CodeWarrior Development Studio for Power Architecture™ Processors

Professional / Linux® Application Editions

Targeting Manual

Revised: 12 August 2010

freescale semiconductor

How to Contact Us

Corporate Headquarters	Freescale Semiconductor, Inc. 6501 William Cannon Drive West Austin, Texas 78735 U.S.A.
World Wide Web	`http://www.freescale.com/codewarrior`
Technical Support	`http://www.freescale.com/support`

Table of Contents

1

Introduction

This manual explains how to install and use the Professional and Linux® Application editions of the CodeWarrior™ Development Studio for Power Architecture™ Processors software development tools.

Use these tools to develop both bare board and embedded Linux software for Power Architecture processors and boards.

The sections of this chapter are:

- Overview of This Manual
- Related Documentation
- CodeWarrior Power Architecture Development Tools
- CodeWarrior Development Process

Overview of This Manual

Table 1.1 lists and describes each chapter in this manual.

Table 1.1 Chapter Contents

Chapter	Description
Introduction	(this chapter)
Working with Projects	Lists the different types of Linux and bare board projects you can create and shows how to create bare board and Linux projects using the EPPC New Project Wizard.
Target Settings Reference	Lists each target settings panel in this CodeWarrior product and defines each setting available on each of these panels.
Working with the Debugger	Lists the remote connections the CodeWarrior debugger supports and covers Power Architecture-specific debugger features.
Working with the Hardware Tools	Explains how to use the flash programmer, hardware diagnostics, and EPPC trace buffer tools.

Table 1.1 Chapter Contents (*continued*)

Chapter	Description
Debugger Limitations and Workarounds	Documents processor-family specific debugger limitations and workarounds.
Target Initialization Files	Explains how to use a target initialization file to initialize a board’s memory and registers prior to a debug session.
Memory Configuration Files	Explains how to use a memory configuration file to define a board’s accessible memory prior to a debug session.
Using the Dhrystone Benchmark Software	Explains how to use the Dhrystone benchmark software.
Using the Linux-hosted Simulators	Explains how to configure a remote connection to communicate over the network with the simulator running on the Linux machine.

Related Documentation

This section provides information where to find more information about your CodeWarrior product and about developing software for the Power Architecture™ processors.

- CodeWarrior Information
- Embedded Power Architecture API Programming Information
- Power Architecture Processor and Board Information
- AltiVec™ Information

CodeWarrior Information

- To view the online help for the CodeWarrior Integrated Development Environment (IDE), select **Help > CodeWarrior IDE** from the menu bar.
- For late-breaking information about new features, bug fixes, known problems, and incompatibilities, read the release notes. They are in this folder:

  `installDir\Release_Notes\`

- For example CodeWarrior projects are in this folder:

  `installDir\(CodeWarrior_Examples)\PowerPC_EABI\`

- For general information about the CodeWarrior IDE and debugger, see the *CodeWarrior™ IDE User’s Guide*. This document is in this folder:

installDir\Help\PDF\

- For information specific to the C/C++ compiler, inline assembler, standalone assembler and linker, read the *Power Architecture Build Tools Reference*. This document is in this folder:

  installDir\Help\PDF\

- For information about the Freescale standard C/C++ libraries, read the *MSL C Reference* and the *MSL C++ Reference*. This document is in this folder:

  installDir\Help\PDF\

- For information about CodeWarrior TRK, including how to customize CodeWarrior TRK for a particular target board, read the *CodeWarrior TRK Reference*. This document is in this folder:

  installDir\Help\PDF\

- For information about the recommended jumper and DIP switch settings for the boards supported by the CodeWarrior for Power Architecture Processors product, refer to the documentation in this folder:

  installDir\PowerPC_EABI_Support\Documentation\

  This folder also contains files that explain how to customize the development tools and list the additional hardware required to allow a board to interact with the tools.

NOTE A project created by the EPPC New Project Wizard includes the documentation file for the board selected during the wizard process.

Embedded Power Architecture API Programming Information

The binaries generated by the CodeWarrior for Power Architecture Processors product conform to the Power Architecture (formerly, PowerPC) Embedded Application Binary Interface (EABI).

Power Architecture EABI specification defines data structure alignment, calling conventions, etc. to which high-level language compilers for Power Architecture chips must adhere. In addition, the specification defines the object file format (ELF) and debugging-information format (DWARF) that Power Architecture linkers must generate.

To learn more the Embedded PowerPC EABI, refer to these documents:

- *System V Application Binary Interface, Third Edition*, published by UNIX System Laboratories, 1994 (ISBN 0-13-100439-5).
- *PowerPC Embedded Binary Interface, 32-Bit Implementation*, published by Freescale Semiconductor, Inc., and available at this web address:

www.freescale.com/files/32bit/doc/app_note/PPCEABI.pdf

- *Executable and Linker Format, Version 1.1*, published by UNIX System Laboratories.
- *DWARF Debugging Standard website* available at:

  www.dwarfstd.org
- *DWARF Debugging Information Format, Revision: Version 1.1.0*, published by UNIX International, Programming Languages SIG, October 6, 1992 and available at this web address:

  www.nondot.org/sabre/os/files/Executables/dwarf-v1.1.0.pdf
- *DWARF Debugging Information Format, Revision: Version 2.0.0*, Industry Review Draft, published by UNIX International, Programming Languages SIG, 7/27/1993.

Power Architecture Processor and Board Information

- For documentation of the processors and boards supported by the CodeWarrior for Power Architecture Processors product, refer to this web page:

  www.freescale.com/powerarchitecture

AltiVec™ Information

To learn more about AltiVec technology, refer to the documents listed below.

- *AltiVec Technology Programming Interface Manual*. This document is available at this web address:

  www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf
- *AltiVec Technology Programming Environments Manual*. This document is available at this web address:

  www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPEM.pdf

CodeWarrior Power Architecture Development Tools

Programming for Power Architecture processors is much like programming for any other CodeWarrior platform target. If you have not used the CodeWarrior IDE before, these are the tools with which you must become familiar:

- CodeWarrior IDE

- Project Manager
- Editor
- C/C++ Compiler
- Standalone Assembler
- Linker
- Debugger
- Main Standard Libraries

If you are an experienced CodeWarrior user, you need to become familiar with the Power Architecture runtime environment.

CodeWarrior IDE

The CodeWarrior IDE is a program that lets you configure and control a set of software development tools for the Power Architecture processor family.

The IDE has a graphical user interface (GUI). You use the GUI to control the development tools included in this CodeWarrior product.

The most important development tools provided by the IDE are the project manager, editor, compiler, linker, and debugger.

For complete documentation of the CodeWarrior IDE, refer to online help or the *CodeWarrior™ IDE User Guide*.

Project Manager

A project is a collection of files and configuration settings that the CodeWarrior IDE uses to generate a final output file.

The project manager is a window that displays the files and targets your project uses.

Table 1.2 defines several project-related terms.

Table 1.2 Project-related Terms

Term	Definition
Host	The system on which you run the CodeWarrior IDE to develop software for one or more platform targets.
Platform target	The operating system, simulator, or target board for which you are writing software. The platform target can be different from the host.
Build target	A named collection of settings and files that the IDE uses to build a final output file. A build target defines all build-specific information, including: • Information that identifies files that belong to the build target • Compiler and linker settings for the build target • Output information for the build target A project can contain multiple build targets. This allows you to define custom builds for different purposes.

The project manager keeps track of dependencies between files in your project. As a result, if you change a file and then build your project, the IDE compiles:

- The file you changed
- All files that are dependent on the file you changed

The project manager lets you define one or more build targets for the same project. A build target is a named set of project settings and files that the IDE uses to build a final output file.

For example, you could create a build target named Debug. For this target, you might choose settings that include information needed by the debugger.

Within the same project, you could also create a second build target, named Release. For this build target, you could exclude all debugging information so the release version of your program is smaller.

For instructions that explain how to use the CodeWarrior project manager, refer to the online help or the *CodeWarrior™ IDE User Guide*.

Editor

The CodeWarrior IDE includes a text editor that includes many features useful to programmers.

For example, the editor highlights language keywords in the color you choose, interfaces with your source code control software, and more.

For complete documentation of the CodeWarrior editor, refer to online help or the *CodeWarrior™ IDE User Guide*.

C/C++ Compiler

The CodeWarrior Power Architecture C/C++ compiler is an ANSI-compliant compiler. It compiles C and C++ statements and assembles inline assembly language statements.

You can generate Power Architecture applications and libraries that conform to the PowerPC EABI by using the CodeWarrior compiler in conjunction with the CodeWarrior linker for Power Architecture 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.

For more information about the CodeWarrior Power Architecture C/C++ compiler and its inline assembler, refer to the *Power Architecture Build Tools Reference*.

Standalone Assembler

The CodeWarrior Power Architecture assembler is a standalone assembler. The macros it supports have an easy-to-use syntax.

For more information about the CodeWarrior Power Architecture assembler, see the *Assembler Reference*.

Linker

The CodeWarrior Power Architecture linker generates binaries that conform to the PowerPC 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.

For more information about the CodeWarrior Power Architecture linker, refer to the *Power Architecture Build Tools Reference*.

Debugger

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

For general information about the debugger, including all of its common features and its visual interface, you should read the *CodeWarrior™ IDE User's Guide*.

The Power Architecture debugger debugs software as it is running on the target board. The debugger communicates with the board through a monitor program (such as CodeWarrior TRK) or through a hardware probe (such as the CodeWarrior USB TAP).

Main Standard Libraries

The Main Standard Libraries (MSL) are ANSI-compliant C and C++ standard libraries.

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

For more information about MSL, see the *MSL C Reference* and the *MSL C++ Reference*.

CodeWarrior Development Process

While working with the CodeWarrior IDE, you proceed through the development stages familiar to all programmers: writing code, compiling and linking, and debugging. See the *CodeWarrior™ IDE User's Guide* for:

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

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

If you are unfamiliar with an integrated environment in general, or with the CodeWarrior IDE in particular, you may find the topics in this section helpful. Each topic explains how one component of the CodeWarrior tools relates to a traditional command-line environment.

Project Files

A CodeWarrior IDE *project* is analogous to a make file. Because a project can have multiple build targets, the project really is analogous to a *collection* of make files. For example, you can have one project that has both a debug version and a release version of your code. You can build one or the other, or both as you wish. In the CodeWarrior IDE, the different builds within a single project are called *build targets*.

The IDE uses the project manager window to list all the files in the project. Among the kinds of files in a project are source code files and libraries.

You can add or remove files easily. You can assign files to one or more different build targets within the project, so files common to multiple targets can be managed simply.

The IDE manages all the interdependencies between files automatically and tracks which files have been changed since the last build. When you rebuild, only those files that have changed are recompiled.

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

Editing Code

The CodeWarrior IDE has an integral text editor designed for programmers. It handles text files in MS-DOS/Windows, UNIX, and Mac OS formats.

To edit a source code file, or any other editable file that is in a project, double-click the filename in the project window to open the file.

The editor window 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.

Compiling

To compile a source code file, it must be among the files that are part of the current build target. If it is, select the source code file in the project window and select **Project > Compile** from the menu bar.

To compile all the files in the current build target that have been modified since they were last compiled, select **Project > Bring Up To Date** from the menu bar.

Linking

Select **Project > Make** from the menu bar to link object code into a final binary file. The **Make** command brings the active project up-to-date, then links the resulting object code into a final output file.

You control the linker through the IDE. There is no need to specify a list of object files. The project manager tracks all the object files automatically. You can use the project manager to specify link order as well.

Use the **EPPC Target settings** panel to set the name of the final output file.

Debugging

Select **Project > Debug** from the menu bar to debug your project. This command downloads the current project’s executable to the target board and starts a debug session.

You can now use the debugger to step through the program's code, view and change the value of variables, set breakpoint. See the *CodeWarrior™ IDE User’s Guide* and the Working with the Debugger chapter of this manual for more information about the debugger.

2

Working with Projects

This chapter explains how to use CodeWarrior™ Development Studio for Power Architecture™ Processors to create projects for boards that contain a Power Architecture processor.

A CodeWarrior *project* contains one or more *build targets*. A build target is a named collection of files and settings that the build tools use to generate an output file.

The sections of this chapter are:

- Types of Projects
- Creating Projects

Types of Projects

The CodeWarrior IDE can create projects for both bare board and embedded Linux® development.

For bare board development, the IDE can create projects that generate applications, libraries, and partially linked (that is, relocatable) binaries.

For Linux, the IDE can create projects that generate applications, shared libraries, static libraries, and kernel loadable modules.

Creating Projects

This section shows you how to create EPPC projects.

