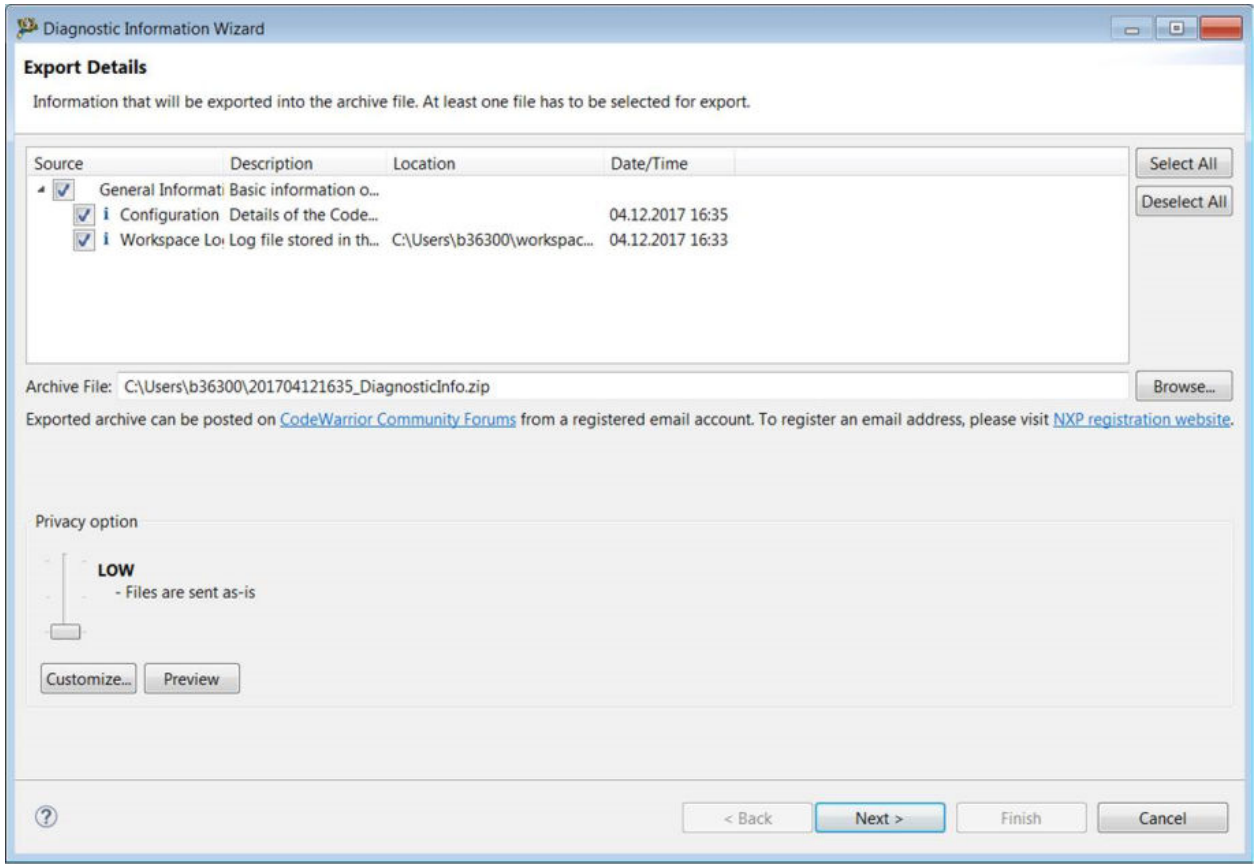


Figure 95. Export - Diagnostic Information Wizard

2. Select the checkbox under the **Source** column to select the information that will be exported into the archive file.

NOTE

You must select at least one file for export.

3. Click **Browse** to select a different archive file location.
4. Select the **Privacy option** or click **Customize** to set your privacy level. The **Customize Filters** dialog appears.

NOTE

You can open the **Customize Filters** dialog through **Customize** button in the **Diagnostic Information Export Wizard** (General settings for Diagnostic Information)or in the **Preferences** dialog (General settings for Diagnostic Information).

5. Click **Preview** to view the text that will be sent to NXP from the wizard.

The **Preview details** dialog appears.

