CodeWarrior for ARMv7 Targeting Manual
Contents

Chapter 1 Introduction.................................................................................................9
  1.1 Release Notes........................................................................................................ 9
  1.2 Contents of this Manual........................................................................................ 9
  1.3 Accompanying documentation.......................................................................... 10
  1.4 ARM Embedded Application Binary Interface................................................10
  1.5 CodeWarrior Development Studio Tools..........................................................11
    1.5.1 Eclipse IDE.....................................................................................................11
    1.5.2 C/C++ Compiler............................................................................................11
    1.5.3 Assembler.......................................................................................................11
    1.5.4 Linker..............................................................................................................12
    1.5.5 Debugger..........................................................................................................12
    1.5.6 GNU newlib libraries......................................................................................12
  1.6 CodeWarrior IDE................................................................................................12
    1.6.1 Project files.....................................................................................................13
    1.6.2 Code editing....................................................................................................13
    1.6.3 Compiling........................................................................................................13
    1.6.4 Linking.............................................................................................................13
    1.6.5 Debugging.......................................................................................................14

Chapter 2 Working with Projects..................................................................................15
  2.1 CodeWarrior Bareboard Project Wizard............................................................15
    2.1.1 Create CodeWarrior Bareboard Project page..............................................16
    2.1.2 Processor page..............................................................................................17
    2.1.3 Debug Target Settings page.........................................................................18
    2.1.4 Build Settings page........................................................................................20
    2.1.5 Configurations page.......................................................................................22
  2.2 CodeWarrior Linux Project Wizard.................................................................23
    2.2.1 Create CodeWarrior Linux Project page.....................................................24
    2.2.2 Processor page..............................................................................................25
    2.2.3 Build Settings page.......................................................................................26
    2.2.4 Linux Application page................................................................................27
  2.3 Creating projects................................................................................................29
    2.3.1 Creating CodeWarrior Bareboard Application project...............................29
    2.3.2 Creating CodeWarrior Bareboard Library project.......................................31
    2.3.3 Creating CodeWarrior Linux Application project......................................32
  2.4 Building projects...............................................................................................34
    2.4.1 Manual-Build mode.......................................................................................34
    2.4.2 Auto-Build mode...........................................................................................35

Chapter 3 Build Properties........................................................................................37
  3.1 Changing build properties................................................................................37
  3.2 Restoring build properties................................................................................38
  3.3 Defining C/C++ build settings and behavior....................................................38
    3.3.1 Define build settings.....................................................................................38
    3.3.2 Define build behavior...................................................................................40
  3.4 Properties for <project>....................................................................................42
    3.4.1 Target Processor.........................................................................................43
Chapter 6 Multicore Debugging ................................................................. 113
  6.1 Debugging Multicore projects............................................................. 113
    6.1.1 Setting launch configurations....................................................... 113
    6.1.2 Debugging multiple cores............................................................ 116
  6.2 Multicore debugging commands......................................................... 119
    6.2.1 Multicore commands in CodeWarrior IDE................................. 119
    6.2.2 Multicore commands in Debugger Shell....................................... 120

Chapter 7 Debugging Embedded Linux Software ..................................... 125
  7.1 Debugging Linux application.............................................................. 125
    7.1.1 Install CodeWarrior TRK on target system................................. 125
    7.1.2 Start CodeWarrior TRK on target system..................................... 126
      7.1.2.1 TCP/IP connections.............................................................. 126
      7.1.2.2 Serial connections............................................................. 127
    7.1.3 Create a CodeWarrior Download Launch Configuration for the Linux Application........ 128
    7.1.4 Specify Console I/O redirections for Linux application.............. 131
    7.1.5 Configure Linux Process Signal Policy....................................... 132
      7.1.5.1 Signal inheritance.............................................................. 132
      7.1.5.2 Default Signal Policy......................................................... 132
      7.1.5.3 Modifying Signal Policy.................................................... 132
    7.1.6 Debug Linux application............................................................ 134
  7.2 Viewing multiple processes and threads............................................... 134
  7.3 Debugging Applications that Use fork() and exec() System Calls........... 135
Chapter 8 JTAG Configuration Files ............................................................. 171
  8.1 JTAG configuration file syntax......................................................................................... 171
  8.2 Using a JTAG configuration file to override RCW.......................................................... 172
  8.3 Using JTAG configuration file to specify multiple linked devices on a JTAG chain......... 173
  8.4 Setting up a remote system to use a JTAG configuration file........................................ 174

Chapter 9 Target Initialization Files .............................................................. 177
  9.1 Using target initialization files......................................................................................... 177

Chapter 10 Memory Configuration Files ....................................................... 181
  10.1 Using memory configuration files.................................................................................. 181

Chapter 11 Working with Hardware Tools .................................................... 183
  11.1 Flash programmer........................................................................................................ 183
    11.1.1 Create flash programmer target task................................................................. 183
    11.1.2 Configure flash programmer target task............................................................. 185
      11.1.2.1 Add flash device.............................................................................................. 186
      11.1.2.2 Specify target RAM settings............................................................................ 186
      11.1.2.3 Add flash programmer actions........................................................................ 186
    11.1.3 Execute flash programmer target task................................................................. 190
    11.1.4 SD/eMMC flash programmer.................................................................................... 192
      11.1.4.1 Programming U-Boot using SD/eMMC flash programmer.............................. 192
    11.1.5 Flash File to Target............................................................................................... 193
      11.1.5.1 Erasing flash device......................................................................................... 194
      11.1.5.2 Programming file............................................................................................ 194
    11.2 Hardware diagnostics............................................................................................... 195
      11.2.1 Creating hardware diagnostics task.................................................................... 195
Chapter 1
Introduction

This manual explains how to use CodeWarrior Development Studio tools to develop software for bareboard applications and embedded Linux® operating system running on NXP QorIQ LS series - ARM V7 ISA.

This chapter provides an overview of this manual and introduces you to the CodeWarrior development tools and development process.

This chapter includes the following sections:

- **Contents of this Manual** on page 9: Describes contents of this manual.
- **Accompanying documentation** on page 10: Describes the documentation included in this version of CodeWarrior Development Studio for QorIQ LS series - ARM V7 ISA.
- **ARM Embedded Application Binary Interface** on page 10: Provides information about the ARM Embedded Application Binary Interface (ARM EABI).
- **CodeWarrior Development Studio Tools** on page 11: Describes some important tools of CodeWarrior Development Studio.
- **CodeWarrior IDE** on page 12: Explains the CodeWarrior IDE and tells how to perform basic IDE operations.

### 1.1 Release Notes

Release notes include information about new features, last-minute changes, bug fixes, incompatible elements, or other sections that may not be included in this manual.

You should read release notes before using the CodeWarrior IDE.

### 1.2 Contents of this Manual

Each chapter of this manual describes a different area of software development.

The table below lists each chapter in the manual.

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Introduction</strong> on page 9</td>
<td>This chapter.</td>
</tr>
<tr>
<td><strong>Working with Projects</strong> on page 15</td>
<td>Describes the different types of projects you can create, provides an overview of CodeWarrior project wizards.</td>
</tr>
<tr>
<td><strong>Build Properties</strong> on page 37</td>
<td>Explains build properties for QorIQ LS series - ARM V7 ISA projects.</td>
</tr>
<tr>
<td><strong>Debug Configurations</strong> on page 53</td>
<td>Describes the different types of launch configurations you can create, provides an overview of the debugger.</td>
</tr>
</tbody>
</table>

*Table continues on the next page...*
Table 1: Organization of this Manual (continued)

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Working with Debugger</strong> on page 79</td>
<td>Explains various aspects of CodeWarrior debugging, such as debugging a project, configuring connections, setting breakpoints and watchpoints, working with registers, viewing memory, viewing cache, and debugging externally built executable files.</td>
</tr>
<tr>
<td><strong>Multicore Debugging</strong> on page 113</td>
<td>Explains multi-core debugging capabilities of CodeWarrior debugger.</td>
</tr>
<tr>
<td><strong>JTAG Configuration Files</strong> on page 171</td>
<td>Explains JTAG configuration files that pass specific configuration settings to the debugger and support chaining of multiple devices.</td>
</tr>
<tr>
<td><strong>Target Initialization Files</strong> on page 177</td>
<td>Discusses how to use a target initialization file, describes target initialization file commands, and explains how to perform target initialization using a tcl script.</td>
</tr>
<tr>
<td><strong>Memory Configuration Files</strong> on page 181</td>
<td>Discusses how to use a memory configuration file and describes memory configuration file commands.</td>
</tr>
<tr>
<td><strong>Working with Hardware Tools</strong> on page 183</td>
<td>Explains CodeWarrior hardware tools used for board bring-up, test, and analysis.</td>
</tr>
</tbody>
</table>

1.3 Accompanying documentation

The Documentation page describes the documentation included in this version of CodeWarrior Development Studio for QorIQ LS series - ARM V7 ISA.

You can access the Documentation page by:

- Opening `START_HERE.html` from the `<CWInstallDir>CW_ARMv7\ARMv7\Help` folder
- Choosing Help > Documentation from the CodeWarrior IDE menu bar

1.4 ARM Embedded Application Binary Interface

The ARM Embedded Application Binary Interface (ARM EABI) specifies data structure alignment, calling conventions, and other information about how high-level languages can be implemented on an ARM processor.

The code generated by CodeWarrior for ARMv7 conforms to the ARM EABI. To learn more about the ARM EABI, information and documentation about all supported ARM hardware is available here: ARM® Technology-Based Solutions

The ARM EABI also specifies the object and symbol file format. It specifies Executable and Linkable Format (ELF) as the output file format and Debug With Arbitrary Record Formats (DWARF) as the debugging information format. For more information about these formats, see:

- DWARF Debugging Standard website available at: infocenter.arm.com
1.5 CodeWarrior Development Studio Tools

This section talks about some important tools of CodeWarrior Development Studio.

Programming for processors is much like programming for any other CodeWarrior platform target. If you have not used CodeWarrior tools before, start by studying the Eclipse IDE, which is used to host the tools. For more details on IDE, see CodeWarrior Development Studio Common Features Guide available in the `<CWInstallDir>`\CW_ARMv7\ARMv7\Help\PDF folder.

The following are some important tools of CodeWarrior Development Studio:

- Eclipse IDE on page 11
- C/C++ Compiler on page 11
- Assembler on page 11
- Linker on page 12
- Debugger on page 12
- GNU newlib libraries on page 12

1.5.1 Eclipse IDE

The Eclipse Integrated Development Environment (IDE) is an open-source development environment that lets you develop and debug your software.

It controls the project manager, the source code editor, the class browser, the compilers and linkers, and the debugger. The Eclipse workspace organizes all files related to your project. This allows you to see your project at a glance and navigate easily through the source code files.

The Eclipse IDE has an extensible architecture that uses plug-in compilers and linkers to target various operating systems and microprocessors. The IDE can be hosted on Microsoft Windows, Linux, and other platforms. There are many development tools available for the IDE, including C, C++, and Java compilers for desktop and embedded processors.

For more information about the Eclipse IDE, read the Eclipse documentation at:
http://www.eclipse.org/documentation/

1.5.2 C/C++ Compiler

The CodeWarrior ARMv7 C/C++ compiler is an ANSI-compliant compiler.

It compiles C and C++ statements and assembles inline assembly language statements. You can generate ARMv7 applications and libraries that conform to the ARM EABI by using the CodeWarrior compiler in conjunction with the CodeWarrior linker for ARMv7 processors.

The IDE manages the execution of the compiler. The IDE invokes the compiler if you:

- Change a source file and issue the make command.
- Select a source file in your project and issue the compile, preprocess, or precompile command.

1.5.3 Assembler

The CodeWarrior ARMv7 assembler is a standalone assembler that translates assembly-language source code to machine-language object files or executable programs.

Either you can provide the assembly-language source code to the assembler, or the assembler can take the assembly-language source code generated by the compiler.
For each assembly-language module in a build target, the ARMv7 assembler can generate a file that lists the generated code side-by-side with the assembly-language source code.

### 1.5.4 Linker

The CodeWarrior ARMv7 linker generates binaries that conform to the ARM EABI (Embedded Application Binary Interface).

The linker combines object modules created by the compiler and/or assembler with modules in static libraries to produce a binary file in executable and linkable (ELF) format.

Among many powerful features, the linker lets you:

- Use absolute addressing
- Create multiple user-defined sections
- Generate S-Record files
- Generate PIC/PID binaries

The IDE runs the linker each time you build your project.

### 1.5.5 Debugger

The CodeWarrior ARMv7 debugger controls the execution of your program and allows you to see what is happening internally as the program runs.

You can use the debugger to find problems in your program. The debugger can execute your program one statement at a time and suspend execution when control reaches a specified point. When the debugger stops a program, you can view the chain of function calls, examine and change the values of variables, and inspect the contents of registers.

The debugger allows you to debug your CodeWarrior project using target hardware. The ARMv7 debugger communicates with the board through a hardware probe (such as the CodeWarrior TAP).

### 1.5.6 GNU newlib libraries

The GNU newlib runtime libraries are ANSI-compliant C and C++ standard libraries.

Use these libraries to help you create applications for processors. The versions of the GNU newlib runtime libraries have been customized and the runtime has been adapted for processor development.

For more information about GNU newlib, see *GNU C Reference* and *GNU C++ Reference*.

### 1.6 CodeWarrior IDE

While working with the CodeWarrior IDE, you will proceed through the development stages familiar to all programmers: writing code, compiling and linking, and debugging.

The difference between the CodeWarrior development environment and traditional command-line environments is how the software, in this case the CodeWarrior IDE, helps you manage your work more effectively.

See *CodeWarrior Development Studio Common Features Guide* for:

- Complete information on tasks, such as editing, compiling, and linking
- Basic information on debugging

The following sections explain the CodeWarrior IDE and describe how to perform basic CodeWarrior IDE operations:
• Project files
• Code editing
• Compiling
• Linking
• Debugging

1.6.1 Project files
A CodeWarrior project is analogous to a set of make files, because a project can have multiple settings that are applied when building the program.

For example, you can have one project that has both a debug version and a release version of your program. You can build one or the other, or both as you wish. The different settings used to launch your program within a single project are called launch configurations.

The CodeWarrior IDE uses the CodeWarrior Projects view to list all the files in a project. A project includes files, such as source code files and libraries. You can add or remove files easily. You can assign files to one or more different build configurations within the project, so files common to multiple build configurations can be managed simply.

The CodeWarrior IDE itself manages all the interdependencies between files and tracks which files have changed since the last build.

The CodeWarrior IDE also stores the settings for the compiler and linker options for each build configuration. You can modify these settings using the IDE, or with the #pragma statements in your code.

1.6.2 Code editing
CodeWarrior IDE has an integral text editor designed for programmers. It handles text files in ASCII, Microsoft® Windows®, and UNIX® formats.

To edit a file in a project, double-click the file name in the CodeWarrior Projects view. CodeWarrior IDE opens the file in the editor associated with the file type.

The editor view has excellent navigational features that allow you to switch between related files, locate any particular function, mark any location within a file, or go to a specific line of code.

1.6.3 Compiling
To compile a source code file, it must be among the files that are part of the current launch configuration.

If the file is in the configuration, select it in the CodeWarrior Projects view and select Project > Build Project from the CodeWarrior IDE menu bar.

To automatically compile all the files in the current launch configuration after you modify them, select Project > Build Automatically from the CodeWarrior IDE menu bar.

1.6.4 Linking
Choose Project > Build Project from the CodeWarrior IDE menu bar to link object code into a final binary file.

The Build Project command makes the active project up-to-date and links the resulting object code into a final output file.

You can control the linker through the IDE. There is no need to specify a list of object files. The workspace tracks all the object files automatically.

You can also modify the build configuration settings to specify the name of the final output file.
1.6.5 Debugging

Choose Run > Debug from the CodeWarrior IDE menu bar to debug your project.

This command downloads the current project's executable to the target board and starts a debug session.

NOTE

The CodeWarrior IDE uses the settings in the launch configuration to generate debugging information and initiate communications with the target board.

You can now use the debugger to step through the program code, view and change the value of variables, set breakpoints, and much more.
Chapter 2
Working with Projects

This chapter explains how to create and build projects for NXP QorIQ LS series - ARM V7 ISA processors using the CodeWarrior tools.

This chapter explains:
• CodeWarrior Bareboard Project Wizard on page 15
• CodeWarrior Linux Project Wizard on page 23
• Creating projects on page 29
• Building projects on page 34

2.1 CodeWarrior Bareboard Project Wizard

The term bareboard refers to hardware systems that do not need an operating system to operate. The CodeWarrior Bareboard Project Wizard presents a series of pages that prompt you for the features and settings to be used when making your program.

This section describes the various pages that the CodeWarrior Bareboard Project Wizard displays as it assists you in creating a bareboard project.

The pages of the CodeWarrior Bareboard Project Wizard are:
• Create CodeWarrior Bareboard Project page on page 16
• Processor page on page 17
• Debug Target Settings page on page 18
• Build Settings page on page 20
• Configurations page on page 22
2.1.1 Create CodeWarrior Bareboard Project page

Use this page to specify the project name and the directory where the project files are located.

Figure 1: Create CodeWarrior Bareboard Project page

The table below describes the various options available on the **Create a CodeWarrior Bareboard Project** page.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project name</td>
<td>Enter the name for the project in this text box.</td>
</tr>
<tr>
<td>Use default location</td>
<td>Select to choose the directory to store the files required to build the program. Use the Location option to select the desired directory.</td>
</tr>
<tr>
<td>Location</td>
<td>Specifies the directory that contains the project files. Use <strong>Browse</strong> to navigate to the desired directory. This option is only available when <strong>Use default location</strong> is cleared.</td>
</tr>
</tbody>
</table>
2.1.2 Processor page

This page displays the target devices supported by the current installation. Use this page to specify the type of processor and the output for the new project.

Figure 2: Processor page

The table below describes the various options available on the Processor page.

Table 3: Processor page settings

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Expand the processor family tree and select a supported target. The toolchain uses this choice to generate code that makes use of processor-specific features, such as multiple cores.</td>
</tr>
<tr>
<td>Project Output</td>
<td>Select any one of the following supported project output:</td>
</tr>
<tr>
<td></td>
<td>• <strong>Application:</strong> Select to create an application with &quot;.elf&quot; extension, that includes information related to the debug over a board.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Static Library:</strong> Select to create a library with &quot;.a&quot; extension, that can be included in other projects. Library files created using this option do not include board specific details.</td>
</tr>
</tbody>
</table>
2.1.3 Debug Target Settings page

Use this page to select debugger connection type, board type, launch configuration type, and connection type for your project.

This page also lets you configure connection settings for your project.

The table below describes the various options available on the **Debug Target Settings** page.

---

**Table:**

<table>
<thead>
<tr>
<th>Option</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debugger Connection Types:</td>
<td>Hardware</td>
</tr>
<tr>
<td>Board</td>
<td>LS1021ATWR</td>
</tr>
<tr>
<td>Launch</td>
<td>Connection</td>
</tr>
<tr>
<td>Download</td>
<td>Default</td>
</tr>
<tr>
<td>Attach</td>
<td>Default</td>
</tr>
<tr>
<td>Attach ROM</td>
<td>Default</td>
</tr>
<tr>
<td>Connect</td>
<td>Default</td>
</tr>
<tr>
<td>Download OCMR</td>
<td>Default</td>
</tr>
<tr>
<td>Connect OCMR</td>
<td>Default</td>
</tr>
<tr>
<td>Connection Type</td>
<td>CMSIS-DAP</td>
</tr>
<tr>
<td>TAP address</td>
<td></td>
</tr>
</tbody>
</table>

---

Figure 3: Debug Target Settings page
### Table 4: Debug Target Settings page

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Debugger Connection Types</strong></td>
<td>Specifies the available target types:</td>
</tr>
<tr>
<td></td>
<td>• <strong>Hardware</strong> - Select to execute the program on the target hardware available.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Emulator</strong> - Select to execute the program on a hardware emulator.</td>
</tr>
<tr>
<td><strong>Board</strong></td>
<td>Specifies the hardware supported by the selected processor. It could be either QDS or Tower board.</td>
</tr>
<tr>
<td><strong>Launch</strong></td>
<td>Specifies the launch configurations and corresponding connection, supported by the selected processor.</td>
</tr>
<tr>
<td><strong>Connection Type</strong></td>
<td>Specifies the interface to communicate with the hardware.</td>
</tr>
<tr>
<td></td>
<td>• <strong>CMSIS-DAP</strong> - Select to use the CMSIS-DAP interface to communicate with the hardware device. This is the default connection type for Tower boards.</td>
</tr>
<tr>
<td></td>
<td>• <strong>CodeWarrior TAP (over USB)</strong> - Select to use the USB interface to communicate with the hardware device.</td>
</tr>
<tr>
<td></td>
<td>• <strong>CodeWarrior TAP (over Ethernet)</strong> - Select to use the Ethernet interface to communicate with the hardware device.</td>
</tr>
<tr>
<td><strong>TAP address</strong></td>
<td>Enter the IP address of the selected TAP device. Enter the IP address of the TAP device here. This option is available only if CodeWarrior TAP (over Ethernet) is selected as the connection type.</td>
</tr>
</tbody>
</table>
2.1.4 Build Settings page

Use this page to select a programming language, toolchain, and the output project type for the new project for your project.

Figure 4: Build Settings page

The table below describes the various options available on the Build Settings page.
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>Specifies the programming language used by the new project. The current installation supports the following languages:</td>
</tr>
<tr>
<td></td>
<td>• C - Select to generate ANSI C-compliant startup code, and initializes global variables.</td>
</tr>
<tr>
<td></td>
<td>• C++ - Select to generate ANSI C++ startup code, and performs global class object initialization.</td>
</tr>
<tr>
<td></td>
<td>• ASM - Select to generate Assembly startup code.</td>
</tr>
<tr>
<td>I/O Support</td>
<td>• <strong>Semihosting I/O Support</strong> - Configures how the application deals with the I/O calls. For this case, it uses a virtual connection with the debugger (also known as 'semi hosting').</td>
</tr>
<tr>
<td></td>
<td>• UART I/O - Configures how the application deals with the I/O calls. With 'UART' it uses the physical serial device connection.</td>
</tr>
<tr>
<td></td>
<td>• No I/O - No I/O support.</td>
</tr>
<tr>
<td>Toolchain</td>
<td>Specifies the toolchains supported by the current installation. Selected toolchain sets up the default compiler, linker, and libraries used to build the new project. Each toolchain generates code targeted for a specific platform.</td>
</tr>
<tr>
<td>Floating Point</td>
<td>Specifies how the compiler handles floating-point operations, encountered in the source code.</td>
</tr>
</tbody>
</table>
2.1.5 Configurations page

Use this page to specify the processing model and the processor core that executes the project.

Figure 5: Configurations page

The table below describes the various options available on the Configurations page.
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing Model</td>
<td>The current installation supports the following processing models:</td>
</tr>
<tr>
<td></td>
<td>• SMP - This option is disabled for this installation.</td>
</tr>
<tr>
<td></td>
<td>• AMP (one project per core) - Select this option to generate a separate project for each selected core. The option will also set the core index for each project based on the core selection.</td>
</tr>
<tr>
<td></td>
<td>• AMP (one build configuration per core) - Select this option to generate one project with multiple targets, each containing an lcf file for the specified core.</td>
</tr>
<tr>
<td></td>
<td><strong>NOTE</strong></td>
</tr>
<tr>
<td></td>
<td>Selecting the AMP (One build configuration per core) option displays a checkbox, Set up build references for build configurations of all cores, just below this option. If you select the Set up build references for build configurations of all cores checkbox, then building the project for one core will automatically build the project for other cores as well. If you do not select this checkbox, then you would need to manually build the project for each core.</td>
</tr>
<tr>
<td>Core Index</td>
<td>Select the processor core that executes the project.</td>
</tr>
</tbody>
</table>

### 2.2 CodeWarrior Linux Project Wizard

The CodeWarrior Linux Project Wizard helps you create a Linux project by displaying various pages that allow you to specify settings for your project.

**NOTE**

The pages that the wizard presents can differ, based upon the choice of project type or execution target.

The pages of the CodeWarrior Linux Project Wizard are:

- [Create CodeWarrior Linux Project page](#) on page 24
- [Processor page](#) on page 25
- [Build Settings page](#) on page 26
- [Linux Application page](#) on page 27
2.2.1 Create CodeWarrior Linux Project page

Use this page to specify the project name and the directory where the project files are located.

Figure 6: Create CodeWarrior Linux Project page

The table below describes the various options available on the Create CodeWarrior Linux Project page.

Table 7: Create CodeWarrior Linux Project page settings

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project name</td>
<td>Enter the name for the project in this text box.</td>
</tr>
<tr>
<td>Use default location</td>
<td>Select to choose the directory to store the files required to build the program. Use the Location option to select the desired directory.</td>
</tr>
<tr>
<td>Location</td>
<td>Specifies the directory that contains the project files. Use Browse to navigate to the desired directory. This option is only available when Use default location is cleared. Ensure that you append the name of the project to the path to create a new location for your project.</td>
</tr>
</tbody>
</table>
2.2.2 Processor page

This page displays the processors supported by the current installation. Use this page to specify the type of processor and the output for the new project.

**Figure 7: Processor page**

The table below describes the various options available on the Processor page.

**Table 8: Processor page settings**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Expand the processor family tree and select a supported target. The toolchain uses this choice to generate code that makes use of processor-specific features, such as multiple cores.</td>
</tr>
<tr>
<td>Project Output</td>
<td>Select any one of the following supported project output:</td>
</tr>
<tr>
<td></td>
<td>• <strong>Application</strong> - Select to create an application with &quot;.elf&quot; extension, that includes information related to the debug over a board.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Library</strong> - Select to create a library with &quot;.a&quot; extension, that can be included in other projects. Library files created using this option do not include board specific details.</td>
</tr>
</tbody>
</table>
2.2.3 Build Settings page
This page displays the toolchains supported by the current installation. Use this page to specify the toolchain for the new project.

NOTE
The current release does not include toolchains for Linux applications by default. To add the required build tools support, you should install the corresponding service pack for the required target.

Figure 8: Build Settings page

The table below describes the various options available on the Build Settings page.
Table 9: Build Settings page setting

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Toolchain</strong></td>
<td>Specifies the toolchains supported by the current installation. Selected toolchain sets up the default compiler, linker, and libraries used to build the new project. Each toolchain generates code targeted for a specific platform.</td>
</tr>
<tr>
<td><strong>Language</strong></td>
<td>Specifies the programming language used by the new project. The current installation supports the following languages:</td>
</tr>
<tr>
<td></td>
<td>• C - Select to generate ANSI C-compliant startup code, and initializes global variables.</td>
</tr>
<tr>
<td></td>
<td>• C++ - Select to generate ANSI C++ startup code, and performs global class object initialization.</td>
</tr>
<tr>
<td><strong>Build Tools Architecture</strong></td>
<td>Specifies the processor used by the new project. The current installation supports the following architectures:</td>
</tr>
<tr>
<td></td>
<td>• 32 bit - 32-bit option is available by default for LS1021A processors.</td>
</tr>
<tr>
<td></td>
<td>• 64 bit - this option is disabled for this release.</td>
</tr>
</tbody>
</table>

2.2.4 Linux Application page

Use this page to specify how the debugger communicates with the host Linux system and controls your Linux application.

**NOTE**

The Linux Application page appears, in the CodeWarrior Linux Project Wizard, only when you add the Linux build tools support, by installing the corresponding service pack for the required target.
When debugging a Linux application, you must use the CodeWarrior TRK to manage the communications interface between the debugger and Linux system.

The table below describes the various options available on the Linux Application page.

### Table 10: Linux Application page setting

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior TRK</td>
<td>Select to use the CodeWarrior Target Resident Kernel (TRK) protocol, to download and control application on the Linux host system.</td>
</tr>
<tr>
<td>IP Address</td>
<td>Specifies the IP address of the Linux host system, the project executes on.</td>
</tr>
<tr>
<td>Port</td>
<td>Specifies the port number that the debugger will use to communicate to the Linux host.</td>
</tr>
<tr>
<td>Remote Download Path</td>
<td>Specifies the host directory into which the debugger downloads the application.</td>
</tr>
</tbody>
</table>
2.3 Creating projects

You can use a project creation wizard provided by CodeWarrior Development Studio to create a CodeWarrior project according to your requirements.

This section explains you how to use the CodeWarrior Bareboard Project Wizard to quickly create new projects with default settings (build and launch configurations). After the project has been created, you can easily change any default setting to suit your needs.

This section explains:

- Creating CodeWarrior Bareboard Application project on page 29
- Creating CodeWarrior Bareboard Library project on page 31
- Creating CodeWarrior Linux Application project on page 32

2.3.1 Creating CodeWarrior Bareboard Application project

You can create a CodeWarrior bareboard application project using the CodeWarrior Bareboard Project Wizard.

To create a CodeWarrior bareboard application project, perform these steps:

1. Select Start > All Programs > Freescale CodeWarrior > CW for ARMv7 vnumber > CodeWarrior, where number is the version number of your product.

   The Workspace Launcher dialog appears, prompting you to select a workspace to use.

   Click Browse to change the default location for workspace folder. You can also select the Use this as the default and do not ask again checkbox to set default or selected path as the default location for storing all your projects.

2. Click OK.

   The default workspace is accepted. The CodeWarrior IDE launches and the Welcome page appears.

   The Welcome page appears only if the CodeWarrior IDE or the selected workspace is started for the first time. Otherwise, the Workbench window appears.

3. Click Go to Workbench from the Welcome page.

   The workbench window appears.


   The CodeWarrior Bareboard Project Wizard launches and the Create a CodeWarrior Bareboard Project page appears.

5. Specify a name for the new project in the Project name text box.

   For example, enter the project name as Hello_World.

6. If you do not want to create your project in the default workspace:
a. Clear the **Use default location** checkbox.

b. Click **Browse** and select the desired location from the **Browse For Folder** dialog.

c. In the **Location** text box, append the location with the name of the directory in which you want to create your project. In the **Location** text box, append the location with the name of the directory in which you want to create your project.

**NOTE**
An existing directory cannot be specified for the project location. If created, the CodeWarrior will prompt an error message.

7. Click **Next**.

   The **Processor** page appears.

8. Select the target processor for the new project, from the **Processor** list.

9. Select **Application** from the **Project Output** group, to create an application with `.elf` extension, that includes information required to debug the project.

10. Click **Next**.

   The **Debug Target Settings** page appears.

11. Select a supported connection type (hardware or emulator), from the **Debugger Connection Types** group. Your selection determines the launch configurations that you can include in your project.

12. Select the board you are targeting, from the **Board** drop-down list. You can select a QDS or Tower, or an IOT or RDB board, depending on the processor selected on the **Processor** page.

13. Select the launch configurations that you want to include in your project and the corresponding connection, from the **Launch** group.

14. Select the interface to communicate with the hardware, from the **Connection Type** drop-down list.

15. Enter the IP address of the TAP device in the **TAP address** text box. This option is disabled and cannot be edited, if you select **CodeWarrior TAP (over USB)** from the **Connection Type** drop-down list. This option is available only if **CodeWarrior TAP (over Ethernet)** is selected as the connection type. **CMSIS-DAP** is the default connection type for Tower boards.

16. Click **Next**.

   The **Build Settings** page appears.

17. Select the programming language, you want to use, from the **Language** group.

   The language you select determines the libraries that are linked with your program and the contents of the main source file that the wizard generates.

18. Select the architecture type used by the new project, from the **Build Tools Architecture** group.

19. Select a toolchain from the **Toolchain** group.

   Selected toolchain sets up the default compiler, linker, and libraries used to build the new project. Each toolchain generates code targeted for a specific platform.

   **NOTE**
   The current release does not include toolchains for Linux applications by default. To add the required Linux build tools support, you should install the corresponding service pack for the required target.

20. Select an option from the **Floating Point** drop-down list, to prompt the compiler to handle the floating-point operations by generating instructions for the selected floating-point unit.
21. Click Next.

The Configurations page appears.