There are four ways:

- Using the Bare Board New Project Wizard
- Using the Linux® New Project Wizard
- Using the External Build Wizard
- Using the Empty Project Template

Using the Bare Board New Project Wizard

This section shows you how to use the EPPC New Project wizard to create a project that generates binaries for execution on a bare board.

Use the EPPC New Project wizard to create a project, if you can accept default target settings (build options) for your project. Once the project has been created, you can change any setting the wizard selected.

NOTE The Linux Application Edition of this product does not support bare board software development.

To use the EPPC New Project Wizard to create a bare board project, follow these steps:

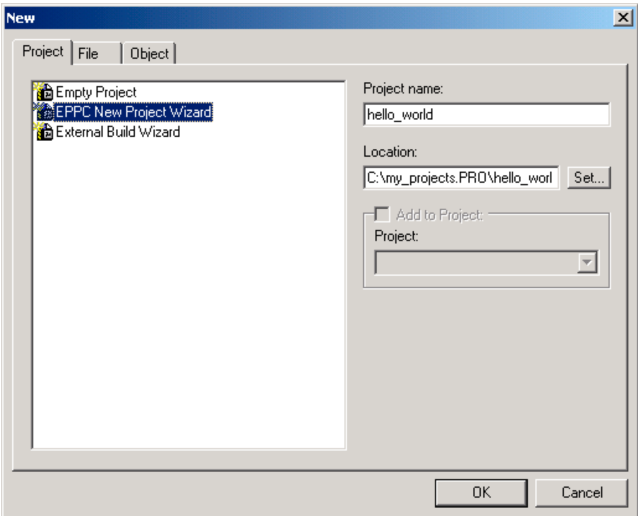
1. From the Windows taskbar, select **Start > Programs > Freescale CodeWarrior > CW for Power Architecture V8.8 > CodeWarrior IDE**.

   The CodeWarrior IDE starts and displays its main window.

2. From the IDE menu bar, select **File > New**.

   The **New** dialog box appears. (See Figure 2.1.)

Figure 2.1 New Dialog Box

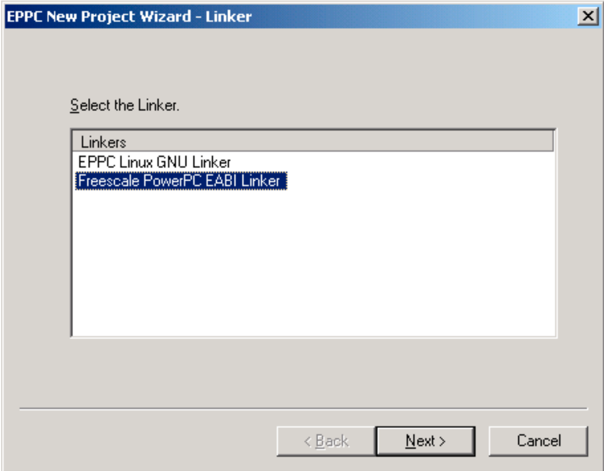


3. From the Project list box, select `EPPC New Project Wizard`.
4. In Project name text box, type `hello_world.`
5. In the Location text box, type the path in which to create this project, or click **Set** to use the **Create New Project** dialog box to find and select this path.

6. Click **OK**.

   The EPPC New Project Wizard starts and displays its **Linker** page. (See Figure 2.2.)

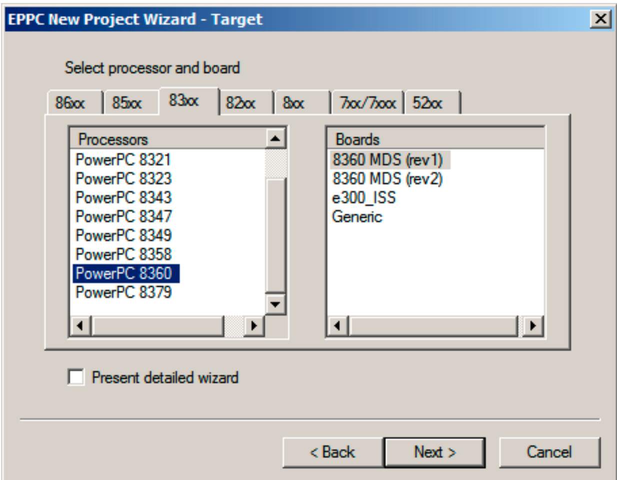
Figure 2.2 EPPC New Project Wizard — Linker Page



7. From the Linkers list box, select `Freescale PowerPC EABI Linker`.
8. Click **OK**.

   The wizard displays its **Target** page. (See Figure 2.3.)

Figure 2.3 EPPC New Project Wizard — Target Page



9. In the **Target** page, click the tab for the processor family to which the processor on your target board belongs.

   The wizard displays the processors and boards in the selected family that this CodeWarrior product supports.

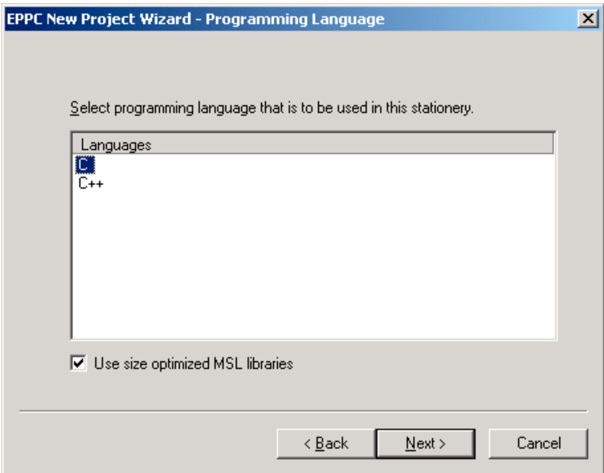
10. From the Processors list box, select the processor on your target board.

    In the Boards list box, the wizard displays the boards that contain the selected processor and which this CodeWarrior product supports.

11. From the **Target** page's Boards list box, select the board you are using.
12. Check the Present detailed wizard check box.
13. Click **Next**.

    The wizard displays the **Programming Language** page. (See Figure 2.4.)

Figure 2.4 EPPC New Project Wizard — Programming Language Page



14. From the Languages list box, select the programming language you want to use.

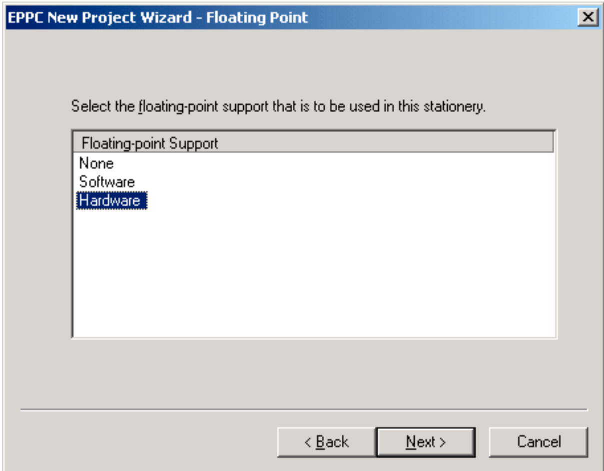
    For example, if you plan to use the C language in your source code files, select `C`.

NOTE The language you select determines the libraries with which the new project links and the contents of the main source file. If you select the C++ language, you can still add C source files to the project (and vice versa).

15. Check the Use size optimized MSL libraries box.
16. Click the **Next**.

    The wizard displays the **Floating Point** page. (See Figure 2.5.)

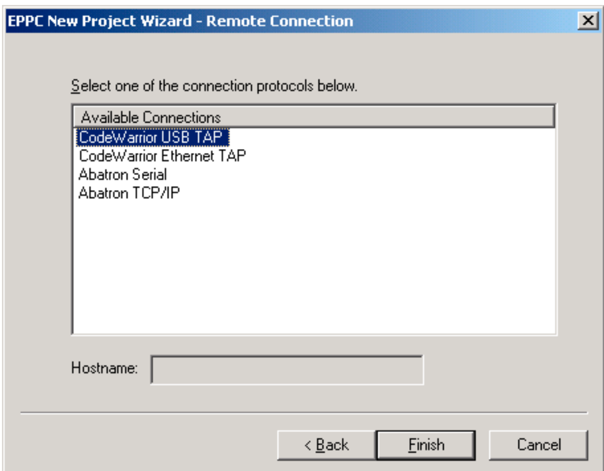
Figure 2.5 EPPC New Project Wizard — Floating Point Page



17. From the Floating-point Support list, select the type of floating-point support your project requires.
18. Click **Next**.

    The wizard displays the **Remote Connection** page. (See Figure 2.6.)

Figure 2.6 EPPC New Project Wizard — Remote Connection Page

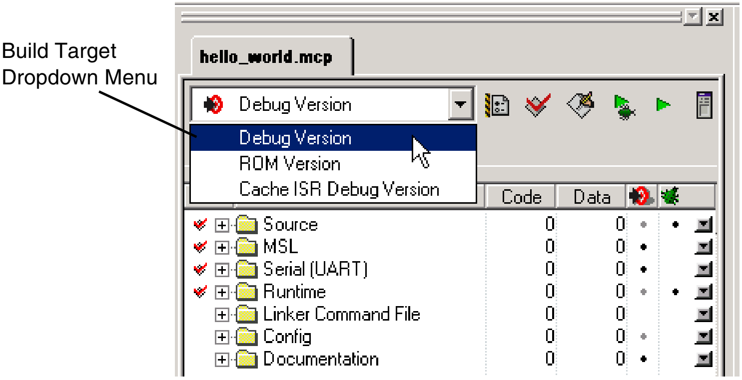


19. From the Available Connections list box, select the remote connection for the run-control hardware or software debug monitor you plan to use.

20. Click **Finish**.

    The wizard creates a project according to your specifications and displays a project window docked to the left, top, and bottom of the IDE main window. (See Figure 2.7.)

Figure 2.7 Project Window — hello_world.mcp



The generated project includes these build targets:

- Debug Version

  This build target lets you get started quickly because you can debug the generated image in RAM. Use this build target until you need interrupt service routines or you must write your program to the ROM.

- ROM Version

  This build target generates an image that can be written to ROM. The image includes exception vectors and is linked in such a way that it will boot from reset and copy the specified sections from ROM to RAM. Further, the image includes a default interrupt handler function that can easily be modified to suit your needs. Finally, this build target generates an S-Record file that standard flash programmers can use to write the image to ROM where you can debug the image. For more information, see Flash Programmer topic.

- Cache ISR Debug Version

  This build target is similar to the Debug Version build target, with the addition of the exception vectors and interrupt handler included in the ROM Version target. In addition, the Cache ISR Debug Version build target configures the MMU to use block address translation and enables the L1 data caches and instruction caches.

That's it—the new project is ready for use. You can now customize it by adding your own source code files, changing target settings, adding libraries, etc.

See Tutorial: Debugging a Bare Board Application for instructions that explain how to make and debug this project.

Using the Linux® New Project Wizard

This section shows you how to use the EPPC New Project wizard to create a project that generates binaries for execution by the embedded Linux operating system.

Use the Linux New Project wizard to create a project, if you can accept default target settings (build options) for your project. Once the project has been created, you can change any setting the wizard selected.

To use the EPPC New Project Wizard to create a Linux project, follow these steps:

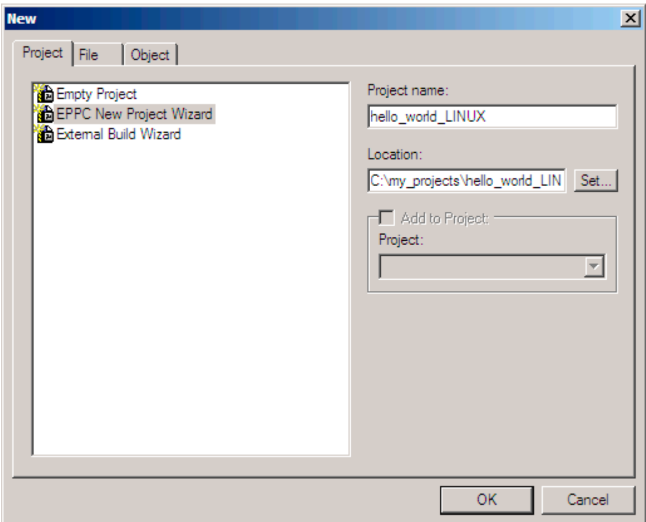
1. From the Windows taskbar, select **Start > Programs > Freescale CodeWarrior > CW for Power Architecture V8.8 > CodeWarrior IDE**.

   The CodeWarrior IDE starts and displays its main window.