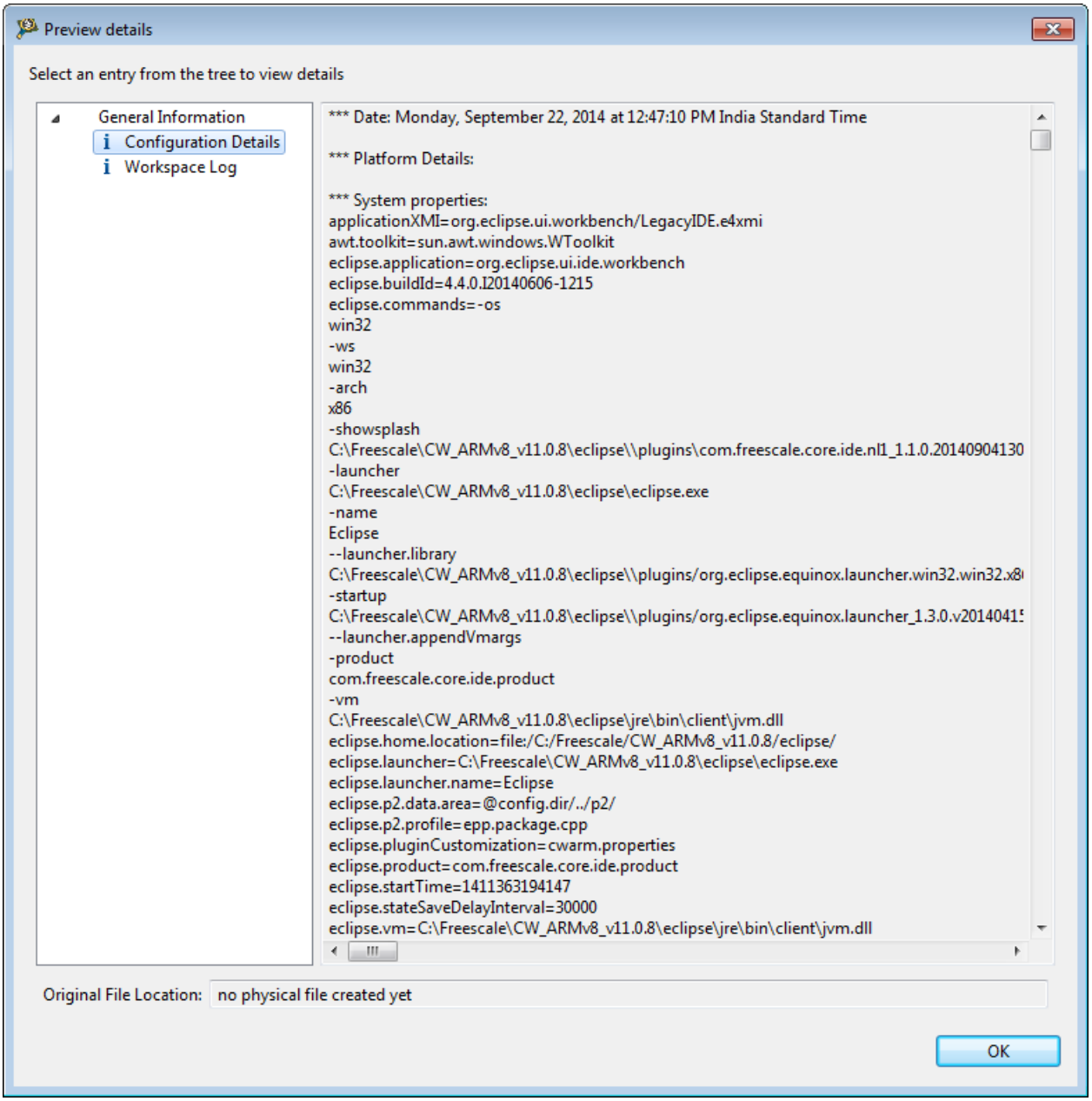


Figure 96. Preview details dialog

You can also check if more filters are needed to protect any sensitive information from leakage.

6. Click **OK** .
7. Click **Next** in the **Diagnostic Information Export Wizard**.

   The **Reproducible Details** page appears.