22. Select a processing model option from the Processing Model group.

- Select SMP (One build configuration for all the cores) to generate a single project for the selected cores. This option is disabled for this release.
- Select AMP (One project per core) to generate a separate project for each selected core. The option will also set the core index for each project based on the core selection.
- Select AMP (One build configuration per core) to generate one project with multiple targets, each containing an .lcf file for the specified core.

23. Select the processor core that executes the project, from the Core index list.

24. Click Next.

25. Click Finish.

The wizard creates an application project according to your specifications. You can access the project from the CodeWarrior Projects view on the Workbench.

The new project is ready for use. You can now customize the project by adding your own source code files, changing debugger settings and adding libraries.

2.3.2 Creating CodeWarrior Bareboard Library project

You can create a CodeWarrior bareboard library project using the CodeWarrior Bareboard Project Wizard.

To create a CodeWarrior bareboard library project, perform these steps:

1. Select Start > All Programs > Freescale CodeWarrior > CW for ARMv7 vnumber > CodeWarrior, where number is the version number of your product.

   The Workspace Launcher dialog appears, prompting you to select a workspace to use.

   **NOTE**

   Click Browse to change the default location for workspace folder. You can also select the Use this as the default and do not ask again checkbox to set default or selected path as the default location for storing all your projects.

2. Click OK.

   The default workspace is accepted. The CodeWarrior IDE launches and the Welcome page appears.

   **NOTE**

   The Welcome page appears only if the CodeWarrior IDE or the selected workspace is started for the first time. Otherwise, the Workbench window appears.

3. Click Go to Workbench, on the Welcome page.

   The workbench window appears.


   The CodeWarrior Bareboard Project Wizard launches and the Create a CodeWarrior Bareboard Project page appears.

5. Specify a name for the new project in the Project name text box.

   For example, enter the project name as library_project.

6. If you do not want to create your project in the default workspace:
a. Clear the **Use default location** checkbox.

b. Click **Browse** and select the desired location from the **Browse For Folder** dialog.

c. In the **Location** text box, append the location with the name of the directory in which you want to create your project.

**NOTE**
An existing directory cannot be specified for the project location.

7. Click **Next**.

The **Processor** page appears.

8. Select the target processor for the new project, from the **Processor** list.

9. Select **Static Library** from the **Project Output** group, to create a library with .a extension, that can be included in other projects. Library files created using this option do not include board specific details.

10. Click **Next**.

The **Build Settings** page appears.

11. Select the programming language, you want to use, from the **Language** group. The language you select determines the libraries that are linked with your program and the contents of the main source file that the wizard generates.

12. Select a toolchain from the **Toolchain** group.

   Selected toolchain sets up the default compiler, linker, and libraries used to build the new project. Each toolchain generates code targeted for a specific platform.

   **NOTE**
   The current release does not include toolchains for Linux applications by default. To add the required build tools support, you should install the corresponding service pack for the required target.

13. Select an option from the **Floating Point** drop-down list, to prompt the compiler to handle the floating-point operations by generating instructions for the selected floating-point unit.

14. Click **Finish**.

The wizard creates a library project according to your specifications. You can access the project from the **CodeWarrior Projects** view on the Workbench.

The new library project is ready for use. You can now customize the project to match your requirements.

### 2.3.3 Creating CodeWarrior Linux Application project

You can create a CodeWarrior Linux application project using the CodeWarrior Linux Project Wizard.

To create a CodeWarrior Linux application project, perform these steps:

1. Select **Start > All Programs > Freescale CodeWarrior > CW for ARMv7 vnumber > CodeWarrior**, where **vnumber** is the version number of your product.

   The **Workspace Launcher** dialog appears, prompting you to select a workspace to use.

   **NOTE**
   Click **Browse** to change the default location for workspace folder. You can also select the **Use this as the default** and do not ask again checkbox to set default or selected path as the default location for storing all your projects.
2. Click **OK**.
   The default workspace is accepted. The CodeWarrior IDE launches and the **Welcome** page appears.

   **NOTE**
   The **Welcome** page appears only if the CodeWarrior IDE or the selected workspace is started for the first time. Otherwise, the Workbench window appears.

3. Click **Go to Workbench**, on the **Welcome** page.
   The workbench window appears.

   The **CodeWarrior Linux Project Wizard** launches and the **Create a CodeWarrior Linux Project** page appears.

5. Specify a name for the new project in the **Project name** text box.
   For example, enter the project name as **linux_project**.

6. If you do not want to create your project in the default workspace:
   a. Clear the **Use default location** checkbox.
   b. Click **Browse** and select the desired location from the **Browse For Folder** dialog.
   c. In the **Location** text box, append the location with the name of the directory in which you want to create your project.

   **NOTE**
   An existing directory cannot be specified for the project location.

7. Click **Next**.
   The **Processor** page appears.

8. Select the target processor for the new project, from the **Processor** list.

9. Select **Application** from the **Project Output** group, to create an application with **.elf** extension, that includes information required to debug the project.

10. Click **Next**.
    The **Build Settings** page appears.

11. Select a toolchain for Linux applications from the **Toolchain** group.
    Selected toolchain sets up the default compiler, linker, and libraries used to build the new project. Each toolchain generates code targeted for a specific platform.

    **NOTE**
    To add the required Linux build tools support, you should install the corresponding service pack for the required target.

12. Select the programming language, you want to use, from the **Language** group.
    The language you select determines the libraries that are linked with your program and the contents of the main source file that the wizard generates.

13. Select the architecture type used by the new project, from the **Build Tools Architecture** group.

14. Click **Next**.
    The **Linux Application** page appears.
15. Select **CodeWarrior TRK** to use the CodeWarrior Target Resident Kernel (TRK) protocol, to download and control application on the Linux host system.

**NOTE**
When debugging a Linux application, you must use the CodeWarrior TRK to manage the communications interface between the debugger and Linux system. For details, see Install CodeWarrior TRK on target system on page 125 topic.


17. In the **IP Address** text box, enter the IP address of the Linux host system, the project executes on.

18. In the **Port** text box, enter the port number that the debugger will use to communicate to the Linux host system.

19. In the **Remote Download Path** text box, enter the absolute path for the host directory, into which the debugger downloads the application.

20. Click **Finish**.

The wizard creates a CodeWarrior Linux application project according to your specifications. You can access the project from the **CodeWarrior Projects** view on the Workbench.

The new CodeWarrior Linux application project is ready for use. You can now customize the project to match your requirements.

### 2.4 Building projects

CodeWarrior IDE supports two modes of building projects.

These modes are:

- **Manual-Build mode** on page 34
- **Auto-Build mode** on page 35

#### 2.4.1 Manual-Build mode

This section explains the manual mode of building projects.

In large workspaces, building the entire workspace can take a long time if users make changes with a significant impact on dependent projects. Often there are only a few projects that really matter to a user at a given time.

To build only the selected projects, and any prerequisite projects that need to be built in order to correctly build the selected projects, select **Project > Build Project** from the CodeWarrior IDE menu bar.
Alternatively, right-click the selected project in the CodeWarrior Projects view and select Build Project from the context menu.

To build all projects available in the CodeWarrior Projects view, select Project > Build All.

### 2.4.2 Auto-Build mode

This section explains the automatic mode of building projects.

CodeWarrior IDE takes care of compiling source files automatically. When auto-build is enabled, project build occurs automatically in the background every time you change files in the workspace (for example, saving an editor).

To automatically build all the projects in a workspace, select Project > Build Automatically from the CodeWarrior IDE menu bar.
Figure 11: Project Menu—Build Automatically

If auto-build is taking too long and is interfering with ongoing development, it can be turned off. Select **Project > Build Automatically** from the CodeWarrior IDE menu bar to disable auto-build mode.

**NOTE**

It is advised that you do not use the **Build Automatically** option for C/C++ development. Using this option will result in building the entire project whenever you save a change to the makefile or source files. This can take a significant amount of time for very large projects.
Chapter 3
Build Properties

This chapter explains build properties for projects. A project can contain multiple build and launch configurations.

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 produced by a Release build configuration might contain no symbols and be highly optimized.

This chapter explains:

• Changing build properties on page 37
• Restoring build properties on page 38
• Defining C/C++ build settings and behavior on page 38
• Properties for <project> on page 42

3.1 Changing build properties

You can modify the build properties of a project to better suit your needs.

Follow these steps to change build properties:

1. Start the CodeWarrior 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 for <project> dialog 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 node.
5. Select Settings.
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 as per the requirements.
10. Click Apply.
    The CodeWarrior 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 for <project> dialog.
3.2 Restoring build properties

You can modify a build configuration of a project and might choose to restore the build properties to have a factory-default configuration, or to revert to a last-known working build configuration.

To undo your modifications to build properties, click the Restore Defaults button at the bottom of the Properties for <project> dialog.

This changes the values of the options to the absolute default of the toolchain.

3.3 Defining C/C++ build settings and behavior

The C/C++ Build page includes all builder-specific property pages.

This section contains the following subsections:

• Define build settings
• Define build behavior

NOTE

Modifying settings such as the Generate makefiles automatically option, might enable or disable some parameters in some situations and change the availability of other property pages.

3.3.1 Define build settings

The Build Settings page of the C/C++ Build page allows you to define build settings for your project.

To define build settings, perform these steps:

1. Start CodeWarrior IDE.
2. In the CodeWarrior Projects view, select the project for which you want to modify the build settings.
3. Select Project > Properties.
   The Properties for <project> 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. Select C/C++ Build.
   The C/C++ Build page appears.
5. In the **Builder Settings** tab, the builder settings for the selected build configuration appears. The table below describes the builder settings options.

**Table 11: Builder settings options**

<table>
<thead>
<tr>
<th>Group</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build Configuration</td>
<td>Configuration</td>
<td>Specifies the type of configurations for the selected project.</td>
</tr>
<tr>
<td>Build Configuration</td>
<td>Manage configurations</td>
<td>Click to open the <strong>Manage Configurations</strong> dialog that lets you set configurations based on the specified toolchains of the selected project. You can also create new configurations, rename an existing configuration, or remove the ones that are no longer required.</td>
</tr>
<tr>
<td>Builder</td>
<td>Builder type</td>
<td>Specifies the type of builder to use:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Internal builder - Builds C/C++ programs using a compiler that implements the C/C++ Language Specifications.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• External builder - External tools let you configure and run programs and Ant buildfiles using the Workbench, which can be saved and run at a later time to perform a build.</td>
</tr>
<tr>
<td>Builder</td>
<td>Use default build command</td>
<td>Select to indicate that you want to use the default make command. Clear when you want to use a new make command. This option is only available when the Builder type option is set to External.</td>
</tr>
</tbody>
</table>

*Table continues on the next page...*
Table 11: Builder settings options (continued)

<table>
<thead>
<tr>
<th>Group</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Builder</td>
<td>Build command</td>
<td>Specifies the default command used to start the build utility for your specific toolchain. Use this field if you want to use a build utility other than the default make command.</td>
</tr>
<tr>
<td>Builder</td>
<td>Variables</td>
<td>Click to open the Select build variable dialog and add the desired environment variables and custom variables to the build command.</td>
</tr>
<tr>
<td>Makefile generation</td>
<td>Generate Makefiles</td>
<td>Automatically Select to enable Eclipse change between two different CDT modes: it either uses the customer's makefile for the build, if one exists, or it generates makefiles for the user.</td>
</tr>
<tr>
<td>Makefile generation</td>
<td>Expand Env. Variable</td>
<td>Select to define whether environment variables should be expanded in makefile.</td>
</tr>
<tr>
<td>Build location</td>
<td>Build directory</td>
<td>Specifies the location where the build operation takes place. This location will contain the generated artifacts from the build process. This option appears disabled when the Generate Makefiles automatically option is enabled.</td>
</tr>
<tr>
<td>Build location</td>
<td>Workspace</td>
<td>Click to open the Folder Selection dialog and select a workspace location for the project. This is the directory that will contain the plug-ins and features to build, including any generated artifacts.</td>
</tr>
<tr>
<td>Build location</td>
<td>File system</td>
<td>Click to open the Browse For Folder dialog and select a folder.</td>
</tr>
<tr>
<td>Build location</td>
<td>Variables</td>
<td>Click to open the Select build variable dialog and select a variable to specify as an argument for the build directory, or create and configure simple build variables which you can reference in build configurations that support variables.</td>
</tr>
</tbody>
</table>

6. Make the desired changes and click OK.  
   The Properties for <project> window will close.

3.3.2 Define build behavior

The Behaviour page of the C/C++ Build page allows you to define build behavior for your project.

To define build behavior, perform these steps:

1. Start CodeWarrior IDE.
2. In the CodeWarrior Projects view, select the project for which you want to modify the build settings.
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. Select C/C++ Build.  
   The C/C++ Build page appears.
5. Click the Behaviour tab.  
   The behavior settings for the selected build configuration appears.  
   The table below describes the behavior settings options.
Table 12: Behavior options

<table>
<thead>
<tr>
<th>Group</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build settings</td>
<td>Enable project specific</td>
<td>Select if you want to enable project specific settings.</td>
</tr>
<tr>
<td></td>
<td>settings</td>
<td></td>
</tr>
<tr>
<td>Build settings</td>
<td>Stop on first build error</td>
<td>Select to stop building when Eclipse encounters an error. Clearing this option is helpful for building large projects as it enables make to continue making other independent rules even when one rule fails.</td>
</tr>
<tr>
<td>Configure Workspace Settings</td>
<td>Enable parallel build</td>
<td>Select to activate the generation of parallel builds. However, you need to determine the number of parallel jobs to perform:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Use optimal jobs number - Lets the system determine the optimal number of parallel jobs to perform.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Use parallel jobs - Lets you specify the maximum number of parallel jobs to perform.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Use unlimited jobs - Lets the system perform unlimited jobs.</td>
</tr>
<tr>
<td>Workbench Build Behavior</td>
<td>Workbench build type</td>
<td>Specifies the builder settings when instructed to build, rebuild, and clean.</td>
</tr>
<tr>
<td>Workbench Build Behavior</td>
<td>Build on resource save</td>
<td>Select to build your project whenever resources are saved. By default, this option is selected and builds occur automatically each time resources are modified. Clear if you do want that the build occurs only manually using a menu item.</td>
</tr>
<tr>
<td></td>
<td>(Auto build)</td>
<td></td>
</tr>
<tr>
<td>Workbench Build Behavior</td>
<td>Build (Incremental Build)</td>
<td>Defines what the standard builder will call when an incremental build is performed.</td>
</tr>
<tr>
<td>Workbench Build Behavior</td>
<td>Variables</td>
<td>Click to open the Select build variable dialog and add variables to the make build target command.</td>
</tr>
<tr>
<td>Workbench Build Behavior</td>
<td>Clean</td>
<td>Defines what the standard builder calls when a clean is performed. The make clean is defined in the makefile.</td>
</tr>
<tr>
<td>Workbench Build Behavior</td>
<td>Variables</td>
<td>Click to open the Select build variable dialog and add variables to the make build target command.</td>
</tr>
</tbody>
</table>

6. Make the desired changes and click **OK**.

The Properties for `<project>` window will close.
3.4 Properties for <project>

The Properties for <project> window shows the corresponding build properties for a bareboard project.

Figure 13: Build properties

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

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.

Table 13: Build properties for bareboard project

<table>
<thead>
<tr>
<th>Tool Settings</th>
<th>Sub Tool Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target Processor on page 43</td>
<td>Target Processor on page 43</td>
</tr>
<tr>
<td>Debugging on page 44</td>
<td>Debugging on page 44</td>
</tr>
<tr>
<td>ARM Sourcery GCC Assembler on page 44</td>
<td>Preprocessor on page 45</td>
</tr>
<tr>
<td></td>
<td>ARM Sourcery GCC C Compiler on page 46</td>
</tr>
</tbody>
</table>

Table continues on the next page...
### 3.4.1 Target Processor

This section describes the options in the **Target Processor** panel.

The options are listed in the table below.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Use to specify the processor.</td>
</tr>
<tr>
<td></td>
<td>Default: cortex-a7</td>
</tr>
<tr>
<td>Thumb (-mthumb)</td>
<td>Select to have the processor generate Thumb code instructions. Clear to</td>
</tr>
<tr>
<td></td>
<td>prevent the processor from generating Thumb code instructions. The IDE</td>
</tr>
<tr>
<td></td>
<td>enables this setting only for architectures and processors that support the</td>
</tr>
<tr>
<td></td>
<td>Thumb instruction set.</td>
</tr>
<tr>
<td></td>
<td>Default: Clear</td>
</tr>
<tr>
<td>Thumb interwork (-mthumb-interwork)</td>
<td>Use to generate suitable interworking veneers when it links the assembler</td>
</tr>
<tr>
<td></td>
<td>output. You must enable this option if you write ARM code that you want to</td>
</tr>
<tr>
<td></td>
<td>interwork with Thumb code or vice versa. The only functions that need to be</td>
</tr>
<tr>
<td></td>
<td>compiled for interworking are the functions that are called from the other</td>
</tr>
<tr>
<td></td>
<td>state. You must ensure that your code uses the correct interworking return</td>
</tr>
<tr>
<td></td>
<td>instructions.</td>
</tr>
<tr>
<td></td>
<td>Default: Clear</td>
</tr>
<tr>
<td>Endianness</td>
<td>Use to specify the byte order of the target hardware architecture:</td>
</tr>
<tr>
<td></td>
<td>• Little Endian: Right-most bytes (those with a higher address) are most</td>
</tr>
<tr>
<td></td>
<td>significant</td>
</tr>
<tr>
<td></td>
<td>• Big Endian: Left-most bytes (those with a lower address) are most</td>
</tr>
<tr>
<td></td>
<td>significant</td>
</tr>
<tr>
<td></td>
<td>Default: Toolchain Default</td>
</tr>
<tr>
<td>Float ABI</td>
<td>Use to specify the float Application Binary Interface (ABI).</td>
</tr>
<tr>
<td></td>
<td>Default: FP instructions (-mfloat-abi=hard)</td>
</tr>
</tbody>
</table>

*Table continues on the next page...*
### Table 14: Target Processor options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPU Type</td>
<td>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: VFPv3-D16 (-mfpu=vfpv3-d16)</td>
</tr>
</tbody>
</table>

#### 3.4.2 Debugging

This section describes the options in the **Debugging** panel.

The options are listed in the table below.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug level</td>
<td>Specify the debug levels:</td>
</tr>
<tr>
<td></td>
<td>• None - No Debug level.</td>
</tr>
<tr>
<td></td>
<td>• Minimal (-g1) - The compiler provides minimal debugging support.</td>
</tr>
<tr>
<td></td>
<td>• Default (-g) - The compiler generates DWARF 1.x-conforming debugging information.</td>
</tr>
<tr>
<td></td>
<td>• Maximum (-g3) - The compiler provides maximum debugging support.</td>
</tr>
<tr>
<td>Debug format</td>
<td>Specify the debug formats for the compiler.</td>
</tr>
<tr>
<td>Generate prof information (-p)</td>
<td>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.</td>
</tr>
<tr>
<td>Generate gprof information (-pg)</td>
<td>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.</td>
</tr>
<tr>
<td>Other debugging flags</td>
<td>Specify additional command line options; type in custom debugging flags that are not otherwise available in the UI.</td>
</tr>
</tbody>
</table>

#### 3.4.3 ARM Sourcery GCC Assembler

This section describes the options in the **ARM Sourcery GCC Assembler** panel.

The options are listed in the table below.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command</td>
<td>Shows the location of the assembler executable file. Default: <code>arm-none-eabi-gcc</code></td>
</tr>
<tr>
<td>All Options</td>
<td>Shows the actual command line the assembler will be called with. Default: <code>-x assembler-with-cpp -Wall -Wa,-adhlns=&quot;$@.lst&quot; -c -fmessage-length=0 -mcpu=cortex-a7 -mfloat-abi=hard -mfpu=vfpv3-d16 -g -gdwarf-2</code></td>
</tr>
</tbody>
</table>

"Table continues on the next page..."
Table 16: ARM sourcery GCC assembler options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
</table>
| Expert settings             | Shows the expert settings command line parameters. Default: "${ARMv7GCCToolsDir}/${COMMAND}" ${INPUTS} ${FLAGS} ${OUTPUT_FLAG} ${OUTPUT_PREFIX}${OUTPUT}

This section contains the following subsections:

- Preprocessor on page 45
- Warnings on page 45

### 3.4.3.1 Preprocessor

Use this panel to specify preprocessor-related settings of the ARM sourcery GCC assembler.

The available options are listed in the table below.

Table 17: Preprocessor settings of ARM sourcery GCC assembler

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use preprocessor</td>
<td>Select this option to use the preprocessor for the assembler.</td>
</tr>
<tr>
<td>Do not search system directories (-nostdinc)</td>
<td>Select 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.</td>
</tr>
<tr>
<td>Preprocess only (-E)</td>
<td>Select 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.</td>
</tr>
<tr>
<td>Defined symbols (-D)</td>
<td>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.</td>
</tr>
<tr>
<td>Undefined symbols (-U)</td>
<td>Undefined the substitution strings you specify in this panel.</td>
</tr>
</tbody>
</table>

### 3.4.3.2 Warnings

Use this panel to specify warning-related settings of the ARM sourcery GCC assembler.

The available options are listed in the table below.

Table 18: Warning options for ARM sourcery GCC assembler

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check syntax only (-fsyntax-only)</td>
<td>Check the code for syntax errors, but don't do anything beyond that.</td>
</tr>
</tbody>
</table>

Table continues on the next page...
### Table 18: Warning options for ARM sourcery GCC assembler (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pedantic (-pedantic)</td>
<td>Issue all the warnings demanded by strict ISO C and ISO C++; reject all programs that use forbidden extensions, and some other programs that do not follow ISO C and ISO C++. For ISO C, follows the version of the ISO C standard specified by any -std option used.</td>
</tr>
<tr>
<td>Pedantic warnings as errors (-pedantic-errors)</td>
<td>Like -Wpedantic, except that errors are produced rather than warnings.</td>
</tr>
<tr>
<td>Inhibit all warnings (-w)</td>
<td>Inhibit all warnings</td>
</tr>
<tr>
<td>All warnings (-Wall)</td>
<td>This enables 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.</td>
</tr>
<tr>
<td>Extra warnings (-Wextra)</td>
<td>This enables some extra warning flags that are not enabled by -Wall.</td>
</tr>
<tr>
<td>Warnings as errors (-Werror)</td>
<td>Make all warnings into errors.</td>
</tr>
</tbody>
</table>

### 3.4.4 ARM Sourcery GCC C Compiler

This section describes the options in the ARM Sourcery GCC C Compiler panel.

The options are listed in the table below.

#### Table 19: ARM sourcery GCC C compiler options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command</td>
<td>Shows the location of the compiler executable file. Default: arm-none-eabi-gcc</td>
</tr>
<tr>
<td>All Options</td>
<td>Shows the actual command line the compiler will be called with.</td>
</tr>
<tr>
<td>Expert settings</td>
<td></td>
</tr>
<tr>
<td>Command line patterns</td>
<td>Shows the expert settings command line parameters. Default: &quot;${ARMv7GCCToolsDir}/${COMMAND} ${INPUTS} ${FLAGS} ${OUTPUT_FLAG} ${OUTPUT_PREFIX} ${OUTPUT}&quot;</td>
</tr>
</tbody>
</table>

This section contains the following subsections:

- **Preprocessor** on page 46
- **Optimization** on page 47
- **Warnings** on page 48
- **Miscellaneous** on page 48

#### 3.4.4.1 Preprocessor

Use this panel to specify preprocessor-related settings of the ARM sourcery GCC C compiler.

The available options are listed in the table below.
Table 20: Preprocessor settings of ARM sourcery GCC C compiler

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do not search system directories (-nostdinc)</td>
<td>Select 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.</td>
</tr>
<tr>
<td>Preprocess only (-E)</td>
<td>Select 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.</td>
</tr>
<tr>
<td>Defined symbols (-D)</td>
<td>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.</td>
</tr>
<tr>
<td>Undefined symbols (-U)</td>
<td>Undefines the substitution strings you specify in this panel.</td>
</tr>
</tbody>
</table>

3.4.4.2 Optimization

Use this panel to specify optimization-related settings of the ARM sourcery GCC C compiler.
The available options are listed in the table below.

Table 21: Optimization settings of ARM sourcery GCC C compiler

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimization level</td>
<td>Select the required optimization level</td>
</tr>
<tr>
<td>• None (-O0) - Reduce compilation time and make debugging produce the expected results. This is the default.</td>
<td></td>
</tr>
<tr>
<td>• Optimize (-O1) - Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function.</td>
<td></td>
</tr>
<tr>
<td>• Optimize more (-O2) - Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. As compared to -O, this option increases both compilation time and the performance of the generated code.</td>
<td></td>
</tr>
<tr>
<td>• Optimize most (-O3) - Optimize yet more. -O3 turns on all optimizations specified by -O2 and also turns on the -finline-functions, -funswitch-loops, -fpredictive-commoning, -fgcse-after-reload, -ftree-loop-vectorize, -ftree-slp-vectorize, -fvect-cost-model, -ftree-partial-pre and -fipa-cp-clone options.</td>
<td></td>
</tr>
<tr>
<td>• Optimize size (-Os) - Optimize for size. -Os enables all -O2 optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size.</td>
<td></td>
</tr>
</tbody>
</table>

Table continues on the next page...
Table 21: Optimization settings of ARM sourcery GCC C compiler (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pack structures</td>
<td>Packed data structures are supported in the compiler with the keyword __packed or <strong>attribute</strong>((packed)). There is no code generation support for accessing un-aligned, packed data members. Users should exercise caution when accessing packed data structures because data might not be aligned.</td>
</tr>
<tr>
<td>Short enumerations</td>
<td>Select to use short enumerated constants and is equivalent to -fshort-enum.</td>
</tr>
<tr>
<td>Function sections</td>
<td>Select to use function sections and is equivalent to -ffunction-sections.</td>
</tr>
<tr>
<td>Data sections</td>
<td>Select to use short data sections and is equivalent to -ffunction-sections.</td>
</tr>
<tr>
<td>Other optimization flags</td>
<td>Specify additional command line options; type in custom optimization flags that are not otherwise available in the UI.</td>
</tr>
</tbody>
</table>

3.4.4.3 Warnings
Use this panel to specify warning-related settings of the ARM sourcery GCC C compiler.

The available options are listed in the table below.

Table 22: Warning options for ARM sourcery GCC C compiler

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check syntax only</td>
<td>Select the code for syntax errors, but don't do anything beyond that.</td>
</tr>
<tr>
<td>No common</td>
<td>Select this option if you want to issue all the warnings demanded by strict ISO C and ISO C++; reject all programs that use forbidden extensions, and some other programs that do not follow ISO C and ISO C++. For ISO C, follows the version of the ISO C standard specified by any `-std' option used.</td>
</tr>
<tr>
<td>Pedantic</td>
<td>Issue all the warnings demanded by strict ISO C and ISO C++; reject all programs that use forbidden extensions, and some other programs that do not follow ISO C and ISO C++. For ISO C, follows the version of the ISO C standard specified by any -std option used.</td>
</tr>
<tr>
<td>Pedantic warnings as errors</td>
<td>Like -Wpedantic, except that errors are produced rather than warnings.</td>
</tr>
<tr>
<td>Inhibit all warnings</td>
<td>Inhibit all warnings</td>
</tr>
<tr>
<td>All warnings</td>
<td>This enables 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.</td>
</tr>
<tr>
<td>Extra warnings</td>
<td>This enables some extra warning flags that are not enabled by -Wall.</td>
</tr>
<tr>
<td>Warnings as errors</td>
<td>Make all warnings into errors.</td>
</tr>
</tbody>
</table>

3.4.4.4 Miscellaneous
You can use this panel to specify miscellaneous settings related to the ARM sourcery GCC C compiler.

The available options are listed in the table below.
### Table 23: Miscellaneous ARM sourcery GCC C compiler options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Language Standard</strong></td>
<td>Select the programming language or standard to which the compiler should conform.</td>
</tr>
<tr>
<td>ISO C90 (-ansi)</td>
<td>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.</td>
</tr>
<tr>
<td>ISO C99 (-std=c99)</td>
<td>Select this option to instruct the compiler to enforce stricter adherence to the ANSI/ISO standard.</td>
</tr>
<tr>
<td>Compiler Default</td>
<td>Select this option to enforce adherence to ISO C90 with GNU extensions.</td>
</tr>
<tr>
<td>ISO C99 with GNU Extensions (-std=gnu99)</td>
<td></td>
</tr>
<tr>
<td><strong>Assembler Listing</strong></td>
<td>Enables the assembler to create a listing file as it compiles assembly language into object code.</td>
</tr>
<tr>
<td>(-Wa, -adflns=&quot;$@.lst&quot;)</td>
<td></td>
</tr>
<tr>
<td><strong>Do not inline functions</strong></td>
<td>Select this option if you do not want to inline function.</td>
</tr>
<tr>
<td><strong>'char' is signed</strong></td>
<td>Select this option if you want to ensure that the char is signed.</td>
</tr>
<tr>
<td><strong>Bitfields are unsigned</strong></td>
<td>Select this option to ensure bitfields are unsigned.</td>
</tr>
<tr>
<td><strong>Verbose (-v)</strong></td>
<td>Select 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.</td>
</tr>
<tr>
<td><strong>Other compiler flags</strong></td>
<td>Specify additional command line options; type in custom flags that are not otherwise available in the UI.</td>
</tr>
</tbody>
</table>

### 3.4.5 ARM Sourcery GCC C Linker

This section describes the options in the ARM Sourcery GCC C Linker panel.

The options are listed in the table below.

### Table 24: ARM sourcery GCC C linker options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Command</strong></td>
<td>Shows the location of the linker executable file. Default: arm-none-eabi-gcc</td>
</tr>
<tr>
<td><strong>All Options</strong></td>
<td>Shows the actual command line the assembler will be called with. Default: -T &quot;${{ProjDirPath}}/Linker_Files/aarch64elf.x -nostartfiles -nodefaultlibs -L&quot;C:\Users\b14174\workspace-15\FirstProjectTest&quot; -Wl,-Map,&quot;FirstProjectTest.map&quot;</td>
</tr>
<tr>
<td><strong>Expert settings</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Command line patterns</strong></td>
<td>Shows the expert settings command line parameters. Default: ${COMMAND} ${cross_toolchain_flags} ${FLAGS} ${OUTPUT_FLAG} ${OUTPUT_PREFIX} ${OUTPUT} ${INPUTS}</td>
</tr>
</tbody>
</table>
This section contains the following subsections:

- **General** on page 50
- **Libraries** on page 50
- **Miscellaneous** on page 50

### 3.4.5.1 General

Use this panel to specify general settings of the ARM sourcery GCC C linker.

The available options are listed in the table below.

**Table 25: General ARM sourcery GCC C linker options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Script file</td>
<td><code>${ProjDirPath}/Linker_Files/LS1021A_QDS_Core0_ram.ld</code></td>
</tr>
<tr>
<td>Do not use standard start files</td>
<td>Marks the object that the search for dependencies of this object will ignore start files</td>
</tr>
<tr>
<td>Do not use default libraries</td>
<td>Marks the object that the search for dependencies of this object will ignore any default library search paths.</td>
</tr>
<tr>
<td>No startup or default libs</td>
<td>Marks the object that the search for dependencies of this object will ignore any default library search paths and start files.</td>
</tr>
<tr>
<td>Remove unused sections</td>
<td>List all sections removed by garbage collection. The listing is printed on stderr. This option is only effective if garbage collection has been enabled via the <code>--gc-sections</code>) option.</td>
</tr>
<tr>
<td>Print removed sections</td>
<td>List all sections removed by garbage collection. The listing is printed on stderr. This option is only effective if garbage collection has been enabled via the <code>--gc-sections</code>) option.</td>
</tr>
<tr>
<td>Omit all symbol information</td>
<td>Omit all symbol information from the output file.</td>
</tr>
</tbody>
</table>

### 3.4.5.2 Libraries

Use this panel to specify library-related settings of the ARM sourcery GCC C linker.