2. From the IDE menu bar, select **File > New**.

   The **New** dialog box appears. (See Figure 2.8.)

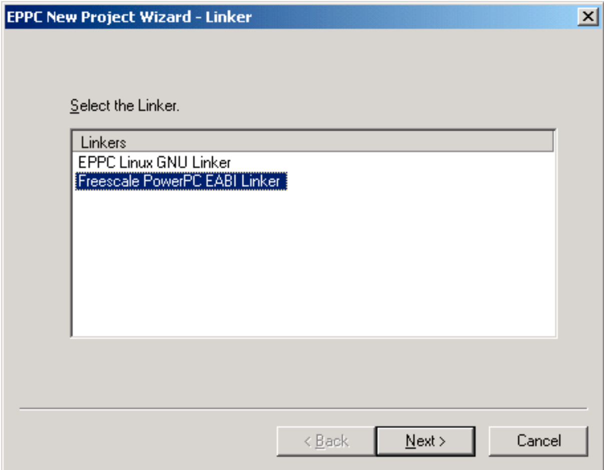
Figure 2.8 New Dialog Box



3. From the Project list box, select `EPPC New Project Wizard`.
4. In the Project Name text box, type `hello_world_LINUX`.
5. In the Location text box, type the path in which to create this project, or click **Set** to use the **Create New Project** dialog box to find and select this path.
6. Click **OK**.

   The EPPC New Project Wizard starts and displays the **Linker** page. (See Figure 2.9.)

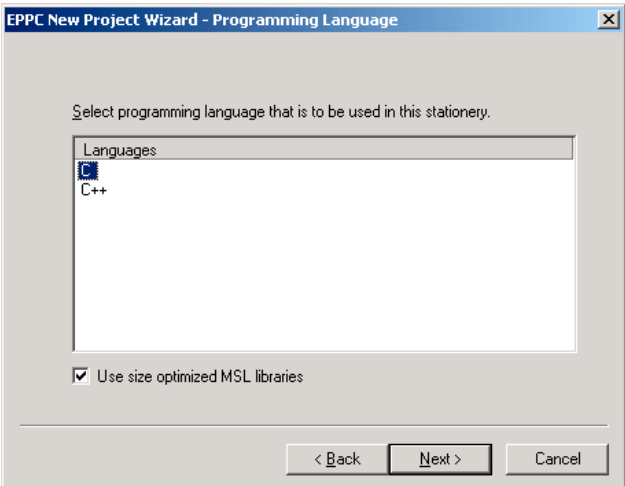
Figure 2.9 EPPC New Project Wizard — Linker Page



7. From the Linkers list box, select `EPPC Linux GNU Linker`.
8. Click **Next**.

   The wizard displays the **Application & Language** page. (See Figure 2.10.)

Figure 2.10 EPPC New Project Wizard — Application & Language Page



9. From the Applications list box, select the type of application you want to create.

10. From the Languages list box, select the programming language you plan to use.

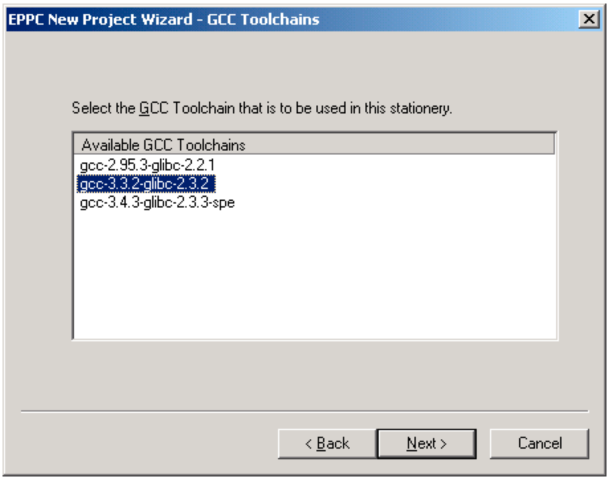
    For example, if you plan to use the C language in your source code files, select `C`.

NOTE The language you select determines the libraries with which the new project links and the contents of the main source file. If you select the C++ language, you can still add C source files to the project (and vice versa).

11. Click **Next**.

    The wizard displays the **GCC Toolchains** page. (See Figure 2.11.)

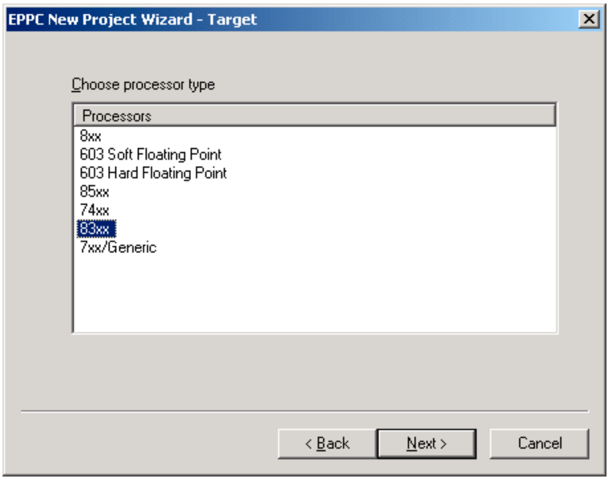
Figure 2.11 EPPC New Project Wizard — GCC Toolchains Page



12. From the Available GCC Toolchains list box, select the GCC toolchain for the new project to use.
13. Click **Next**.

    The wizard displays the **Target** page. (See Figure 2.12.)

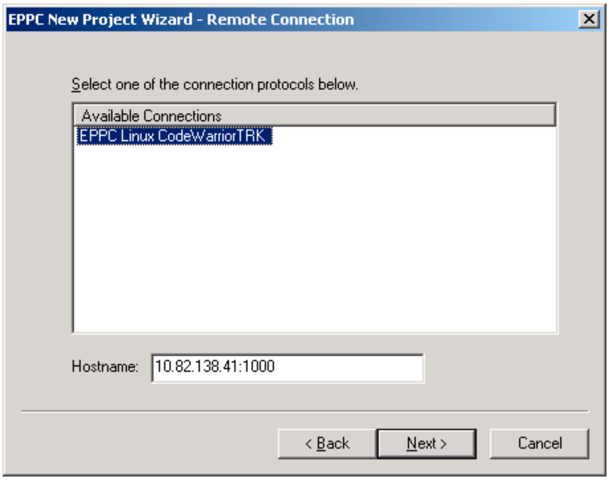
Figure 2.12 EPPC New Project Wizard — Target Page



14. From the Processors list box, select the processor family to which the processor on your target board belongs.
15. Click **Next**.

    The wizard displays the **Remote Connection** page. (See Figure 2.13.)

Figure 2.13 EPPC New Project Wizard — Remote Connection Page



16. From the Available Connections list box, select `EPPC Linux CodeWarriorTRK`.

    The Hostname text box activates.

17. In the Hostname text box, type the IP address assigned to your target board, followed by a colon, followed by the port number on which (running on the board) will listen for CodeWarrior debugger connections.

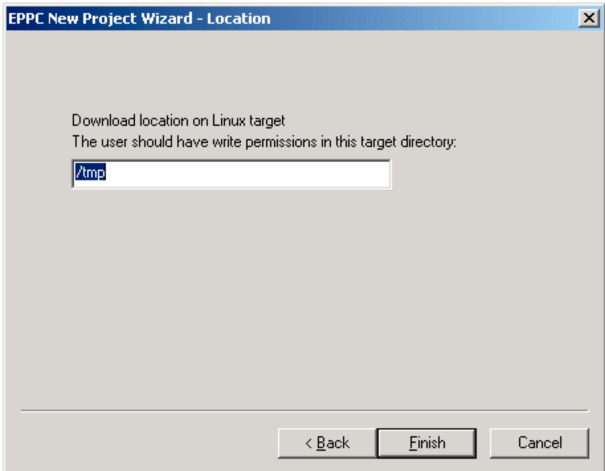
    For example, you would enter this: `10.82.191.3:1000`

    if the IP address of your board is `10.82.191.3` and will listen on port `1000` for debugger connections.

18. Click **Next**.

    The wizard displays the **Location** page. (See Figure 2.14.)

Figure 2.14 EPPC New Project Wizard — Location Page

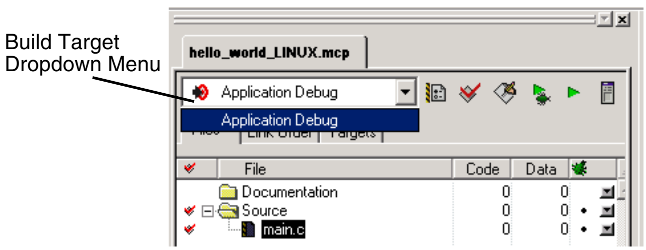


19. In the text box, type the path of the desired download location on the target board.

20. Click **Finish**.

    The wizard creates a project according to your specifications and displays a project window docked to the left, top, and bottom of the IDE's main window. (See Figure 2.15.)

Figure 2.15 Project Window for the hello_world_Linux Project



That's it—the new project is ready for use. You can now customize it by adding your own source code files, changing target settings, adding libraries, etc.

See Tutorial: Debugging an Embedded Linux® Application for instructions that explain how to make and debug this project.

Using the External Build Wizard

The External Build Wizard creates a CodeWarrior project that lets the CodeWarrior IDE manipulate an external project built by an external make system.

Create a project with the External Build Wizard if you have an existing project that is built by an external make system and would like to use some of the CodeWarrior IDE's powerful features (such as the debugger) with this project.

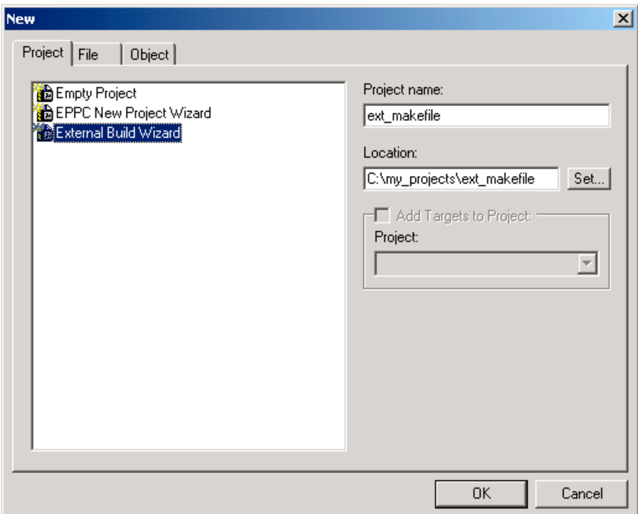
NOTE The External Build Wizard does not "import" the information within an external make file; instead, the wizard creates a CodeWarrior project that invokes the specified external make utility on the specified make file.

To use the External Build Wizard, follow these steps:

1. Start the CodeWarrior IDE.
2. From the IDE's menu bar, select **File > New**.

   The **New** dialog box appears. (See Figure 2.16.)

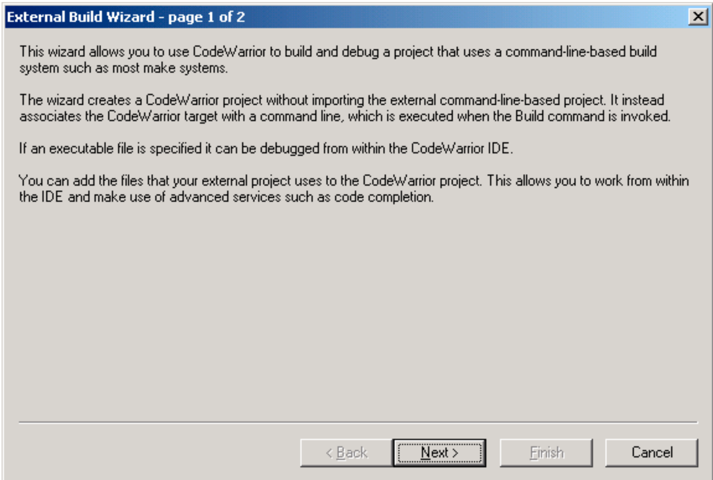
Figure 2.16 New Dialog Box



3. Select External Build Wizard.
4. In the Project Name text box, type `ext_makefile`.
5. In the Location text box, type the location where you want to save this project, or click **Set** to use a standard file dialog box to select a location.
6. Click **OK**.

   The External Build Wizard starts and displays its first page. (See Figure 2.17.)