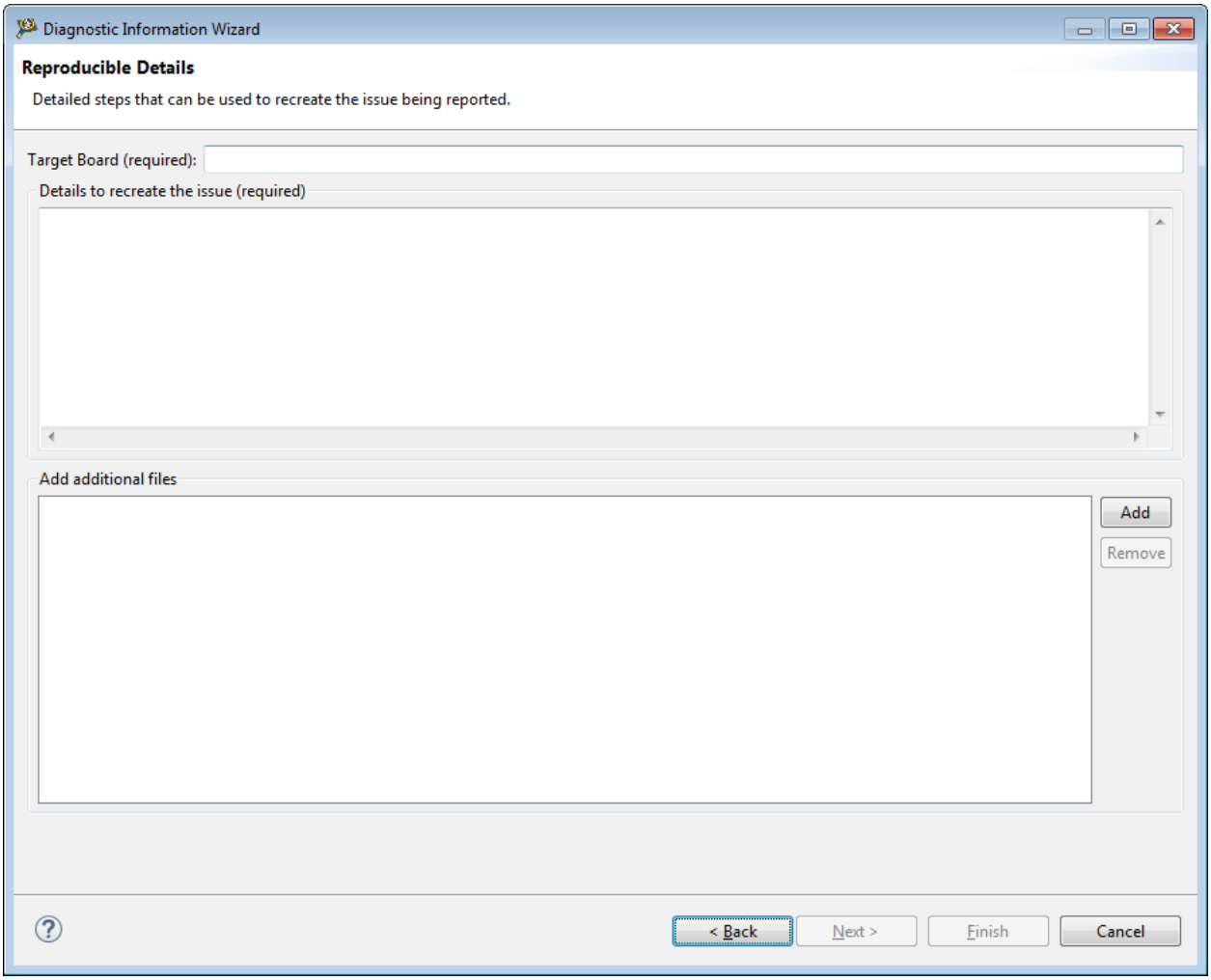


Figure 97. Reproducible Details page

8. Enter the reproducible steps and any other relevant information in the **Details to recreate the issue** textbox.
9. Click **Add** to add additional files to the archive file for diagnosis.
10. Click **Finish**.

9.2 Connection diagnostics

This topic explains how to use connection diagnostic and define custom diagnostic tests.

- Using connection diagnostics
- User-defined connection diagnostics tests

9.2.1 Using connection diagnostics

This feature is used to diagnose the selected connection. It can be launched in two ways:

- From the **Target Connections** view, using the **Diagnose Connection** button

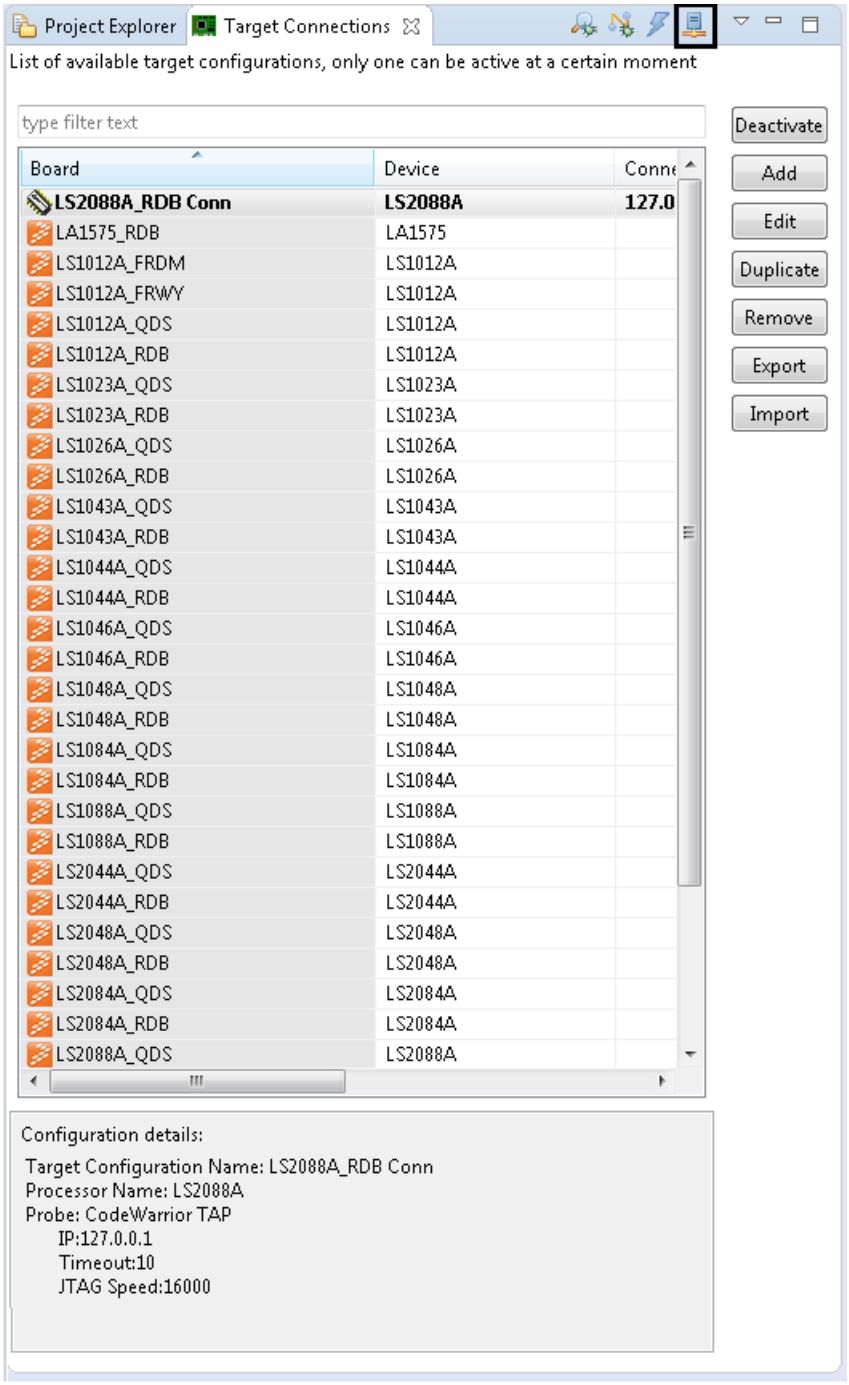


Figure 98. Diagnose connection from Target Connections view

- Directly from the error message shown by the CodeWarrior software when the connection to the target fails, using the **Diagnose** button.

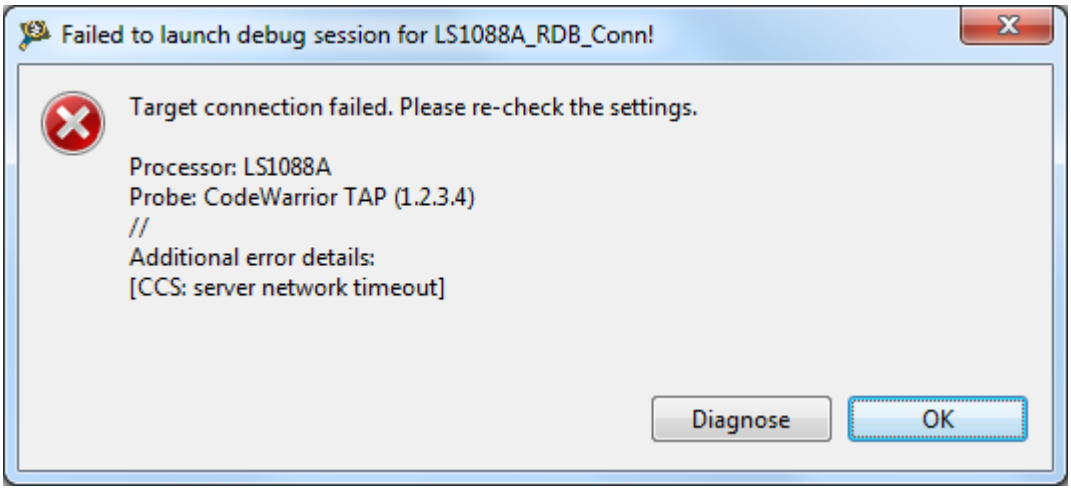


Figure 99. Diagnose connection from the error message

When a connection is diagnosed, the **Connection Diagnostics** view appears showing all the tests that are executed.

If the diagnostic is successful, all the tests in the list are preceded with a green icon.