The available options are listed in the table below.

**Table 26: Library settings for ARM sourcery GCC C linker**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Libraries (-l)</td>
<td>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&quot;xyz&quot;, the compiler first searches user paths, then the system paths. For include statements of the form #include&lt;xyz&gt;, the compiler searches only system paths. This option is global.</td>
</tr>
<tr>
<td>Library search path (-L)</td>
<td>Use this option to specify the include library search path.</td>
</tr>
</tbody>
</table>

### 3.4.5.3 Miscellaneous

You can use this panel to specify miscellaneous settings related to the ARM sourcery GCC C linker.

The available options are listed in the table below.
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linker flags</td>
<td>This option specifies the flags to be passed with the linker file.</td>
</tr>
<tr>
<td>Other objects</td>
<td>The linker searches the paths in the order shown in this list.</td>
</tr>
<tr>
<td>Generate Map</td>
<td>This option specifies the map filename. Default: <code>{BuildArtifactFileBaseName}.map</code></td>
</tr>
<tr>
<td>Cross Reference (-Xlinker --cref)</td>
<td>Select 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.</td>
</tr>
<tr>
<td>Print link map (-Xlinker --printf-map)</td>
<td>Select this option to instruct the linker to print the map file.</td>
</tr>
<tr>
<td>Verbose (-v)</td>
<td>Select this option to show verbose information, including hex dump of program segments in applications; default setting</td>
</tr>
<tr>
<td>Other flags</td>
<td>Specify additional command line options for the linker; type in custom flags that are not otherwise available in the UI.</td>
</tr>
</tbody>
</table>
Properties for <project>
Chapter 4
Debug Configurations

A CodeWarrior project can have multiple associated debug configurations. A debug configuration is a named collection of settings that the CodeWarrior tools use.

Debug configurations let you specify settings, such as:

• The files that belong to the debug configuration
• Behavior of the debugger and the related debugging tools

This chapter explains:

• Using Debug Configurations dialog on page 53
• Customizing Debug Configurations on page 75
• Reverting Debug Configuration settings on page 77

4.1 Using Debug Configurations dialog

The Debug Configurations dialog allows you to specify debugger-related settings for your CodeWarrior project.

As you modify a launch configuration's debugger settings, you create pending, or unsaved, changes to that launch configuration. To save the pending changes, you must click the Apply button of the Debug Configurations dialog or click the Close button and then the Yes button.

Table 28: Debug Configuration tabs

<table>
<thead>
<tr>
<th>Main on page 54</th>
<th>Arguments on page 58</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debugger on page 59</td>
<td>Debug on page 60</td>
</tr>
<tr>
<td></td>
<td>Exceptions on page 62</td>
</tr>
<tr>
<td></td>
<td>Interrupts on page 62</td>
</tr>
<tr>
<td></td>
<td>Download on page 63</td>
</tr>
<tr>
<td></td>
<td>PIC on page 65</td>
</tr>
<tr>
<td></td>
<td>Other Executables on page 66</td>
</tr>
<tr>
<td></td>
<td>Symbolics on page 67</td>
</tr>
<tr>
<td></td>
<td>OS Awareness on page 68</td>
</tr>
<tr>
<td>Source on page 72</td>
<td>Environment on page 73</td>
</tr>
<tr>
<td>Common on page 74</td>
<td></td>
</tr>
</tbody>
</table>
4.1.1 Main

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

You can also specify a remote system configuration on this tab. The remote system configuration is separated into connection and system configurations allowing you to define a single system configuration that can be referred to by multiple connection configurations. The launch configurations refer to a connection configuration, which in turn refers to a system configuration.

NOTE
The options displayed on the Main tab vary depending on the selected debug session type.

The following figure shows the Main tab.

Figure 14: Debug Configurations-Main tab

The table below lists the various options available on the Main page.
### Table 29: Main tab options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug session type</td>
<td>Specifies the options to initiate a debug session using pre-configured debug configurations. The options include:</td>
</tr>
<tr>
<td></td>
<td>- <strong>Download</strong> - Resets the target if the debug configuration specifies the action. Further, the command stops the target, (optionally) runs an initialization script, downloads the specified ELF file, and modifies the program counter (PC).</td>
</tr>
<tr>
<td></td>
<td>- <strong>Attach</strong> - Assumes that code is already running on the board and therefore does not run a target initialization file. The state of the running program is undisturbed. The debugger loads symbolic debugging information for the current build target's executable. The result is that you have the same source-level debugging facilities you have in a normal debug session (the ability to view source code and variables, and so on). The function does not reset the target, even if the launch configuration specifies this action. Further, the command loads symbolics, does not stop the target, run an initialization script, download an ELF file, or modify the program counter (PC).</td>
</tr>
<tr>
<td></td>
<td>- <strong>Connect</strong> - Runs the target initialization file specified in the RSE configuration to set up the board before connecting to it. The Connect debug session type does not load any symbolic debugging information for the current build target's executable thereby, denying access to source-level debugging and variable display. The Connect command resets the target if the launch configuration specifies this action. Further, the command stops the target, (optionally) runs an initialization script, does not load symbolics, download an ELF file, or modify the program counter (PC).</td>
</tr>
<tr>
<td></td>
<td><strong>NOTE</strong> The debugger does not support restarting debugging sessions that you start by attaching the debugger to a process.</td>
</tr>
<tr>
<td></td>
<td><strong>NOTE</strong> The default debugger configuration causes the debugger to cache symbolics between sessions. However, selecting the Connect option invalidates this cache. If you must preserve the contents of the symbolics cache, and you plan to use the Connect option, clear the Cache Symbolics Between Sessions checkbox in the Symbolics tab page.</td>
</tr>
<tr>
<td></td>
<td><strong>NOTE</strong> <strong>Custom</strong> - Provides user an advantage to create a custom debug configuration.</td>
</tr>
</tbody>
</table>

*Table continues on the next page...*
### Table 29: Main tab options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++ application</td>
<td>Specifies the settings for the C/C++ application. The options include:</td>
</tr>
<tr>
<td></td>
<td>• <strong>Project</strong> - Specifies the name of the project associated with the selected debug launch configuration. Click Browse to select a different project.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Application</strong> - Specifies the name of the C or C++ application executable.</td>
</tr>
<tr>
<td></td>
<td><strong>NOTE</strong></td>
</tr>
<tr>
<td></td>
<td>This option is disabled when Connect debug session type is selected.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Search Project</strong> - Click to open the Program Selection dialog box and select a binary.</td>
</tr>
<tr>
<td></td>
<td><strong>NOTE</strong></td>
</tr>
<tr>
<td></td>
<td>This option is disabled when Connect debug session type is selected.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Variables</strong> - Click to open the Select build variable dialog and select the build variables to be associated with the program.</td>
</tr>
<tr>
<td></td>
<td>The dialog displays an aggregation of multiple variable databases and not all these variables are suitable to be used from a build environment. Given below are the variables that should be used:</td>
</tr>
<tr>
<td></td>
<td>ProjDirPath - returns the absolute path of the current project location in the file system</td>
</tr>
<tr>
<td></td>
<td><code>${ProjDirPath}/Source/main.c&quot;</code></td>
</tr>
<tr>
<td></td>
<td>workspace_loc - returns the absolute path of a workspace resource in the file system, or the location of the workspace if no argument is specified</td>
</tr>
<tr>
<td></td>
<td><code>${workspace_loc:/ProjectName/Source main.c&quot;</code>${workspace_loc}</td>
</tr>
<tr>
<td></td>
<td>Gnu_Make_Install_Dir - returns the absolute path of the GNU make.exe tool</td>
</tr>
<tr>
<td></td>
<td><code>${Gnu_Make_Install_Dir}\make.exe</code></td>
</tr>
<tr>
<td></td>
<td><strong>NOTE</strong></td>
</tr>
<tr>
<td></td>
<td>This option is disabled when Connect debug session type is selected.</td>
</tr>
</tbody>
</table>

*Table continues on the next page...*
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build (if required) before launching</td>
<td>Controls how auto build is configured for the launch configuration. Changing this setting overrides the global workspace setting and can provide some speed improvements.</td>
</tr>
<tr>
<td><strong>NOTE</strong></td>
<td>These options are set to default and collapsed when Connect debug session type is selected.</td>
</tr>
<tr>
<td></td>
<td>The options include:</td>
</tr>
<tr>
<td></td>
<td>• <strong>Build configuration</strong> - Specifies the build configuration either explicitly or use the current active configuration.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Select configuration using &quot;C/C++ Application&quot;</strong> - Select/clear to enable/disable automatic selection of the configuration to be built, based on the path to the program.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Enable auto build</strong> - Enables auto build for the debug configuration which can slow down launch performance.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Disable auto build</strong> - 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.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Use workspace settings (default)</strong> - Uses the global auto build settings.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Configure Workspace Settings</strong> - Opens the Launching preference panel where you can change the workspace settings. It will affect all projects that do not have project specific settings.</td>
</tr>
<tr>
<td>Target settings</td>
<td>Specifies the connection and other settings for the target. The options include:</td>
</tr>
<tr>
<td></td>
<td>• <strong>Connection</strong> - Specifies the applicable Remote System configuration.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Edit</strong> - Click to edit the selected Remote System configuration.</td>
</tr>
<tr>
<td></td>
<td>• <strong>New</strong> - Click to create a new Remote System configuration for the selected project and application.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Execute reset sequence</strong> - Select to apply reset settings, specified in the target configuration, when attaching to a target. Alternatively, clear the option to ignore reset settings.</td>
</tr>
<tr>
<td></td>
<td><strong>NOTE</strong> This option is not available when Attach debug session type is selected.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Execute initialization script(s)</strong> - Select to execute the initialization script(s), specified in the target configuration, when attaching to a target. Alternatively, clear the option to ignore the initialization script(s).</td>
</tr>
<tr>
<td></td>
<td>• <strong>Target</strong> (multicore only) - Select the core to be debugged. For SMP debugging, select all cores in the SMP group.</td>
</tr>
</tbody>
</table>
4.1.2 Arguments

Use this tab to specify the program arguments that an application uses and the working directory for a run or debug configuration.

Figure 15: Debug Configurations-Arguments tab

The table below lists the various options available on the Arguments page.

Table 30: Arguments tab options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program arguments</td>
<td>Specifies the arguments passed on the command line.</td>
</tr>
<tr>
<td>Variables</td>
<td>Click to select variables by name to include in the program arguments list.</td>
</tr>
<tr>
<td>Working Directory</td>
<td>Specifies the run/debug configuration working directory.</td>
</tr>
<tr>
<td>Use default</td>
<td>Select to specify the default run/debug configuration working directory, which is a directory within the current project directory, or clear to specify a different workspace, a file system location, or a variable. For Linux applications, the default working directory is the current directory on the process that started CodeWarrior TRK on the target. This should not be confused with the directory where the CodeWarrior TRK binary resides.</td>
</tr>
<tr>
<td>Workspace</td>
<td>Click to specify the path of, or browse to, a workspace relative working directory.</td>
</tr>
<tr>
<td>File System</td>
<td>Click to specify the path of, or browse to, a file system directory.</td>
</tr>
<tr>
<td>Variables</td>
<td>Click to specify variables by name to include in the working directory.</td>
</tr>
</tbody>
</table>
4.1.3 Debugger

Use this tab to configure the debugger settings.

The **Debugger** tab presents different pages for specifying different configuration settings specific to the selected debugger type.

**NOTE**

The content in the **Debugger Options** panel changes, depending on the Debug session type selected on the Main page.

![Figure 16: Debug Configurations-Debugger tab](image)

The different pages available on the **Debugger** tab are grouped in the **Debugger options** group. The pages are listed below:

- **Debug** on page 60
- **Exceptions** on page 62
- **Interrupts** on page 62
- **Download** on page 63
- **PIC** on page 65
- **Other Executables** on page 66
- **Symbolics** on page 67
- **OS Awareness** on page 68
4.1.3.1 Debug

Use this page to specify the program execution options, breakpoint and watchpoint options, and target access behavior.

Figure 17: Debugger options-Debug page

The options displayed on the Debug tab varies depending on the selected launch configuration.

The table below lists the various options available on the Debug page.

Table 31: Debugger options - Debug

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize program counter at</td>
<td>Controls the initialization of program counter.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Program entry point</strong> - Select to initialize the program counter at</td>
</tr>
<tr>
<td></td>
<td>a specified program entry point.</td>
</tr>
<tr>
<td></td>
<td>• <strong>User specified</strong> - Select to initialize the program counter at a</td>
</tr>
<tr>
<td></td>
<td>user-specified function. The default location is <strong>main</strong>.</td>
</tr>
<tr>
<td></td>
<td><strong>NOTE</strong></td>
</tr>
<tr>
<td></td>
<td>Disabling this option will also disable the Resume program and Stop on</td>
</tr>
<tr>
<td></td>
<td>startup at options.</td>
</tr>
</tbody>
</table>

*Table continues on the next page...*
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resume program</td>
<td>Select to resume the execution after the program counter is initialized.</td>
</tr>
<tr>
<td></td>
<td><strong>NOTE</strong> Disabling this option will also disable the Stop on startup at option.</td>
</tr>
<tr>
<td>Stop on startup at</td>
<td>Stops program at specified location. When cleared, the program runs until you interrupt it manually, or until it hits a breakpoint.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Program entry point</strong> - Select to stop the debugger at a specified program entry point.</td>
</tr>
<tr>
<td></td>
<td>• <strong>User specified</strong> - Select to stop the debugger at a user-specified function. The default location is <code>main</code>.</td>
</tr>
<tr>
<td>Stop on exit</td>
<td>Select this option to have the debugger set a breakpoint at the code's exit point. For multicore projects, when you set this option for one project on one core, it is set for projects on the other cores. Clear this option to prevent the debugger from setting a breakpoint at the code's exit point.</td>
</tr>
<tr>
<td>Install regular breakpoints as</td>
<td>Select this option to install breakpoints as either:</td>
</tr>
<tr>
<td></td>
<td>• Regular</td>
</tr>
<tr>
<td></td>
<td>• Hardware</td>
</tr>
<tr>
<td></td>
<td>• Software</td>
</tr>
<tr>
<td></td>
<td>Clear this option to install breakpoints as Regular breakpoints.</td>
</tr>
<tr>
<td>Restore watchpoints</td>
<td>Select this option to restore previous watchpoints.</td>
</tr>
<tr>
<td>Disable display of variable values by default</td>
<td>Select this option to disable the display of variable values. Clear this option to enable the display of variable values</td>
</tr>
<tr>
<td>Disable display of register values by default</td>
<td>Select this option to disable the display of register values. Clear this option to enable the display of register values</td>
</tr>
<tr>
<td>Refresh while running period (seconds)</td>
<td>Specifies the refresh period used when a view is configured to refresh, while the application is running. By default, the refresh period is set to two seconds.</td>
</tr>
</tbody>
</table>
4.1.3.2 Exceptions
Use this page to specify the processor exceptions you want the CodeWarrior debugger to catch.

Figure 18: Debugger options- Exceptions page

Selecting the checkboxes, available on the Exceptions page, configures the core to automatically halt when the corresponding exception is taken. The debugger stops at the entry point of the interrupt handler for the selected exception, allowing you to inspect the processor state and continue debugging from there.

4.1.3.3 Interrupts
Use this tab to inhibit or allow interrupts.

Debugging an application involves single-stepping through code. However, if you do not modify the behavior of interrupts that are part of normal code execution, an interrupt may occur and the debugger jumps to the interrupt handler code, rather than single-stepping to the next instruction. Therefore, you must mask, or inhibit, certain interrupt levels to prevent the interrupts from happening. When inhibiting interrupts, you can mask interrupts below a level that you specify.
Figure 19: Debugger options-Interrupts page

The table below lists the various options available on the Interrupts page.

### Table 32: Debugger options - Interrupts

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disable interrupts during stepping</td>
<td>Select this checkbox if you want to disable interrupts during stepping.</td>
</tr>
</tbody>
</table>

#### 4.1.3.4 Download

Use this page to specify which executable code sections the debugger downloads to the target, and whether the debugger should read back those sections and verify them.

**NOTE**

Selecting all options in the Program Download Options group significantly increases download time.

Initial Launch options apply to the first debugging session. Successive Runs options apply to subsequent debugging sessions.

The Download options control whether the debugger downloads the specified Program Section Data type to the target hardware. The Verify options control whether the debugger reads the specified Program Section Data type from the target hardware and compares the read data against the data written to the device.
The table below lists the various options available on the Download page.

**Table 33: Debugger options - Download**

<table>
<thead>
<tr>
<th>Section</th>
<th>Data Type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perform</td>
<td>standard</td>
<td>Controls download of the target application using memory write command.</td>
</tr>
<tr>
<td>First</td>
<td>download</td>
<td>Represents a group of settings that are used when an application is debugged for the first time.</td>
</tr>
<tr>
<td>Subsequent</td>
<td>download</td>
<td>Represents a group of settings that are used when the application is debugged subsequent times. To make these settings be used during debugging, you</td>
</tr>
<tr>
<td></td>
<td>verification</td>
<td>need to select the <strong>Cache Symbolics Between Sessions</strong> option on the <strong>Symbolics</strong> page.</td>
</tr>
<tr>
<td>Executable</td>
<td>control</td>
<td>Controls downloading and verification for executable sections. Check appropriate checkboxes to specify downloading and verifications, for initial</td>
</tr>
<tr>
<td></td>
<td>for</td>
<td>launch and for successive runs.</td>
</tr>
<tr>
<td>Constant</td>
<td>control</td>
<td>Controls downloading and verification for constant-data sections. Check appropriate checkboxes to specify downloading and verifications, for initial</td>
</tr>
<tr>
<td>Data</td>
<td>for</td>
<td>launch and for successive runs.</td>
</tr>
</tbody>
</table>

*Table continues on the next page...*
Table 33: Debugger options - Download (continued)

<table>
<thead>
<tr>
<th>Section Data Type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialized Data</td>
<td>Controls downloading and verification for initialized-data sections. Check appropriate checkboxes to specify downloading and verifications, for initial launch and for successive runs.</td>
</tr>
<tr>
<td>Uninitialized Data</td>
<td>Controls downloading and verification for uninitialized-data sections. Check appropriate checkboxes to specify downloading and verifications, for initial launch and for successive runs.</td>
</tr>
<tr>
<td>Execute Tasks</td>
<td>Enables the execution of target tasks.</td>
</tr>
<tr>
<td>Name</td>
<td>For target tasks, this is the name of the target task as seen in the Target Task view. For Debugger Shell scripts, this is the path to the CLDE script.</td>
</tr>
<tr>
<td>Task Type</td>
<td>Contains either Debugger Shell scripts or target tasks (such as Flash Programmer).</td>
</tr>
<tr>
<td>First</td>
<td>Represents a task that is executed first.</td>
</tr>
<tr>
<td>Subsequent</td>
<td>Represents a task that is executed subsequent times.</td>
</tr>
</tbody>
</table>

4.1.3.5 PIC

Use this page to specify an alternate address at which the debugger loads the PIC module onto target memory.

Usually, Position Independent Code (PIC) is linked in such a way that the entire image starts at address 0x00000000.

Figure 21: Debugger options-PIC page

The table below lists the various options available on the PIC page.
Table 34: PIC page options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alternate Load Address</td>
<td>Specify the starting address at which the debugger loads your program. You can also use this setting when you have an application which is built with ROM addresses and then relocates itself to RAM (such as U-Boot). Specifying a relocation address lets the debugger map the symbolic debugging information contained in the original ELF file (built for ROM addresses) to the relocated application image in RAM. Clear the checkbox to have the debugger load your program at a default starting address. The debugger does not verify whether your code can execute at the specified address. As a result, the PIC generation settings of the compiler, linker and your program's startup routines must correctly set any base registers and perform any required relocations.</td>
</tr>
</tbody>
</table>

**NOTE**

4.1.3.6 Other Executables

Use this page to specify additional ELF files to download or debug in addition to the main executable file associated with the launch configuration.

Figure 22: Debugger options-Other Executables page

The table below lists the various options available on the Other Executables page.
### Table 35: Debugger options - Other Executables

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>File list</td>
<td>Shows files and projects that the debugger uses during each debug session.</td>
</tr>
<tr>
<td></td>
<td><strong>Debug column:</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>Selected:</strong> The debugger loads symbolics for the file.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Cleared:</strong> The debugger does not load symbolics for the file.</td>
</tr>
<tr>
<td></td>
<td><strong>Download column:</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>Selected:</strong> The debugger downloads the file to the target device.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Cleared:</strong> The debugger does not download the file to the target device.</td>
</tr>
<tr>
<td>Add</td>
<td>Click to open the Debug Other Executable dialog, and add other executable file to debug while debugging this target. Use this dialog to specify the following settings:</td>
</tr>
<tr>
<td></td>
<td>• Specify the location of the additional executable - Enter the path to the executable file that the debugger controls in addition to the current project's executable file. Alternatively, click Workspace, File System, or Variables to specify the file path.</td>
</tr>
<tr>
<td></td>
<td>• Load symbols - Check to have the debugger load symbols for the specified file. Clear to prevent the debugger from loading the symbols. The Debug column of the File list corresponds this setting.</td>
</tr>
<tr>
<td></td>
<td>• Download to device - Check to have the debugger download the specified file to the target device. Specify the path of the file in the Specify the remote download path text box. Clear the Download to device checkbox to prevent the debugger from downloading the file to the device. The Download column of the File list corresponds to the Download to device setting.</td>
</tr>
<tr>
<td></td>
<td>• OK - Click to add the information that you specify in the Debug Other Executable dialog to the File list.</td>
</tr>
<tr>
<td>Change</td>
<td>Click to change the settings for the entry currently selected in the File list column. Change this information as needed, then click the OK button to update the entry in the File list.</td>
</tr>
<tr>
<td>Remove</td>
<td>Click to remove the entry currently selected in the File list.</td>
</tr>
</tbody>
</table>

### 4.1.3.7 Symbolics

Use this page to specify whether the IDE keeps symbolics in memory.

Symbolics represent an application's debugging and symbolic information. Keeping symbolics in memory, known as caching symbolics, is beneficial when you debug a large-size application.
Consider a situation in which the debugger loads symbolics for a large application, but does not download content to a hardware device and the project uses custom makefiles with several build steps to generate this application. In such a situation, caching symbolics helps speed up the debugging process. The debugger uses the readily available cached symbolics during subsequent debugging sessions. Otherwise, the debugger spends significant time creating an in-memory representation of symbolics during subsequent debugging sessions.

NOTE
Caching symbolics provides the most benefit for large applications, where doing so speeds up application-launch time. If you debug a small application, caching symbolics does not significantly improve the launch times.

Figure 23: Debugger options-Symbolics page

The table below lists the various options available on the Symbolics page.

Table 36: Debugger options - Symbolics

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache Symbolics Between Sessions</td>
<td>Select this option to have the debugger cache symbolics between debugging sessions. If you select this checkbox and clear the Create and Use Copy of Executable checkbox, the executable file remains locked after the debugging session ends. In the Debug view, right-click the locked file and select Un-target Executables to have the debugger delete its symbolics cache and release the file lock. The IDE enables this menu command when there are currently unused cached symbolics that it can purge. Clear this option so that the debugger does not cache symbolics between debugging sessions.</td>
</tr>
<tr>
<td>Create and Use Copy of Executable</td>
<td>Select this option to have the debugger create and use a copy of the executable file. Using the copy helps avoid file-locking issues with the build system. If you select this checkbox, the IDE can build the executable file in the background during a debugging session. Clear this option so that the debugger does not create and use a copy of the executable file.</td>
</tr>
</tbody>
</table>

4.1.3.8 OS Awareness
Use this page to specify the operating system (OS) that resides on the target device.

The table below lists the options available on the OS Awareness page.
### Table 37: OS Awareness page options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target OS</td>
<td>Use the <strong>Target OS</strong> list box to specify the OS that runs on the target device, or specify <strong>None</strong> to have the debugger use the bareboard.</td>
</tr>
</tbody>
</table>
| Boot Parameters tab        | Enable Command Line Settings - Select this option to specify settings for regular initialization. Enter the specific command line parameters in the **Command Line** and **Base Address** text boxes. Enable Initial RAM Disk Settings - Select this option to specify settings for flattened device tree initialization that downloads parameters to the kernel during its initialization. You specify a `.dts` file that contains initialization information.  
  - File Path - Specifies the path of the ramdisk that you transferred from the Linux machine  
  - Address - Specifies the address specified in Linux, initrd-start from the dts file  
  - Size - Specifies the size of the dts file  
  - Download to target - Downloads the initial RAM disk settings to the target  
  Open Firmware Device Tree Settings - Select this option to load parameters to the kernel from a bootloader on the ARMv7 processors  
  - File Path - Specifies the path to the dtb file for kernel debug  
  - Address - Specifies the address specified in Linux, initrd-start from the dts file |

*Table continues on the next page...*
### Table 37: OS Awareness page options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Debug tab</strong></td>
<td>Specifies the parameters required for Linux kernel debug.</td>
</tr>
<tr>
<td></td>
<td>• Enable Memory Translation - Select this option to translate memory by specifying the following values Physical Base Address - this is the CONFIG_PHYSICAL_START option of the kernel configuration Virtual Base Address - this is the CONFIG_KERNEL_START option of the kernel configuration Memory Size - this is the CONFIG_LOWMEM_SIZE option of the kernel configuration</td>
</tr>
<tr>
<td></td>
<td><strong>Note</strong>: The virtual memory space should not overflow the 32-bit memory space. This indicates that the Virtual Base Address + Memory Size shouldn't be greater than 0xFFFFFFFF. CodeWarrior displays an error when this happens.</td>
</tr>
<tr>
<td></td>
<td>• Enable Threaded Debugging Support - Select this option to enable support for Linux kernel threaded debugging</td>
</tr>
<tr>
<td></td>
<td>• Update Background Threads on Stop - Select this option only if you want to update the background threads on stop. Keep this option unchecked as it may increase debug speed.</td>
</tr>
<tr>
<td></td>
<td>• Enable Delayed Software Breakpoint Support - Select this option to enable support for delayed software breakpoints during kernel debug</td>
</tr>
<tr>
<td><strong>Modules tab</strong></td>
<td>This tab allows you to add modules to the Linux kernel project and configure the module's symbolics mapping. For more information on the Modules tab, see Configure the Modules' Symbolics Mapping topic.</td>
</tr>
</tbody>
</table>

### 4.1.4 Trace and Profile

CodeWarrior Tracing and Performance Analysis Tools provide visibility into an application as it runs on the hardware. This visibility can help you understand how your application runs, as well as identify operational problems.

The UI Platform Configurator tool reads the user settings from the input XML file and transforms them into target access memory writes to the configuration registers.

**Figure 24: Core Trace Path**

![Core Trace Path Diagram](Cortex A7 CPU diagram)
The figure above shows the core trace path within the ARMv7 architecture. The UI Platform Configurator configures the ARMv7 cores embedded trace macrocell (ETM), Funnel, embedded trace FIFO (ETF) and embedded trace router (ETR) modules. The ETF stores and forwards trace data using a dedicated RAM buffer. This reduces trace loss by absorbing spikes in trace data. The ETR redirects trace to the system bus for collection from alternative channels. For ARMv7, it is double data rate (DDR) memory.

The destination of the raw trace is either the internal trace buffer of the ETF module or a user defined buffer in DDR.

The basic setup can be done using the **Trace and Profile** tab in the Debug launch configuration:

- To enable tracing for any new project created, go to the **Debug Configuration**, select a launch configuration from CodeWarrior in the left panel. Select **Trace and Profile** tab from the right panel.

  ![Main tab](image)

  **Figure 25:** Main tab

- **The Overview** tab displays the information about the ARMv7 architecture.

- **The Basic** tab displays the predefined settings. For **Platform Configuration Settings**, you can apply different values based on what you want to achieve. The values are called profiles, and the framework allows creation of different profiles for a configuration block. The profiles dialog is used to create new, rename, delete or edit the settings of a profile for a configuration provider.
4.1.5 Source

Use this tab 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.

The table below lists the various options available on the Source page.
Table 38: Source tab options

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

4.1.6 Environment

Use this tab to specify the environment variables and values to use when an application runs.

Figure 28: Debug Configuration-Environment tab

The table below lists the various options available on the Environment page.

Table 39: Environment tab options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Environment variables to set</td>
<td>Lists the environment variable name and its value.</td>
</tr>
</tbody>
</table>

Table continues on the next page...
Table 39: Environment tab options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>Click to create a new environment variable.</td>
</tr>
<tr>
<td>Select</td>
<td>Click to select an existing environment variable.</td>
</tr>
<tr>
<td>Edit</td>
<td>Click to modify the name and value of a selected environment variable.</td>
</tr>
<tr>
<td>Remove</td>
<td>Click to remove selected environment variables from the list.</td>
</tr>
<tr>
<td>Append environment to native environment</td>
<td>Select to append the listed environment variables to the current native environment.</td>
</tr>
<tr>
<td>Replace native environment with specified environment</td>
<td>Select to replace the current native environment with the specified environment set.</td>
</tr>
</tbody>
</table>

4.1.7 Common

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

Figure 29: Debug Configuration-Common tab

The table below lists the various options available on the Common page.
Table 40: Common tab options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local file</td>
<td>Select to save the launch configuration locally.</td>
</tr>
<tr>
<td>Shared file</td>
<td>Select to specify the path of, or browse to, a workspace to store the launch configuration file, and be able to commit it to a repository.</td>
</tr>
<tr>
<td>Display in favorites menu</td>
<td>Select to add the configuration name to Run or Debug menus for easy selection.</td>
</tr>
<tr>
<td>Allocate console (necessary for input)</td>
<td>Select to assign a console view to receive the output. You must select this option if you want to use the host CodeWarrior to view the output of the debugged application.</td>
</tr>
<tr>
<td>File</td>
<td>Specify the file name to save output. For Linux applications, this option provides a way to select a host-side file to redirect the output forwarded by CodeWarrior TRK to host CodeWarrior (if redirections are specified in the Arguments tab, then this feature makes no sense because redirections are using target-side files).</td>
</tr>
<tr>
<td>Append</td>
<td>Select to append output. Clear to recreate file each time. Selecting this option means that the file (host-side file, in case of Linux applications) mentioned in the File text box will not be overwritten for new content. Instead, the new content will be appended to the file.</td>
</tr>
<tr>
<td>Port</td>
<td>Select to redirect standard output (stdout, stderr) of a process being debugged to a user specified socket.</td>
</tr>
<tr>
<td>Act as Server</td>
<td>Select to redirect the output from the current process to a local server socket bound to the specified port.</td>
</tr>
<tr>
<td>Hostname/IP Address</td>
<td>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</td>
</tr>
<tr>
<td>Launch in background</td>
<td>Select to launch configuration in background mode.</td>
</tr>
</tbody>
</table>

4.2 Customizing Debug Configurations

When you use the CodeWarrior wizard to create a new project, the wizard sets the project’s launch configurations to default values. You can change the default values of your project’s launch configurations, according to your program’s requirements.

To modify the launch configurations:

1. Start the CodeWarrior IDE.
2. From the main menu bar of the IDE, select Run > Debug Configurations.
The **Debug Configurations** dialog appears. The left side of this dialog has a list of debug configurations that apply to the current application.

3. Expand the **CodeWarrior** configuration.

4. From the expanded list, select the debug configuration that you want to modify.

   The following figure shows the **Debug Configurations** dialog with the settings for the debug configuration you selected.

   
   ![CodeWarrior Debug Configuration-Main tab](image)

   Figure 30: CodeWarrior Debug Configuration-Main tab

5. In the group of tabs in the upper-right side of the dialog, click a tab.

6. Change the settings on the debug configuration page as per your requirements. See *Using Debug Configurations dialog* on page 53 for details on the various settings of this page.

7. Click **Apply** to save the new settings.

   When you finish, you can click **Debug** to start a new debugging session, or click **Close** to save your changes and close the **Debug Configurations** dialog.
4.3 Reverting Debug Configuration settings

After making some modifications in a debug configuration's settings, you can either save the pending (unsaved) changes or revert to last saved settings.