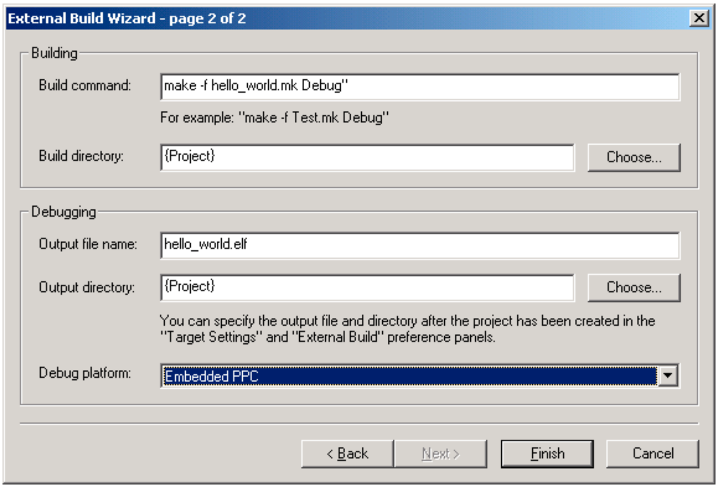
Figure 2.17 External Build Wizard — Page 1 of 2



7. Read the information on the first wizard page.
8. Click **Next**.

   The External Build Wizard displays its second page. (See Figure 2.18.)

Figure 2.18 External Build Wizard — Page 2 of 2



9. In the Build command text box, type the command-line string you enter at the command prompt to build the project.

   For example, if your project uses the Cygwin `make` utility, you might enter this:

   ```
   make -f hello_world.mk Debug
   ```

10. In the Build directory text box, type the path in which the make file (entered in the previous step) resides.

    Alternatively, click **Browse** to select the build directory path using the **Browse for Folder** dialog box.

TIP You can use any of the built-in CodeWarrior symbolic paths (such as `{Project}`) for the Build directory path.

11. In the Output file name text box, type the root file name of the executable that the CodeWarrior project (not the make file) will generate.

NOTE This file name does not have to match the name of the file generated by the external make file.

12. In the Output directory text box, type the path to which you want the new CodeWarrior project to write the output file specified in the previous step.

    Alternatively, click **Browse** to select the output directory using the **Browse for Folder** dialog box.

TIP You can use any of the built-in CodeWarrior symbolic paths (such as `{Project}`) for the Output directory path.

13. From the Debug platform dropdown menu, select Embedded PPC.
14. Click **Finish**.
15. The wizard displays its **Summary** page.
16. Click **Generate**.

    The External Build Wizard creates a CodeWarrior project consisting of a single build target that executes the specified Build command line each time you invoke one of the IDE's build commands (for example, Make and Debug).

That's it. You have created a CodeWarrior project that builds a binary using an external make utility.

Now, you should add the source files referenced by the external make file to the new CodeWarrior project. Doing so lets you take advantage of more IDE features, such as code completion.

To learn more about the External Build Wizard, see the *CodeWarrior™ IDE User's Guide* This document is in this folder:

`installDir\Help\PDF\`

Using the Empty Project Template

Finally, you can create a project "by-hand." To do this, choose Empty Project from the EPPC New Project Wizard. The result is a project that contains no files and only the most obvious target settings choices.

Create an empty project if you want control over everything. See the *CodeWarrior™ IDE User's Guide* and the Target Settings Reference chapter of this manual for the information you will need to choose the target settings your new project requires.

To make the empty project approach easier, your CodeWarrior product includes template source code files for each supported board. To get started faster, you can add the files from the appropriate template directory to your empty project.

The EABI template source code files are here:

`installDir\Templates\PowerPC_EABI\Sources`

Beneath the `Sources` directory, there is one directory for each supported board.

NOTE Some template source files are stubs; you must replace them with full implementations of your own.

3

Target Settings Reference

This chapter documents the target settings panels that are specific to the CodeWarrior™ Development Studio for Power Architecture™ Processors product. Use these panels to control the behavior of the compiler, linker, debugger, and other software development tools included in this CodeWarrior product.

NOTE For documentation of the target settings panels common to all CodeWarrior products, refer to the *IDE User's Guide* and the *Power Architecture™ Build Tools Reference*.

The sections of this chapter are:

- Working with Target Settings
- General Purpose Target Settings Panels
- Power Architecture™-specific Target Settings Panels
- PC-lint Target Settings Panels

Working with Target Settings

This section explains what target settings are and shows you how to change, restore, and save a copy of them.

What are Target Settings?

A CodeWarrior project contains one or more *build targets*. A build target is a named collection of files and settings that the CodeWarrior IDE uses to generate an output file.

A *platform target* is an operating system or processor with which the output file generated by a build target is compatible. For example, a build target might generate an executable and linkable (ELF) format file that the Linux® operating system can execute. In this example, Linux is the platform target.

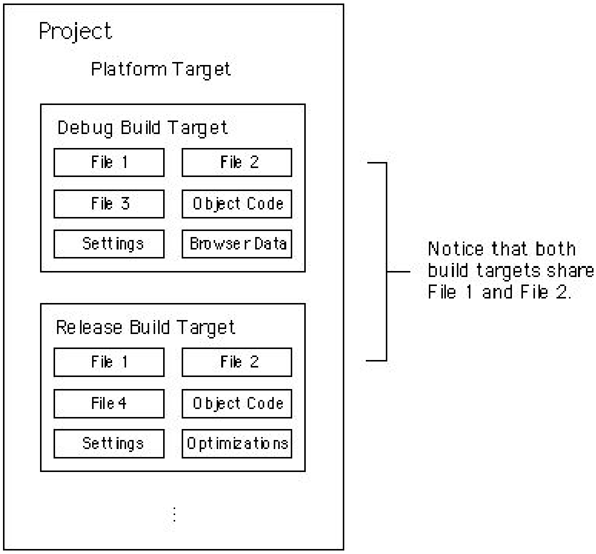
A build target contains all build-specific *target settings*. Target settings define:

- The files that belong to a build target.
- The behavior of the compiler, assembler, linker, and other build tools.

The build target feature lets you create different versions of your program for different purposes. For example, you might have a *debug* build target. This build target would include no optimizations so it is easy to debug. You might also have a *release* build target. This build target would be heavily optimized so it uses less memory or runs faster.

Figure 3.1 shows how a CodeWarrior project, a platform target, and build targets relate.

Figure 3.1 Relationship between a Project, a Platform Target, and Build Targets



Changing Target Settings

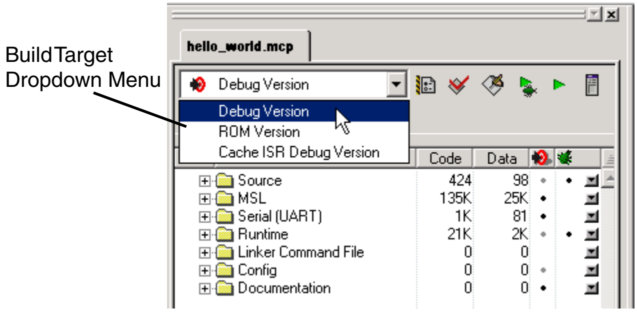
If you create a project using the New Project Wizard, the wizard sets the target settings of each build target in the project to reasonable defaults. That said, you may need to change some of them.

To change a build target's target settings, you use the **Target Settings** window. These steps show you how:

1. Start the CodeWarrior IDE.
2. Open the project that contains the build target to be modified.

   The IDE displays the project in a project window (docked to the left and bottom of the IDE's main window).
3. From the build target dropdown menu of the project window, select the build target that you want to modify. (See Figure 3.2.)

Figure 3.2 Project Window Showing the Selection of a Build Target



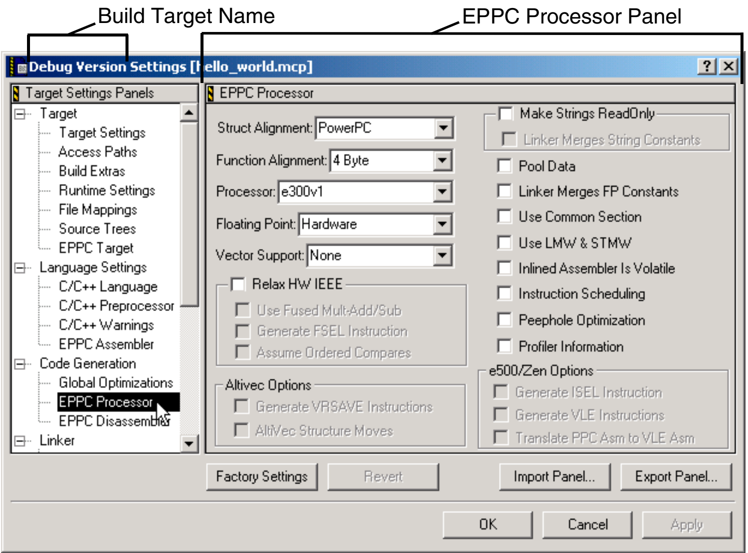
4. Press **ALT-F7**

   The **Target Settings** window appears. (See Figure 3.3.)

NOTE In the sentence above, the word *Target* is in italics because it is a placeholder for the name of the current build target. For example, in Figure 3.3, the string `Debug Version` appears in place of *Target*.

The settings you make in the panels of the **Target Settings** window apply to the project build target currently selected.

Figure 3.3 *Target* Settings Window Showing the EPPC Processor Target Settings Panel



On the left side of the **Target Settings** window is the Target Settings Panels list. This list contains the name of each target settings panel available for the current build target. Your selections for Linker, Pre-linker, and Post-linker in the Target Settings panel determine the panel names in this list.

5. In the Target Settings Panels list, click a target settings panel name.

   The selected panel appears in the right side of the **Target Settings** window.

   Figure 3.3 shows the EPPC Processor target settings panel on the right side of the **Target Settings** window.

6. Change the settings in the displayed panel as dictated by the build target's purpose.
7. Click **Apply**.

   The IDE saves your new settings.

8. In the Target Settings Panels list, click a different target settings panel name.

   The selected panel replaces the EPPC Processor panel.

9. Again, change the settings in the panel as dictated by the build target's purpose.
10. Click **Apply**.

    The IDE saves your new settings.

11. Continue this process for each target settings panel until you have made all settings your build target requires.

12. When you are done making settings, click **OK**.

    The IDE saves your settings and closes the **Target Settings** window.

Restoring Target Settings

If you change any of a build target's settings, you can recover the original values.

To restore a build target's original settings, use one of these methods:

- To restore the previous settings, click the **Revert** button at the bottom of the **Target Settings** window.
- To restore the factory default settings, click the **Factory Settings** button at the bottom of the **Target Settings** window.

Importing/Exporting Target Settings

If you want to save any of a build target's settings, you can export them and save them. You can also import a build target's predefined settings.

To export or import a build target's settings, use these methods:

- To export a build target's settings, click the **Export Panel** button at the bottom of the **Target Settings** window.
- To import the predefined target settings, click the **Import Panel** button at the bottom of the **Target Settings** window.

Making a Copy of a Project

Once you have made the required settings for each build target of a project, you might want to make a copy of the project before changing it any further. Doing so lets you create a project template that you and others can use as a starting point for new projects.

To create a project template, follow these steps:

1. Create a project.
2. For each build target in the project, change the target settings as desired.
3. Select **File > Save a Copy As**.

   The **Save a copy of project as** dialog box appears.
4. Use this dialog box to save a copy of the current project on your hard disk or on a network disk (if you want others to be able to use the project template).

General Purpose Target Settings Panels

Some target settings panels are needed for all development done with a CodeWarrior product, no matter what the product. Other panels are specific to the CodeWarrior for Power Architectures product.

Table 3.1 lists each target settings panel that is *not* Power Architecture-specific and identifies the manual that documents the panel.

Table 3.1 General Purpose Target Settings Panels

Target Settings Panel	Description
CodeWarrior™ IDE User's Guide	
Access Paths	Use this panel to define the list of directories that the build tools search for files, such as include files.
Build Extras	Use this panel to select options that affect the performance of the software development tools. In addition, use this panel to set up a third-party debugger.
Runtime Settings	Use this panel to supply information, such as command-line arguments, that your program needs when run under control of the CodeWarrior debugger.
File Mappings	Use this panel to associate a file extension with a tool designed to manipulate files that have that extension.
Source Trees	Use this panel to define aliases for paths that change from one developer's workstation to another's. Using source trees makes it easier to share a project.
External Build	Use this panel to configure a build target to use an external make file to build the target's output file.
Global Optimizations	Use this panel to define the optimizations the compiler performs.
Custom Keywords	Use this panel to define up to four sets of custom keywords along with the color the editor uses for each.
Other Executables	Use this panel to define the list of other projects and executable files for the debugger to use in addition to the executable generated by the current build target.