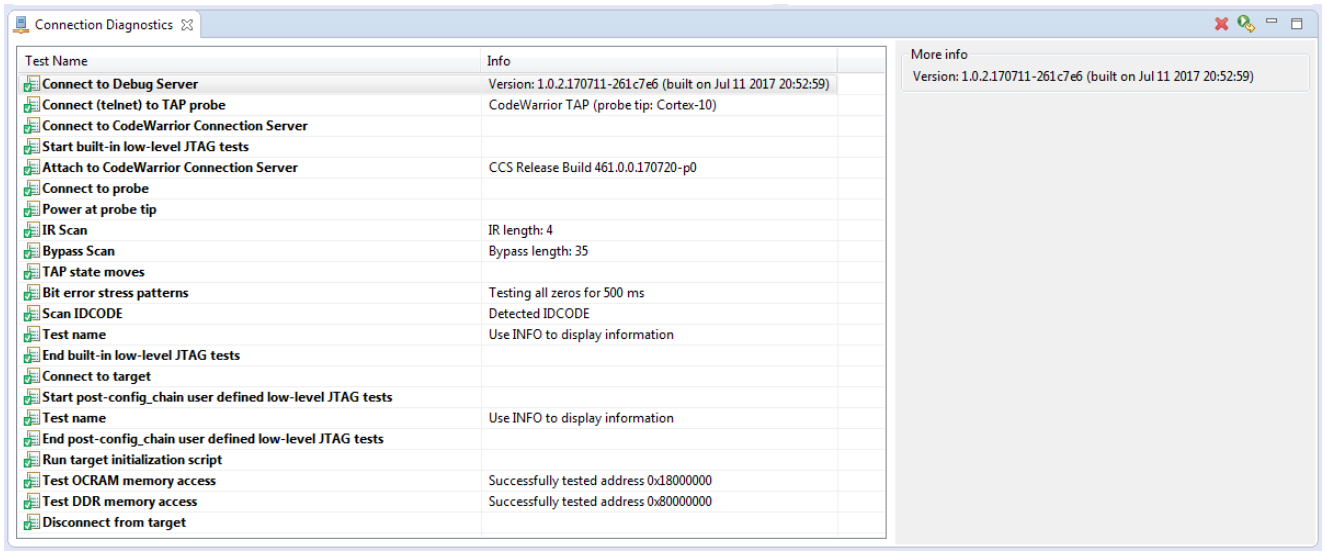


Figure 100. Successful connection

If one of the test fails, its entry appears with a red icon, as shown in the fiure below. When you select the entry, the right pane displays additional information and also some steps the you can try to address the problem.

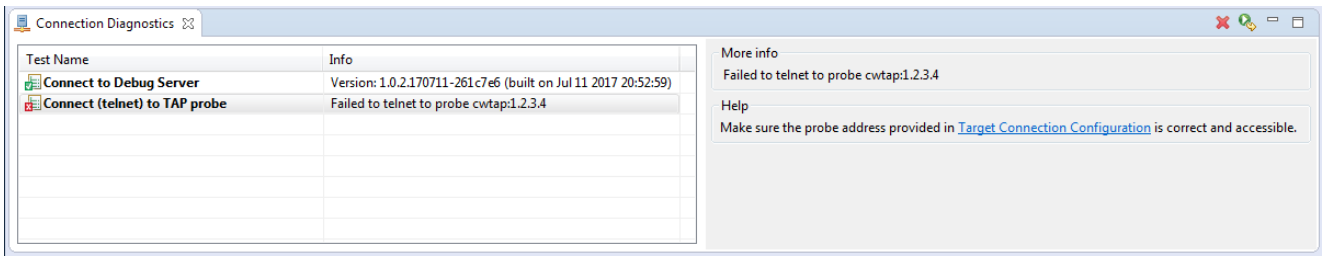


Figure 101. Unsuccessful connection

9.2.2 User-defined connection diagnostics tests

Connection Diagnostics allows the user to define custom tests.

Existing tests are defined in `{CW Folder}/CW_ARMv8/ARMv8/gdb_extensions/diagnostics/diag.py`. To add a new test, user has to add a new class derived from `Test` in this file. As example template of such a test is shown below:

```
class CustomTests(Test):

    name = "Test name"
    optional = False

    def __init__(self):
        Test.__init__(self, CustomTests.name)

    def body(self):
        self.fail_msg = "Message displayed in case test failed"
        self.help_msg = """
Displayed only in case the test failed. Meant to inform the user about the steps necessary to address
the issue.
It can be multiline.
"""

        try:
            INFO("Information message")
        except CWException as e:
            raise TestError()
```

After defining the test, you need to add the test to the list of tests to be executed. The lists are `suites_hw` for hardware tests and they can be found in the same `diag.py` file. The position of the test in the list of tests is important because the test will execute using the connection state from the previous test.

After the tests are added and executed, the **Connections Diagnostics** window shows the new tests in the list.