To save the pending changes, click the **Apply** button of the **Debug Configurations** dialog, or click the **Close** button and then the **Yes** button.

To undo pending changes and restore the last saved settings, click the **Revert** button at the bottom of the **Debug Configurations** dialog.

The IDE restores the last set of saved settings to all pages of the **Debug Configurations** dialog. Also, the IDE disables the **Revert** button until you make new pending changes.
Debug Configurations
Reverting Debug Configuration settings
Chapter 5
Working with Debugger

This chapter explains various aspects of CodeWarrior debugging, such as debugging a project, configuring connections, setting breakpoints and watchpoints, working with registers, viewing memory, viewing cache, and debugging externally built executable files.

NOTE
This chapter documents debugger features that are specific to CodeWarrior Development Studio for QorIQ LS series - ARM V7 ISA.

This chapter explains:
- Debugging CodeWarrior project on page 79
- Consistent debug control on page 80
- Secure Debug mode on page 80
- Connection types on page 80
- Editing remote system configuration on page 81
- CodeWarrior command-line debugger on page 84
- Working with Breakpoints on page 86
- Working with Watchpoints on page 90
- Working with Registers on page 93
- Viewing Memory on page 101
- Changing Program Counter Value on page 103
- Hard Resetting on page 103
- Setting Stack Depth on page 103
- Importing CodeWarrior Executable file Wizard on page 104
- Debugging Externally Built Executable Files on page 108

5.1 Debugging CodeWarrior project

This section explains how to change the debugger settings and how to debug a CodeWarrior project.

The CodeWarrior Bareboard Project Wizard or the CodeWarrior Linux Project Wizard sets the debugger settings of a project's launch configurations to default values. You can change these default values as per your requirements.

To change the debugger settings and start debugging a CodeWarrior project, perform these steps:

1. From the CodeWarrior IDE menu bar, select Run > Debug Configurations. The CodeWarrior IDE uses the settings in the launch configuration to generate debugging information and initiate communications with the target board.

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

2. Expand the CodeWarrior configuration.

3. From the expanded list, select the debug configuration that you want to modify.
4. Click **Apply** to save the new settings.

**TIP**

You can click **Revert** to undo any of the unsaved changes. The CodeWarrior IDE restores the last set of saved settings to all pages of the **Debug Configurations** dialog. Also, the IDE disables **Revert** until you make new pending changes.

5. Click **Debug** to start the debugging session.

You just modified the debugger settings and initialized a debugging session.

5.2 **Consistent debug control**

This section describes the **consistent debug control** feature of the CodeWarrior debugger.

When you attempt to stop the target during a debugging session, the **consistent debug control** feature enables the debugger to report core's Doze and Nap low power management states.

In addition, the debugger at the same time grants you access to the system states, such as core registers, TLB registers, caches, and so on.

When you attempt to resume the debugging session, the debugger displays a warning message and puts the respective core in the same power management state (Doze or Nap, whichever is the previous one). The debugger waits for the core to exit out of Doze or Nap state to continue with the attempted operation.

5.3 **Secure Debug mode**

This section describes the secure debug feature of the CodeWarrior debugger.

If the processor is in the Secure Debug mode and if the unlock key is not provided, then a popup is displayed requesting the unlock key. If you provide a wrong key and an unlock sequence is run by the debugger with the erroneous key, then the associated part will be locked until a rest occurs, and you will need to hard reset the target board to connect to it again.

5.4 **Connection types**

This section describes the different connection types provided by the CodeWarrior debugger for connecting the target board to the host computer.

The CodeWarrior debugger provides the following three connection types for connecting the target board to the host computer:

- CMSIS-DAP: Available only for TWR and IOT boards
- CodeWarrior TAP (over USB): Available for QDS, RDB, TWR, and IOT boards
- CodeWarrior TAP (over Ethernet): Available for QDS, RDB, TWR, and IOT boards
5.5 Editing remote system configuration

The remote system configuration model defines the connection and system configurations where you can define a single system configuration that can be referred to by multiple connection configurations.

To edit the system configuration, perform these steps:

1. Select Run > Debug Configurations.
   
The Debug Configurations dialog appears.

2. In the Target Settings panel, click Edit next to the Connection drop-down list.
   
The Properties for <connection launch configuration> window appears.

   For targets that support the secure debug feature, the Secure debug key option is available on the Advanced tab of the Properties for <connection launch configuration> window, as shown in the figure below.

   ![Figure 31: Setting Secure debug key option](image)

   Select the Secure debug key checkbox to enable the CodeWarrior debugger to unlock the secured board with the secure debug key provided in the associated text box. If this option is not selected, then you will receive a secure debug violation error when you try to perform debugging on the locked board, and a window will appear where you can provide a secure debug key.

   If you provide a wrong key and an unlock sequence is run by the debugger with the erroneous key, then the associated part will be locked until a rest occurs, and you will need to hard reset the target board to connect to it again.

   CodeWarrior connection server (CCS) checks if the board is secured or not during chain configuration and/or reset to handle automatic unlocking with the provided secure debug key.

3. Click Edit next to the Target drop-down list.
   
The Properties for <system launch configuration> window appears.

4. Select the appropriate system type from the Target type drop-down list.
5. Make the respective settings in **Initialization tab** on page 82 and **Memory tab** on page 83.
6. Click **OK** to save the settings.
7. Click **OK** to close the **Properties** window.

This section contains the following subsections:

- **Initialization tab** on page 82
- **Memory tab** on page 83

### 5.5.1 Initialization tab

Use the Initialization tab to specify the target initialization file for various cores.

The figure below shows the settings on the Initialization tab.

![Figure 32: CodeWarrior TAP connection type - Initialization tab](image)

The table below lists the various options available on the **Initialization** page.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute target reset</td>
<td>Select to execute target system reset.</td>
</tr>
<tr>
<td>Target</td>
<td>Lists the targets and the supported cores.</td>
</tr>
<tr>
<td>Run out of reset</td>
<td>Select to include the respective core for run out of reset operation.</td>
</tr>
</tbody>
</table>

*Table continues on the next page...*
Table 41: Initialization tab options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize target</td>
<td>Click to specify a target initialization file for the respective core.</td>
</tr>
<tr>
<td>Initialize target script</td>
<td>Lists the path to a Debugger Shell Tcl script that runs when launching a debug session for the respective core. To edit, select a cell, then click the ellipsis button to open the Target InitializationFile dialog. The settings for a group of cores can be changed all at once by editing the cell of a common ancestor node in the Target hierarchy.</td>
</tr>
</tbody>
</table>

5.5.2 Memory tab

Use the Memory tab to specify the memory configuration file for various cores.

The figure below shows the settings on the Memory tab.

Figure 33: CodeWarrior TAP connection type - Memory tab

The table below lists the various options available on the Memory page.

Table 42: Memory tab options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target</td>
<td>Lists the targets and the supported cores.</td>
</tr>
</tbody>
</table>

Table continues on the next page...
Table 42: Memory tab options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory configuration</td>
<td>Select to specify a memory configuration file for the respective core.</td>
</tr>
<tr>
<td>Memory configuration file</td>
<td>Lists the path to the memory configuration file for the respective core. To edit, select a cell, then click the ellipsis button to open the Memory Configuration File dialog. The settings for a group of cores can be changed all at once by editing the cell of a common ancestor node in the Target hierarchy.</td>
</tr>
</tbody>
</table>

5.6 CodeWarrior command-line debugger

CodeWarrior supports a command-line interface that you can use to interact with the CodeWarrior debugger, by issuing commands.

You can use the command-line interface together with various scripting engines, such as the Microsoft® Visual Basic® script engine, the Java™ script engine, TCL, Python, and Perl. You can even issue a command that saves your command-line activity to a log file.

You use the Debugger Shell view to issue command lines to the IDE. For example, you enter the command `debug` in this window to start a debugging session. The window displays the standard output and standard error streams of command-line activity.

To open the Debugger Shell view, follow these steps:

1. Switch the IDE to the Debug perspective and start a debugging session.
2. Select Window > Show View > Other.
   The Show View dialog appears.
3. Expand the Debug group.
4. Select Debugger Shell.
5. Click OK.
   The Debugger Shell view appears in the view stack at the bottom of the IDE.

To issue a command-line command, type the desired command at the command prompt (`%`) in the Debugger Shell view, then press Enter or Return. The command-line debugger executes the specified command.

NOTE
To display a list of the commands the command-line debugger supports, type `help` at the command prompt and press Enter. The `help` command lists each supported command along with a brief description of each command.
If you work with hardware as part of your project, you can use the command-line debugger to issue commands to the debugger while the hardware is running.

**TIP**

To view page-wise listing of the debugger shell commands, right-click in the Debugger Shell view and select **Paging** from the context menu. Alternatively, click the **Enable Paging** icon from the view toolbar.

The table below lists the instructions for common command-line debugging tasks.

**Table 43: Common command-line debugging tasks**

<table>
<thead>
<tr>
<th>Task</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open the Debugger Shell view.</td>
<td>Choose <strong>Window &gt; Show View &gt; Other &gt; Debugger Shell.</strong></td>
<td>The Debugger Shell view appears.</td>
</tr>
<tr>
<td>Use the help command.</td>
<td>1. On the Debugger shell command prompt (%&gt;), type help.</td>
<td>The command list for CodeWarrior appears.</td>
</tr>
<tr>
<td></td>
<td>2. Press Enter key.</td>
<td></td>
</tr>
<tr>
<td>Enter a command.</td>
<td>1. On the <strong>Debugger shell</strong>, type a command followed by a space.</td>
<td>You can use shortcuts instead of complete command names, such as <strong>k</strong> for <strong>kill.</strong></td>
</tr>
<tr>
<td></td>
<td>2. Type any valid command-line options, separating each with a space.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. Press Enter key.</td>
<td></td>
</tr>
<tr>
<td>View debug command hints.</td>
<td>Type alias followed by a space.</td>
<td>The syntax for the rest of the command appears.</td>
</tr>
</tbody>
</table>

*Table continues on the next page...*
Table 43: Common command-line debugging tasks (continued)

<table>
<thead>
<tr>
<th>Task</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Review previous commands.</td>
<td>Press Up Arrow and Down Arrow keys.</td>
<td></td>
</tr>
<tr>
<td>Clear command from the command line.</td>
<td>Press the Esc key.</td>
<td></td>
</tr>
<tr>
<td>Stop an executing script.</td>
<td>Press the Esc key.</td>
<td></td>
</tr>
<tr>
<td>Toggle between insert/overwrite mode.</td>
<td>Press the Insert key.</td>
<td></td>
</tr>
<tr>
<td>Scroll up/down a page.</td>
<td>Press Page Up or Page Down key.</td>
<td></td>
</tr>
<tr>
<td>Scroll left/right one column.</td>
<td>Press Ctrl+Left Arrow or Ctrl+Right Arrow keys.</td>
<td></td>
</tr>
<tr>
<td>Scroll to beginning or end of buffer.</td>
<td>Press Ctrl+Home or Ctrl+End keys.</td>
<td></td>
</tr>
</tbody>
</table>

5.7 Working with Breakpoints

A breakpoint is set on an executable line of a program; if the breakpoint is enabled when you debug, the execution suspends before that line of code executes.

The different breakpoint types that you can set are listed below:

- **Software breakpoints**: The debugger sets a software breakpoint into target memory. When program execution reaches the breakpoint, the processor stops and activates the debugger. The breakpoint remains in the target memory until the user removes it.

  The breakpoint can only be set in writable memory, such as SRAM or DDR. You cannot use this type of breakpoints in ROM.

- **Hardware breakpoints**: Selecting the Hardware menu option causes the debugger to use the internal processor breakpoints. These breakpoints are usually very few and can be used with all types of memories (ROM/RAM) because they are implemented by using processor registers.

  **TIP**
  
  You can also set breakpoint types by issuing the `bp` command in the **Debugger Shell** view.

In this section:

- **Setting Breakpoints** on page 86
- **Setting Hardware Breakpoints** on page 89
- **Removing Breakpoints** on page 90

5.7.1 Setting Breakpoints

This section explains how to set breakpoints within a program in CodeWarrior IDE.

To set a breakpoint, perform the following steps:

1. Switch to the **Debug** perspective in CodeWarrior IDE.
2. Open the **Debug** view if it is not already open by selecting **Window > Show View > Debug**.

The **Debug** view appears, shown in the figure below.

![Debug view](Figure 35)

3. Expand the **Thread** group.

4. Under the **Thread** group, select the thread that has the `main()` function.

The source code appears in the **Editor** view (shown in the figure below). The small blue arrow to the left of the source code indicates which code statement the processor's program counter is set to execute next.

![Editor view](Figure 36)

5. In the **Editor** view, place the cursor on the line that has this statement: `counter++;`

6. Select **Run > Toggle Line Breakpoint**.

A blue dot appears in the marker bar to the left of the line (shown in the figure below). This dot indicates an enabled breakpoint. After the debugger installs the breakpoint, a blue checkmark appears beside the dot. The debugger installs a breakpoint by loading into the Java™ virtual machine the code in which you set that breakpoint.
An alternate way to set a breakpoint is to double-click the marker bar to the left of any source-code line. If you set the breakpoint on a line that does not have an executable statement, the debugger moves the breakpoint to the closest subsequent line that has an executable statement. The marker bar shows the installed breakpoint location. If you want to set a hardware breakpoint instead of a software breakpoint, use the `bp` command in the Debugger Shell view. You can also right-click the marker bar to the left of any source-code line, and select **Set Special Breakpoint** from the context menu that appears.

### TIP

Figure 37: Editor view - after setting breakpoints

7. From the menu bar, select **Run > Resume**.

   The debugger executes all lines up to, but not including, the line at which you set the breakpoint. The editor view highlights the line at which the debugger suspended execution (shown in the figure below). Note also that the program counter (blue arrow) is positioned here.
5.7.2 Setting Hardware Breakpoints

This section explains how to set hardware breakpoints within a program in CodeWarrior IDE.

There are two ways to set hardware breakpoints:

• Using IDE to set Hardware Breakpoints on page 89
• Using Debugger Shell to set Hardware Breakpoints on page 89

5.7.2.1 Using IDE to set Hardware Breakpoints

This section explains how to set a hardware breakpoint using CodeWarrior IDE.

The steps are as follows:

1. In the CodeWarrior IDE, select Run > Breakpoint Types > C/C++ Hardware Breakpoints.
2. In the Editor view, click in the source line where you want to place the breakpoint.
3. Select Run > Toggle Breakpoint.

A hardware breakpoint appears in the marker bar on the left side of the source line.

5.7.2.2 Using Debugger Shell to set Hardware Breakpoints

This section explains how to set a hardware breakpoint using the Debugger Shell view.

The steps are as follows:

1. Open the Debugger Shell view.
2. Begin the command line with the text:
   ```
   bp -hw
   ```
3. Complete the command line by specifying the function, address, or file at which you want to set the hardware breakpoint.
For example, to set a breakpoint for line 6 in your program, type:

```bash
bp -hw 6
```

4. Press the Enter key.

The debugger shell executes the command and sets the hardware breakpoint.

---

**TIP**

Enter help `bp` at the command-line prompt to see examples of the `bp` command syntax and usage.

---

### 5.7.3 Removing Breakpoints

This section explains how to remove breakpoints from a program in CodeWarrior IDE.

To remove a breakpoint from your program, you have two options:

- **Remove Breakpoints using Marker Bar** on page 90
- **Remove Breakpoints using Breakpoints view** on page 90

#### 5.7.3.1 Remove Breakpoints using Marker Bar

This section explains how to remove an existing breakpoint using the marker bar.

The steps are as follows:

1. Right-click the breakpoint in the marker bar.
2. Select **Toggle Breakpoint** from the menu that appears.

#### 5.7.3.2 Remove Breakpoints using Breakpoints view

This section explains how to remove an existing breakpoint using the Breakpoints view.

The steps are as follows:

1. Open the **Breakpoints** view if it is not already open by selecting **Window > Show View > Breakpoints**.
   
   The **Breakpoints** view appears, displaying a list of breakpoints.
2. Right-click the breakpoint you wish to remove and select **Remove** from the menu that appears.
   
   The selected breakpoint is removed, and it disappears from the both the marker bar and the list in the view.

---

**NOTE**

To remove all of the breakpoints from the program at once, select **Remove All** from the menu.

---

### 5.8 Working with Watchpoints

A watchpoint is another name for a data breakpoint that you can set on an address or a range of addresses in the memory.

The debugger halts execution each time the watchpoint location is read, written, or accessed (read or written). You can set a watchpoint using the **Add Watchpoint** dialog. To open the **Add Watchpoint** dialog, use one of the following views:

- **Breakpoints** view
• Memory view
• Variables view

The debugger handles both watchpoints and breakpoints in similar manners. You can use the Breakpoints view to manage both watchpoints and breakpoints. It means, you can use the Breakpoints view to add, remove, enable, and disable both watchpoints and breakpoints. The debugger attempts to set the watchpoint if a session is in progress based on the active debugging context (the active context is the selected project in the Debug view).

If the debugger sets the watchpoint when no debugging session is in progress, or when re-starting a debugging session, the debugger attempts to set the watchpoint at startup as it does for breakpoints. The Problems view displays error messages when the debugger fails to set a watchpoint. For example, if you set watchpoints on overlapping memory ranges, or if a watchpoint falls out of execution scope, an error message appears in the Problems view. You can use this view to see additional information about the error.

The following sections explain how to set or remove watchpoints:
• Setting Watchpoints on page 91
• Removing Watchpoints on page 93

5.8.1 Setting Watchpoints

Use the Add Watchpoint dialog to create a watchpoint for a memory range.

You can specify these parameters for a watchpoint:
• An address (including memory space)
• An expression that evaluates to an address
• A memory range
• An access type on which to trigger

To open the Add Watchpoint dialog, follow these steps:
1. Open the Debug perspective.
2. Click one of these tabs:
   • Breakpoints
   • Memory
   • Variables

   The corresponding view appears.
3. Right-click the appropriate content inside the view as mentioned in the table below.

<table>
<thead>
<tr>
<th>In the view...</th>
<th>Right-click...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breakpoints</td>
<td>An empty area inside the view.</td>
</tr>
<tr>
<td>Memory</td>
<td>The cell or range of cells on which you want to set the watchpoint.</td>
</tr>
<tr>
<td>Variables</td>
<td>A global variable.</td>
</tr>
</tbody>
</table>

NOTE
The debugger does not support setting a watchpoint on a stack variable or a register variable.

4. Select Add Watchpoint (C/C++) from the context menu that appears.
The **Add Watchpoint** dialog appears (shown in the figure below). The debugger sets the watchpoint according to the settings that you specify in the **Add Watchpoint** dialog. The **Breakpoints** view shows information about the newly set watchpoint. The **Problems** view shows error messages when the debugger fails to set the watchpoint.

![Add Watchpoint dialog](image)

The table below describes the options available in the **Add Watchpoint** dialog.

**Table 45: Add Watchpoint dialog options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
</table>
| Expression to watch   | Enter an expression that evaluates to an address on the target device. When the specified expression evaluates to an invalid address, the debugger halts execution and displays an error message. You can enter these types of expressions:  
  • An r-value, such as &variable  
  • A register-based expression. Use the $ character to denote register names. For example, enter $SP-12 to have the debugger set a watchpoint on the stack pointer address minus 12 bytes.  
  The **Add Watchpoint** dialog does not support entering expressions that evaluate to registers. |
| Memory space          | Select this option to specify an address, including memory space, at which to set the watchpoint. Use the text box to specify the address or address range on which to set the watchpoint. If a debugging session is not active, the text/list box is empty, but you can still type an address or address range. |
| Units                 | Enter the number of addressable units that the watchpoint monitors.                                                                                                                                          |
| Write                 | Select this option to enable the watchpoint to monitor write activity on the specified memory space and address range. Clear this option if you do not want the watchpoint to monitor write activity.                     |

*Table continues on the next page...*
Table 45: Add Watchpoint dialog options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>Select this option to enable the watchpoint to monitor read activity on the specified memory space and address range. Clear this option if you do not want the watchpoint to monitor read activity.</td>
</tr>
</tbody>
</table>

5.8.2 Removing Watchpoints

This section explains how to remove a watchpoint.

The steps are as follows:

1. Open the Breakpoints view if it is not already open by selecting Window > Show View > Breakpoints.
   
   The Breakpoints view appears, displaying a list of watchpoints.

2. Right-click the watchpoint you wish to remove and select Remove from the menu that appears.
   
   The selected watchpoint is removed, and it disappears from the list in the Breakpoints view.

5.9 Working with Registers

Use the Registers view to display and modify the contents of the registers of the processor on your target board.

To display the Registers view, select Window > Show View > Other > Debug > Registers. The Registers view appears (shown in the figure below). The default state of the Registers view provides details on the processor's registers.

The Registers view displays categories of registers in a tree format. To display the contents of a particular category of registers, expand the tree element of the register category of interest. The figure below shows the Registers view with the General Purpose Registers tree element expanded.

You can also view and update registers by issuing the reg, change, and display commands in the Debugger Shell view.

Figure 40: Registers view

In this section:

- Changing bit value of register on page 94
5.9.1 Changing bit value of register

This section explains how to change a bit value in a register.

The steps are as follows:

1. Switch the IDE to the Debug perspective and start a debug session.
2. Open the Registers view by selecting Window > Show View > Other > Debug > Registers.
3. In the Registers view, expand the register group that contains the register with the bit value that you want to change.
4. Click the register’s current bit value in the view’s Value column.
   The value becomes editable.
5. Type in the new value.
6. Press the Enter key.
   The debugger updates the bit value. The bit value in the Value column changes to reflect your modification.

5.9.2 Viewing Register details

You can view the registers and their descriptions using the Registers view.

The steps are as follows:

1. Start a debug session.
2. In the Debug perspective, click the Registers view tab.
   The Registers view appears.
3. Click the View Menu button (the inverted triangle) on the Registers view toolbar.
4. Select Layout > Vertical or Layout > Horizontal to show register details.

---

**NOTE**

Selecting Layout > Registers View Only hides the register details.

The details of the register, selected by default in the Registers view, are displayed, as shown in the figure below.
5. Expand a register group to see individual registers.

6. Select a specific register by clicking it.

   The details of the selected register get displayed.

   **NOTE**
   Use the **Format** list box to change the format of data displayed for the selected register.

7. Examine register details. For example,
   - Use the **Bit Fields** on page 96 group to see a graphical representation of the selected register's bit fields. You can use this graphical representation to select specific bits or bit fields.
   - Use the **Actions** on page 97 group to perform operations, such as update bit field values and format the displayed register data.
   - Use the **Description** on page 98 group to see an explanation of the selected register, bit field, or bit value.

   **TIP**
   To enlarge the **Registers** view, click **Maximize** on the view's toolbar. After you finish looking at the register details, click **Restore** on the view's toolbar to return the view to its previous size. Alternatively, right-click the **Registers** tab and select **Detached**. The **Registers** view becomes a floating window that you can resize. After you finish looking at the register details, right-click the **Registers** tab of the floating window and select **Detached** again. You can rearrange the re-attached view by dragging its tab to a different collection of view tabs.

This section contains the following subsections:
- **Bit Fields** on page 96
- **Actions** on page 97
- **Description** on page 98
5.9.2.1 Bit Fields

The Bit Fields group of the Registers view shows a graphical representation of the selected register's bit values.

This graphical representation (shown in the figure below) shows how the register organizes bits. You can use this representation to select and change the register's bit values. Hover the cursor over each part of the graphical representation in order to see additional information.

![Figure 42: Register Details - Bit Fields group](image)

A bit field is either a single bit or a collection of bits within a register. Each bit field has a mnemonic name that identifies it. You can use the Field list box to view and select a particular bit field of the selected register. The list box shows the mnemonic name and bit-value range of each bit field. In the Bit Fields graphical representation, a box surrounds each bit field. A red box surrounds the bit field shown in the Field list box.

After you use the Field list box to select a particular bit field, you see its current value in the = text box. If you change the value shown in the text box, the Registers view shows the new bit field value.

To change a bit field in a register, you must first start a debugging session, and then open the Registers view. To change a bit field, perform these steps:

1. In the Registers view, view register details.
2. Expand the register group that contains the bit field you want to change.

   Register details appear in the Registers view (shown in the figure below).
3. From the expanded register group above the register details, select the name of the register that contains the bit field that you want to change.

The **Bit Fields** group displays a graphical representation of the selected bit field. The **Description** group displays explanatory information about the selected bit field and parent register.

4. In the **Bit Fields** group, click the bit field that you want to change. Alternatively, use the **Field** list box to specify the bit field that you want to change.

5. In the **=** text box, type the new value that you want to assign to the bit field.

6. In the **Action** group, click **Write**.

   The debugger updates the bit field value. The bit values in the **Value** column and the **Bit Fields** group change to reflect your modification.

   **NOTE**

   Click **Revert** to discard your changes and restore the original bit field value.

### 5.9.2.2 Actions

Use the **Actions** group of the **Registers** view to perform various operations on the selected register’s bit field values.

The figure below shows the **Actions** group.

![Figure 44: Register view - Actions Group](image)

The table below lists each item in the **Actions** group and explains the purpose of each.
Table 46: Actions Group Items

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Revert</td>
<td>Discard your changes to the current bit field value and restore the original value. The debugger disables this button if you have not made any changes to the bit field value.</td>
</tr>
<tr>
<td>Write</td>
<td>Save your changes to the current bit field value and write those changes into the register’s bit field. The debugger disables this button after writing the new bit field value, or if you have not made any changes to that value.</td>
</tr>
<tr>
<td>Reset</td>
<td>Change each bit of the bit field value to its register-reset value. The register takes on this value after a target-device reset occurs. To confirm the bit field change, click Write. To cancel the change, click Revert.</td>
</tr>
<tr>
<td>Summary</td>
<td>Display Description group content in a pop-up window. Press the Esc key to close the pop-up window.</td>
</tr>
<tr>
<td>Format</td>
<td>Specify the data format of the displayed bit field values.</td>
</tr>
</tbody>
</table>

5.9.2.3 Description

The Description group of the Registers view shows explanatory information for the selected register. The figure below shows the Description group.

![Figure 45: Register view - Description group](image)

The register information covers:
- Current value
- Description
- Bit field explanations and values

Some registers have multiple modes (meaning that the register's bits can have multiple meanings, depending on the current mode). If the register you examine has multiple modes, you must select the appropriate mode.

5.9.3 Registers view context menu

The registers view context menu provides you various options for working with registers. To display the registers view context menu, right-click a register in the Registers view. The table below lists the options of the registers view context menu.
### Table 47: Registers view context menu options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select All</td>
<td>Selects the entire contents of the current register.</td>
</tr>
<tr>
<td>Copy Registers</td>
<td>Copies to the system clipboard the contents of the selected register.</td>
</tr>
<tr>
<td>Enable</td>
<td>Allows the debugger to access the selected register.</td>
</tr>
<tr>
<td>Disable</td>
<td>Prevents the debugger from accessing the selected register.</td>
</tr>
<tr>
<td>View Memory</td>
<td>Displays the corresponding memory for the selected register.</td>
</tr>
<tr>
<td>Cast to Type</td>
<td>Opens a dialog that you can use to cast the selected register value to a different data type.</td>
</tr>
<tr>
<td>Restore Original Type</td>
<td>Reverts the selected register value to its default data type.</td>
</tr>
<tr>
<td>Find</td>
<td>Opens a dialog that you can use to select a particular register.</td>
</tr>
<tr>
<td>Change Value</td>
<td>Opens a dialog that you can use to change the current register value.</td>
</tr>
<tr>
<td>Add Register Group</td>
<td>Opens a dialog that you can use to create a new collection of registers to display in the Registers view.</td>
</tr>
<tr>
<td>Restore Default Register Groups</td>
<td>Resets the custom groups of registers created using the Add Register Group option, and restores the default groups provided by the debugger as they were when CodeWarrior was installed. Note that if you select this option, all custom groupings of registers done by you are lost.</td>
</tr>
</tbody>
</table>

### 5.9.4 Working with Register Groups

This section describes different operations that can be performed on register groups.

You can perform the following operations on the register groups:

- [Adding Register Group](#) on page 99
- [Editing Register Group](#) on page 100
- [Removing Register Group](#) on page 101

#### 5.9.4.1 Adding Register Group

The default display of the Registers view groups related registers into a tree structure. You can add a custom group of registers to the default tree structure.

To add a new register group, perform these steps:

1. Right-click in the Registers view.
   - A context menu appears.
2. Select Add Register Group from the context menu.
   - The Register Group dialog appears, as shown in the figure below.
3. Enter in the **Group Name** text box a descriptive name for the new group.

4. Select the checkbox next to each register you want to appear in the new group.

   **TIP**
   
   Click **Select All** to check all of the checkboxes. Click **Deselect All** to clear all of the checkboxes.

5. Click **OK**.

   The **Register Group** dialog closes. The new group name appears in the **Registers** view.

### 5.9.4.2 Editing Register Group

In the **Registers** view, you can edit both the default register groups and the groups that you add.

To do so, use the following steps:

1. In the **Registers** view, right-click the name of the register group you want to edit.
   
   A context menu appears.

2. Select **Edit Register Group** from the context menu.
   
   The **Register Group** dialog appears.

3. If you wish, enter in the **Group Name** text box a new name for the group.
4. Select the checkbox next to each register you want to appear in the group.

**TIP**

Click **Select All** to check all of the checkboxes. Click **Deselect All** to clear all of the checkboxes.

5. Click **OK**.

The **Register Group** dialog closes. The new group name appears in the **Registers** view.

### 5.9.4.3 Removing Register Group

In the **Registers** view, you can remove register groups.

To remove a register group, follow these steps:

1. In the **Registers** view, right-click the name of register group that you wish to remove.
   
   A context menu appears.

2. Select **Remove Register Group** from the context menu.

   The selected register group disappears from the **Registers** view.

### 5.10 Viewing Memory

This section explains how to view memory of a target processor.

The debugger allocates multiple memory spaces in the IDE for flexible control over the memory access. The number of supported memory spaces and their properties depends upon the debugged processor.

You can display and access the supported memory spaces for a target in the **Memory** and **Memory Browser** views, in the Import/Export/Fill Memory Action Task View or in the **Debugger Shell** view using the designated memory space prefix. Use the `mem -ms` command to list the supported memory spaces for a processor.

To display the **Memory** view, select **Window > Show View > Other > Debug > Memory**. The figure below shows a **Memory** view displaying physical memory address space.
The Memory view seamlessly displays 32-bit, 36-bit, 40-bit, and 64-bit addresses depending upon the selected memory space and the target processor currently under debug process.

This section contains the following subsection:

- **Adding Memory Monitor** on page 102

### 5.10.1 Adding Memory Monitor

The Memory view provides you option to display the supported memory spaces for a target.

Perform these steps:

1. In the Memory view, click the Add Memory Monitor icon.
   
   The Monitor Memory dialog appears.

2. Specify the address in the Enter address or expression to monitor drop-down list.

3. Select one of the supported memory spaces from the Memory space drop-down list.
   
   - Virtual Secure (x)
     
     Indicates that the specified address space is same as the address space in which the processor executes in Secure mode.
   
   - Virtual Non Secure (nx)
     
     Indicates that the specified address space is same as the address space in which the processor executes in Non Secure mode.
   
   - Physical (p)
Indicates that the specified address is interpreted as a physical address and the access disregards the cache and accesses whatever is in the memory.

- Physical Cache-coherent (c)
  Indicates that the specified address is interpreted as a physical address. If the data is in cache, the debugger gets it from there, otherwise the access goes to the memory.

### 5.11 Changing Program Counter Value

This section explains how to change the program counter value in CodeWarrior IDE to make the debugger execute a specific line of code.

To change the program-counter value, follow these steps:

1. Start a debugging session.
2. In the Editor view, place the cursor on the line that you want the debugger to execute next.
3. Right-click in the Editor view.
   A context menu appears.
4. From the context menu, select Move To Line.