Table 3.1 General Purpose Target Settings Panels (*continued*)

Target Settings Panel	Description
Debugger Settings	Use this panel to configure the general (that is, not EPPC-specific) behavior of the debugger.
Remote Debugging	Use this panel to select and configure the connection that the CodeWarrior debugger uses to communicate with the target board or simulator. For multi-core debugging, use this panel to specify the index of the core to which the debugger should download the executable generated by the current build target.
Power Architecture™ Build Tools Reference	
C/C++ Language	Use this panel to control how the compiler handles certain C/C++ language features as well as certain object code storage features.
C/C++ Preprocessor	Use this panel to control the operation of the CodeWarrior compiler's preprocessor.
C/C++ Warnings	Use this panel to control the warning messages the CodeWarrior C/C++ compiler issues.

Power Architecture™-specific Target Settings Panels

This section explains the purpose and effect of each setting in the target settings panels that are specific to the CodeWarrior for Power Architecture Processors product.

Table 3.2 lists and briefly describes each Power Architecture-specific target settings panels. In addition the table shows which panels are available when you select a particular linker, pre-linker, or post-linker.

Table 3.2 Power Architecture™-Specific Target Settings Panels

Target Settings Panel	Description
All Linkers	
Target Settings	Use this panel to define the name of the current build target, and the linker, pre-linker, post-linker, and output directory this build target uses.

Table 3.2 Power Architecture™-Specific Target Settings Panels (*continued*)

Target Settings Panel	Description
Debugger PIC Settings	Use this panel to specify an alternate address for the debugger to load a PIC module on the target.
Source Folder Mapping	Use this panel to enable source-level debugging when debugging a binary that was built in one place, but which is being debugged from another.
System Call Service Settings	Use this panel to activate debugger support for system services and to select options for handling requests for system services.
PowerPC EABI Linker	
OSEK Sysgen	Use this panel to configure the behavior of the OSEK sysgen utility. **NOTE:** You should have the CodeWarrior OSEKturbo Sysgen tool installed on your machine to use this panel.
EPPC Target	Use this panel to specify the name the linker gives to the final output file generated by the current build target. In addition, use the panel to define compiler and linker options such as the version of DWARF debugging information generated and the ABI or code model used.
EPPC Assembler	Use this panel to define the syntax allowed in assembly language source code files.
EPPC Processor	Use this panel to make processor-specific code generation settings.
EPPC Disassembler	Use this panel to control the information included in the results of a disassembly.
EPPC Linker	Use this panel to select options related to linking object code into its final form.
EPPC Linker Optimizations	Use this panel to configure the bare board linker's code merging feature.
EPPC Debugger Settings	Use this panel to provide information the debugger needs to work with the target and to define how and when the debugger downloads portions of your binary to the target.

Table 3.2 Power Architecture™-Specific Target Settings Panels (*continued*)

Target Settings Panel	Description
EPPC Exceptions	Use this panel to define the processor exceptions that the CodeWarrior debugger will handle. **NOTE:** this panel applies only to processors that have a BDM debug module.
EPPC Trace Buffer	Use this panel to configure the trace events you want to capture while debugging a target equipped with a trace buffer.
EPPC Linux GNU Linker	
GNU Target	Use this panel to select a project type, define the name of the build target's final output file and, for shared libraries, to define the library's SONAME.
GNU Assembler	Use this panel to specify command-line arguments to be passed to the GNU assembler.
GNU Disassembler	Use this panel to specify command-line arguments to be passed to the GNU disassembler.
GNU Compiler	Use this panel to specify command-line arguments to be passed to the GNU compiler
GNU Linker	Use this panel to specify command-line arguments to be passed to the GNU linker.
GNU Environment	Use this panel to define environment variables that the GNU build tools can reference.
GNU Tools	Use this panel to specify the path to the GNU build tools and to define the particular tools the IDE uses.
Console I/O Settings	Use this panel to define the locations to which `stdin`, `stdout`, and `stderr` are redirected when a Linux application is run under control of the debugger.
Debugger Signals	Use this panel to define how CodeWarrior TRK (also known as) handles Linux signals on behalf of the debugger.
BatchRunner PreLinker	
BatchRunner PreLinker	Use this panel to specify the batch file that the BatchRunner PreLinker runs.

Table 3.2 Power Architecture™-Specific Target Settings Panels (*continued*)

Target Settings Panel	Description
BatchRunner PostLinker	
BatchRunner PostLinker	Use this panel to specify the batch file that the BatchRunner PostLinker runs.
EPPC Linux GNU PostLinker	
GNU Post Linker	Use this panel to specify command-line arguments to be passed to the GNU post-linker utility.

Target Settings

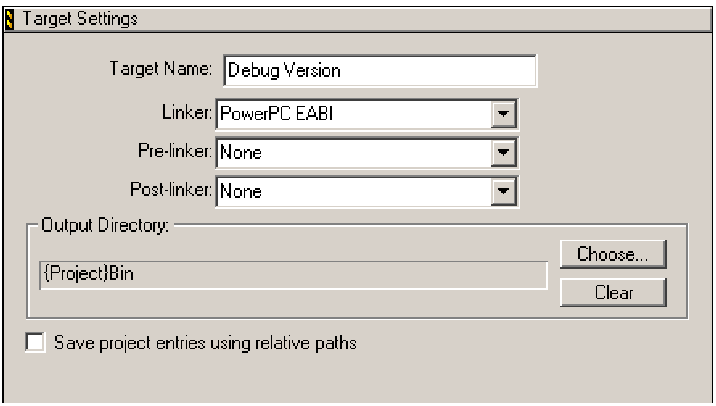
The **Target Settings** panel is the most important target settings panels. This is the panel where you select the linker, pre-linker, and post-linker a build target uses. These choices, in turn, define which target settings panels appear in the **Target Settings** window's panel list.

As your linker, pre-linker, and post-linker choices determine the availability of other target settings panels, always make these choices first.

NOTE The **Target Settings** panel is *not* the same as the EPPC Target panel. You select a linker in the **Target Settings** panel; you select other target-specific options in the EPPC Target panel.

Figure 3.4 shows the **Target Settings** panel.

Figure 3.4 Target Settings Panel



Target Name

Use the Target Name text box to assign a name to the a build target. The name you specify appears in the project window's build target dropdown menu and in this window's **Targets** tab.

NOTE Target name is the name of the current build target, *not* the name of the file this build target generates. You specify a build target's output file name in the Output File Name text box of the EPPC Target panel.

Linker

Use the Linker dropdown menu to select the linker a build target uses. The choices are:

- None

  Choose this option if for a build target that generates no binary. For example, when you create a CodeWarrior project just so you can debug and existing binary (such as U-Boot), you would select None for linker.

- External Build Linker

  Choose this option to configure a build target to use an external make file to build the target's output file.

  If you select this linker, the External Build panel appears in the left pane of the **Target Settings** window. See the *IDE User's Guide* for instructions that explain how to use this target settings panel.

  Also, see Using the External Build Wizard for instructions that explain how to use a wizard to create a project whose build targets use an external make file.

- EPPC Linux GNU Linker

  Choose this linker to configure a build target to generate a file in Executable and Linkable (ELF) format for execution on the embedded Linux operating system.

- PowerPC EABI

  Choose this linker to configure a build target to use Freescale's PowerPC EABI linker to generate a file in Executable and Linkable (ELF) format for execution on a bare board.

- PCLint Linker

  Choose this option to configure a build target to use PC-lint to check your C/C++ source code files for bugs, inconsistencies, and non-portable constructs.

  PC-lint is a third-party software development tool developed by Gimpel Software (www.gimpel.com). As a result, you must obtain and install a copy of PC-lint before a CodeWarrior build target can use this tool.

NOTE The sections below document the panels used by the “real” linkers, that is, linkers that generate a binary. See PC-lint Target Settings Panels for documentation of panels used by the PCLint “linker.”

Pre-linker

A pre-linker is a tool that performs its work immediately before the linker runs.

Use the Pre-linker dropdown menu to select the pre-linker the current build target uses. The choices are:

- None

  Use no pre-linker.

- BatchRunner PreLinker

  If you select the BatchRunner PreLinker, a new panel, named BatchRunner PreLinker, appears in the left panel of the **Target Settings** window. Use this panel to select the Windows® batch file for the pre-linker to run.

Post-linker

A post-linker is a tool that performs its work immediately after the linker runs.

Use the Post-linker dropdown menu to select the post-linker the current build target uses. The choices are:

- None

  Use no post-linker.

- BatchRunner Postlinker

  If you select this pre-linker, the BatchRunner PostLinker panel appears in the left pane of the **Target Settings** window. Use this panel to select the Windows® batch file for the post-linker to run.

- EPPC Linux GNU Postlinker

  If you select the this pre-linker, the GNU Post Linker panel appears in the left pane of the **Target Settings** window. Use this panel supply the command-line switches to pass to the program specified Post Linker text box of the GNU Tools panel.

NOTE The Post-linker dropdown menu contains the EPPC Linker GNU Postlinker option only if you select the EPPC Linux GNU Linker from the Linker dropdown menu.

Output Directory

This read-only text box contains the path in which the linker places a build target's final output file (application, library, etc.)

The `{Project}` directory is the default output directory.

Click **Choose** to display a dialog box that lets you select the desired output path.
Click **Clear** to restore the default directory (the project directory).

Save Project Entries Using Relative Paths

Check this box to instruct the IDE to save the relative path of each file in a build target along with the root file name of the file.

If this box is checked, you can add two or more files that have the same name to a project. This is so because, when searching for files, the IDE prepends the directory names in the **Access Paths** target settings panel to the relative path of each project file, thereby producing a unique filename.

If this box is unchecked, each file in a project must have a unique name because, when searching for files, the IDE combines the directory names in the **Access Paths** panel with just the root filename of each project file. As a result, the IDE cannot discriminate between two files that have the same name but different relative paths.

OSEK Sysgen

Use the **OSEK Sysgen** settings panel to control the output of the OSEK Sysgen tool. OSEK (Open Systems and their Interfaces for the Electronics in Motor Vehicles) is a standards body that has produced specifications for an embedded operating system, a communications stack, and a network management protocol for automotive embedded systems. OSEK System Generator (Sysgen) is a special tool for system generation which reads the operating system configuration file and configures the OS.

NOTE OSEK Sysgen can be used for only 52xx projects.

When you build a CodeWarrior build target that contains an object implementation language (OIL) file, the OSEK Sysgen tool compiles the OIL file and generates C language files used in the generation of an OSEK operating system image as well as other types of files. The OSEK Sysgen panel lets you define the names, locations, and other attributes of these files.

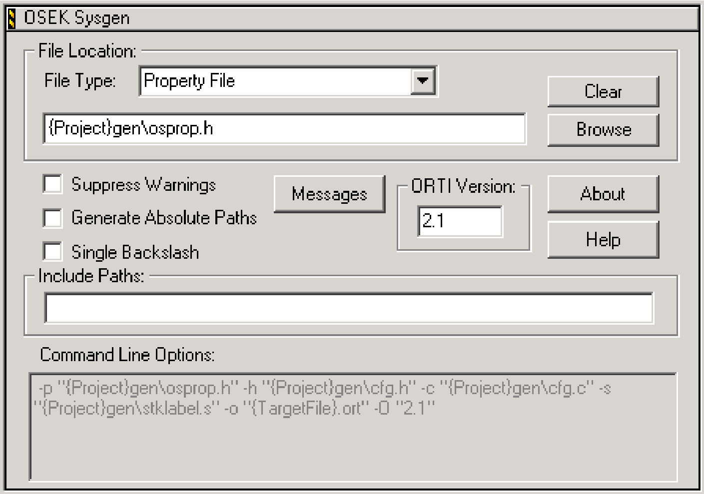
Next, the CodeWarrior C compiler compiles the generated C language files, the OSEK operating system's source code, and any application source code files the build target contains. Finally, the CodeWarrior linker links the resulting object code into an executable OSEK operating system image that contains your application.

Compilation of the OSEK operating system source code depends on the definition of several macros; the OSEK Sysgen tool helps with these macro definitions. Specifically, the tool generates file `options.h`, *which you must include in your build target's prefix file*. The tool also defines macros `APPTYPESH`, `OSPROPH`, and `OSCFGH`, extracting macro values from corresponding user types, property, and object-declaration files.

NOTE We recommend that you not edit the generated files. Doing so may lead to data inconsistency, compilation errors, or unpredictable application behavior.

Figure 3.5 shows the **OSEK Sysgen** settings panel.

Figure 3.5 OSEK Sysgen Panel



File Type

Use the File Type dropdown menu to select the type of file referenced by the File Location text box. Table 3.3 lists and describes each File Type option.