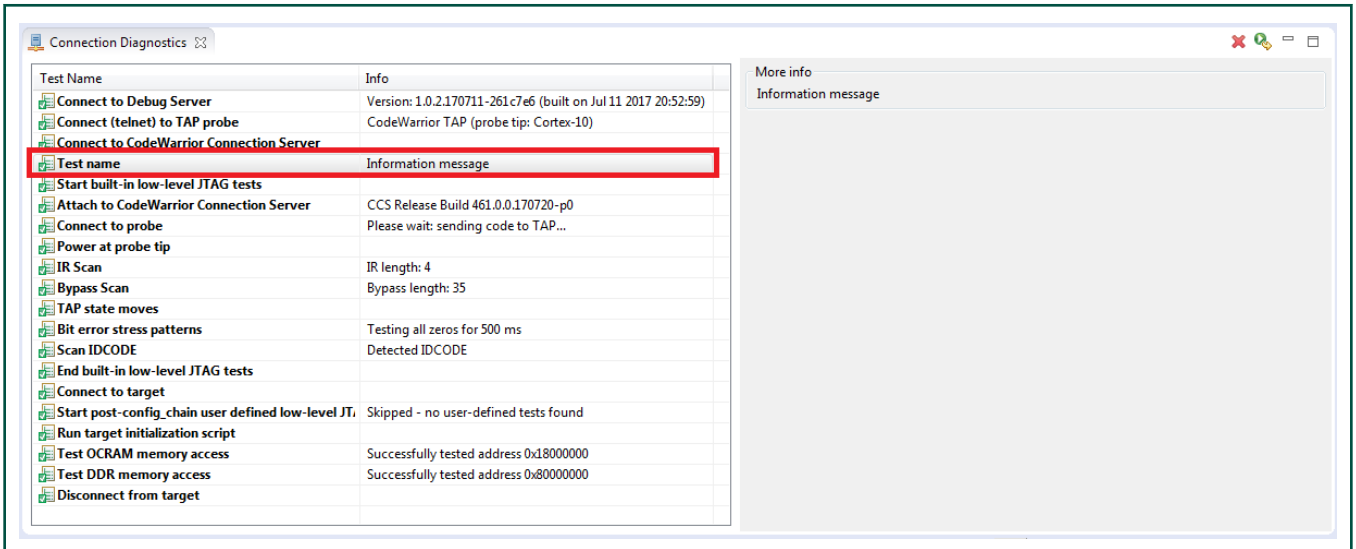


Figure 102. User-defined connection diagnostic

9.3 Prevent core from entering non-recoverable state due to unmapped memory access

The Arm core can enter in a non-recoverable state when a speculative access to an unmapped memory happens.

Also this can happen for accesses to memory regions that are marked as valid in the MMU, but the underlying memory interface is either misconfigured or absent. For example, access to a memory range dedicated to PCIe without a proper initialization for

the PCIe controller or access to memory holes as defined in the SoC memory map can cause core to enter in a non-recoverable state.

If the debugger detects a failed attempt to stop the core in such situation, it samples the value of the External PC Debug register (EDPCSR) in order to provide the program location where the program has hanged. An error message is displayed informing the user that the stop attempt has failed and listing the collected PC sample value.

Although the debug session is not reliable from this point onwards and must be terminated, the PC value allows the user to identify and fix the application problem that has caused the core to enter into the non-recoverable state. The user needs to make sure that the MMU is configured from the application in such a way that all valid translations point to the actual memory.

9.4 Board recovery in case of missing/corrupt RCW in IFC memory

Describes the process for recovering board when SoC fails to complete the reset sequence.

The principal mechanism for configuring the SoC during reset is the Reset Configuration Word (RCW). The RCW data can be pulled in from an external memory interface, such as flash memory, SD/MMC, I2C or from a set of hard-coded RCW options defined by the SoC. The most common setup for a board is using flash memory as the source for RCW data. If the flash memory is initially empty, or if the sector containing the RCW data is erased or corrupted due to flash programming operations, the SoC won't be able to complete the reset sequence.

The following subsections describe the procedure for recovering a board in this situation, depending on the available configuration options.

9.4.1 Board recovery using a hard-coded RCW option

This is the recommended procedure for recovery, if the board offers the possibility to select the RCW source.

All NXP evaluation boards include the necessary DIP switches that allow RCW source selection.

A hard-coded RCW option allows the reset sequence to bypass its RCW loading phase (the loading of RCW data from a non-volatile memory device). Instead the device is automatically configured according to the specific RCW field encodings pre-assigned for the given hard-coded RCW option.

1. Select one of the available hard-coded RCW options based on the rest of the board configuration (like SYSCLK, DDRCLK) such that the multipliers used by the hard-coded RCW option result in a valid PLL configuration.

   NOTE

   See the SoC Reference Manual for the list of available hard-coded RCW options.

2. Change the onboard DIP switches to select the above hard-coded RCW option as RCW_SRC.

   NOTE

   See the board User's Guide for the switch configuration information.

3. Use CodeWarrior Flash Programmer to burn a valid RCW configuration in the flash memory.

   NOTE

   See Flash Programmer for Flash Programming instructions.

4. Revert the on board DIP switches to select the flash `RCW_SRC` again and reset the board.

9.4.2 Board recovery by overriding RCW through JTAG

It may not be possible to use the above recommended procedure under all circumstances, for example in cases when a custom board design does not include switches to select `RCW_SRC`, or when physical access to the board is not possible.

In this situation, an alternative procedure involving overriding RCW though JTAG can be used.

1. Select one of the available hard-coded RCW options based on the rest of the board configuration (like SYSCLK, DDRCLK) such that the multipliers used by the hard-coded RCW option result in a valid PLL configuration

NOTE

See the SoC Reference Manual for the list of available hard-coded RCW options.

2. Edit the target initialization script corresponding to the board, for example:
   - `{CW Install Dir}\CW_ARMv8\Config\boards\<board_name>_init.gdb`
   - Specify that a safe RCW should be used by changing `useSafeRCW` from `False` to `True`
   - `useSafeRCW = True`
   - Also, in the Reset function make sure the desired RCW source is set (according to step 1).

   NOTE

   The initialization script already contains the value for the hard-coded RCW option that is appropriate for the default board configuration, but it may need to be changed if the board has been configured for different SYSCLK, DDRCLK frequencies.

3. Use CodeWarrior Flash Programmer to burn a valid RCW configuration in the flash memory.

   NOTE

   See Flash Programmer for Flash Programming instructions.

4. Revert the changes performed in the target initialization script in Step 2 and reset the board.

9.5 Logging

GDB logs are used to save output of the GDB commands to a file. There are two types of logs: GDB and GDB RSP server.