The CodeWarrior IDE modifies the program counter to the specified location. The Editor view shows the new location.

### 5.12 Hard Resetting

Use the reset hard command in the Debugger Shell view to send a hard reset signal to the target processor.

<table>
<thead>
<tr>
<th>NOTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>The <strong>Hard Reset</strong> command is enabled only if the debug hardware you are using supports it.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>You can also perform a hard reset by clicking <strong>Reset</strong> on the Debug perspective toolbar.</td>
</tr>
</tbody>
</table>

### 5.13 Setting Stack Depth

This section describes how to control the depth of the call stack displayed by the debugger.

Select **Window > Preferences > C/C++ > Debug > Maximum stack crawl depth** option to set the depth of the stack to read and display. Showing all levels of calls when you are examining function calls several levels deep can sometimes make stepping through code more time consuming. Therefore, you can use this menu option to reduce the depth of calls that the debugger displays.
5.14 Importing CodeWarrior Executable file Wizard

The **Import a CodeWarrior Executable file** wizard helps you to import a CodeWarrior executable file and create a new project.

To use the **Import a CodeWarrior Executable file** wizard, perform these steps:

1. From the CodeWarrior IDE menu bar, select **File > Import**.
   
   The **Import** wizard launches and the **Select** page appears, as shown in the figure below.

   ![Import Wizard - Selecting CodeWarrior Executable Importer](image)

2. Expand the **CodeWarrior** group.

3. Select the **CodeWarrior Executable Importer** to import a .elf file.

4. Click **Next**.

   The wizard name changes to **Import a CodeWarrior Executable file**.

The following sections describe the various pages that the wizard displays as it assists you in importing an executable (.elf) file:

- **Import a CodeWarrior Executable file page** on page 105
- **Import C/C++/Assembler Executable Files page** on page 105
- **Processor page** on page 105
- **Linux Application Launch Configurations page** on page 106
- **Debug Target Settings page** on page 107
5.14.1 Import a CodeWarrior Executable file page

The Import a CodeWarrior Executable file page allows you to specify the name and location for your project. The table below describes the options available on this page.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project name</td>
<td>Specify the name of the project. The specified name identifies the project created for debugging (but not building) the executable file.</td>
</tr>
<tr>
<td>Use default location</td>
<td>If you select this option, the project files required to build the program are stored in the current workspace directory of the workbench. If you clear this option, the project files are stored in the directory that you specify in the Location option.</td>
</tr>
<tr>
<td>Location</td>
<td>Specifies the directory that contains the project files. Use the Browse button to navigate to the desired directory. This option is only available when the Use default location option is cleared.</td>
</tr>
</tbody>
</table>

5.14.2 Import C/C++/Assembler Executable Files page

Use the Import C/C++/Assembler Executable Files page to select an executable file or a folder to search for C/C++/assembler executable files.

The table below explains the options available on the page.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>File to import</td>
<td>Specifies the C/C++/assembler executable file. Click Browse to choose an executable file.</td>
</tr>
<tr>
<td>Copy the selected file to current project folder</td>
<td>Select this option to copy the executable file in the project folder.</td>
</tr>
</tbody>
</table>

5.14.3 Processor page

Use the Processor page to specify the processor family for the imported executable file and also specify the toolchain to be used.

The table below describes the options available on the page.
Table 50: Processor page settings

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Expand the processor family and select the appropriate target processor for the execution of the specified executable file. The toolchain uses this choice to generate code that makes use of processor-specific features, such as multiple cores.</td>
</tr>
<tr>
<td></td>
<td><strong>TIP</strong></td>
</tr>
<tr>
<td></td>
<td>You can also type the processor name in the text box.</td>
</tr>
<tr>
<td>Toolchain</td>
<td>Chooses the compiler, linker, and libraries used to build the program. Each toolchain generates code targeted for a specific platform. These are:</td>
</tr>
<tr>
<td></td>
<td>• <strong>Bareboard Application</strong>: Targets a hardware board without an operating system.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Linux Application</strong>: Targets a board running the Linux operating system.</td>
</tr>
<tr>
<td>Target OS</td>
<td>Select if the board runs no operation system or imports a Linux kernel project to be executed on the board. The option is applicable only for bareboard application projects.</td>
</tr>
</tbody>
</table>

5.14.4 Linux Application Launch Configurations page

Use the **Linux Application Launch Configurations page** to specify how the debugger communicates with the host Linux system and controls your Linux application.

The table below lists the options available on the page.

Table 51: Linux Application Launch Configurations page setting

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior TRK</td>
<td>Select to use the CodeWarrior Target Resident Kernel (TRK) protocol, to download and control application on the Linux host system.</td>
</tr>
<tr>
<td>TAP Address</td>
<td>Specifies the IP address of the Linux host system, the project executes on.</td>
</tr>
<tr>
<td>Port</td>
<td>Specifies the port number that the debugger will use to communicate to the Linux host.</td>
</tr>
<tr>
<td>Remote Download Path</td>
<td>Specifies the host directory into which the debugger downloads the application.</td>
</tr>
</tbody>
</table>
5.14.5 Debug Target Settings page

Use the Debug Target Settings page to specify debugger connection type, board type, launch configuration type, and connection type for your project.

This page also allows you to configure connection settings for your project. The table below describes the options available on the page.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debugger Connection Types</td>
<td>Specifies what target the program executes on.</td>
</tr>
<tr>
<td>• Hardware</td>
<td>Select to execute the program on the hardware available for the product.</td>
</tr>
<tr>
<td>• Emulator</td>
<td>Select to execute the program on a hardware emulator.</td>
</tr>
<tr>
<td>Board</td>
<td>Specifies the hardware (board) supported by the selected processor.</td>
</tr>
<tr>
<td>Launch</td>
<td>Specifies the launch configurations and corresponding connection configurations, supported by the selected processor.</td>
</tr>
<tr>
<td>Connection Type</td>
<td>Specifies the interface to communicate with the hardware.</td>
</tr>
<tr>
<td>• CodeWarrior USB TAP</td>
<td>Select to use the CodeWarrior USB TAP interface to communicate with the hardware device.</td>
</tr>
<tr>
<td>• CodeWarrior Ethernet TAP</td>
<td>Select to use the CodeWarrior Ethernet TAP interface to communicate with the hardware device.</td>
</tr>
<tr>
<td>TAP address</td>
<td>Enter the IP address of the selected TAP device.</td>
</tr>
</tbody>
</table>

NOTE
The Debug Target Settings page may prompt you to either create a new remote system configuration or select an existing one.

A remote system is a system configuration that defines connection, initialization, and target parameters. The remote system explorer provides data models and frameworks to configure and manage remote systems, their connections, and their services.

5.14.6 Configurations page

Use the Configurations page to select the processor core that executes the project.

The table below lists the options available on the page.

<table>
<thead>
<tr>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core Index</td>
<td>Select the processor core that executes the project.</td>
</tr>
</tbody>
</table>
5.15 Debugging Externally Built Executable Files

You can use the Import a CodeWarrior Executable file wizard to debug an executable (.elf) file that has no associated CodeWarrior project.

For example, you can debug a .elf file that was generated using a different IDE. The process of debugging an externally built executable file can be divided into the following tasks:

- Import Executable File on page 108
- Edit Launch Configuration on page 110
- Specify Source Lookup Path on page 110
- Debug Executable File on page 112

5.15.1 Import Executable File

First of all, you need to import the executable file that you want the CodeWarrior IDE to debug.

The IDE imports the executable file into a new project. To import an externally built executable file, follow these steps:

1. From the CodeWarrior IDE menu bar, select File > Import.
   
   The Import wizard appears.
2. Expand the CodeWarrior group.
4. Click Next.
   
   The wizard name changes to Import a CodeWarrior Executable file and the Import a CodeWarrior Executable file page appears.
5. In the Project name text box, enter the name of the project. This name identifies the project that the IDE creates for debugging (but not building) the executable file.
6. Clear the Use default location checkbox and click Browse to specify a different location for the new project.
   
   By default, the Use default location checkbox is selected.
7. Click Next.
   
   The Import C/C++/Assembler Executable Files page appears.
8. Click Browse.
   
   The Select file dialog appears. Use the dialog to navigate to the executable file that you want to debug.
9. Select the required file and click Open.
   
   The Select file dialog closes. The path to the executable file appears in the File to import text box.

   **TIP**

   You can also drag and drop a .elf file in the CodeWarrior Eclipse IDE. When you drop the .elf file in the IDE, the Import a CodeWarrior Executable file wizard appears with the .elf file already specified in the Project Name and File to Import text box.

10. Select the Copy the selected file to current project folder checkbox to copy the executable file in the current workspace.
11. Click Next.
The Processor page appears.

12. Select the processor family for the executable file.

13. Select a toolchain from the Toolchain group.

Selected toolchain sets up the default compiler, linker, and libraries used to build the new project. Each toolchain generates code targeted for a specific platform.

14. Select if the board runs no operation system or imports a Linux kernel project to be executed on the board. The Target OS options are applicable only for bareboard application projects.

15. Click Next.

The Debug Target Settings page appears.

16. Select a supported connection type, from the Debugger Connection Types group. Your selection determines the launch configurations that you can include in your project.

17. Select the hardware or simulator, you plan to use, from the Board drop-down list.

**NOTE**

Hardware or Simulators that supports the target processor selected on the Processors page are only available for selection.

18. Select the launch configurations that you want to include in your project and the corresponding connection.

19. Select the interface to communicate with the hardware, from the Connection Type drop-down list.

20. Enter the IP address of the TAP device in the TAP address text box. This option is disabled and cannot be edited, if you select USB TAP from the Connection Type drop-down list.

21. Click Next.

The Configurations page appears.

22. Select the processor core that executes the project, from the Core index list.

23. Click Finish.

The Import a CodeWarrior Executable file wizard ends. The project for the imported .elf file appears in the CodeWarrior Projects view. You can now open the Debug Configurations dialog by selecting Run > Debug Configurations. The Debug Configurations dialog shows the current settings for the launch configuration that you just created. A remote system is created with details of all the connection, initialization, and target parameters you had set while importing the .elf file.
5.15.2 Edit Launch Configuration

Using the tabs of the **Debug Configurations** dialog box, you can change the launch configuration settings that you specified while importing the `.elf` file.

To edit the launch configuration for your executable file, follow these steps:

1. On the **Main** tab, click **Edit** in the **Connection** panel.

   The corresponding **Connection** page appears.

2. Use the **Connection type** list box to modify the current connection type.

3. Configure the various connection options as appropriate for your executable file by using the various tabs available on the **Connection** page.

   For example, specify the appropriate target processor, any initialization files, and connection protocol.

4. Click **OK** to close the **Connection** page.

5.15.3 Specify Source Lookup Path

Source lookup path is specified in terms of the compilation path and the local file system path.

The CodeWarrior debugger uses both these paths to debug the executable file. The compilation path is the path to the original project that built the executable file. If the original project is from an IDE on a different computer, you need to specify the compilation path in terms of the file system on that computer.

The local file system path is the path to the project that the CodeWarrior IDE creates to debug the executable file.
To specify a source lookup path for your executable file, perform the following steps:

1. Click the **Source** tab of the **Debug Configurations** dialog.  
   The corresponding page appears.

2. Click **Add**.  
   The **Add Source** dialog appears.

3. Select **Path Mapping** from the available list of sources.

   **Figure 50:**  Add Source dialog

4. Click **OK**.  
   The **Add Source** dialog closes. The **Path Mappings** dialog appears.

5. In the **Name** text box, enter the name of the new path mapping.

6. Click **Add**.  
   The cursor blinks in the **Compilation path** column.

7. In the **Compilation path** column, enter the path to the parent project of the executable file, relative to the computer that generated the file.

   Suppose the computer on which you debug the executable file is not the same computer that generated that executable file. On the computer that generated the executable file, the path to the parent project is `D:\workspace\originalproject`. Enter this path in the **Compilation path** text box.
You can use the IDE to discover the path to the parent project of the executable file, relative to the computer that generated the file. In the C/C++ Projects view of the C/C++ perspective, expand the project that contains the executable file that you want to debug. Next, expand the group that has the name of the executable file itself. A list of paths appears, relative to the computer that generated the file. Search this list for the names of source files used to build the executable file. The path to the parent project of one of these source files is the path you should enter in the Compilation path column.

8. In the Local file system path text box, enter the path to the parent project of the executable file, relative to your computer. Click the ellipsis button to specify the parent project.

Suppose the computer on which you debug the executable file is not the same computer that generated that executable file. On your current computer, the path to the parent project of the executable file is `C:\projects\thisproject`. Enter this path in the Local file system path text box.

9. Click OK.

The Path Mapping dialog closes. The mapping information now appears under the path mapping shown in the Source Lookup Path list of the Source page.

10. If needed, change the order in which the IDE searches the paths.

The IDE searches the paths in the order shown in the Source Lookup Path list, stopping at the first match. To change this order, select a path, then click Up or Down to change its position in the list.

11. Click Apply.

The IDE saves your changes.

5.15.4 Debug Executable File

You can use the CodeWarrior debugger to debug the externally built executable file.

To debug the executable file:
1. Select the project in the CodeWarrior Projects view.
2. Click the Debug button from the IDE toolbar.

The IDE switches to Debug perspective listing the debugging output.
Chapter 6
Multicore Debugging

This chapter explains how to use the multicore debugging capability of the CodeWarrior debugger.

In this chapter:

• Debugging Multicore projects on page 113
• Multicore debugging commands on page 119

6.1 Debugging Multicore projects

This section explains how to set launch configurations and how to debug multiple cores in a multi-core project.

The CodeWarrior debugger provides the facility to debug multiple ARMv7 processors using a single debug environment. The run control operations can be operated independently or synchronously. A common debug kernel facilitates multicore, run control debug operations for examining and debugging the interaction of the software running on the different cores on the system.

To debug a multicore project, perform the steps given in the following sections:

• Setting launch configurations on page 113
• Debugging multiple cores on page 116

6.1.1 Setting launch configurations

Setting a launch configuration allows you to specify all core-specific initializations.

To set up the launch configurations, follow these steps:

1. Open the CodeWarrior project you want to debug.
2. Switch to the Debug perspective.
3. Select Run > Debug Configurations.

The Debug Configurations dialog appears (shown in the figure below) with a list of debug configurations that apply to the current application.

4. Expand the CodeWarrior tree control.
5. From the expanded list, select the debug configuration for which you want to modify the debugger settings. For example, FirstProject-Test-core0_RAM_LS1021AQDS_Download
6. On the **Main** tab, select a connection from the **Connection** drop-down list.

7. Select a core from the **Target** list.

8. Click **Edit** next to the **Connection** drop-down list.

   The **Properties for <connection>** dialog appears.
9. Select a target from the Target drop-down list.

10. Select the required TAP connection from the Connection type drop-down list. For example, CodeWarrior TAP.

11. On the Connection tab, specify the hostname/IP of the target board in the Hostname/IP text box.

12. Enter the JTAG clock speed in the JTAG clock speed text box.

13. Specify the port number of the CCS server in the Server port number text box.

14. Click OK.

15. Click the Debugger tab in the Debug Configurations dialog.

   The Debugger page appears.

16. Ensure that the Stop on startup at checkbox is selected and main is specified in the User specified text box.

17. Click Apply to save the changes.

   You have successfully configured a debug configuration.

18. Similarly, configure remaining debug configurations.
To successfully debug multiple cores, the connection settings must be identical for all debug configurations.

### 6.1.2 Debugging multiple cores

The CodeWarrior debugger enables system developers to simultaneously develop and debug applications on a system with multiple processors, within the same debug environment.

**NOTE**

Ensure that you have attached a debug probe to the target board and to the computer hosting the CodeWarrior IDE before performing the steps listed in this section.

To debug multiple cores, follow these steps:

1. Select a multicore project in the **CodeWarrior Projects** view.
2. Select **Run > Debug**.

   The debugger downloads core 0 and switches to the **Debug** perspective. The debugger halts execution at the first statement of `main()`. The **Debug** view displays all the threads associated with the core.

   ![Figure 53: Multicore Debugging - Debug Core 0](image)

3. Download all other cores associated with the project.
4. Select a thread from core 1 in the **Debug** view.

   All the views in the **Debug** perspective will be updated to display the debug session for the selected core. The figure below displays the debug session for a selected thread in core 1.
5. Select and expand the **Core Registers** group.

6. Select **Run > Step Over**.

   The following actions occur:
   - Debugger executes the current statement and halts at the next statement.
   - The program counter (PC) indicator moves to the next executable source line in the **Source** view.
   - In the **Debug** view, the status of the program changes to (Suspended).
   - Modified register values are highlighted in yellow.

7. Select a thread from core 1 in the **Debug** view.

   All the views in the **Debug** perspective will be updated to display the debug session for the selected core.

8. Select and expand the **Core Registers** group.

9. Select **Run > Step Over**.

   The following actions occur:
   - Debugger executes the current statement and halts at the next statement.
   - The program counter (PC) indicator moves to the next executable source line in the **Source** view.
10. Issue several more **Step Over** commands and watch the register values change.

11. Select `main()` thread from core 0 again.
   
   Notice that the register values remain unchanged. This is because the CodeWarrior debugger controls each core’s execution individually.

12. With core 0 still selected, click the **Step Over** button several times until you reach the `printf()` statement.
   
   Debugger executes the current statement, the following statements, and halts at the `printf()` statement.

13. Switch to the other debug window.

14. Select the `main()` thread for core 1 by clicking it. Notice that the program counter icon in the **Source** view did not move. The debugger controls the execution of each core individually.

15. In the **Debug** view, click the **Resume** button.
   
   Core 1 enters an infinite loop. The status of the program changes to (**Running**).

16. In the **Debug** view, click the `main()` thread for core 0 and click the **Resume** button.
   
   Core 0 enters an infinite loop and core 1 continues to execute in its loop.

17. Select `main()` thread from core 1 and click the **Suspend** button.
   
   The debugger halts core 1 at the current statement and the status of the program changes to (**Halted**). Core 0 continues to execute.

18. Select **Run > Multicore Terminate**.
   
   The debugger terminates the active debug session. The threads associated with each core in the **Debug** view disappear.

   Similarly, you can collect multicore trace data. In the **Debug** view, click **Multicore Resume**; the execution begins and data measurement starts. Wait for some time. When output starts displaying for all the cores, click **Multicore Terminate**. Trace viewer will show trace of all the cores as shown below.
6.2 Multicore debugging commands

This section describes the multi-core commands available in the Run menu of CodeWarrior IDE and in the Debugger Shell.

If you are debugging a multicore project, you can use single and multicore debugging commands to debug parts of each core project.

This section contains the following subsections:

- Multicore commands in CodeWarrior IDE on page 119
- Multicore commands in Debugger Shell on page 120

6.2.1 Multicore commands in CodeWarrior IDE

When you start a multicore debug session, multicore commands are enabled on the CodeWarrior IDE Run menu. These commands, when issued, affect all cores simultaneously.

The table below describes each menu choice.
### Table 54: Multi-core debugging commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Icon</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multicore Resume</td>
<td>![Icon]</td>
<td>Starts all cores of a multicore system running simultaneously.</td>
</tr>
<tr>
<td>Multicore Suspend</td>
<td>![Icon]</td>
<td>Stops execution of all cores of a multicore system simultaneously.</td>
</tr>
<tr>
<td>Multicore Restart</td>
<td>![Icon]</td>
<td>Restarts all the debug sessions for all cores of a multicore system simultaneously.</td>
</tr>
<tr>
<td>Multicore Terminate</td>
<td>![Icon]</td>
<td>Kills all the debug sessions for all cores of a multicore system simultaneously.</td>
</tr>
</tbody>
</table>
| Multicore Groups   | ![Icon] | **Use All Cores:** If the selected debug context is a multicore system, then all cores are used for multicore operations.  
**Disable Halt Groups:** Disables breakpoint halt groups.  
**Limit new breakpoints to current group:** If selected, all new breakpoints set during a debug session are reproduced only on cores belonging to the group of the core on which the breakpoint is set.  
**Edit Target Types:** Opens Target Types dialog that lets you add and remove system types.  
**Edit Multicore Groups:** Opens the Multicore Groups dialog to create multicore groups. You can also use this option to modify the existing multicore groups. |

To use the multi-core commands from the **Debug** perspective, follow these steps:

1. Start a debugging session by selecting the appropriately configured launch configuration.
2. If necessary, expand the desired core's list of active threads by clicking on the tree control in the **Debug** view.
3. Click the thread you want to use with multicore operations.
4. From the **Run** menu, specify the multicore operation to perform on the thread.

**NOTE**

The keyboard shortcut for the **Multicore Resume** operation is Alt+Shift+F8.

### 6.2.2 Multicore commands in Debugger Shell

In addition to the multicore-specific toolbar buttons and menu commands available in the **Debug** view, the **Debugger Shell** has multicore specific commands that can control the operation of one or more processor cores at the same time.

Similar to the menu commands, the multicore debugger shell commands allow you to select, start, and stop a specific core. You can also restart or kill sessions executing on a particular core. The table below lists and defines the affect of each multicore debugging command.
<table>
<thead>
<tr>
<th>Command</th>
<th>Shortcut</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mc::config</td>
<td>mc::c</td>
<td>List or edit multicore group options.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Syntax</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>mc::config</td>
</tr>
<tr>
<td>mc::go</td>
<td>mc::g</td>
<td>Resume multiple cores</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Syntax</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>mc::go</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Examples</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>mc::go</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Resumes the selected cores associated with the current thread context.</td>
</tr>
<tr>
<td>mc::group</td>
<td>mc::gr</td>
<td>Display or edit multicore groups</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Syntax</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>group group new &lt;type-name&gt; [ &lt;name&gt; ] group rename &lt;name&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Examples</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>mc::group</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Shows the defined groups, including indices for use in the set of commands.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mc::group new 8572</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Creates a new group for system type 8572. The group name will be based on the system name and will be unique. The enablement of the group elements will be all non-cores enabled, all cores disabled.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mc::group rename 0 &quot;My Group Name&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Renames the group at index 0 to &quot;My Group Name&quot;.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mc::group enable 0 0.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Enables the group at index 0 and the element at index 0.0 of the command.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mc::group remove &quot;My Group Name&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Removes the group named &quot;My Group Name&quot;.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mc::group removeall</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Removes all groups.</td>
</tr>
</tbody>
</table>

Table continues on the next page...
### Table 55: Multi-Core debugging commands (continued)

<table>
<thead>
<tr>
<th>Command</th>
<th>Shortcut</th>
<th>Description</th>
</tr>
</thead>
</table>
| mc::kill  | mc::kill | Terminates the debug session for selected cores associated with the current thread context.  
**Syntax**  
mc::kill  
**Examples**  
mc::kill  
Terminates multiple cores. |
| mc::reset | mc::reset | Resets multiple cores.  
**Syntax**  
mc::reset |
| mc::restart | mc::restart | Restarts the debug session for selected cores associated with the current thread context.  
**Syntax**  
mc::restart  
**Examples**  
mc::restart  
Restarts multiple cores. |
| mc::stop  | mc::stop | Stops the selected cores associated with the current thread context.  
**Syntax**  
mc::stop  
**Examples**  
mc::stop  
Suspends multiple cores. |

*Table continues on the next page...*
<table>
<thead>
<tr>
<th>Command</th>
<th>Shortcut</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mc::type</td>
<td>mc::t</td>
<td>Shows the system types available for multicore debugging as well as type indices for use by the mc::type remove and mc::group new commands.</td>
</tr>
</tbody>
</table>

**Syntax**

```
type type import <filename> type remove <filename> | <type-index> ... type removeall
```

**Examples**

- `mc::type` Display or edit system types.
- `mc::type import 8572_jtag.txt` Creates a new type from the JTAG configuration file.
- `mc::type remove 8572_jtag.txt` Removes the type imported from the specified file.
- `mc::type removeall` Removes all imported types.
Chapter 7
Debugging Embedded Linux Software

This chapter explains how to use the CodeWarrior Development Studio tools to debug embedded Linux® software for ARMv7 processors.

This chapter includes the following sections:

- Debugging Linux application on page 125
- Viewing multiple processes and threads on page 134
- Debugging applications that use fork() and exec() System Calls on page 135
- Debugging shared library on page 145
- Debugging Linux Kernel on page 153
- Debugging loadable kernel modules on page 165

7.1 Debugging Linux application

This section describes CodeWarrior Target-Resident Kernel (TRK) and provides information related to using it with CodeWarrior projects.

For embedded Linux development, CodeWarrior TRK is a user-level application that resides on target embedded Linux systems and accepts connections from the CodeWarrior debugger. You use the CodeWarrior remote connections feature to download and debug applications built with CodeWarrior projects. The CodeWarrior debugger connects to CodeWarrior TRK on the remote target system through a serial or ethernet connection.

On embedded Linux systems, CodeWarrior TRK is packaged as a regular Linux application, named apptrak. This application runs on the remote target system along side the program you are debugging to provide application-level debug services to the CodeWarrior debugger.

To debug a Linux application using CodeWarrior TRK:

- Install CodeWarrior TRK on target system on page 125
- Start CodeWarrior TRK on target system on page 126
- Create CodeWarrior download launch configuration for Linux application on page 128
- Specify Console I/O redirections for Linux application on page 131
- Configure Linux Process Signal Policy on page 132
- Debug Linux application on page 134

7.1.1 Install CodeWarrior TRK on target system

This section explains how to install CodeWarrior TRK on target system.

To connect the CodeWarrior debugger to CodeWarrior TRK, the CodeWarrior TRK binary executable file must be installed and running on the remote target system. Once CodeWarrior TRK is running on the target system, the debugger can upload your application and debug the application on the target system.
If CodeWarrior TRK is not present on a given target system, you need to use a file transfer facility, such as Secure Copy (SCP) or File Transfer Protocol (FTP) to download the CodeWarrior TRK binary executable file, AppTRK.elf, to a suitable location on the file system of the target system. You also need to place the unstripped versions of the ld.so, libpthread.so, and libthread_db.so files in the /lib directory of the target system to debug shared library code or multi-threaded code with CodeWarrior TRK.

### 7.1.2 Start CodeWarrior TRK on target system

This section explains how to start CodeWarrior TRK on target system.

How you start CodeWarrior TRK on the target hardware depends on the type of connection between the host computer and that target hardware:

- **Transmission Control Protocol/Internet Protocol (TCP/IP):** The host computer communicates with the target hardware over a TCP/IP connection
- **Serial cable:** A serial cable connecting the host computer to the target hardware

CodeWarrior TRK can be started as either a root user or a normal user; however, if the application to be debugged requires root permission, then you need to start CodeWarrior TRK as a root user. In other words, CodeWarrior TRK must have all the privileges required by the application that it will debug. You also need to ensure that the download directory specified in the Remote tab of the launch configuration matches the user privileges of the CodeWarrior TRK running on the target system.

This section contains the following subsections:

- **TCP/IP connections** on page 126
- **Serial connections** on page 127

### 7.1.2.1 TCP/IP connections

This section explains how to start CodeWarrior TRK through a TCP/IP connection.

The steps are as follows:

1. **Connect to the remote target system.**
   a. On the host computer, open a new terminal window.
   b. At the command prompt in the terminal window, enter the following command, where IPAddress represents the target system's IP address:

   ```
telnet IPAddress
   ```
   
   The telnet client connects to the telnet daemon on the target system.

2. **Navigate to the directory that contains the AppTRK.elf binary executable file.**
   The system changes the current working directory.

3. **Type the following command (where Port is the listening port number optionally specified in the Connections panel of Debug window- typically 1000):**

   ```
./AppTRK.elf :Port
   ```
   
   CodeWarrior TRK starts on the target system, and listens to the specified TCP/IP port for connections from the CodeWarrior IDE.
To continue use of the terminal session after launching CodeWarrior TRK, start CodeWarrior TRK as a background process by appending the ampersand symbol (&) to the launch command. For example, to start CodeWarrior TRK as a background process listening to TCP/IP port number 6969, you would enter the following command:

```
./AppTRK.elf:6969 &
```

### 7.1.2.2 Serial connections

This section explains how to launch CodeWarrior TRK through a serial connection.

The steps are as follows:

1. Connect a serial cable between the host computer's serial port and the second serial port of the target system.
2. On the host computer, start a terminal emulation program (such as minicom).
3. Configure the terminal emulation program with baud rate, stop bit, parity, and handshake settings appropriate for the target system.
4. Connect the terminal emulator to the target system.
   - A command prompt appears in the terminal emulation program.
5. Boot the system. Log in as the root user.
6. Use the `cd` command at the command prompt to navigate to the directory where the CodeWarrior TRK binary executable file, `AppTRK.elf`, resides on the target system.
   - The system changes the current working directory.
7. Configure the serial port on which CodeWarrior TRK is going to connect.
   a. Enter this command: `stty -F /dev/ttyS1 raw`
      - This command configures the serial port for raw mode of operation. If you do not use raw mode, special characters sent as part of packets may be interpreted (dropped), causing the connection to break.
   b. Enter this command: `stty -F /dev/ttyS1 ispeed 115200`
      - The serial input speed is set to 115200 baud.
   c. Enter this command: `stty -F /dev/ttyS1 ospeed 115200`
      - The serial output speed is set to 115200 baud.
   d. Enter this command: `stty -F /dev/ttyS1 crtscts`
      - The terminal emulation program enables handshake mode.
   e. Enter this command: `stty -a -F /dev/ttyS1`
      - The system displays the current device settings.
8. Enter the command: 

   ```
   ./AppTRK.elf /dev/ttyS1
   ```

   CodeWarrior TRK launches on the remote target system.

7.1.3 Create CodeWarrior download launch configuration for Linux application

This section explains how to create a CodeWarrior download launch configuration for debugging a Linux application on target system.

Follow these steps to create a CodeWarrior download launch configuration:

1. In the CodeWarrior Projects view of the C/C++ perspective, select the name of the project that builds the Linux application.

2. Select Run > Debug Configurations.

   The Debug Configurations dialog appears.


4. Click the New launch configuration toolbar button of the Debug Configurations dialog.

   The IDE creates a new launch configuration under the CodeWarrior group. The settings pages for this new launch configuration appear on the right-hand side of the Debug Configurations dialog.

5. In the Main tab of the Debug Configuration dialog:

   a. Select Download from the Debug session type group.

   b. Click New next to the Connection drop-down list.

   The New Connection wizard appears.

6. Expand the CodeWarrior Application Debugging group and select Linux AppTRK, as shown in the figure below.
7. Click **Next**.

   The **Linux AppTRK** page appears.

8. Specify the connection name, description, template and connection type on this page.

9. When you select the connection type, the corresponding **Connection** tab appears (shown in the figure below).
10. Specify the settings as appropriate for the connection between the host computer and the target hardware on this page.

11. Click **Finish**.

The new remote system that you just created appears in the **Connection** drop-down list.

12. Click the **Debugger** tab.

The **Debugger options** panel appears with the respective tabs.

13. On the **Debug** tab, if required, specify a function or address in the application where you want the program control to stop first in the debug session:
   
   a. Select the **Stop on startup at** checkbox.
      
      The IDE enables the corresponding text box.
   
   b. Enter in the text box an address or a function inside the application.

14. Click the **Remote** tab.

The corresponding sub-page comes forward.

15. Enter in the **Remote download path** text box the path of a target-computer directory to which the Linux application, running on the target hardware, can read and write files.

**NOTE**

The specified directory must exist on the target system.
16. If required, specify information about other executable files to debug in addition to the Linux application:
   a. Click the Other Executables tab.
      The corresponding sub-page appears.
   b. Use the sub-page settings to specify information about each executable file.
17. Click Apply.
    The IDE saves the pending changes you made to the launch configuration.

You just finished creating a CodeWarrior download launch configuration that you can use to debug the Linux application.

7.1.4 Specify Console I/O redirections for Linux application

CodeWarrior TRK allows you to specify I/O redirections as arguments for applications to be debugged.
This feature allows users to use a file on the target or even the target console for file descriptors, including stdin, stdout, and stderr. By default, the CodeWarrior TRK running on the target forwards the output of the application to the host CodeWarrior. The host CodeWarrior will be able to print the received output only if the Allocate Console (necessary for input) checkbox is selected in the Common tab of the Debug Configurations dialog.