Table 3.3 OSEK Sysgen File Type Options

Option	Description
Property File	Select this file type so you can specify the path and name of an OSEK property file in the related text box. The property file is a C language header file that describes the current configuration of the operating system — in other words, system properties. This file contains the preprocessor directives `#define` and `#undef` and is used at compile-time to build the OS kernel with the specified properties. The default is `{Project}gen\osprop.h`, but you can use another path and name.
Objects Declaration File	Select this file type so you can specify the path and name of an OSEK objects declaration file in the related text box. The objects declaration file is a header file that contains definitions of data types, constants, and external declarations of variables needed to describe system objects. The default is `{Project}gen\cfg.h`, but you can use another path and name.
Objects Definition File	Select this file type so you can specify the path and name of an OSEK objects definition file in the related text box. The objects definition file is a source file that contains initialized data and allocates memory for system objects. The default is `{Project}gen\cfg.c`, but you can use another path and name.
Stack Labels File	Select this file type so you can specify the path and name of an OSEK stack labels file in the related text box. The stack labels file defines labels for the bottom and top of the stack for extended tasks implemented in the OSEK OS. You can see these labels in the debugger during application execution and can use them for dynamic control of task stack usage. The default is `{Project}gen\stklabel.s`, but you can use another path and name.

Table 3.3 OSEK Sysgen File Type Options

Option	Description
ORTI File	Select this file type so you can specify the path and name of an OSEK ORTI (OSEK Run Time Interface) file in the related text box. The ORTI file contains internal OSEK operating system data, which is available to an ORTI Aware Debugger. The debugger can display and update the system object information in the ORTI file. The default path and filename is the same as the path and name of the `.abs` file, but you can use another name.
Sysgen Tool	Select this file type so you can specify the path to and name of the OSEK Sysgen utility. This utility processes a OIL file. If you do not define the location of the Sysgen utility, the IDE looks for this information in the Windows® registry. If the registry does not contain this information, the IDE next looks at the `PATH` environment variable. The default is `{Compiler}osek\shared\bin\sysgen.exe`, but you can use another path and name.
Sysgen Command Line File	Select this file type so you can specify the path to and name of the OSEK Sysgen command-line file in the related text box. The Sysgen command-line file contains additional command-line options for the Sysgen utility. Use of the command-line file is optional and is intended for advanced users. There is no default filename for this file type.
User Types File	Select this file type so you can specify the path to and name of the OSEK user types file in the related text box. The user types file contains definitions of a users' message types. Also, the file defines the macro `APPTYPESH` equal to the location of this file. The default is `{Project}Sources\usertypes.h`, but you can use another path and name.
Prefix File Path (for `option.h`)	Select this file type so you can specify the path in which the Sysgen utility writes the prefix file (`options.h`) in the related text box. If you do not specify a path, `options.h` file is put in the `{Project}` directory. The `options.h` file contains macro definitions that other OS files use to find other generated OSEK configuration files. You must include `options.h` in a build target's prefix file or as an additional include file in the compiler's configuration. The default is `{Project}gen`, but you can use another path.

File Location

Use the File Location text box to type the path to and name of a file of the type currently selected in the File Type dropdown menu. Alternatively, click **Browse** to display a dialog box with which you can to navigate to and find this file.

Suppress Warnings

Check the Suppress Warnings checkbox so the OSEK Sysgen utility does not display warning and informational messages in the log window.

Generate Absolute Paths

Check the Generate Absolute Paths box if the file location macros in `options.h` be assigned absolute paths.

Uncheck this box, if these macros must be assigned relative paths.

Single Backslash

Check the Single Backslash checkbox to use a single backslash as the path delimiter in the paths assigned to the file location macros in `options.h`.

Uncheck this box to use two backslash characters as the path delimiter in the paths assigned to the file location macros in `options.h`.

Messages

Click the Messages button to display the **Suppress Messages** dialog box. Use this dialog box to define the warning messages and informational messages you want suppressed.

Clicking **Disable All** is equivalent to checking the Suppress Warnings checkbox.

Clicking **Enable All** is equivalent to unchecking the Suppress Warnings checkbox.

ORTI Version

Use the ORTI Version text box to enter the supported ORTI (OSEK Run Time Interface) version.

About Button

Click the **About** button to display a dialog box containing OSEK Sysgen utility version information.

Help Button

Click the **Help** button to display an online help window. This window contains information that explains how to use the OSEK Sysgen utility and the **OSEK Sysgen** target settings panel.

Include Paths

Use the Include Paths text box to enter directories for the OSEK Sysgen utility to search for include OIL files. Separate each directory path with a comma or a semicolon.

Command Line Options

The Command Line Options read-only text box displays all system generation options currently defined.

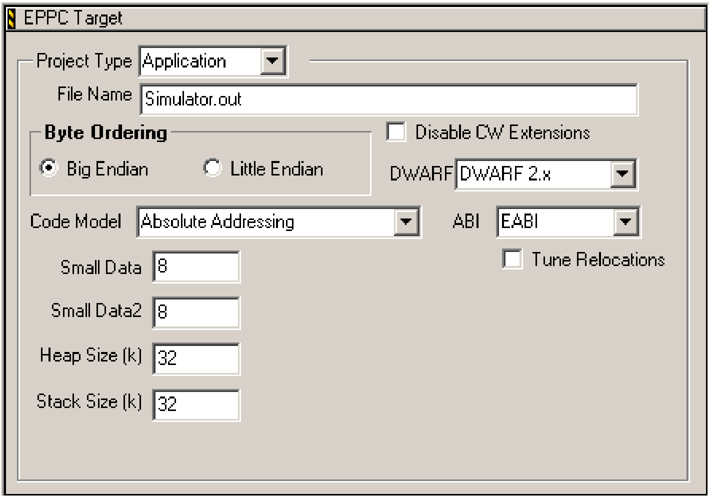
EPPC Target

Use the **EPPC Target** settings panel to specify the name the linker assigns to the final output file (application, library, etc.) generated by the current build target. In addition, use this panel to tell the linker how to structure this file.

In addition, use the panel to define compiler and linker options, such as the version of DWARF debugging information generated and the ABI or code model used.

Figure 3.6 shows the **EPPC Target** settings panel.

Figure 3.6 EPPC Target Panel



Project Type

Use the Project Type dropdown menu to define the kind of project that the build target creates. The options are:

- Application
- Library
- Partial Link

The project type you choose determines which other items appear in this panel.

If you choose Library or Partial Link, the Heap Size, Stack Size, and Tune Relocations items disappear because they are not relevant. The Partial Link item lets you generate a relocatable output file that a dynamic linker or loader can use as input. If you choose Partial Link, the items Optimize Partial Link, Deadstrip Unused Symbols, and Require Resolved Symbols appear in the panel.

File Name

Use the File Name text box to define the name of the application or library a build target creates.

By convention, application names should end with the extension `.elf`, and library names should end with the extension `.a`.

If the build target is configured to generate an S-Record file and/or a map file, and the in the File Name text box ends in `.elf` or `.ELF`, this extension is stripped and `.mot` is appended to the S-Record file name and `.MAP` is appended to the map file name.

Byte Ordering

Use the option buttons in the Byte Ordering group box to select big-endian or little-endian byte ordering. The Big-endian option generates object code and links an executable image that uses big-endian byte ordering. This is the default setting for the compiler and linker. The little-endian option generates object code and links an executable image that uses little-endian byte ordering.

If you choose big endian byte ordering, within a given multi-byte numeric representation, the most significant byte has the lowest address (the word is stored "big-end-first"). Listing 3.1 shows how the value `0x0A0B0C0D` is stored in memory if big-endian byte ordering is chosen.

Listing 3.1 Big-Endian Byte Ordering

```
Memory Address: 0x1000      0x1001     0x1002    0x1003
Byte:             0xA         0xB        0xC       0xD
```

If you choose little endian byte ordering, within a given multi-byte numeric representation, bytes at lower addresses have lower significance (the word is stored "little-end-first"). Listing 3.2 shows how the value `0x0A0B0C0D` is stored in memory if little-endian byte ordering is chosen.

Listing 3.2 Little-Endian Byte Ordering

```
Memory Address: 0x1000      0x1001      0x1002     0x1003
Byte:             0xD         0xC         0xB        0xA
```

NOTE You can create little endian project from this panel, but debugging such a project is not supported.

Disable CW Extensions

Check the Disable CW Extension box if you are using a third-party linker, and it cannot link object files generated by the CodeWarrior C/C++ compiler.

Object modules generated from C-language files compiled by the CodeWarrior compiler contain extra information that lets the CodeWarrior linker eliminate unused code, data, and DWARF symbols. This feature is called *deadstripping*.

Most third-party linkers have no problem with the extra information the CodeWarrior compiler puts in its object modules (although they do not use it to perform deadstripping).

That said, if a third-party linker issues errors, the errors might go away if you:

- Check the Disable CW Extensions box
- Recompile all your C language source code files
- Relink

NOTE Even if Disable CW Extensions is checked, the compiler may generate some sections that a third-party linker cannot handle. In particular, if the Enable C++ Exceptions box of the C/C++ Language panel is checked, the compiler generates the `.extab` and `.extabindex` sections. If, after checking Disable CW Extensions, your link still fails, try unchecking Enable C++ Exceptions.

DWARF

Use the DWARF dropdown menu to select the version of the Debug With Arbitrary Record Format (DWARF) debugging information format the compiler and assembler generate. If in doubt about the DWARF version to use, you can use the default setting of DWARF 2.x.

The linker ignores debugging information that is not in the selected format.

ABI

Use the ABI dropdown menu to select the Application Binary Interface (ABI) the compiler and assembler use for function calls and structure layout. For more information on the Application Binary Interface (ABI), see the Embedded Power Architecture API Programming Information topic.

Tune Relocations

The tune relocations option pertains to object relocation and is available for just these application binary interfaces:

- EABI
- SDA PIC/PID

NOTE The Tune Relocations checkbox appears only if you select Application from the Project Type dropdown menu.

Check the Tune Relocations checkbox when you receive link warning about out of range relocations. Checking the Tune Relocations checkbox has these effects:

- For the EABI application binary interface, a 14-bit branch relocation is converted to a 24-bit branch relocation only if the 14-bit relocation cannot reach the calling site from the original relocation.
- For the SDA PIC/PID application binary interface, the absolute addressed references of data from code are changed to use a small data register (such as `r13`) instead of `r0`; absolute code is changed to code references to use the PC relative relocations.

Code Model

Use the Code Model dropdown menu to select the addressing mode for the binary generated by the current build target.

The options are:

- Absolute Addressing

  Select to instruct the build tools to generate a non-relocatable binary.

- SDA Based PIC/PID Addressing

  Select to instruct the build tools to generate a relocatable binary that uses position-independent-code (PIC)/position-independent-data (PID) addressing. The resulting binary can be loaded at any address.

Small Data

Use the Small Data text box to specify the threshold size (in bytes) for an item to be considered small data by the linker. The linker stores small data items in the Small Data address space.

Data in the Small Data address space can be accessed more quickly than data in the "normal" address space.

Small Data2

Use the Small Data2 text box to specify the threshold size (in bytes) for an item to be considered small data by the linker. The linker stores read-only small data items in the Small Dasta2 address space.

Constant data in the Small Data2 address space can be accessed more quickly than data in the "normal" address space.

Heap Size

Use the Heap Size text box to define the amount of memory (in kilobytes) the build tools allocate for the heap. The heap is used when your program calls `malloc` or `new`. You can define the address of the heap segment in a linker command file or in the EPPC Linker target settings panel.

NOTE Heap size does not apply to libraries; only applications have a heap.

Stack Size

Use the Stack Size text box to define the amount of memory (in kilobytes) the build tools allocate for the stack. You can define the address of the stack segment in a linker command file or in the EPPC Linker target settings panel.

NOTE Stack size does not apply to libraries; only applications have a stack.

NOTE Consider the amount of RAM your target has, as you choose heap and stack size. If you allocate too much RAM to the heap and stack, your program may run out of memory; if you allocate to little RAM to heap and stack, your program might run out of these critical resources.

Optimize Partial Link

Check the Optimize Partial Link checkbox to instruct the linker to perform final work on a partial link.

An unoptimized partial link (also known as a partial link without qualifiers) is a relocatable file that can be linked again just like any `.o` file. An optimized partial link is almost the same as a unoptimized partial link except that the linker creates the symbols `_ctors` and `_dtors`. A loader needs these symbols to initialize C++ exceptions and static constructors after the loader relocates the file.