- GDB logs - Configured with standard GDB log control commands.

  For details about GDB log control commands, refer https://sourceware.org/gdb/onlinedocs/gdb/Logging-Output.html

- GDB RSP server log - Configured with GDB monitor commands. For details about GDB monitor commands, run the command `monitor help log`.

  The log messages from the GDB RSP server are grouped in different categories, and each category can be associated with one or more log destinations, such as console, file, and socket.

9.6 Recording

GDB provides the possibility to record all commands typed during a command-line debug session and save these to a file.

To enable this feature from command line GDB:

- (gdb) set history size unlimited – command history size defaults to 256; “unlimited” recommended
- (gdb) set history filename <filename> - the file where to save the recording (default: “.gdb_history”, located in the GDB executable home directory)
- (gdb) set history save on – all following commands will be recorded;

NOTE

The recorded command history is written to a file only upon exiting GDB.

After ending a debug session and exiting GDB, the “.gdb_history” file can be inspected and eventually edited. Optionally, when restarting the debug session, all commands from the recording may be replayed as a gdb script:

```
(gdb) source .gdb_history
```

9.7 NXP Licensing

The NXP Eclipse licensing feature lets the user see and manage the available licenses for the installed NXP products.

The NXP Eclipse Licensing feature appears to the user in two different ways:

- A *warning dialog box* appears after each time the CodeWarrior starts if a licensed product is going to expire soon, hasn't been activated yet, or is disabled because of license expiration.

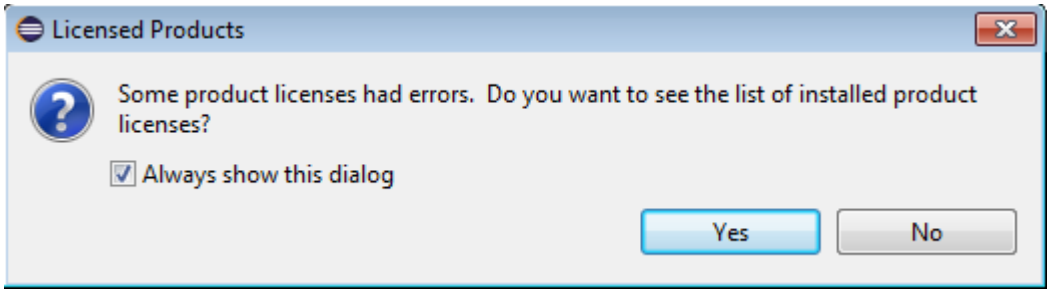


Figure 103. NXP Licensing warning dialog

- The *NXP Licenses* window displays all installed licensed products and their status ("licensed", "expiring in X days", "expired"). It can be opened from **Help > NXP Licenses**.

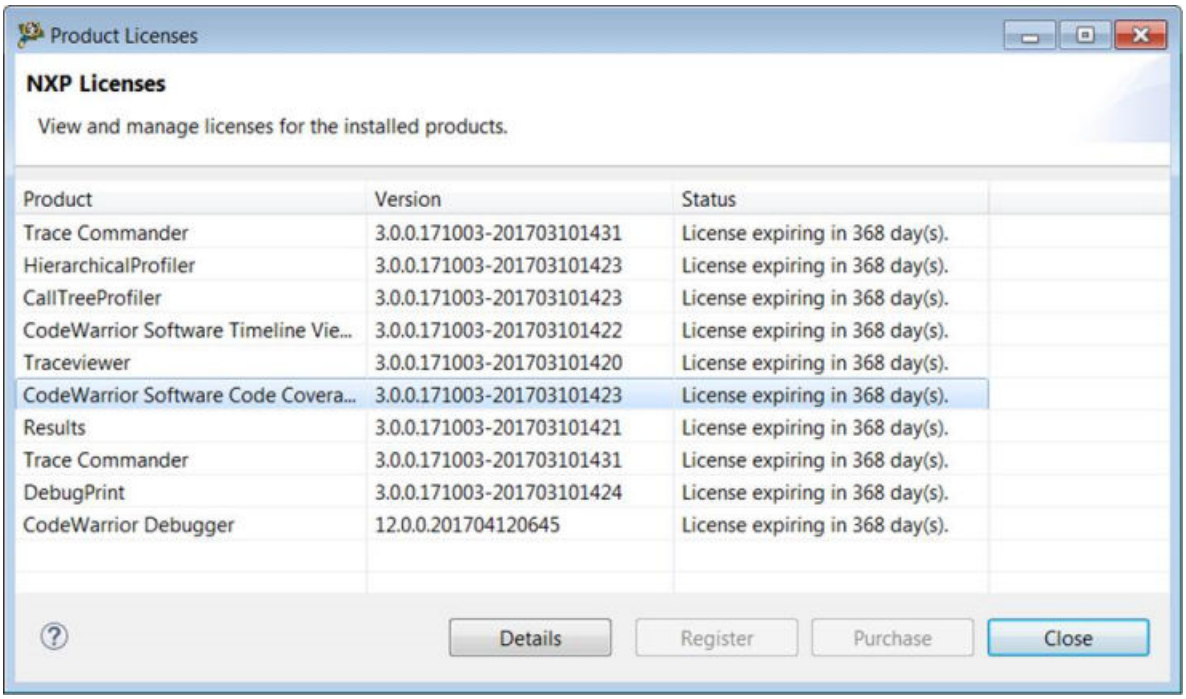


Figure 104. Product Licenses dialog

There is also a NXP Licenses preference panel which allows the user to customize specific aspects of the license plugins:

- whether the license expiration warning window should be displayed or not
- after how much delay, the expiration warning window should appear

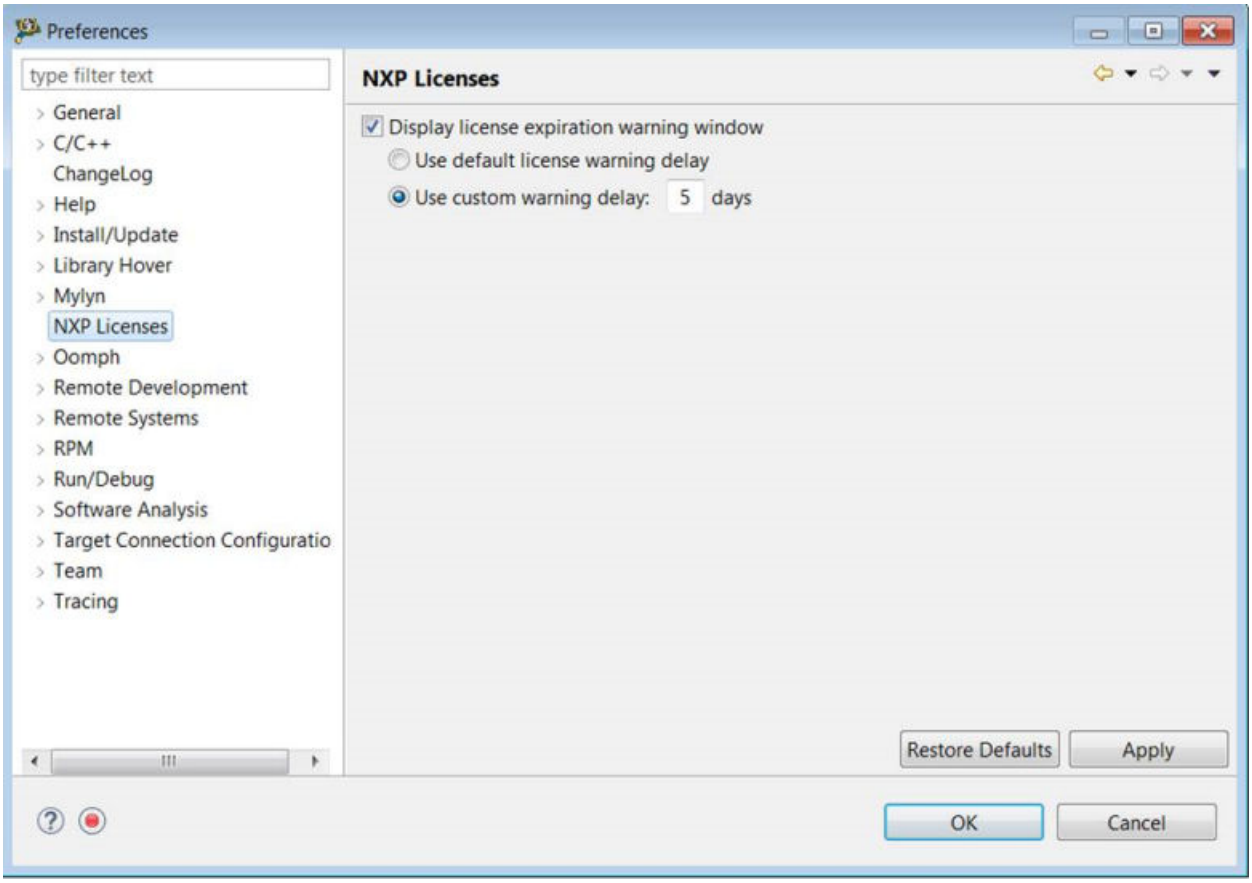


Figure 105. NXP Licensing preference page

NOTE

The NXP License plugin is not responsible for enabling or disabling a feature based on its license status, but only to monitor that status, and display it to the user. The plugin itself is responsible to enable or disable itself.

Index

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.

While NXP has implemented advanced security features, all products may be subject to unidentified vulnerabilities. Customers are responsible for the design and operation of their applications and products to reduce the effect of these vulnerabilities on customer’s applications and products, and NXP accepts no liability for any vulnerability that is discovered. Customers should implement appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP, the NXP logo, Freescale, the Freescale logo, and QorIQ are trademarks of are trademarks of NXP B.V. All other product or service names are the property of their respective owners. Arm, Cortex are trademarks or registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. The related technology may be protected by any or all of patents, copyrights, designs and trade secrets. All rights reserved.

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 06/2020
Document identifier: CWARMv8TM