NOTE
The CodeWarrior console, allocated for the debugged application, can only be used to view the output of the application running on the target; forwarding the input from a CodeWarrior console to the debugged application is not supported currently for Linux applications.

The listing below displays the syntax to specify I/O redirections for the stdin, stdout, and stderr file descriptors.

```
- '< <filename>' - stdin redirection from <filename>
- '> <filename>' - stdout redirection to <filename>
- '2> <filename>' - stderr redirection to <filename>
```

Figure 58: Specifying I/O redirections

To specify I/O redirections for a Linux application:

1. In the CodeWarrior Projects view of the C/C++ perspective, select the name of the project that builds the Linux application.
2. Select Run > Debug Configurations.
   The Debug Configurations dialog appears.
3. Expand CodeWarrior group and select the launch configuration associated with the project.
   The settings pages for the selected launch configuration appears on the right-hand side of the Debug Configurations dialog.
4. Click the Arguments tab.
5. Specify the I/O redirections in the Program arguments text box.
6. Click Apply to save the changes.

The listing below displays an example of redirections, added to the list of arguments, to forward the output to the console where CodeWarrior TRK was started.
7.1.5 Configure Linux Process Signal Policy

This section explains how to control applications being debugged using signals and how to manage signals, using CodeWarrior IDE.

AppTRK and CodeWarrior can be configured to stop the application being debugged, whenever the application receives a signal. A user can send signals to the application directly from CodeWarrior, when the application resumes execution. To send a signal to an application, right-click the signal name in the Signals view and select Resume With Signal from the context menu that appears.

This section contains the following subsections:

- Signal inheritance on page 132
- Default Signal Policy on page 132
- Modifying Signal Policy on page 132

7.1.5.1 Signal inheritance

When a new process is forked, it inherits the signal settings from the parent process.

For example, if a process has a setting that if the SIGUSR1 signal is received, the application being debugged will be stopped, then a child process forked by this process will also inherit this setting. It will stop the application being debugged if the SIGUSR1 signal is received.

All the threads created by a process share the signal settings of that process. Signal settings cannot be configured at thread level.

7.1.5.2 Default Signal Policy

By default, the SIGINT, SIGILL, SIGTRAP, SIGSTOP, and SIGSEGV signals are caught by the debugger.

The debugger stops the application being debugged if any of these signals is received.

7.1.5.3 Modifying Signal Policy

CodeWarrior IDE provides a view, Signals, which can be used to view signals and change the debugger’s policy for a signal.

To open the Signals view, perform the following steps:

1. Select Window > Show View > Other in the CodeWarrior IDE.
   
   The Show View dialog appears.

2. Select Debug > Signals.

   The Signals view appears, as shown in the figure below.

---

**Figure 59: Sample I/O redirections**

- `< /proc/self/fd/0` --> use target console for stdin (this way, stdin is functional and can be used - using a CW console it isn't)
- `>/proc/self/fd/1` --> use target console for stdout
- `'/proc/self/fd/2` --> use target console for stderr
To send a signal to a stopped process or thread, right-click the signal in the **Signals** view and select **Resume With Signal**, as shown in the figure below.

**Figure 61: Sending Signal to Process or Thread**

To catch a signal, perform the following steps:

1. Right-click the signal in the **Signals** view and select **Signal Properties**.
   - The **Properties for** window appears (shown in the figure below).
2. Select the **Suspend the program when this signal happens** checkbox, as shown in the figure below.

**Figure 62: Catching Signal**

The figure below shows a child process stopped on receiving the SIGUSR1 signal.

**Figure 63: Stopped Child Process**

Some signals cannot be caught, but they can be passed to the debugged application. These signals have read-only properties. One such signal is SIGKILL.
7.1.6 **Debug Linux application**

You can use the CodeWarrior download launch configuration created earlier to debug the Linux application on the target system.

You can use that launch configuration to start a debug session that can be used to debug a Linux application.

The steps are as follows:

1. On the left-hand side of the **Debug Configurations** dialog, ensure to select the CodeWarrior download launch configuration that you created to debug the Linux application.

2. Click **Debug** in the **Debug Configurations** dialog.

   The IDE uses the selected CodeWarrior download launch configuration to start a debugging session and opens the **Debug** view, as shown in the figure below.

   ![Debug view - Sample Linux application](image)

You just finished using the CodeWarrior download launch configuration to debug a Linux application.

### 7.2 Viewing multiple processes and threads

This section explains how to view all processes and threads on a target.

When you debug an application, the CodeWarrior debugger opens the **Debug** view. In this view, you can see only processes and threads/tasks on which debugger is attached, as shown in the figure below.
For Linux debugging, you can view all processes on target in the System Browser view.

To view processes and threads in System Browser view:
1. Open a Linux application in the CodeWarrior IDE.
2. Select Run > Debug.
   The Debug perspective appears.
3. While the application is running, select Window > Show View > Other.
   The Show View dialog appears.
4. From the Debug group, select System Browser.
5. Click OK.
   The System Browser window appears with the process and thread information (shown in the figure below). Keeping this view open increases the time required to suspend a core.

Figure 66: System Browser window

7.3 Debugging applications that use fork() and exec() System Calls

This section shows you how to use the CodeWarrior debugger to debug programs that contain fork() and exec() system calls.

The table below describes the fork() and exec() system calls.
Table 56: fork() and exec() description

<table>
<thead>
<tr>
<th>System Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork()</td>
<td>This generic Linux system call creates a new process that is the exact replica of the process that creates it. This call returns 0 to the child process and returns the PID (Process ID) of the newly-created child process to the parent process.</td>
</tr>
<tr>
<td>exec()</td>
<td>This Linux system call launches a new executable in an already running process. The debugger destroys the instance of the previous executable loaded into that address space and a new instance is created.</td>
</tr>
</tbody>
</table>

NOTE
You can also pick up sample Linux applications from the following folder:

`<CWInstallDir>CW_ARMv7\ARMv7\ARM_Linux\AppDebug\Examples`

For CodeWarrior debugging purposes, when applications call the fork() system call, the debugger instead calls the clone() system call with the flag CLONE_PTRACE. This causes:

• The operating system to attach CodeWarrior TRK to the child process
• The child process to stop with a SIGTRAP on return from the clone() system call

To make this happen, you must add a static library to your CodeWarrior project. The source code for building the static library is described later in this section.

Before you start the tutorial, ensure that you have:

• Installed the BSP on Linux
• Created a TCP/IP connection between the host computer and the remote target
• Launched CodeWarrior TRK on the target system

These steps demonstrate how to use the CodeWarrior IDE to debug programs that contain fork() and exec() system calls:

1. Create a CodeWarrior project with the settings listed in the table below.

Table 57: Static Library Project Settings

<table>
<thead>
<tr>
<th>Option Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project name</td>
<td>Fork</td>
</tr>
<tr>
<td>Location</td>
<td><code>&lt;workspace-dir&gt;Fork</code></td>
</tr>
<tr>
<td>Project type</td>
<td>Linux application</td>
</tr>
<tr>
<td>Language</td>
<td>C</td>
</tr>
</tbody>
</table>

The IDE creates a project with a debug launch configuration.

2. Create a new build configuration. Right-click the project folder and select Build Configurations > Manage.

   The Fork: Manage Configurations dialog appears.

3. Rename the default debug configuration to Fork.

4. Click New to create a new build configuration.

   The Create New Configuration dialog appears.
5. In the **Name** field, enter the configuration name, **Fork2Clone**.

6. From the **Copy settings from** options, select **Existing configuration**.

7. Click **OK**. The **Fork: Manage Configurations** dialog appears (shown in the figure below).

```
Figure 67:  Fork: Manage Configurations dialog
```

8. **Activate** the **Fork2Clone** build configuration.

9. **Build** the **Fork2Clone** build configuration by right-clicking it in the **CodeWarrior Projects** view and selecting **Build Project** from the context menu. The CodeWarrior IDE builds the project and stores the support library, **libfork2clone.a**, in the **Output directory** within the project directory.

   **NOTE**

   Remember to build the **Fork2Clone** build configuration before the **Fork** build configuration to avoid getting a library file missing error as the **libfork2clone.a** is used in the Fork project.

10. To specify the linker settings and add the support library to the project.
   a. Right-click the **Fork** build configuration in the **CodeWarrior Projects** view.
   b. Select **Properties** from the context menu. The **Properties** window for the shared library project appears.
   c. From the **C/C++ Build** group, select **Settings**.
   d. On the **Tool Settings** page, from the **ARM Sourcery Windows GCC Linker** container, select **Libraries**.
   e. In the **Libraries (-l)** panel, click **Add** ( ). The **Enter Value** dialog appears.
   f. Enter the library file name in the **Libraries** field.
   g. In the **Libraries search path (-L)** panel, click **Add** ( ). The **Add directory path** dialog appears.
   h. Enter the library path in the **Directory** field, as shown in the figure below.

   **NOTE**

   These settings enable the CodeWarrior IDE linker to locate the shared library **libfork2clone.a**. For detailed information on other linker command line arguments, refer to GNU linker manuals. You can find GNU documentation here: `http://www.gnu.org`.
11. Remove the default main.c file from the project.
12. Add a new db_fork.c file to the project.
13. Enter the below code in the editor window of db_fork.c file.

```
/*
   User Include files

*/
#include "db_fork.h"

/*
   Main Program

*/
```
int __libc_fork(void)
{
    return( __db_fork() );
}

extern __typeof (__libc_fork) __fork __attribute__((weak, alias ("__libc_fork")));
extern __typeof (__libc_fork) fork __attribute__((weak, alias ("__libc_fork")));

14. Create a header file db_fork.h in your project directory and add the below code in the header file.

Figure 70: Source code for db_fork.h

#include <asm/unistd.h>
#include <sys/syscall.h>
#include <errno.h>
#include <signal.h>
#include <sched.h>
#define __NR___db_clone__NR_clone
#define __db_fork()
syscall(__NR___db_clone, SIGCHLD | CLONE_PTRACE, 0);

15. Enter the below code in the editor window of fork.c file.

Figure 71: Source code for fork.c

/*
 */
* fork.c
*
*/
/*---------------------------------------------*/
System Include files
*---------------------------------------------*/
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/ptrace.h>
#include <sys/syscall.h>
#include <sys/errno.h>
#include <sys/types.h>
#include <signal.h>
#include <sched.h>
#include <fcntl.h>
#include <dlfcn.h>

/*-------------------------------------------------------------------*
 Function Prototypes
 * -----------------------------------------------------------------*/
int fn1(int j);
int fn2(int i);

/*------------------------------------------------------------------*
 Global Variables
 *------------------------------------------------------------------*/
int gint;

/*------------------------------------------------------------------*
 Main Program
 *------------------------------------------------------------------*/
int main(void)
{
  int pid,x;
  int shared_local;
  printf( "Fork Testing!\r\n" );
  fflush( stdout );
  gint = 5;
  shared_local =5;
  pid = fork();
  if(pid == 0)
  {
    x=0;
    gint = 10;
shared_local = fn1(9);
printf("\nForked : Child");
printf("\nChild:Global=%d,Shared_Local=%d",gint,shared_local);
printf("\nChild pid = %d, parent pid =%d \n", getpid(),getppid());
fflush( stdout );
}
else
{
x=0;
gint = 12;
shared_local = fn2(11);
printf("\nForked : Parent");
printf("\nParent:Global=%d,Shared_Local=%d",gint,shared_local);

printf("\nParent pid = %d, Parent's parent pid =%d \n", getpid(),getppid());
fflush( stdout );
}
return 0;
}
int fn1(int j)
{
    j++;
    return j;
}

int fn2(int i)
{
    i++;
    return i;
}
The code of the parent process creates a forked process (child process) when the \_db\_fork function executes. The debugger opens a separate thread window for the child process. When the child process finishes executing, the debugger closes the thread window. To debug the code of the child process, you need to set a breakpoint in the child process code. You can debug the code of the child process the same way you debug code of any other process.

16. Create another project, Exec, and create two new build configurations with the following settings:

<table>
<thead>
<tr>
<th>Option Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project name</td>
<td>Exec</td>
</tr>
<tr>
<td>Location</td>
<td>&lt;workspace-dir&gt;\Exec</td>
</tr>
<tr>
<td>Project type</td>
<td>Linux application</td>
</tr>
<tr>
<td>Language</td>
<td>C</td>
</tr>
</tbody>
</table>
| Build configurations | * Exec  
|                   | * Exec-1                      |

17. Add the source files exec.c and exec-1.c to the Exec project.

- exec.c: The code demonstrating exec() functionality
- exec-1.c: Generates the executable file exec-1.elf

As you step through the code of the exec.elf file, the exec() function call executes and a separate debugger window for the exec-1.elf appears. You can perform normal debug operations in this window. The debugger destroys the instance of the previous file (exec.elf) and creates a new instance for the exec-1.elf file.

18. Enter the below code in the editor window of Exec.c file.

```c
/**  Exec.c
 *   Demonstrates Exec system call functionality
 */

/* System Include files */
#include <stdio.h>
#include <unistd.h>
/* Constant Definitions */
#define EXEC_1  
"/tmp/Exec-1.elf"

int main(void) {
```
char *argv[2];
printf("Exec Testing!\r\n" );
printf("Before exec my ID is %d\n", getpid());
printf("My parent process's ID is %d\n", getppid());
fflush( stdout );

/**Calling another program exec-1.elf*/
argv[0] = EXEC_1;
argv[1] = NULL;
printf("exec starts\n");
execv(argv[0], argv);
printf("This will not print\n");
fflush( stdout );
return 0;
}

19. Enter the below code in the editor window of Exec-1.c file.

** Figure 73: Source code for Exec-1.c **

```c
/**  Exec-1.c *
Demonstrates Exec system call functionality */
/*----------------------------------------*/
System Include files
*----------------------------------------*/
#include <stdio.h>
#include <unistd.h>
/*----------------------------------------*/
Main Program
*----------------------------------------*/
int main(void){
    printf("After exec my process ID is %d\n", getpid());
    printf("My parent process's ID is %d\n", getppid());
    printf("exec ends\n");
    fflush( stdout );
    return 0;
}
```

20. Create the build configurations for building Exec.elf and Exec-1.elf (similar to creating the build configurations for the Fork project).

   a. Select the Exec build configuration, if not selected.
   b. Select Project > Build Project.

      The CodeWarrior IDE generates the exec.elf, and exec-1.elf executable files and places them in the project folder.

22. Specify the remote download path of the executable files to be launched by the exec() system call.
   a. Select Run > Debug Configurations to open the Debug Configurations dialog.
   b. In the left panel from the CodeWarrior group, select the Exec launch configuration.
   c. On the Debugger page, click the Remote tab.
   d. Type /tmp in the Remote Download Path field, as shown in the figure below. This specifies that the final executable file will be downloaded to this location on the target platform for debugging.
For this tutorial, the remote download path is specified as /tmp. If you wish, you may specify an alternate remote download path for the executable file.

**NOTE**

Figure 74: Remote download path - shared library project

---

23. Specify the host-side location of the executable files to be launched by the `exec()` system call.
   a. Click the **Other Executables** tab.
   b. Click **Add**. The **Debug Other Executable** dialog appears.
   c. Click **File System**. The **Open** dialog appears.
   d. Navigate to the location of the `exec-1.elf` file in your project directory.
   e. Select the `exec-1.elf` file name.
   f. Click **Open**. The host-side location of `exec-1.elf` appears in the **Additional Executable File** text box.
   g. Select the **Load Symbols** checkbox.
   h. Select the **Download to Device** checkbox. The **Specify the remote download path** field is activated.
   i. Type `/tmp` in the **Remote download path** text box. The shared library will be downloaded to this location when you debug or run the executable file.
   j. Click **OK**. The settings are saved.

24. Click **Apply** to save the settings made to the launch configuration.

25. Set breakpoints in the child and parent processes.
   a. Double-click the `fork.c` file name in the **CodeWarrior Projects** view.
   b. Set a breakpoint in the code of the child process at this line: `x=0;`
   c. Set a breakpoint in the code of the parent process.
   d. Close the `fork.c` file.

26. Select **Run > Debug**.

The debugger window appears and the **Fork** project starts debugging. As a result, the `Fork.elf` and `libfork2clone.a` files are downloaded on the target system.

27. Step over the code until you reach the line of code that calls the `fork()` system call: `pid = fork ();`

When the `fork()` system call is called, the child process debugger window appears. You can now perform normal debugging operations in this window.
28. Step over the code in the child process debugger window a couple of times.
29. Next, step over the code in the parent process debugger window a couple of times.

NOTE
The console window of the parent process is shared by the child process.

30. Terminate the debug session.
31. Clear previously set breakpoints.
32. Select Run > Debug for Exec project.
33. Set a breakpoint in the Exec.c file on the line containing the execv() function call.
34. Click Resume. The target stops at the line where you set the breakpoint.
35. Click Resume. The exec() call is executed and the debugger stops in the main() function of the Exec-1.elf file.
36. Execute some steps in Exec-1.c file.
37. Terminate the debug session and remove all breakpoints.

7.4 Debugging shared library

This section demonstrates how to debug a shared library that is implicitly linked to an application.

CodeWarrior allows you to do source-level debugging of shared libraries. When you debug an executable file using a shared library, you can step into the shared library code.

This tutorial demonstrates how to debug a shared library that is implicitly linked to an application. In this tutorial:

- Create an example project on page 145
- Configure shared library build configuration on page 148
- Configure executable build configuration on page 148
- Build shared library on page 149
- Build executable on page 149
- Configure launch configuration on page 149
- Debug shared library on page 151

7.4.1 Create an example project

First of all, you need to create an example Linux project that uses a shared library.

To create an example Linux project, perform the following steps:

1. In the CodeWarrior IDE, use File > New > CodeWarrior Linux Project Wizard to create a new Linux project with the settings given in the table below.
Table 59: Example Project Settings

<table>
<thead>
<tr>
<th>Option Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project name</td>
<td>SharedLibraryExample</td>
</tr>
<tr>
<td>Location</td>
<td>&lt;workspace-dir&gt;\SharedLibraryExample</td>
</tr>
<tr>
<td>Project type</td>
<td>Linux application</td>
</tr>
<tr>
<td>Language</td>
<td>C</td>
</tr>
</tbody>
</table>
| Build configurations | • LibExample (generates the dynamic library needed by the launch configurations)  
                      | • SharedLib.IM (used to demonstrate implicit linking with the library generated by LibExample build configuration) |
| Launch configurations | SharedLib.IM (launches the application that demonstrates implicit linking with a shared library) |

**NOTE**
In this tutorial, we only mention implicit library linking; however, in the example project shipped with CodeWarrior, SharedLibrary, we have also demonstrated explicit library loading. For explicit library loading, we have used another build/launch configuration, SharedLib_EX.

2. Remove the default main.c file and add the source files (SharedLib_IM.c and LibExample.c) to your project.

3. In the CodeWarrior IDE, create a header file, LibExample.h, as depicted in the listing below.

Figure 75: Source Code for LibExample.h

/* LibExample.h */
int add_example(int x, int y);
int add_example_local(int x, int y);

4. Save the LibExample.h file in the project directory.

5. Enter the below code into the editor window of the SharedLib_IM.c file.

Figure 76: Source Code for SharedLib_IM.c

/*
   Sharedlib_IM.c

   Demonstrates implicit linking.
*/

// User Include files
#include "LibExample.h"

// function prototype declaration
int temp (int, int);
// main program
int main ()
{
  int ret;
  int a, b;
  a = 10;
  b = 20;
  ret = temp (a, b);
  ret = add_example (a, b);   // step in here
  return ret;
}
int temp(int i, int j)
{
  return i + j;
}

6. Enter the below code into the editor window of the LibExample.c file.

Figure 77: Source Code for LibExample.c

/*
 LibExample.c
 */

// user include files#include "LibExample.h"// functions definitions
int add_example( int x, int y)
{
  int p,q;
  p=100;
  q=p+200;
  add_example_local (2, 3);  // step in here
  return x+y+q;
}

int add_example_local (int x, int y)
7.4.2 Configure shared library build configuration

The next action is to configure the LibExample build configuration, which generates libexample.so.

The steps are given below:

1. Select the SharedLibraryExample project in the CodeWarrior Projects view.
2. Select the LibExample build configuration by selecting Project > Build Configurations > Set Active > <Build Configuration Name>.
3. Check LibExample.c and LibExample.h in the Build column.

TIP

Use the CodeWarrior example project, SharedLibrary, as a reference to set up the build configuration settings of the LibExample build configuration.

7.4.3 Configure executable build configuration

Now, you need to set up the SharedLib_IM build configuration.

For this, perform the following steps:

1. Select the SharedLibraryExample project in the CodeWarrior Projects view.
2. Select the SharedLib_IM build configuration by selecting Project > Build Configurations > Set Active > <Build Configuration Name>.
3. Specify the linker settings.
   a. Select the SharedLib_IM build configuration in the CodeWarrior Projects view.
   b. Select Project > Properties. The Properties window for the shared library project appears.
   c. In the Tool settings page, from the ARM Sourcery Windows GCC C Linker container, select Libraries.
   d. In the Libraries (-l) panel, click Add ( ). The Enter Value dialog appears.
   e. Enter the library file name, example, in the Libraries field.
   f. In the Library search path (-L) panel, click Add ( ). The Add directory path dialog appears.
   g. Enter the library path in the Directory field. The library path is the path of the Output directory that is used by LibExample build configuration.
7.4.4 Build shared library
The next action is to build the shared library.

To build the shared library, perform the following steps:
1. Select the SharedLibraryExample project in the CodeWarrior Projects view.
2. Select the LibExample build configuration by selecting Project > Build Configurations > Set Active > <Build Configuration Name>.
3. Select Project > Build Project. The CodeWarrior IDE builds the project and stores the output file libexample.so in the Output directory within the project directory.

7.4.5 Build executable
Now, you need to build the executable that uses the shared library.

To build the executable that uses the shared library, perform the following steps:
1. Select the SharedLibraryExample project in the CodeWarrior Projects view.
2. Select the SharedLib_IM build configuration by selecting Project > Build Configurations > Set Active > <Build Configuration Name>.
3. Select Project > Build Project. The CodeWarrior IDE builds the project and stores the output file SharedLib_IM.elf in the Output directory within the project directory.

7.4.6 Configure launch configuration
The next action is to configure the SharedLib_IM launch configuration.

Configuring the SharedLib_IM launch configuration involves:
• Specifying the remote download path of the final executable file
• Specifying the host-side location of the executable file to be used for debugging the shared library
• Specifying the environment variable that enables the shared object loader to locate the shared library on the remote target at run time

The steps are as follows:
1. Activate the SharedLib_IM launch configuration in the project.
2. Specify the remote download path of the final executable file.
   a. Select Run > Debug Configurations to open the Debug Configurations dialog.
   b. In the left pane from the CodeWarrior group, select the SharedLib_IM launch configuration.
   c. On the Debugger page, click the Remote tab.
   d. Type /tmp in the Remote Download Path field, as shown in the figure below. This specifies that the final executable file will be downloaded to this location on the target platform for debugging.
For this tutorial, the remote download path is specified as /tmp. You can replace /tmp with any other directory for which CodeWarrior TRK has the necessary access permissions.

NOTE

3. Specify the host-side location of the executable file to be used for debugging the shared library.
   a. Click the Other Executables tab in the Debugger page.
   b. Click Add. The Debug Other Executable dialog appears.
   c. Click Workspace. The Open dialog appears.
   d. Navigate to the location where you have stored the libexample.so file in your project directory.
   e. Select the libexample.so file name.
   f. Click Open. The host-side location of the shared library appears in the Specify the location of the other executable field.
   g. Select the Load Symbols checkbox, so that the debugger has visibility of symbols within the library.
   h. Select the Download to Device checkbox. The Specify the remote download path field is activated.
   i. Type /tmp in the Remote download path text box. The shared library will be downloaded to this location when you debug or run the executable file.

   The default location of shared libraries on the embedded Linux operating system is /usr/lib. For this tutorial, the remote download location of libexample.so is /tmp.
   j. Click OK. The settings (shown in the figure below) are saved.

Figure 78: Remote download path - shared library project

Figure 79: Debug Other Executable dialog
4. Specify the environment variable that enables the shared object loader to locate the shared library on the remote target at run time.

At run time, the shared object loader first searches for a shared library in the path specified by the `LD_LIBRARY_PATH` environment variable's value. In this case, the value of this environment variable will be `/tmp`, which is the remote download path for the shared library you specified in the Debug Other Executable dialog. If you have not specified the environment variable or have assigned an incorrect value, the shared object loader searches for the shared library in the default location `/usr/lib`.

a. In the Debug window, click Environment to open the Environment page.

b. Click New to open the New Environment Variable dialog.

c. In the Name field, type `LD_LIBRARY_PATH`.

d. In the Value field, type `/tmp`.

NOTE
Ensure that you type the same remote download path in the Value field that you specified in the Debug Other Executable dialog.

e. Click OK. The environment variable is added to the launch configuration.

f. Add another environment variable with name, `AVOID_SYSTEM_PATH` and value `YES`.

NOTE
The `AVOID_SYSTEM_PATH` variable sets the launch configuration to use the library path settings you specify. By specifying the value `YES` you avoid the launch configuration from picking up any other system path.

g. Click Apply to save the launch configuration settings. The target settings are saved (shown in the figure below).

![Figure 80: Environment variables - shared library project](image)

h. Click OK to close the Debug view.

7.4.7 Debug shared library

Finally, you need to debug the shared library.

In the steps that follow, you will launch the debugger. Next, you will step through the code of the executable file `SharedLib_IM.elf` until you reach the code that makes a call to the `add_example` function implemented in the shared library. At this point, you will step into the code of the `add_example` function to debug it.

1. Activate the `SharedLib_IM` launch configuration in the project.

2. Select Run > Debug.
The debugger starts and downloads the `SharedLib_IM.elf` and `libexample.so` files to the specified location on the remote target, one after another. The Debug perspective appears.

3. Click Step Over in the debugger window until you reach the following line of code (shown in the figure below):

   ```c
   ret=add_example(a,b)
   ```

   **TIP**

   Before you set breakpoints in the code of an imported shared library to step into the code, you can use the Executables view to navigate and check the source files of the library. For more information on the Executables view, open CodeWarrior Eclipse Help by selecting Help > Help Contents in the CodeWarrior IDE, and then select Third Party References > C/C++ Development User Guide > Reference > C/C++ Views and Editors > Executables view in the Contents pane.

   ![Figure 81: SharedLib_IM.c - step in location](image)

4. In the Debug view, click Step Into to step into the code of the `add_example` function.

   The debugger steps into the source code of the `add_example` function in the `LibExample.c` file (shown in the figure below).

   ![Figure 82: LibExample.c - add_example function](image)

5. After stepping in, you can step through the rest of the code.
The **Debug** view shows the function calls to the `add_example` function (shown in the figure below).

**Figure 83: Debug view - shared library project**

![Debug view - shared library project](image)

6. View the output of the program.

The rest of the code is executed and the output appears in the **Variables** view (shown in the figure below).

**Figure 84: Variables view - shared library project**

![Variables view - shared library project](image)

### 7.5 Debugging Linux Kernel

This section shows you how to use the CodeWarrior debugger to debug the Linux kernel.

The Linux operating system (OS) works in two modes - **kernel mode** and **user mode**. The Linux kernel operates in kernel mode and resides at the top level of the OS memory space, or **kernel space**. The kernel performs the function of a mediator among all the currently running programs and between the programs and the hardware. The kernel manages the memory for all the programs (processes) currently running and ensures that the processes share the available memory such that each process has enough memory to function adequately. In addition, the kernel allows application programs to manipulate various hardware architectures via a common software interface.

User mode uses the memory in the lowest level of the OS memory space, called the **user space** or the application level. At the application level, a program accesses memory or other hardware through system calls to the kernel as it does not have permission to directly access these resources.

Debugging the Linux kernel involves the following major actions:

1. **Setting up target hardware** on page 154
2. **Installing Board Support Package (BSP)** on page 155
3. Configuring build tool on page 156
4. Configuring Linux kernel on page 156
5. Creating a CodeWarrior Project using Linux kernel image on page 158
6. Configuring kernel project for debugging on page 160
7. Debugging kernel by attaching to a running U-Boot on page 163

7.5.1 Setting up target hardware
Before you use the CodeWarrior IDE to debug the Linux kernel, you need to set up the target hardware.

One requirement of the setup is to have a debug probe connected between the CodeWarrior debug host and target board.

The figure below illustrates the setup required to use the IDE to debug the Linux kernel running on a ARMv7 target board.

![Setup for kernel debugging using CodeWarrior IDE](image)

Connect the hardware debug probe between the target board and the CodeWarrior debug host. Kernel debugging is possible using a Linux-hosted or Windows-hosted CodeWarrior installation. There are a variety of debug probes. The kernel debugging tutorial uses the USB Tap. Connection information for other debug probes can be determined from documentation provided with the probes.

This section contains the following subsection:

- Establishing console connection on page 154

7.5.1.1 Establishing console connection
This section explains how to establish a console connection to the target hardware.

You need to establish a console connection to the target hardware to:

- View target generated log and debug messages
- Confirm successful installation of the bootloader (U-Boot)
• Use the bootloader to boot the Linux OS
• Halt the booting of the Linux OS

The bootloader receives keyboard input through a serial port that has default settings 115,200-8-N-1.

Follow these steps to establish a console connection to the target hardware.

1. Connect a serial cable from a serial port of the CodeWarrior debug host to a serial port of the target board.
2. On the CodeWarrior debug host computer, open a terminal-emulator program of your choice (for example, minicom for a Linux host).
3. From the terminal-emulator program, open a console connection to the target hardware.

Use the connection settings given in the table below.

<table>
<thead>
<tr>
<th>Name</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baud rate</td>
<td>115, 200 bits per second</td>
</tr>
<tr>
<td>Data bits</td>
<td>8</td>
</tr>
<tr>
<td>Parity</td>
<td>None</td>
</tr>
<tr>
<td>Stop bits</td>
<td>1</td>
</tr>
<tr>
<td>Flow control</td>
<td>Hardware</td>
</tr>
</tbody>
</table>

**Table 60: Terminal connection settings**

**NOTE**
Refer to the board specific README file inside the stationery wizard project to find out more details on the serial connection settings, changing the serial port on the board, and the type of serial cable to use.

4. Test the connection by turning on the test board with the power switch and viewing the boot messages in the console connection.

### 7.5.2 Installing Board Support Package (BSP)

This section describes how to install a BSP on a Linux computer.

**NOTE**
The BSP versions keep changing frequently. For different BSP versions, you might encounter build environments based on ltib, bitbake, or other tools. The subsequent sections will describe necessary procedures and use specific examples from real NXP BSPs for illustration. The examples in these sections will need to be adapted based on the BSP versions or build tools you are currently using.

1. On the Linux computer, download the Board Support Package (BSP) for your target hardware to install kernel files and Linux compiler toolchains on your system.

   Board Support Package image files for target boards are located at [Linux Board Support Packages](#).

2. Download the BSP image file for your target board.

   **NOTE**
   You will need to log in or register to download the BSP image file.

   The downloaded image file has an `.iso` extension.
For example,

`LS1021A-SDK-<yyyymmdd>-yocto.iso`

3. Mount the image file to the CDROM as root, or using "sudo":

```
<sudo> mount -o loop LS1021A-SDK-<yyyymmdd>-yocto.iso /mnt/cdrom
```

**NOTE**

`sudo` is a Linux utility that allows users to run applications as root. You need to be setup to run `sudo` commands by your system administrator to mount the BSP image files.

4. Execute the BSP install file to install the build tool files to a directory of your choice, where you have privileges to write files:

```
/mnt/cdrom/install
```

**NOTE**

The BSP must be installed as a non-root user, otherwise the install will exit.

5. Answer the questions from the installation program until the file copy process begins.

You will be prompted to input the required build tool install path. Ensure you have the correct permissions for the install path.