The linker generates four symbols:

- `__ctors` — an array of static constructors
- `__dtors` — an array of destructors
- `__rom_copy_info` — an array of a structure that contains all of the necessary information about all initialized sections to copy them from ROM to RAM
- `__bss_init_info` — a similar array that contains all of the information necessary to initialize all of the bss-type sections.

NOTE The Optimize Partial Link checkbox is available only if you select Partial Link from the Project Type dropdown menu.

Enabling this option instructs the linker to:

- Use a linker command file (LCF)

  The commands in an LCF let you merge the sections of your program into the `.text`, `.data`, or `.bss` segment. If you do not use an LCF to perform this merge, the CodeWarrior debugger will probably not display the application's source code correctly.

- Perform deadstripping

  Deadstripping is strongly recommended.

NOTE An application must have at least one entry point for the linker to be able to deadstrip it.

- Collect all static constructors and destructors in a way similar to the `munch` utility.

NOTE Refer to any Unix documentation for an explanation of the `munch` utility.

NOTE It is essential that you not use `munch` yourself because the linker must put C++ exception handling initialization code in the first constructor.
If you see `munch` in a make file that you are importing into the CodeWarrior IDE, it is a clue that you need an optimized build, that is, that you need to enable the Optimize Partial Link option.

- Change common symbols to `.bss` symbols.

  As a result, you can examine the common symbols in the debugger.

- Perform a special type of partial link that has no unresolved symbols.

  Wind River's Diab linker can perform the same kind of special link.

If you do not check the Optimize Partial Link box, the build target's output file is equivalent to the file produce by the command-line linker when it is passed the `-r` flag.

Deadstrip Unused Symbols

Check the Deadstrip Unused Symbols checkbox to instruct the linker to remove any symbols that are not used. Deadstripping makes your program smaller by removing code and data not referenced by an application's main entry point (or any entry points specified in a `force_active` linker command file directive).

NOTE The Deadstrip Unused Symbols checkbox is available only if you select Partial Link from the Project Type dropdown menu.

Require Resolved Symbols

Check the Require Resolved Symbols checkbox to instruct the linker to resolve all symbols in a partial link.

NOTE The Require Resolved Symbols checkbox is available only if you select Partial Link from the Project Type dropdown menu.

If this option is checked, the linker emits an error message if any symbol referenced by your program is not defined in any source code file or library in the project.

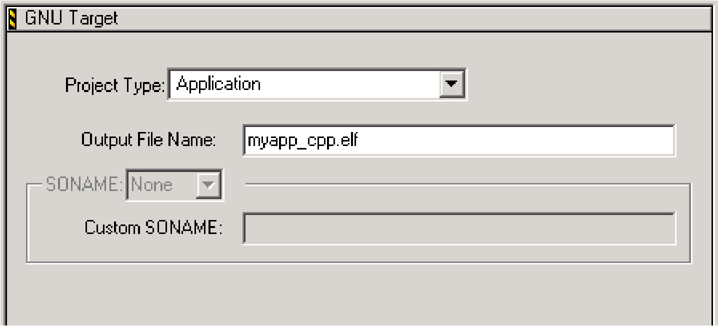
NOTE Some real-time operating systems require that there be no unresolved symbols in a partial link file. In this case, enable this option.

GNU Target

Use the **GNU Target** panel to select a project type, define the name of the build target's final output file and, for shared libraries, to define the library's SONAME.

Figure 3.7 shows the **GNU Target** settings panel.

Figure 3.7 GNU Target Panel



Project Type

Use the Project Type dropdown menu to select the project type for the build target.

Table 3.4 lists and describes each option the Project Type menu provides.

Table 3.4 Project Types

Project Type	Description
Application	A standalone application (such as `cw.elf`)
Shared Library	A library that can be shared by multiple processes or dynamically loaded into a process (for example, `sharedlib.so`)
Library	A static library (such as `staticlib.a`)
Loadable Module	A Linux kernel module that can be loaded into the kernel at runtime (for example, `printdriver.o`)

Output File Name

In the Output File Name text box, type the file name for the build target to assign to its final output file.

The linker creates this file in the Output Directory defined in the Target Settings panel. To get the linker to put the final output file elsewhere, type a relative path and file name in the Output File Name text box.

Table 3.5 shows the default output file names for each project type.

Table 3.5 Default File Names for each Project Type

Project Type	Default Output File Name
Application	`myapp_cpp.elf`
Shared Library	`my_sharedLib.so`
Library	`my_staticLib.a`
Loadable Module	`mod.o`

SONAME

Use the SONAME dropdown menu to define the SONAME (shared object name) to embed in the shared library.

The menu choices are:

- None

  No SONAME is embedded in the shared library.
- Default

  The name in the Output File Name text box is embedded in the shared library.
- Custom

  The string you enter in the Custom SONAME text box is embedded in the shared library.

NOTE The SONAME dropdown menu is only available for shared library projects.

The SONAME feature lets the library creator provide the system with version compatibility information.

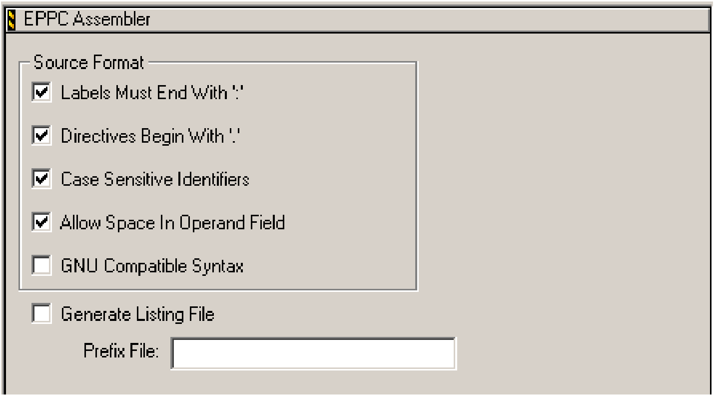
The Linux dynamic loader compares the SONAME requested by a program to the SONAME embedded in each shared library the loader finds. The loader will load only a shared library whose SONAME matches the SONAME requested by the program, thereby ensuring that the program and shared library are compatible.

EPPC Assembler

Use the **EPPC Assembler** target settings panel to define the syntax allowed in assembly language source code files, whether the assembler generates a listing file, and the name of the prefix file for the assembler to use (if any).

Figure 3.8 shows the **EPPC Assembler** target settings panel.

Figure 3.8 EPPC Assembler Panel



NOTE Previous versions of this panel included processor-related options. These options are now defined using the Processor dropdown menu of the EPPC Processor target settings panel.

Source Format

Use the checkboxes in the Source Format area to define some syntax requirements for assembly language source files. For more information about the syntax that the EPPC assembler requires, refer to the *Assembler Reference*.

GNU Compatible Syntax

Check the GNU compatible syntax checkbox to indicate that your application uses GNU-compatible assembly language syntax.

GNU-compatibility allows:

- Redefining all equates regardless of whether they were defined using `.equ` or `.set`

- Ignoring the `.type` directive
- Treating undefined symbols as imported
- Using GNU-compatible arithmetic operators

  The symbols < and > mean left-shift and right-shift instead of less than and greater than. Additionally, the symbol `!` means bitwise-or-not instead of logical not
- Using GNU-compatible operator precedence rules
- Implementing GNU-compatible numeric local labels from 0 to 9
- Treating numeric constants that start with the ‘0’ character as octal values
- Using semicolons as statement separators
- Using a single unbalanced quote for character constants. For example, `.byte 'a`

Generate Listing File

A listing file contains source code statements along with line numbers, relocation information, and macro expansions.

Check the Generate Listing File checkbox to instruct the assembler to generate a listing file for each assembly language source code file in a build target.

Prefix File

The Prefix File text box lets you enter the name of a prefix file. The assembler automatically includes this file at the beginning of each assembly language source code file in a build target.

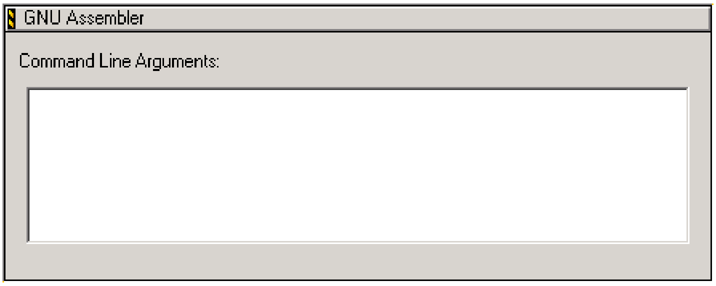
This feature lets you include common definitions without having to include the file that contains these definitions in every source code file

GNU Assembler

Use the **GNU Assembler** target settings panel to specify the command-line arguments to be passed to the GNU assembler.

Figure 3.9 shows the **GNU Assembler** target settings panel.

Figure 3.9 GNU Assembler Panel



Command Line Arguments

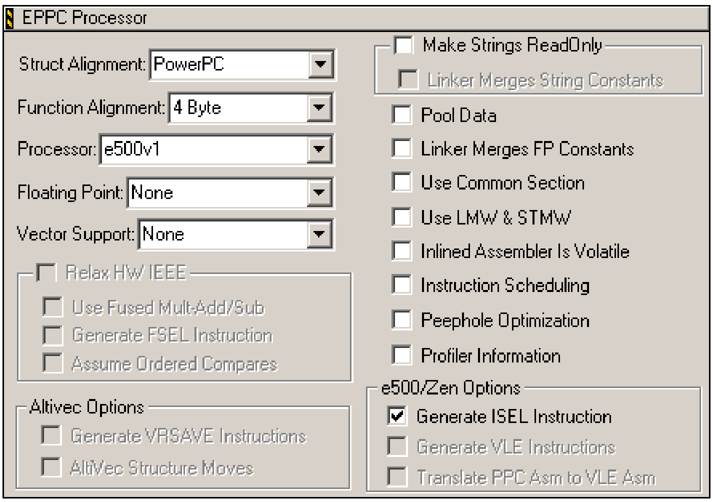
In the Command Line Argument text box, type the command-line arguments to be passed to the GNU assembler.

EPPC Processor

Use the **EPPC Processor** panel to make processor-specific code generation settings.

Figure 3.10 shows the **EPPC Processor** target settings panel.

Figure 3.10 EPPC Processor Panel



Struct Alignment

Select an option from the Struct Alignment dropdown menu to define how the compiler aligns structures.

The default option for Struct Alignment is PowerPC.

If your code must conform to the PowerPC EABI specification and inter-operate with third party object code, you must select PowerPC for the Struct Alignment option. Other choices may lead to reduced performance or alignment violation exceptions.

For more information, refer to the documentation of the `pack` pragma in the *Power Architecture Build Tools Reference*.

NOTE If you choose a Struct Alignment setting other than PowerPC, your program may not run correctly.

Function Alignment

If your board has hardware capable of fetching multiple instructions at a time, you may achieve slightly better performance by aligning functions to the width of the fetch.

Use the Function Alignment dropdown menu to select alignments from 4 (the default) to 128 bytes. These selections correspond to `#pragma function_align`. For more information, see the `function_align` pragma topic in the *Power Architecture Build Tools Reference*.

NOTE The `st_other` field of the `.symtab` entries in ELF files generated by the CodeWarrior build tools has been overloaded to ensure that dead-stripping functions does not interfere with the chosen function alignment. This may result in code that is incompatible with some third-party linkers.

Processor

Use the Processor dropdown menu to select a target processor.

Choose `Generic` if the processor you are working with is not listed, or if you want to generate code that runs on any EPPC processor. Choosing `Generic` allows the use of all optional instructions and the core instructions for the 603, 604, 740, and 750 processors.

Choosing a particular processor (as opposed to `Generic`) has these effects:

- Improved instruction scheduling

  Specifying the specific processor being targeted lets the build tools do a better job of optimizing instruction scheduling. Of course, the Instruction Scheduling option must be enabled for this effect to be realized.

- A preprocessor symbol for the selected target is defined

  If you select a particular processor, a preprocessor symbol is defined that allows code that applies just to this processor to be conditionally compiled.

  The symbol is defined as shown below, where *number* is the identification number of the processor selected:

  ```
  #define __PPCnumber__ 1
  ```

  If you select the 823 processor, for example, the symbol `__PPC823__` is defined. If you select `Generic`, the symbol `__PPCGENERIC__` is defined to 1.

- Floating-point support verification

  You can select any of the options in the Floating Point dropdown menu (Software, Hardware, SPFP, and DPFP), no matter what processor you select.

  Selecting a specific processor (as opposed to `Generic`) lets the build tools warn you if the selected floating-point option is not supported by the processor you select.