6. Upon successful installation, you will be prompted to install the ISO for the core(s) you want to build.

For example, if you want to build the SDK for ARMv7, that is a Cortex-A7 core, then you have to install the ISO images for Cortex-A7 core:

**NOTE**

You can refer to the SDK User Manual for instructions about how to build the BSP images and run different scenarios from the `iso/help/documents/pdf` location.

### 7.5.3 Configuring build tool

After installing the BSP, you need to configure the build tool and build the Linux kernel and U-boot images for CodeWarrior debug.

For more information on configuring the build tool, refer to the SDK User Manual from `iso/help/documents/pdf`.

### 7.5.4 Configuring Linux kernel

After you complete the BSP configuration, configure the Linux kernel to enable CodeWarrior support.

Follow these steps:

1. Launch a terminal window and navigate to the `<yocto_installtion_path>/build_<board>_release` folder.

2. Execute the following command to get a new and clean kernel tree:

   `bitbake -c configure -f virtual/kernel`
3. Configure the Linux kernel using the various configuration options available in the kernel configuration user interface. For example, run the following command to display the kernel configuration user interface:

    bitbake -c menuconfig virtual/kernel

The kernel configuration user interface appears.

4. CodeWarrior supports both SMP and non-SMP debug. To change the default settings, you can make changes by selecting the Processor support options.

5. To run a monolithic kernel, you do not need to enable loadable module support. However, during the debug phase of drivers, it is easier to debug them as loadable modules to avoid rebuilding the Linux kernel on every debug iteration. If you intend to use loadable modules, select the Loadable module support menu item.

6. Select the Enable loadable module support option.

7. Select the Module unloading option.

   NOTE
   
   If you want to use the `rmmod -f <mod_name>` command for kernel modules under development, select the Forced module unloading option.

8. Select Exit to return to the main configuration menu.


10. Select Compile-time checks and compiler options.

11. Select Compile the kernel with debug info by pressing Y. Enabling this option allows the CodeWarrior debugger to debug the target.

12. Select other desired configuration options for Linux kernel debug.

13. Select Exit to return to the main configuration menu.

14. Select the General Setup option.

15. Select Configure standard kernel features (expert users) and ensure that the Sysctl syscall support option is selected.

16. If you are using the Open Source Device Tree debugging method, under the General Setup > Configure standard kernel features (expert users) option, then select:

   • Load all symbols for debugging/ksymoops.
   • Include all symbols in kallsyms.

   NOTE
   
   These settings are optional. They aid the debugging process by providing the vmlinux symbols in `/proc/kallsyms`.

17. Select Exit to exit the configuration screen.

18. Select Yes when asked if you want to save your configuration.

19. Execute the following command to rebuild the Linux kernel:

    bitbake virtual/kernel

The uncompressed Linux kernel image with debug symbols, `vmlinux`, is created.
You just created a Linux kernel image that contains symbolic debugging information. Now, you can use this image and create a CodeWarrior project for debugging the Linux kernel. The various use cases for the Linux kernel debug scenario are:

- **CodeWarrior** allows you to download this Linux kernel image (vmlinux.elf), ramdisk, and dtb files to the target
- You can start the Linux kernel and ramdisk manually from U-Boot. The U-Boot, the kernel, ramdisk, and dtb images are written into flash memory.
- You can perform an early kernel debug before the mmu is enabled or debug after the Linux kernel boots and the login prompt is shown.

The Linux kernel debug scenarios are explained in the following sections:

- **Creating a CodeWarrior Project using Linux kernel image** on page 158
- **Configuring kernel project for debugging** on page 160

### 7.5.5 Creating a CodeWarrior Project using Linux kernel image

After creating a Linux kernel image with symbolic debugging information, you need to create a CodeWarrior project using the kernel image.

To create a CodeWarrior project, follow these steps:

1. Start the CodeWarrior IDE from the Windows system.
2. Select File > Import. The Import wizard appears.
3. Expand the CodeWarrior group and select CodeWarrior Executable Importer.
4. Click Next.
   - The Import a CodeWarrior executable file page appears.
5. Specify a name for the project, to be imported, in the Project name text box.
6. If you do not want to create your project in the default workspace:
   a. Clear the Use default location checkbox.
   b. Click Browse and select the desired location from the Browse For Folder dialog.
   c. In the Location text box, append the location with the name of the directory in which you want to create your project.

   **NOTE**

    An existing directory cannot be specified for the project location.

7. Click Next.
   - The Import C/C++/Assembler Executable Files page appears.
8. Click Browse next to the Executable field.
9. Select the vmlinux file obtained.
10. Click Open.
11. From the Processor list, expand the processor family and select the required processor.
12. Select the **Bareboard Application** toolchain from the **Toolchain** group.
   Selected toolchain sets up the default compiler, linker, and libraries used to build the new project. Each toolchain generates code targeted for a specific platform.

13. Select the **Linux Kernel** option from the **Target OS** list.

   **NOTE**
   Selecting Linux Kernel will automatically configure the initialization file for kernel download, the default translation settings (these settings need to be adjusted according to the actual Linux kernel configuration) in the OS Awareness tab, and the startup stop function to `start_kernel`.

14. Click **Next**.
   The **Debug Target Settings** page appears.

15. From the **Debugger Connection Types** list, select the required connection type.

16. Specify the settings, such as board configuration, launch configuration, connection type, and TAP address if you are using CodeWarrior Ethernet or USB TAP.

17. Click **Next**.
   The **Configurations** page appears.

18. From the **Core index** list, select the required core.

19. Click **Finish**.
   The wizard creates a project according to your specifications.

You can access the project from the **CodeWarrior Projects** view on the workbench.

This section contains the following subsection:

- Updating Linux kernel image on page 159

### 7.5.5.1 Updating Linux kernel image
By modifying the Linux kernel image, you can update the project you just created.

There are two use cases for this scenario:

- **Cache symbolics between sessions is enabled** on page 159
- **Cache symbolics between sessions is disabled** on page 160

#### 7.5.5.1.1 Cache symbolics between sessions is enabled
This section provides steps to replace the current `vmlinux.elf` file with the new `vmlinux.elf` file when the cache symbolics between sessions is enabled.

Follow these steps:

1. Terminate the current debug session.
2. Right-click in the Debug window.
3. From the context menu, select **Purge Symbolics Cache**. The old `vmlinux.elf` file is being used by the debugger, but after you select this option, the debugger stops using this file in the disk.
4. Copy the new `vmlinux.elf` file over the old file.
   Now, when you re-initiate a debug session, the updated `vmlinux.elf` file is used for the current debug session.
7.5.5.1.2 Cache symbolics between sessions is disabled
This section provides steps to replace the current `vmlinux.elf` file with the new `vmlinux.elf` file when the cache symbolics between sessions is disabled.

Follow these steps:
1. Terminate the current debug session.
2. Copy the new `vmlinux.elf` file over the old file.

Now, when you re-initiate a debug session, the updated `vmlinux.elf` file is used for the current debug session.

7.5.6 Configuring kernel project for debugging
After you have created a CodeWarrior project using the Linux kernel image, the next action is to configure this project for debugging.

This section contains the following subsection:
- Configuring attach kernel debug scenario on page 160

7.5.6.1 Configuring attach kernel debug scenario
This section describes how to configure an attach debug scenario.

For the attach scenario, CodeWarrior does not download any file on the target. The kernel is started directly from U-Boot. You need to burn the U-Boot image to the flash memory of the hardware.

After the boot process, the U-Boot console is available and the Linux kernel can be started manually from U-Boot. For this, the following files can be either written into flash memory or can be copied from U-Boot using TFTP.

- Binary kernel image file, `uImage`
- Ramdisk to be started from U-Boot, for example, `<target version>.rootfs.ext2.gz.u-boot`
- `dtb` file, for example, `uImage-<target version>.dtb`

After the Linux boot process, the Linux login appears and you can connect to debug the kernel using the CodeWarrior Attach launch configuration. As all the files are manually loaded from U-Boot, these files must not be specified in the launch configuration.

The table below describes the settings you need to provide in the launch configuration.

To specify the launch configuration settings in CodeWarrior:

1. Select Run > Debug Configurations.
2. Enter the launch configuration settings, given in the table below, in the Debug Configurations dialog.
Table 61: Kernel Project Attach Launch Configuration Settings

<table>
<thead>
<tr>
<th>Debug Window Component</th>
<th>Settings</th>
</tr>
</thead>
</table>
| **Main Tab** | Select an appropriate system (if existing) from the **Connection** drop-down list or define a new system.  
  - To define a new system, click **New**.  
  - Select Hardware or Simulator Connection from the CodeWarrior Bareboard Debugging list. Click **Next**.  
  - Specify a name and a description for the connection.  
  - Select an appropriate target (if existing) from the **Target** drop-down list or define a new target.  
  - To define a new target, click **New** on the **Hardware or Simulator Connection** dialog.  
  - Select **Hardware or Simulator Target** from the CodeWarrior Bareboard Debugging list. Click **Next**.  
  - Specify a name and a description for the target.  
  - Select a target from the **Target type** drop-down list. On the **Initialization** tab, ensure there are no initialization files selected.  
  - Click **Finish** to create the target and close the **Hardware or Simulator Target** dialog.  
  - Select the type of connection you will use from the **Connection type** drop-down list.  
  - Click **Finish**.  
  - Select all the cores on which Linux is running (for example, core 0 for single-core or cores 0-7 for 8-core SMP). |
| **Debugger Tab > Debugger options > Symbolics Tab** | Select the **Cache Symbolics between sessions** checkbox. The symbolics are loaded from the elf file to the debugger for the first session only. This shows a speed improvement for **vmlinux.elf** as the size is bigger than around 100 MB. |
| **Debugger Tab > Debugger options > OS Awareness Tab** | Select **Linux** from the **Target OS** drop-down list. |
| **Debugger Tab > Debugger options > OS Awareness Tab > Boot Parameters** | Disable all settings on the **Boot Parameters** tab. |

*Table continues on the next page...*
Table 61: Kernel Project Attach Launch Configuration Settings (continued)

<table>
<thead>
<tr>
<th>Debug Window Component</th>
<th>Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debugger Tab &gt; Debugger options &gt; OS</td>
<td>Debug tab</td>
</tr>
<tr>
<td>Awareness Tab &gt; Debug Tab</td>
<td>• Select the <strong>Enable Memory Translation</strong> checkbox</td>
</tr>
<tr>
<td></td>
<td><strong>Physical Base Address</strong> is set to value CONFIG_KERNEL_START (0x80000000)</td>
</tr>
<tr>
<td></td>
<td><strong>Virtual Base Address</strong> is set to value CONFIG_KERNEL_START (0x80000000 for 32-bits)</td>
</tr>
<tr>
<td></td>
<td>• <strong>Memory Size</strong> is the kernel space translation size.</td>
</tr>
<tr>
<td></td>
<td><strong>NOTE</strong></td>
</tr>
<tr>
<td></td>
<td>The values shown above should be set as configured in the Linux configuration file (.config). You can read the MMU registers to verify what you have configured and do a correction, if required.</td>
</tr>
<tr>
<td></td>
<td>• Select <strong>Enable Threaded Debugging Support</strong> checkbox</td>
</tr>
<tr>
<td></td>
<td>• Select <strong>Enable Delayed Software Breakpoint Support</strong></td>
</tr>
<tr>
<td></td>
<td>• If required, also select <strong>Update Background Threads on Stop</strong>.</td>
</tr>
<tr>
<td></td>
<td>When enabled, the debugger reads the entire thread list when the target is suspended. This decreases the speed. If the option is disabled, the speed is increased but the Debug window might show non-existent threads, as the list is not refreshed.</td>
</tr>
</tbody>
</table>

3. Click the **Source** page to specify path mappings. Path mappings are not required if the debug host is similar to the compilation host. If the two hosts are separate, the .elf file contains the paths for the compilation host. Specifying the path mappings helps establish paths from compilation host to where the sources are available to be accessed by the debugger on the debugger host. If no path mapping is specified, when you perform a debug on the specified target, a source file missing message appears (shown in the figure below).
You can specify the path mappings, either by adding a new path mapping on the Source tab or by clicking the appropriate buttons (Locate File, Edit Source Lookup Path) that appear when a source path mapping is not found.

4. Click Apply to save the settings.

5. Click Close.

7.5.7 Debugging kernel by attaching to a running U-Boot

This section explains how to debug the Linux kernel by attaching it to a running U-Boot.

To debug the kernel, perform these steps:

1. Create a project for the Linux kernel image. For more details, See Creating a CodeWarrior Project using Linux kernel image on page 158 topic.

2. Configure the launch configuration for Linux kernel debug. For more details, see Configuring attach kernel debug scenario on page 160 topic.


4. From the left pane, expand the CodeWarrior Attach tree and select the appropriate launch configuration.

5. From the Debugger tab, select the PIC tab.

6. Clear the Alternate Load Address checkbox.

7. Click Apply.

8. Click Debug to start the debug session. The Debug perspective appears.

9. While the U-Boot is running, attach the target.

   The debugger displays a warning, in the console, as the kernel is not being executed on the target.
For multi-core processors, only core0 is targeted in the Debug view. This is normal as the secondary cores are initialized in the Linux kernel after MMU initialization. CodeWarrior will automatically add other cores, in the Debug view, after the kernel initializes the secondary cores.

10. Set software or hardware breakpoints for any stage (before or after MMU initialization).

   Set a HW breakpoint at address 0x80008000. Issue the following command in the Debugger Shell view.

   ```
   bp –hw 0x80008000
   ```

11. Using the U-boot console, load the Linux Kernel, DTB file and ramdisk/rootfs from flash or from TFTP.

12. Debug the kernel.

   The debugger halts execution of the program at whatever breakpoints have been set in the project. Typical stages involved in debugging the kernel are discussed below:

   a. **Debugging the kernel at the entry point**

      The CodeWarrior debugger will stop at the kernel entry point, if any software or hardware breakpoint has been set for entry point.

      NOTE

      For the debugger to stop at the kernel entry point, set a breakpoint before loading the kernel from the U-boot console.

      At the entry point, the MMU is not initialized and therefore debugging before MMU initialization also applies in this stage.

   b. **Debugging the Kernel before the MMU is enabled**

      Being in early debug stage, the user should set the correct PIC value, in order to see the source correspondence, in the Debugger Shell view.

      Before setting a breakpoint for the stage after MMU initialization (for example, breakpoint at `start_kernel`) the correct PIC should be set, in the Debugger Shell view. This is required to ensure that the new breakpoint is set with the correct PIC for the stage after MMU initialization.

      The user can set breakpoints and run/step to navigate, before MMU initialization. The correct PIC should be set in the Debugger Shell view, before the debugger enters the next stage.

   c. **Debugging the Kernel after the MMU is enabled**

      After the MMU is initialized, the PIC value must be reset in the Debugger Shell view. During the Linux Kernel booting, you can debug this stage directly, if no breakpoint has been set for the stage before MMU initialization. Alternatively, you can also debug this stage after run or step from the stage before initialization.

      NOTE

      In case of SMP, all the secondary cores are targeted and displayed in the Debug view.

13. When finished, you can either:

   a. Kill the process by selecting **Run > Terminate**.

   b. Leave the kernel running on the hardware.
7.6 Debugging loadable kernel modules

This section explains how to use the CodeWarrior debugger to debug a loadable kernel module. This section contains the following subsections:

• Loadable Kernel Modules - An Introduction on page 165
• Creating CodeWarrior project for Linux Kernel image on page 165
• Configuring Modules' Symbolics Mapping on page 168

7.6.1 Loadable Kernel Modules - An Introduction

The Linux kernel is a monolithic kernel, that is, it is a single, large program in which all the functional components of the kernel have access to all of its internal data structures and routines.

Alternatively, you may have a micro kernel structure where the functional components of the kernel are broken into pieces with a set communication mechanism between them. This makes adding new components to the kernel using the configuration process very difficult and time consuming. A more reliable and robust way to extend the kernel is to dynamically load and unload the components of the operating system using Linux loadable kernel modules.

A loadable kernel module is a binary file that you can dynamically link to the Linux kernel. You can also unlink and remove a loadable kernel module from the kernel when you no longer need it. Loadable kernel modules are used for device drivers or pseudo-device drivers, such as network drivers and file systems.

When a kernel module is loaded, it becomes a part of the kernel and has the same rights and responsibilities as regular kernel code.

Debugging a loadable kernel module consists of several general actions, performed in the following order:

1. Create a CodeWarrior Linux kernel project for the loadable kernel module to be debugged. See Creating CodeWarrior project for Linux Kernel image on page 165 topic.
2. Add the modules and configure their symbolics mapping. See Configuring Modules' Symbolics Mapping on page 168 topic.

7.6.2 Creating CodeWarrior project for Linux Kernel image

The steps in this section show you how to create a CodeWarrior project from a Linux kernel image that contains symbolic debugging information.

The following procedure assumes that you have made an archive of the Linux kernel image and transferred it to the Windows machine. For kernel modules debugging, ensure that you build the kernel with loadable module support and also make an archive for the rootfs directory, which contains the modules for transferring to Windows.

The steps are as follows:

1. Launch CodeWarrior IDE.
2. Select File > Import. The Import wizard appears.
3. Expand the CodeWarrior group and select CodeWarrior Executable Importer.
4. Click Next.
   The Import a CodeWarrior Executable file page appears.
5. Specify a name for the project, to be imported, in the Project name text box.
6. If you do not want to create your project in the default workspace:
   a. Clear the Use default location checkbox.
   b. Click Browse and select the desired location from the Browse For Folder dialog.
   c. In the Location text box, append the location with the name of the directory in which you want to create your project.

   **NOTE**
   An existing directory cannot be specified for the project location.

7. Click **Next**.
   The Import C/C++/Assembler Executable Files page appears.

8. Click **Browse** next to the Executable field.

9. Select the vmlinux.elf file.

10. Click **Open**.

11. From the Processor list, expand the processor family and select the required processor.

12. Select **Bareboard Application** from the Toolchain group.

13. Select **Linux Kernel** from the Target OS list.

14. Click **Next**.
   The Debug Target Settings page appears.

15. From the Debugger Connection Types list, select the required connection type.

16. Specify the settings, such as board, launch configuration, connection type, and TAP address if you are using Ethernet or USB TAP.

17. Click **Next**.
   The Configuration page appears.

18. From the Core index list, select the required core.

19. Click **Finish**.
   The wizard creates a project according to your specifications. You can access the project from the CodeWarrior Projects view on the Workbench.

20. Configure the launch configuration for Linux kernel debug.
   a. Select Run > **Debug Configurations**.
   The Debug Configurations dialog appears.

21. Enter the launch configuration settings in the **Debug Configurations** dialog. The table below lists the launch configuration settings.
Table 62: Kernel Project Attach Launch Configuration Settings

<table>
<thead>
<tr>
<th>Debug Window Component</th>
<th>Settings</th>
</tr>
</thead>
</table>
| **Main tab**           | Select an appropriate system (if existing) from the **Connection** drop-down list or define a new system.  
  • To define a new system, click **New**.  
  • Select **Hardware or Simulator Connection** from the **CodeWarrior Bareboard Debugging** list. Click **Next**.  
  • Specify a name and a description for the connection.  
  • Select an appropriate target (if existing) from the **Target** drop-down list or define a new target.  
  • To define a new target, click **New** on the **Hardware or Simulator Connection** dialog.  
  • Select **Hardware or Simulator Target** from the **CodeWarrior Bareboard Debugging** list. Click **Next**.  
  • Specify a name and a description for the target.  
  • Select a processor from the **Target type** drop-down list. On the **Initialization** tab, ensure that there are no initialization files selected.  
  • Click **Finish** to create the target and close the **Hardware or Simulator Target** dialog.  
  • Select the type of connection you will use from the **Connection type** drop-down list.  
  • Click **Finish**.  
  • Select all the cores on which Linux is running (for example, core 0 for single-core or cores 0-7 for 8-core SMP). |
| **Debugger tab > Debugger options > Symbolics tab** | Select the **Cache Symbolics between sessions** checkbox. The symbolics are loaded from the elf file to the debugger for the first session only. This shows a speed improvement for vmlinux.elf as the size is bigger than around 100 MB. |
| **Debugger tab > Debugger options > OS Awareness tab** | Select **Linux** from the **Target OS** drop-down list. |
| **Debugger tab > Debugger options > OS Awareness tab > Boot Parameters tab** | All options should be cleared. |

*Table continues on the next page...*
Table 62: Kernel Project Attach Launch Configuration Settings (continued)

<table>
<thead>
<tr>
<th>Debug Window Component</th>
<th>Settings</th>
</tr>
</thead>
</table>
| Debugger tab > Debugger options > OS Awareness tab > Debug tab | • Select the Enable Memory Translation checkbox  
Physical Base Address is set to value CONFIG_KERNEL_START (0x80000000)  
Virtual Base Address is set to value CONFIG_KERNEL_START (0x80000000 for 32-bits).  
• Memory Size is the kernel space translation size.  
NOTE  
The values shown above should be set as configured in the Linux configuration file (.config). |
| Debugger tab > Debugger options > OS Awareness tab > Modules tab | • Select the Detect module loading checkbox  
• Click Add to insert the kernel module file. See Configuring Modules’ Symbolics Mapping on page 168 topic  
• Select the Prompt for symbolics path if not found checkbox |

22. Click the Source page to add source mappings for rootfs and linux-<version>.
23. Click Apply to save the settings.

7.6.3 Configuring Modules' Symbolics Mapping

You can use the Modules tab in the Debug Configurations dialog to add modules to the Linux kernel project and configure the module's symbolics mapping.

The figure below shows the Detect module loading group of the Modules page.
The table below describes the various options available on the **Modules** tab.

**Table 63: Kernel Module Project Launch Configuration - Modules tab settings**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detect module loading</td>
<td>Enables the debugger to detect module load events and insert an eventpoint in the kernel. Disabling this setting delays the module loading. This is useful in scenarios where multiple modules are loaded to the kernel and not all of them need to be debugged. You can enable this setting again in the <strong>Modules</strong> dialog. The dialog is available during the Debug session from the <strong>System Browser View</strong> toolbar &gt; <strong>Module</strong> tab.</td>
</tr>
<tr>
<td>Add</td>
<td>Adds a module name along with the corresponding symbolic path. This option displays a dialog box in the following scenarios:</td>
</tr>
<tr>
<td></td>
<td>• The file that you have selected is not a valid compiled kernel module</td>
</tr>
<tr>
<td></td>
<td>• If the selected module already exists in the list with the same path</td>
</tr>
<tr>
<td>Scan</td>
<td>Automatically searches for module files and populates the kernel module list.</td>
</tr>
<tr>
<td>Remove</td>
<td>Removes the selected items. This button will be enabled only if a row is selected.</td>
</tr>
<tr>
<td>Remove All</td>
<td>Removes all items. This button will be enabled only if the kernel list contains any entries.</td>
</tr>
</tbody>
</table>

*Table continues on the next page*...
## Table 63: Kernel Module Project Launch Configuration - Modules tab settings (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prompt for symbolics path if not found</td>
<td>Prompts to locate the symbolics file if a mapping for it is not available in the settings. A <strong>Browse</strong> dialog appears that allows you to browse for a module file containing symbolics. The debugger will add the specified symbolics to the modules’ symbolics mapping.</td>
</tr>
<tr>
<td>Keep target suspended</td>
<td>Keeps the target suspended after the debugger loads the symbolics file for a module. This option is useful if you want to debug the module’s initialization code. It allows you to set breakpoints in the module’s initialization code before running it.</td>
</tr>
<tr>
<td><strong>NOTE</strong></td>
<td>This option is automatically enabled when activating the <strong>Prompt for symbolics path if not found</strong> option.</td>
</tr>
</tbody>
</table>

**NOTE**

Breakpoints are resolved each time a symbolics file is loaded and the debugger uses the modules unload events for symbolics disposal and breakpoints cleanup.
Chapter 8
JTAG Configuration Files

This chapter explains about JTAG configuration files that pass specific configuration settings to the debugger and support chaining of multiple devices.

A JTAG configuration file is a text file, specific to the CodeWarrior debugger, which describes a custom JTAG scan chain. You can specify the file in the remote system settings.

This chapter explains:
• JTAG configuration file syntax on page 171
• Using a JTAG configuration file to override RCW on page 172
• Using JTAG configuration file to specify multiple linked devices on a JTAG chain on page 173
• Setting up a remote system to use a JTAG configuration file on page 174

8.1 JTAG configuration file syntax

This section describes the syntax of a JTAG configuration file.

You can create a JTAG configuration file that specifies the type, chain order, and various settings for the devices you want to debug. To create the JTAG configuration file, list each device on a separate line, starting with the device that is directly connected to the transmit data out (TDO) signal (Pin 1) of the 16-pin COP/JTAG debug connector on the hardware target, and conclude with a blank line.

The listing below shows the complete syntax for a JTAG configuration file.

Figure 88: JTAG configuration file syntax

```
cfgfile:
  '\n'

  '#' 'any other characters until end of line'

  line

  cfgfile line

line:

  target

  target filter_list_or_params

target:

  target_name

  target_name = target_id

  'Generic' number number number

filter_list_or_params:

  filter_list_entity
```
8.2 Using a JTAG configuration file to override RCW

You can use a JTAG configuration file to override reset configuration word (RCW) for a processor, such as LS1021A, revision 2.0.

In the following scenarios, the JTAG configuration files are used for overriding RCW:

- Programming RCW in a target board that does not have RCW programmed already
- New board bring-up
- Recovering a target board having a blank or corrupted flash: This feature is only available for LS1021A processor revision 2.0 and LS2085A. Board recovery scenarios involving a blank/corrupted flash require an external CodeWarrior TAP probe connection for overriding RCW; RCW override is not supported with a CMSIS-DAP connection.

For more information on RCW, see the reference manual for your processor.

The CodeWarrior software includes example JTAG configuration files that can be used to override the RCW (see the listing below). The JTAG Configuration files are available at the following location:

<CWInstallDir>\CW_ARMv7\ARMv7\ARM_Support\Configuration_Files\jtag_chains

As specified in the listing above, the JTAG configuration files can be used to override a portion of the RCW for LS1021A, by specifying (index, value) pairs for some of the 16 x (32-bit words) of the RCW. For some targets

```
# Example file to allow overriding a portion of the RCW
#
# Syntax:
#   LS102xA (0 RCW_source) (0x1000 RCW_option) (RCWn value) ...
#
#   where:
#   x = Processor version (LS1020/1/2A); default is LS1021A
#   RCW_source = The RCW source that you want to override (for example, 0x9b for hard-coded mode)
#   RCW_option = 0 [RCW override disabled]
#                1 [RCW override enabled]
#   RCWn = 4096+n (n = 0 .. 15; index of RCW value)
#   value = 32-bit value
```

Figure 89: Sample JTAG configuration file for overriding RCW
(for example, LS2085A), you do not need to specify `RCW_source`; therefore, you can remove `(0 RCW_source)` from the JTAG configuration file.

NOTE
You can use the pre-boot loader (PBL) tool to configure the various settings of the RCW and output the RCW in multiple formats, including CodeWarrior JTAG configuration files. For more information on the PBL tool, see QCVS PBL Tool User Guide.

8.3 Using JTAG configuration file to specify multiple linked devices on a JTAG chain

This section explains how to connect multiple processors through a single JTAG chain and how to describe such a JTAG chain in a JTAG configuration file.

The listing and figure below show a sample JTAG initialization file with a single core.

**Figure 90: Sample JTAG initialization file for LS1021A processor**

```plaintext
# A single device in the chain
LS1021A
```

**Figure 91: A single device in a JTAG chain**

![Diagram of a single device in a JTAG chain](image)

The listing below show a sample JTAG initialization file with two devices in a JTAG chain.

**Figure 92: Sample JTAG initialization file for LS1020A and LS1021A processors**

```plaintext
# Two devices in a JTAG chain
LS1020A
LS1021A
```

**NOTE**
The devices are enumerated in the direction starting from TDO output to TDI input.

The listing below show two devices connected in a JTAG chain.

**Figure 93: Sample JTAG initialization file for LS1020A and LS1021A processors**

```plaintext
# Two devices in a JTAG chain
LS1020A (0x80000000 1)
LS1021A (2 1) (210005 0x90404000)
```
The listing below show two devices connected in a JTAG chain with a filter applied for the second device.

Figure 94: Sample JTAG initialization file for two devices with filter for second device

```
# Two devices in a JTAG chain
8306 (1 1) (2 0x44050006) (3 0x00600000)
8309 log
```

In the above example, the entry for the 8306 also includes the Hard Reset Control Word (HRCW) data that will overwrite the HRCW fetched by the 8306 upon power up or Hard Reset. The Hard Reset Control Word parameters are optional.

The CodeWarrior debugger not only supports NXP devices but also supports non-NXP devices in a JTAG scan chain. Each non-NXP device used in a scan chain is declared as "Generic" and it takes the following three parameters:

- JTAG Instruction Length
- Bypass Command
- Bypass Length

The values for these three parameters are available in the device's data sheet or can be obtained from the manufacturer of the device.

The listing below show an NXP device, 8560, connected with a non-NXP device, PLA, in a JTAG scan chain. From the PLA's data sheet, the JTAG Instruction Length = 5, the Bypass Command = 1, and the Bypass Length = 0x1F.

Figure 95: Sample JTAG initialization file including non-NXP devices

```
8560
Generic 5 1 0x1F
```

8.4 Setting up a remote system to use a JTAG configuration file

This section explains how to configure a remote system to use a JTAG configuration file.

To connect to a JTAG chain, specify these settings in the launch configurations:

1. Create a JTAG initialization file that describes the items on the JTAG chain. For more information on how to create a JTAG initialization file, see JTAG configuration file syntax on page 171 and Using JTAG configuration file to specify multiple linked devices on a JTAG chain on page 173.
2. Open the CodeWarrior project you want to debug.
3. Select Run > Debug Configurations.
   The Debug Configurations dialog appears with a list of debug configurations that apply to the current application.
4. Expand the CodeWarrior tree control.
5. From the expanded list, select the debug configuration for which you want to modify the debugger settings.
   The Debug view shows the settings for the selected configuration.
6. Select a remote system from the Connection drop-down list.
7. Select a core from the Target list.
8. In the **Connection** group, click **Edit**.
   The **Properties for <project>** window appears.

9. Click **Edit** next to the **Target** list.
   The **Properties for <remote system>** window appears.

10. Click **Edit** next to the **Target type** drop-down list.
    The **Target Types** dialog appears.

11. Click **Import**.

12. The **Import Target Type** dialog appears.

13. Select the JTAG initialization file that describes the items on the JTAG chain from this location:

   ![CWInstallDir]\\CW_ARMv7\\ARMv7\\ARM_Support\\Configuration_Files\\jtag_chains

14. Click **OK**.
   The items on the JTAG chain described in the file appear in the **Target Types** dialog.

15. Click **OK**.
   The selected JTAG configuration file appears on the **Advanced** tab.

16. Click **OK**.

17. Click the **Debugger** tab.
    The **Debugger** page appears.

18. Ensure that the **Stop on startup at** checkbox is selected and **main** is specified in the **User specified** text box.

19. Click **Apply** to save the changes.
    You have successfully configured a debug configuration.
JTAG Configuration Files

Setting up a remote system to use a JTAG configuration file
Chapter 9
Target Initialization Files

A target initialization file is a file that contains commands that initialize registers, memory locations, and other components on a target board.

The most common use case is to have the CodeWarrior debugger execute a target initialization file immediately before the debugger downloads a bareboard binary to a target board. The commands in a target initialization file put a board in the state required to debug a bareboard program.

NOTE
The target board can be initialized either by the debugger (by using an initialization file), or by an external bootloader or OS (U-Boot, Linux). In both cases, the extra use of an initialization file is necessary for debugger-specific settings (for example, silicon workarounds needed for the debug features).

This chapter includes the following section:
• Using target initialization files on page 177

9.1 Using target initialization files

This section describes how to configure the CodeWarrior debugger to use a specific target initialization file.

A target initialization file is a command file that the CodeWarrior debugger executes each time the launch configuration to which the initialization file is assigned is debugged. You can use the target initialization file for all launch configuration types (Attach, Connect and Download). The target initialization file is executed after the connection to the target is established, but before the download operation takes place.

The debugger executes the commands in the target initialization file using the target connection protocol, such as a JTAG run-control device.

NOTE
You do not need to use an initialization file if you debug using the CodeWarrior TRK debug protocol.

To instruct the CodeWarrior debugger to use a target initialization file:
1. Start the CodeWarrior IDE.
2. Open a bareboard project.
3. Select one of this project's build targets.
4. Select Run > Debug Configurations.
   The Debug Configurations dialog appears.
5. Select the appropriate launch configuration from the left panel.
6. In the Main tab, from the Connection panel, click Edit next to the Connection drop-down list.
   The Properties for <Launch Configuration Name> window appears.
7. Click Edit next to the Target drop-down list.
   The Properties for <remote system> window appears.
8. In the Initialization tab, select the appropriate cores checkboxes from the Initialize target column, as shown in the figure below.
9. In the Initialize target script column, click the ellipsis button, as shown in the figure above.

**TIP**
Click in the specified cell of the Initialize target script column for the ellipsis button to appear.

The Target Initialization File dialog appears, as shown in the figure below.

**Figure 97:** Target initialization file dialog
10. Select the target initialization file by using the buttons provided in the dialog and click OK.
The target initialization files are available at the following path:

<CWInstallDir>CW_ARMvc7\ARMv7\ARM_Support\Initialization_Files\n
You can also write your own target initialization files.
Chapter 10
Memory Configuration Files

A memory configuration file contains commands that define the rules the debugger follows when accessing a target board's memory.

NOTE
Memory configuration files do not define the memory map for the target. Instead, they define how the debugger should treat the memory map the target has already established. The actual memory map is initialized either by a target resident boot loader or by a target initialization file. For more information, see Target Initialization Files on page 177 chapter of this manual.

If necessary, you can have the CodeWarrior debugger execute a memory configuration file immediately before the debugger downloads a bareboard binary to a target board. The memory configuration file defines the memory access rules (restrictions, translations) used each time the debugger needs to access memory on the target board.

NOTE
Assign a memory configuration file to bareboard build targets only. The memory of a board that boots embedded Linux® is already set up properly. A memory configuration file defines memory access rules for the debugger; the file has nothing to do with the OS running on a board. If needed, a memory configuration file should be in place at all times. The Linux Kernel Aware Plugin performs memory translations automatically, relieving the user from specifying them in the memory configuration file. In addition, for certain processors, the debugger can automatically read the translations from the target in a bareboard scenario, relieving the user from specifying them in the memory configuration file.

This chapter includes the following section:
• Using memory configuration files on page 181

10.1 Using memory configuration files

This section describes how to configure the CodeWarrior debugger to use a specific memory configuration file.

A memory configuration file is a command file that contains memory access rules that the CodeWarrior debugger uses each time the build target to which the configuration file is assigned is debugged.

You specify a memory configuration file in the Memory tab of the remote system configuration (shown in the figure below).
You can also write your own memory configuration files.
Chapter 11
Working with Hardware Tools

This chapter explains how to use the CodeWarrior hardware tools. You can use these tools for board bring-up, test, and analysis.

The following tools are described in this chapter:

• Flash programmer on page 183
• Hardware diagnostics on page 195

11.1 Flash programmer

Flash programmer is a CodeWarrior plug-in that lets you program the flash memory of the supported target boards from within the IDE.

The flash programmer can program the flash memory of the target board with code from a CodeWarrior IDE project or a file.

The flash programmer runs as a target task in the Eclipse IDE. To program the flash memory on a target board, you need to create, configure, and execute a flash programmer target task. You can click the Save button or press Ctrl+S to save task settings.

The following subsections describe how to work with flash programmer:

• Create flash programmer target task on page 183
• Configure flash programmer target task on page 185
• Execute flash programmer target task on page 190
• SD/eMMC flash programmer on page 192
• Flash File to Target on page 193

11.1.1 Create flash programmer target task

You can create a flash programmer task using the Create New Target Task wizard.

Follow these steps:

1. Choose Window > Show View > Other from the CodeWarrior IDE menu bar.
   
   The Show View dialog appears.
2. Expand the **Debug** group and select **Target Tasks**.

3. Click **OK**.

   The **Target Tasks** view appears.
4. Click the **Create New Target Task** button in the **Target Tasks** view toolbar. The **Create New Target Task** wizard appears.

![Create New Target Task window](image)

**Figure 101: Create New Target Task window**

5. In the **Task Name** text box, enter a name for the new flash programming target task.

6. Choose a launch configuration from the **Run Configuration** pop-up menu.
   - Choose **Active Debug Context** when flash programmer is used over an active debug session.
   - Choose a project-specific debug context when flash programmer is used without an active debug session.

7. Choose **Flash Programmer for ARM** from the **Task Type** pop-up menu.

8. Click **OK**.

   The target task is created in the **Target Tasks** view. Click **Edit Task Configuration** button and the ARM Flash Programmer Task editor window appears. You use this window to configure the flash programmer target task.

   - **Flash Devices** - Lists the devices added in the current task.
   - **Target RAM** - Lets you specify the settings for Target RAM.
   - **Flash Program Actions** - Displays the programmer actions to be performed on the flash devices.

### 11.1.2 Configure flash programmer target task

You can add flash devices, specify Target RAM settings, and add flash program actions to a flash programmer task to configure it.

This section contains the following subsections:

- **Add flash device** on page 186
- **Specify target RAM settings** on page 186
- **Add flash programmer actions** on page 186
11.1.2.1 Add flash device
This section explains how to add a flash device to the Flash Devices table.

In the ARM Flash Programmer Task editor window, to add a flash device to the Flash Devices table:
1. Click the Add Device button.
   The Add Device dialog appears.
2. Select a flash device from the device list.
3. Click the Add Device button.
   The flash device is added to the Flash Devices table in the ARM Flash Programmer Task editor window.

You can select multiple flash devices to add to the Flash Devices table. To select multiple devices, hold down the Control key while selecting the devices.

4. Click Done.
   The Add Device dialog closes and the flash device appears in the Flash Devices table in the ARM Flash Programmer Task editor window.

For NOR and QSPI flashes, the base address indicates the location where the flash is mapped in the memory. For NAND flashes, the base address is usually 0x0.

11.1.2.2 Specify target RAM settings
The Target RAM is used by Flash programmer to download its algorithms.

The Target RAM (Add flash device on page 186) group contains fields to specify settings for the Target RAM.

- Address text box: Specify the address from the target memory. The Address text box should contain the first address from target memory used by the flash algorithm running on a target board.
- Size text box: Specify the size of the target memory. The flash programmer does not modify any memory location other than the target memory buffer and the flash memory.
- Verify Target Memory Writes checkbox: Select this checkbox to verify all write operations to the hardware RAM during flash programming.

11.1.2.3 Add flash programmer actions
The Flash Programmer Actions group in the ARM Flash Programmer Task editor window allows you to add flash programmer actions.

The Flash Programmer Actions group contains the following UI controls to work with flash programmer actions:

- Add Action pop-up menu:
  - Erase / Blank Check: Allows you to add erase or blank check actions for a flash device
  - Program / Verify: Allows you to add program or verify flash actions for a flash device
  - Checksum: Allows you to add checksum actions for a flash device
  - Diagnostics: Allows you to add diagnostics actions for a flash device
• **Dump Flash**: Lets you add a dump flash action
• **Protect/Unprotect**: Lets you add protect or unprotect action
• **Secure/Unsecure**: Lets you add secure or unsecure action
• **Duplicate Action** button: Allows you to duplicate a flash program action in the Flash Programmer Actions table.
• **Remove Action** button: Allows you to remove a flash program action from the Flash Programmer Actions table.
• **Move Up** button: Allows you to move up the selected flash action in the Flash Programmer Actions table.
• **Move Down** button: Allows you to move down the selected flash action in the Flash Programmer Actions table.

NOTE

Actions can also be enabled or disabled using the **Enabled** column. The **Description** column contains the default description for the flash programmer actions. You can also edit the default description.

This section contains the following subsections:

• **Erase/Blank check actions** on page 187
• **Program/Verify actions** on page 188
• **Checksum actions** on page 188
• **Diagnostics actions** on page 189
• **Dump Flash actions** on page 189
• **Protect/Unprotect actions** on page 190

### 11.1.2.3.1 Erase/Blank check actions

The Erase action erases sectors from the flash device.

You can also use the erase action to erase the entire flash memory without selecting sectors. The blank check action verifies if the specified areas have been erased from the flash device.

NOTE

Flash Programmer will not erase a bad sector in the NAND flash. After the erase action a list of bad sectors is reported (if any).

To add an erase/blank check action:

1. Choose **Erase/Blank Check Action** from the Add Action pop-up menu.

   The Add Erase/Blank Check Action dialog appears.

2. Select a sector from the Sectors table and click the Add Erase Action button to add an erase operation on the selected sector.

   **NOTE**
   
   Press the **Control** or the **Shift** key for selecting multiple sectors from the Sectors table.

3. Click the Add Blank Check Action button to add a blank check operation on the selected sector.

4. Select the **Mass erase all devices** checkbox to erase the entire flash memory.
After selecting the **Mass erase all devices** checkbox, you need to add either erase or blank check action to erase all sectors.

5. Click **Done**.

The **Add Erase/Blank Check Action** dialog closes and the added erase/blank check actions appear in the **Flash Programmer Actions** table in the **ARM Flash Programmer Task** editor window.

### 11.1.2.3.2 Program/Verify actions

The program action allows you to program the flash device and the verify action verifies the programmed flash device.

To add a program/verify action:

1. Choose **Program/Verify Action** from the **Add Action** pop-up menu.

   The **Add Program / Verify Action** dialog appears.

2. Select the file to be written to the flash device.

3. Select the **Use File from Launch Configuration** checkbox to use the file from the launch (run) configuration associated with the task.

4. Specify the file name in the **File** text box. You can use **Workspace**, **File System**, or **Variables** buttons to select the desired file.

5. Choose a file type from the **File Type** pop-up menu. You can select any one of the following file types:
   - **Auto** - Detects the file type automatically.
   - **Elf** - Specifies executable in ELF format.
   - **Srec** - Specifies files in Motorola S-record format.
   - **Binary** - Specifies binary files.

6. Select the **Erase sectors before program** checkbox to erase sectors before program.

7. [Optional] Select the **Verify after program** checkbox to verify after the program.

   **NOTE**
   The **Verify after program** checkbox is available only with the processors supporting it.

8. Select the **Restricted To Address in this Range** checkbox to specify a memory range. The write action is permitted only in the specified address range. In the **Start** text box, specify the start address of the memory range sector and in the **End** text box, specify the end address of the memory range.

9. Select the **Apply Address Offset** checkbox and set the memory address in the **Address** text box. Value is added to the start address of the file to be programmed or verified.

10. Click the **Add Program Action** button to add a program action on the flash device.

11. Click the **Add Verify Action** button to add a verify action on the flash device.

12. Click **Done**.

   The **Add Program / Verify Action** dialog closes and the added program/verify actions appear in the **Flash Programmer Actions** table in the **ARM Flash Programmer Task** editor window.

### 11.1.2.3.3 Checksum actions

The checksum can be computed over host file, target file, memory range, or entire flash memory.

To add a checksum action:
1. Choose **Checksum Action** from the **Add Action** pop-up menu.
   
   The **Add Checksum Action** dialog appears.

2. Select the file for checksum action.

3. Select the **Use File from Launch Configuration** checkbox to use the file from the launch (run) configuration associated with the task.

4. Specify the filename in the **File** text box. You can use the **Workspace, File System**, or **Variables** buttons to select the desired file.

5. Choose the file type from the **File Type** pop-up menu.

6. Select an option from the **Compute Checksum Over** options. The checksum can be computed over the host file, the target file, the memory range, or the entire flash memory.

7. Specify the memory range in the **Restrict To Addresses in this Range** group. The checksum action is permitted only in the specified address range. In the **Start** text box, specify the start address of the memory range sector and in the **End** text box, specify the end address of the memory range.

8. Select the **Apply Address Offset** checkbox and set the memory address in the **Address** text box. Value is added to the start address of the file to be programmed or verified.

9. Click the **Add Checksum Action** button.

10. Click **Done**.

    The **Add Checksum Action** dialog closes and the added checksum actions appear in the **Flash Programmer Actions** table in the **ARM Flash Programmer Task** editor window.

### 11.1.2.3.4 Diagnostics actions

The diagnostics action generates the diagnostic information for the selected flash device.

**NOTE**

Flash Programmer will report bad blocks, if they are present in the NAND flash.

To add a diagnostics action:

1. Choose **Diagnostics** from the **Add Action** pop-up menu.

   The **Add Diagnostics Action** dialog appears.

2. Select a device to perform the diagnostics action.

3. Click the **Add Diagnostics Action** button to add diagnostic action on the selected flash device.

   **NOTE**

   Select the **Perform Full Diagnostics** checkbox to perform full diagnostics on a flash device.

4. Click **Done**.

    The **Add Diagnostics Action** dialog closes and the added diagnostics action appears in the **Flash Programmer Actions** table in the **ARM Flash Programmer Task** editor window.

### 11.1.2.3.5 Dump Flash actions

The dump flash action allows you to dump selected sectors of a flash device or the entire flash device.

To add a dump flash action:

1. Choose **Dump Flash Action** from the **Add Action** pop-up menu.
The Add Dump Flash Action dialog appears.

2. Specify the file name in the File text box. The flash is dumped in this selected file.

3. Choose the file type from the File Type pop-up menu. You can choose any one of the following file types:
   • Srec: Saves files in Motorola S-record format.
   • Binary: Saves files in binary file format.

4. Specify the memory range for which you want to add dump flash action.
   • Enter the start address of the range in the Start text box.
   • Enter the end address of the range in the End text box.

5. Click the Add Dump Flash Action button to add a dump flash action.

6. Click Done.

The Add Dump Flash Action dialog closes and the added dump flash action appear in the Flash Programmer Actions table in the ARM Flash Programmer Task editor window.

11.1.2.3.6 Protect/Unprotect actions
The protect/unprotect actions allow you to change the protection of a sector in the flash device.

To add a protect/unprotect action:
1. Choose the Protect/Unprotect Action from the Add Action pop-up menu.

   The Add Protect/Unprotect Action dialog appears.

2. Select a sector from the Sectors table and click the Add Protect Action button to add a protect operation on the selected sector.

   **NOTE**
   Press the Control or Shift key for selecting multiple sectors from the Sectors table.

3. Click the Add Unprotect Action button to add an unprotect action on the selected sector.

4. Select the All Device checkbox to add action on full device.

5. Click Done.

The Add Protect/Unprotect Action dialog closes and the added protect or unprotect actions appear in the Flash Programmer Actions table in the ARM Flash Programmer Task editor window.

11.1.3 Execute flash programmer target task
You can execute the flash programmer tasks using the Target Tasks view.

To execute the configured flash programmer target task, select a target task and click the Execute button in the Target Tasks view toolbar. Alternatively, right-click a target task and choose Execute from the shortcut menu.

You can use predefined target tasks for supported boards. To load a predefined target task, click Import button in the Target Task view. Select the task, for example, LS102xAQDS_NAND_FLASH.xml from {CW folder} \ARMv7\bin\plugins\Support\Target Task\Flash_Programmer\ARM\ folder. Change the Run Configuration and then double-click the imported task, the ARM Flash Programmer Task view appears.
In the **ARM Flash Programmer Task** view, click **Add Action** button to add the **Flash Programmer Actions** as shown in above figure. Select the checkbox for the operation that you want to execute. Click **Execute** button in the **Target Task** view.

**NOTE**

To load a predefined target task, right-click in the **Target Tasks** view and choose **Import** from the shortcut menu. To save your custom tasks, right-click in the **Target Tasks** view and then choose **Export** from the shortcut menu.

You can check the results of flash batch actions in the **Console** view. The green color indicates the success and the red color indicates the failure of the task.
11.1.4 SD/eMMC flash programmer

This section explains what is SD/eMMC flash programmer and how to use it to program U-Boot.

The SD/eMMC card uses the notion of blocks, not addresses. One block has 512 bits. The flash programmer UI still uses address at operations; therefore, 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). In case of a program operation, if you select the Erase sectors before program checkbox (which is always recommended) in the Add Program / Verify Action dialog, then the flash programmer will only erase the sectors that are written.

Because the size of a block is small and the size of the card is large, all the available blocks cannot be displayed in the Add Erase/Blank Check Action dialog; therefore, the card has been divided into sectors of 4 MB each. So, when you choose an erase/blank check operation, you will erase/check sectors of 4 MB.

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, change in the ARM Flash Programmer Task editor window the running address of the algorithm (Target RAM field) for the target task with the OCRAM address of your processor, which will enable the use of the internal buffer of the eSDHC controller. This will reduce the performance as DMA is not used in this mode.

This section contains the following subsection:

- Programming U-Boot using SD/eMMC flash programmer on page 192

11.1.4.1 Programming U-Boot using SD/eMMC flash programmer

This section explains how to program U-Boot using the SD/eMMC flash programmer.

The example described in this section uses the LS1021AQDS Rev. 2.0 target. The steps given here are applicable to all targets that support SD/eMMC flash programmer.

The steps to program U-Boot using SD/eMMC flash programmer are as follows:

1. Import the preconfigured target task for the target (in the current example, eMMC flash programmer is used; for the SD flash programmer, the steps are same but only the preconfigured target task is different):
   `ARMv7/bin/plugins/support/TargetTask/Flash_Programmer/ARM/LS102xAQDS_MMC_FLASH.xml.`
2. Choose **Program/Verify Action** from the **Add Action** pop-up menu.

The **Add Program / Verify Action** dialog appears, as shown in the figure below.

**Figure 104: Add Program / Verify Action dialog**

3. Select the U-Boot image you want to write by clicking **File System**.

4. Select the **Erase sectors before program** checkbox to erase sectors before U-Boot is programmed.

5. Select the **Apply Address Offset** checkbox and set the start address of the U-Boot image to be programmed.

   **NOTE**

   The U-Boot image for the LS1 targets is written from block 8 (the address that you have to specify in the **Address** field is computed as described earlier in this section: block 8 x 512 = 4096 = 0x00001000).

6. Click **Add Program Action** to add a program action on the flash device.

7. Click **Done** to close the **Add Program / Verify Action** dialog.

8. Execute the flash programmer target task.

### 11.1.5 Flash File to Target

You can use the **Flash File to Target** feature to perform flash operations, such as erasing a flash device or programming a file.

You do not need any project for using **Flash File to Target** feature, only a valid **Remote System** is required.

To open the **Flash File to Target** dialog, click the **Flash Programmer** button on the IDE toolbar.

- **Connection** pop-up menu - Lists all run configurations defined in Eclipse. If a connection to the target has already been made the control becomes inactive and contains the text Active Debug Configuration.

- **Flash Configuration File** pop-up menu - Lists predefined target tasks for the processor selected in the Launch Configuration and tasks added by user with the **Browse** button. The items in this pop-up menu are updated based on the processor selected in the launch configuration.

- **Unprotect flash memory before erase** checkbox - Select to unprotect flash memory before erasing the flash device. This feature allows you to unprotect the flash memory from **Flash File To Target** dialog.

- **File to Flash** group - Allows selecting the file to be programmed on the flash device and the location.
• File text box - Used for specifying the filename. You can use the Workspace, File System, or Variables buttons to select the desired file.

• Offset:0x text box - Used for specifying offset location for a file. If no offset is specified the default value of zero is used. The offset is always added to the start address of the file. If the file does not contain address information then zero is considered as start address.

• Save as Target Task - Select to enable Task Name text box.

• Task Name text box - Lets you to save the specified settings as a Flash target task. Use the testbox to specify the name of the target task.

• Erase Whole Device button - Erases the flash device. In case you have multiple flash blocks on the device, all blocks are erased. If you want to selectively erase or program blocks, use the Flash programmer on page 183 feature.

• Erase and Program button - Erases the sectors that are occupied with data and then programs the file. If the flash device can not be accessed at sector level then the flash device is completely erased.

This feature helps you perform these basic flash operations:

• Erasing flash device on page 194
• Programming file on page 194

11.1.5.1 Erasing flash device
This section explains how to erase a flash device using the Flash File to Target feature.

To erase a flash device, follow these steps:

1. Click the Flash Programmer button on the IDE toolbar.
   The Flash File to Target dialog appears.

2. Choose a connection from the Connection pop-up menu.
   
   NOTE
   If a connection is already established with the target, this control is disabled.
   
   The Flash Configuration File pop-up menu is updated with the supported configurations for the processor from the launch configuration.

3. Choose a flash configuration from the Flash Configuration File pop-up menu.

4. Select the Unprotect flash memory before erase checkbox to unprotect flash memory before erasing the flash device.

5. Click the Erase Whole Device button.

11.1.5.2 Programming file
This section explains how to program a file using the Flash File to Target feature.

The steps are as follows:

1. Click the Flash Programmer button on the IDE toolbar.
   The Flash File to Target dialog appears.

2. Choose a connection from the Connection pop-up menu.
   
   NOTE
   If a connection is already established with the target, this control is disabled.
The Flash Configuration File pop-up menu is updated with the supported configurations for the processor from the launch configuration.

3. Choose a flash configuration from the Flash Configuration File pop-up menu.

4. Select the Unprotect flash memory before erase checkbox to unprotect flash memory before erasing the flash device.

5. Type the file name in the File text box. You can use the Workspace, File System, or Variables buttons to select the desired file.

6. Type the offset location in the Offset text box.

7. Click the Erase and Program button.

11.2 Hardware diagnostics

The Hardware Diagnostics utility lets you run a series of diagnostic tests that determine if the basic hardware is functional.

These tests include:

- Memory read/write: This test only makes a read or write access to the memory in order to read or write a byte, word (2 bytes) and long word (4 bytes) to or from the memory. For this task, the user needs to set the options in the Memory Access group.

- Scope loop: This test makes read and write accesses to memory in a loop at the target address. The time between accesses is given by the loop speed settings. The loop can only be stopped by the user, which cancels the test. For this type of test, the user needs to set the memory access settings and the loop speed.

- Memory tests: This test requires the user to set the access size and target address from the access settings group and the settings present in the Memory Tests group.

This section contains the following subsections:

- Creating hardware diagnostics task on page 195
- Working with Hardware Diagnostic Action editor on page 196

11.2.1 Creating hardware diagnostics task

You can create a hardware diagnostic task using the Create New Target Task wizard.

To create a task for hardware diagnostics:

1. Choose Window > Show View > Other from the IDE menu bar.
   
   The Show View dialog appears.

2. Expand the Debug group and select Target Tasks.

3. Click OK.

4. Click the Create New Target Task button on the Target Tasks view toolbar. Alternatively, right-click in the Target Tasks view and choose New Task from the shortcut menu.
   
   The Create a Target Task dialog appears.
5. Type name for the new task in the **Task Name** text box.

6. Choose a launch configuration from the **Run Configuration** pop-up menu.

   **NOTE**
   If the task does not successfully launch the configuration that you specify, the **Execute** button on the **Target Tasks** view toolbar appears disabled.

7. Choose **Hardware Diagnostic for ARM** from the **Task Type** pop-up menu.

8. Click **Finish**.

   A new hardware diagnostic task is created in the **Target Tasks** view.

   **NOTE**
   You can perform various actions on a hardware diagnostic task, such as renaming, deleting, or executing the task, using the shortcut menu that appears on right-clicking the task in the **Target tasks** view.

### 11.2.2 Working with Hardware Diagnostic Action editor

The **Hardware Diagnostic Action** editor is used to configure a hardware diagnostic task.

To open the **Hardware Diagnostic Action** editor for a particular task, double-click the task in the **Target Tasks** view.

The following figure shows the **Hardware Diagnostics Action** editor.
The Hardware Diagnostics Action editor window includes the following groups:

- **Action Type** on page 197
- **Memory Access** on page 197
- **Loop Speed** on page 198
- **Memory Tests** on page 198

**11.2.2.1 Action Type**

You can choose the action type from the Action Type group in the Hardware Diagnostics Action editor window.

You can choose any one of the following actions:

- Memory read/write - Enables the options in the Memory Access group
- Scope loop - Enables the options in the Memory Access and the Loop Speed groups
- Memory test - Enables the access size and target address from the access settings group and the settings present in the Memory Tests group

**11.2.2.2 Memory Access**

The Memory Access pane configures diagnostic tests for performing memory reads and writes over the remote connection interface.

The table below lists and describes the items in the pane.
Table 64: Memory Access pane items

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>Select to have the hardware diagnostic tools perform read tests.</td>
</tr>
<tr>
<td>Write</td>
<td>Select to have the hardware diagnostic tools perform write tests.</td>
</tr>
<tr>
<td>1 Byte</td>
<td>Select to have the hardware diagnostic tools perform byte-size operations.</td>
</tr>
<tr>
<td>2 Bytes</td>
<td>Select to have the hardware diagnostic tools perform word-size operations.</td>
</tr>
<tr>
<td>4 Bytes</td>
<td>Select to have the hardware diagnostic tools perform long-word-size operations.</td>
</tr>
<tr>
<td>Memory space and address</td>
<td>Specify the address of an area in RAM that the hardware diagnostic tools should analyze. The tools must be able to access this starting address through the remote connection (after the hardware initializes).</td>
</tr>
<tr>
<td>Value</td>
<td>Specify the value that the hardware diagnostic tools write during testing. Select the Write option to enable this text box.</td>
</tr>
<tr>
<td>Verify Memory Writes</td>
<td>Select the checkbox to verify success of each data write to the memory.</td>
</tr>
</tbody>
</table>

11.2.2.3 Loop Speed

The Loop Speed pane configures diagnostic tests for performing repeated memory reads and writes over the remote connection interface.

The tests repeat until you stop them. By performing repeated read and write operations, you can use a scope analyzer or logic analyzer to debug the hardware device. After the first 1000 operations, the Status shows the estimated time between operations.

**NOTE**

For all values of Speed, the time between operations depends heavily on the processing speed of the host computer.

For Read operations, the Scope Loop test has an additional feature. During the first read operation, the hardware diagnostic tools store the value read from the hardware. For all successive read operations, the hardware diagnostic tools compare the read value to the stored value from the first read operation. If the Scope Loop test determines that the value read from the hardware is not stable, the diagnostic tools report the number of times that the read value differs from the first read value. The following table lists and describes the items in Loop Speed pane.

Table 65: Loop Speed pane items

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specify the speed</td>
<td>Enter a numeric value between 0 to 1000 in the text box to adjust the speed. You can also move the slider to adjust the speed at which the hardware diagnostic tools repeat successive read and write operations. Lower speeds increase the delay between successive operations. Higher speeds decrease the delay between successive operations.</td>
</tr>
</tbody>
</table>

11.2.2.4 Memory Tests

The Memory Tests pane lets you perform three hardware tests: Walking Ones, Address, and Bus Noise.

You can specify any combination of tests and number of passes to perform. For each pass, the hardware diagnostic tools performs the tests in turn, until all passes are complete. The tools compare memory test failures
and display them in a log window after all passes are complete. Errors resulting from memory test failures do not stop the testing process; however, fatal errors immediately stop the testing process.

The following table explains the items in the Memory Tests pane.

<table>
<thead>
<tr>
<th>Item</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Walking 1's</td>
<td>Select the checkbox to have the hardware diagnostic tools perform the Walking Ones on page 199 test. Deselect to have the diagnostic tools skip the Walking Ones on page 199 test.</td>
</tr>
<tr>
<td>Address</td>
<td>Select to have the hardware diagnostic tools perform the Address on page 200 test. Deselect to have the diagnostic tools skip the Address on page 200 test.</td>
</tr>
<tr>
<td>Bus Noise</td>
<td>Select to have the hardware diagnostic tools perform the Bus Noise on page 200 test. Deselect to have the diagnostic tools skip the Bus Noise on page 200 test.</td>
</tr>
<tr>
<td>Test Area Size</td>
<td>Specify the size of memory to be tested. This setting along with Target Address defines the memory range being tested.</td>
</tr>
<tr>
<td>Number of Passes</td>
<td>Enter the number of times that you want to repeat the specified tests.</td>
</tr>
</tbody>
</table>

This section contains the following subsections:

- Walking Ones on page 199
- Address on page 200
- Bus Noise on page 200
- Memory test use cases on page 200

### 11.2.2.4.1 Walking Ones

This section describes the Walking Ones test.

This test detects these memory faults:

- **Address Line:** The board or chip address lines are shorting or stuck at 0 or 1. Either condition could result in errors when the hardware reads and writes to the memory location. Because this error occurs on an address line, the data may end up in the wrong location on a write operation, or the hardware may access the wrong data on a read operation.

- **Data Line:** The board or chip data lines are shorting or stuck at 0 or 1. Either condition could result in corrupted values as the hardware transfers data to or from memory.

- **Retention:** The contents of a memory location change over time. The effect is that the memory fails to retain its contents over time.

The Walking Ones test includes four sub-tests:

- **Walking Ones:** This subtest first initializes memory to all zeros. Then the subtest writes, reads, and verifies bits, with each bit successively set from the least significant bit (LSB) to the most significant bit (MSB). The subtest configures bits such that by the time it sets the MSB, all bits are set to a value of 1. This pattern repeats for each location within the memory range that you specify. For example, the values for a byte-based Walking Ones subtest occur in this order:

  0x01, 0x03, 0x07, 0x0F, 0x1F, 0x3F, 0x7F, 0xFF
• Ones Retention: This subtest immediately follows the Walking Ones subtest. The Walking Ones subtest should leave each memory location with all bits set to 1. The Ones Retention subtest verifies that each location has all bits set to 1.

• Walking Zeros: This subtest first initializes memory to all ones. Then the subtest writes, reads, and verifies bits, with each bit successively set from the LSB to the MSB. The subtest configures bits such that by the time it sets the MSB, all bits are set to a value of 0. This pattern repeats for each location within the memory range that you specify. For example, the values for a byte-based Walking Zeros subtest occur in this order:

\[0xFE, 0xFC, 0xF8, 0xE0, 0xC0, 0x80, 0x00\]

• Zeros Retention: This subtest immediately follows the Walking Zeros subtest. The Walking Zeros subtest should leave each memory location with all bits set to 0. The Zeros Retention subtest verifies that each location has all bits set to 0.

### 11.2.2.4.2 Address

This section describes the Address test.

This test detects memory aliasing. *Memory aliasing* exists when a physical memory block repeats one or more times in a logical memory space. Without knowing about this condition, you might conclude that there is much more physical memory than what actually exists.

The address test uses a simplistic technique to detect memory aliasing. The test writes sequentially increasing data values (starting at one and increasing by one) to each successive memory location. The maximum data value is a prime number and its specific value depends on the addressing mode so as to not overflow the memory location.

The test uses a prime number of elements to avoid coinciding with binary math boundaries:

- For byte mode, the maximum prime number is 28-5 or 251.
- For word mode, the maximum prime number is 216-15 or 65521.
- For long word mode, the maximum prime number is 232-5 or 4294967291.

If the test reaches the maximum value, the value rolls over to 1 and starts incrementing again. This sequential pattern repeats throughout the memory under test. Then the test reads back the resulting memory and verifies it against the written patterns. Any deviation from the written order could indicate a memory aliasing condition.

### 11.2.2.4.3 Bus Noise

This section describes the Bus Noise test.

This test stresses the memory system by causing many bits to flip from one memory access to the next (both addresses and data values). *Bus noise* occurs when many bits change consecutively from one memory access to another. This condition can occur on both address and data lines.

### 11.2.2.4.4 Memory test use cases

This section describes some memory test use cases.

The memory read/write, scope loop tests, and memory tests are host-based tests. The host machine issues read and write action to the memory through the connection protocol. For example, CodeWarrior connection server (CCS). Memory tests are the complex tests that can be executed in two modes: Host based and Target based depending upon the selection made for the *Use Target CPU* checkbox. The Host based tests are slower. The target based tests are not available.