Floating Point

Use the Floating Point dropdown menu to define how the compiler handles floating-point operations it encounters in your source code.

NOTE Some Floating Point menu options require that you include the corresponding version of the runtime library in your build target. For example, if you select `None`, you must include `Runtime.PPCEABI.N.a` in your build target.

Table 3.6 lists and describes each floating-point support option.

Table 3.6 Floating-Point Support Options

Item	Description
None	Select to disable floating-point support.
Software	Select to have the compiler emulate floating-point operations by calling functions that perform floating-point math. The C runtime library contains the functions the compiler invokes. If you use software floating-point emulation, you must include the appropriate C runtime library in your project. Enabling this option without including the appropriate C runtime library causes link errors.
Hardware	Select to have the compiler handle floating-point operations by generating instructions for the hardware floating-point unit. Do not select this option if your target processor does not have a hardware floating-point unit.

Table 3.6 Floating-Point Support Options (*continued*)

Item	Description
SPFP	Select to have the compiler handle single-precision floating-point operations by generating instructions for the e500-EFPU floating point unit, and perform double-precision floating-point operations by calling functions that perform double-precision floating-point math. Do not select this option if your target processor does not have a e500-EFPU floating-point unit.
DPFP	Select to have the compiler handle both single- and double-precision floating-point operations by generating instructions for the e500 DPFP APU (Double-Precision Floating-Point Auxiliary Processing Unit). Do not select this option if your target processor does not have a DPFP unit.

NOTE If the selected processor does not handle a floating-point exception, you should select `None` or `Software` floating-point support.

Vector Support

Use the Vector Support dropdown menu to select the type of vector execution unit your target processor has. The CodeWarrior Power Architecture C/C++ compiler supports both AltiVec™ and SPE vector execution units.

If your target processor includes a vector execution unit and you want the compiler to generate instructions for this unit, select the vector type your processor supports from the Vector Support dropdown menu. If your processor does not have a vector execution unit or you do not want the compiler to emit vector instructions, select `None`.

If you select Altivec from the Vector Support menu, the checkboxes in the Altivec Options area enable. These options let you select the type of AltiVec support required.

Relax HW IEEE

Check the The Relax HW IEEE checkbox to instruct the compiler to generate faster code by ignoring some of the more strict requirements of the IEEE floating-point standard. You control the particular requirements that are relaxed with the options Use Fused Multi-Add/Sub, Generate FSEL Instruction, and Assume Ordered Compares.

NOTE The Relax HW IEEE checkbox is available only if you select Hardware from the Floating Point dropdown menu.

Use Fused Multi-Add/Sub

Check this box to instruct the compiler to generate EPPC Fused Multi-Add/Sub instructions. If enabled, this option lets the compiler generate smaller and faster floating-point code than it generates if it adheres to the IEEE floating-point specification.

NOTE Enabling the Use Fused Multi-Add/Sub option may produce unexpected results because of the greater precision of the intermediate values these instructions produce. The results are slightly more accurate than those produced by the IEEE floating-point standard because of an extra rounding bit between the multiply operation and the add/subtract operation.

Generate FSEL Instruction

Check this box to instruct the compiler to generate the FSEL instruction. This instruction executes more quickly than corresponding instructions allowed by the IEEE floating-point specification.

Enabling Generate FSEL Instruction option lets the compiler optimize the pattern

```
x = (condition ? y : z)
```

where `x` and `y` are floating-point values.

NOTE The FSEL instruction is not accurate for denormalized numbers and may cause problems related to unordered compares.

Assume Ordered Compares

Check this box to instruct the compiler to ignore issues associated with unordered numbers (such as NAN) when comparing floating-point values. In strict IEEE mode, any comparison with NAN except not-equal-to, returns false. The assume ordered compares optimization ignores this requirement, thereby allowing this conversion:

```
if (a <= b)
```

to

```
if !(a > b)
```

Altivec Options

Use the checkboxes of the Altivec Options group box to instruct the compiler to generate specific categories of instructions for an Altivec vector execution unit.

NOTE The options in the Altivec Options group are disabled unless you select Altivec from the Vector Support dropdown menu.

Altivec Structure Moves

Check the Altivec Structure Move checkbox to instruct the compiler to use Altivec instructions to copy structures.

Generate VRSAVE Instructions

The value of the VRSAVE register tells the operating system which vector registers to save and reload when a context switch occurs—the bits of the VRSAVE register that correspond to the vector registers to save/reload are set to 1.

When a function call occurs, the value of the VRSAVE register is saved in a part of the stack frame called the vrsave word. In addition, the function saves the values of any non-volatile vector registers in the stack frame in an area called the vector register save area before changing the values in any of these registers.

Checking the Generate VRSAVE Instructions checkbox tells the compiler to use the Altivec VRSAVE instruction to save these vector register values, thereby reducing the time required to complete a context switch.

NOTE Check the Generate VRSAVE Instructions box only if the resulting binary will run on multi-tasking operating system that supports the Altivec vector unit.

Make Strings Read Only

Check the Make Strings Read Only box to instruct the compiler to store string constants in the `.rodata` (read-only data) section. Uncheck this box to instruct the compiler to store string constants in the `.data` section.

The Make Strings Read Only option corresponds to `#pragma readonly_strings`. The default setting of this pragma is `OFF`.

Linker Merges Strings Constants

If you check the Make Strings Read Only checkbox, the Linker Merges String Constants checkbox becomes available.

Check the Linker Merges String Constants box to have the compiler pool strings defined within a given source file. If this checkbox is clear, the compiler treats each string as an individual string.

NOTE The linker can deadstrip unused, unpooled strings, but cannot deadstrip unused pooled strings.

Pool Data

Check the Pool Data checkbox to instruct the compiler to organize some of the data in the large data sections (`.data`, `.bss`, and `.rodata`) such that a program can access the data more quickly.

The Pool Data option affects only data that is defined in the current source file; the option does not affect external declarations or any small data.

NOTE The linker is aggressive about stripping unused data and functions from a binary; however, the linker cannot strip any large data that has been pooled.

NOTE If your program uses tentative data, you get a warning that you need to force the tentative data into the common section.

Linker Merges FP Constants

Check the Linker Merges FP Constants checkbox to instruct the compiler to name floating-point constants in such a way that the name contains the constant. This lets the linker merge floating-point constants automatically.

Use Common Section

Check the Use Common Section checkbox to have the compiler place global, uninitialized data in the common section. This section is similar to a Fortran common block.

If this box is checked and the linker finds two or more variables with the same name and at least one of them is in a common section, the linker assigns these variables the same memory address. If this checkbox is clear, two variables with the same name generate a link error.

The compiler never places small data, pooled data, or variables declared static in the common section.

The `section` pragma provides fine control over which symbols the compiler includes in the common section.

To have the desired effect, this feature must be enabled during the definition of the data, as well as during the declaration of the data. Common section data is converted to use the `.bss` section at link time. The linker supports common section data in libraries even if the switch is disabled at the build-target level.

NOTE You must initialize a common section variable in each source file that uses this variable; otherwise you get unexpected results.

NOTE We recommend that you develop with the Use Common Section box clear. Once you have debugged your program, look at its data for especially large variables that are used in just one file. Change the names of such variables so they are the same, and make sure that you initialize them before you use them. Once you have completed this process, you can enable the Use Common Section feature.

Use LMW & STMW

`LMW` (Load Multiple Word) is a single EPPC instruction that loads a group of registers; `STMW` (Store Multiple Word) is a single EPPC instruction that stores a group of registers. If the Use LMW & STMW box is checked, the compiler sometimes uses these instructions in a function's prologue and epilogue to save and restore volatile registers.

A function that uses the `LMW` and `STMW` instructions is always smaller, but usually slower, than a function that uses an equivalent series of `LWZ` and `STW` instructions. Therefore, in general, check the Use LMW & STMW box if compact code is your goal, and leave this box unchecked if execution speed is your objective.

That said, because a smaller function might fit better in the processor's cache lines than a larger function, it is possible that a function that uses `LMW`/`STMW` will execute faster than one that uses multiple `LWZ`/`STW` instructions.

As a result, to determine which instructions produce faster code for a given function, you must try the function with and without `LMW/STMW` instructions. To make this determination, use these pragmas to control the instructions the compiler emits for the function in question:

- `#pragma no_register_save_helpers on|off|reset`

  If this pragma is on, the compiler always inlines instructions.

- `#pragma use_lmw_stmw on|off|reset`

This pragma has the same effect as the Use LMW & STMW checkbox, but operates at the function level.

NOTE The compiler never uses the `LMW` and `STMW` instructions if little-endian byte ordering is enabled, even if the Use LMW & STMW checkbox is checked. This restriction is necessary because execution of an `LMW` or `STMW` instruction while the processor is in little-endian mode causes an alignment exception.

See the *Programming Environments Manual For 32-Bit Implementations of the PowerPC Architecture* for more information about `LMW` and `STMW` efficiency issues.

Inlined Assembler is Volatile

Check the Inlined Assembler is Volatile checkbox to instruct the compiler to treat all `asm` blocks (including inline `asm` blocks) as if the `volatile` keyword were present. This prevents the `asm` block from being optimized.

You can use the `.nonvolatile` directive to selectively enable optimization on `asm` blocks, as required.

Instruction Scheduling

If the Instruction Scheduling checkbox is checked, scheduling of instructions is optimized for the specific processor you are targeting (as defined by which processor selected in the Processor dropdown menu).

NOTE Enabling the Instruction Scheduling checkbox can make source-level debugging more difficult (because the source code may not correspond to the execution order of the underlying instructions). Sometimes, it is helpful to clear this checkbox when debugging, and then check it once you have finished the bulk of your debugging.

Peephole Optimization

Check the Peephole Optimization checkbox to instruct the compiler to perform peephole optimizations.

Peephole optimizations are small, local optimizations that can reduce several instructions to one target instruction, eliminate some compare instructions, and improve branch sequences.

This checkbox corresponds to `#pragma peephole`. See the *Power Architecture Build Tools Reference* for more information about this pragma.

Profiler Information

Check the Profiler Information checkbox to instruct the compiler to generate object code that collects information at runtime that the code profiler can use.

This checkbox corresponds to `#pragma profile`.

e500/Zen Options

The options in the e500/Zen Options group box apply only to processors that have an e500 or an e200z (formerly, Zen) core.

The e500/Zen options are:

NOTE This CodeWarrior product supports the e500 core. The product does not support the e200z core.

- Generate ISEL Instruction

  Check this box to instruct the compiler to generate ISEL instructions.

  The ISEL instruction can improve program performance by reducing conditional branching.

NOTE The Generate ISEL Instruction checkbox is disabled unless you select `e500v1` or `e500v2` from the Processor dropdown menu of this panel. This is because only processors that have an e500 core have an ISEL auxiliary processing unit (APU).

- Generate VLE Instructions

  CodeWarrior for Power Architecture Processors does not support this feature.

- Translate PPC Asm to VLE Asm

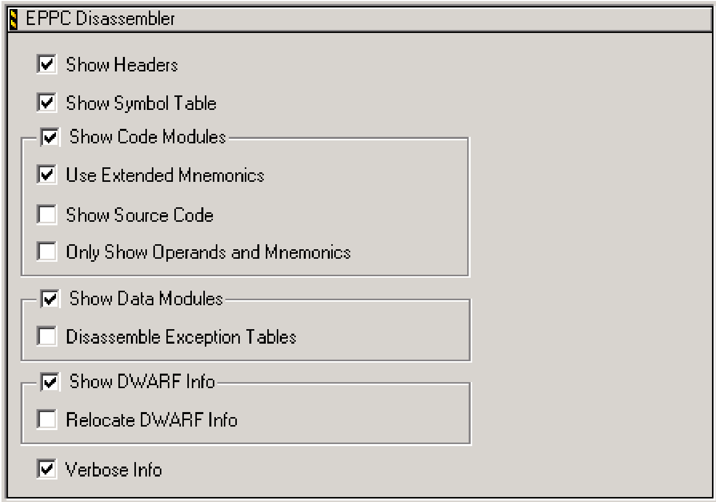
  CodeWarrior for Power Architecture Processors does not support this feature.

EPPC Disassembler

Use the **EPPC Disassembler** target settings panel to define the information to include in the results of a disassembly.

Figure 3.11 shows the **EPPC Disassembler** target settings panel.

Figure 3.11 EPPC Disassembler Panel



Show Headers

Check the Show Headers checkbox to have the disassembler include ELF header information in the results of the disassembly.

Show Symbol Table

Check the Show Symbol Table checkbox to have the disassembler include the symbol table in the module being disassembled in the results of the disassembly.

Show Code Modules

Check the Show Code Modules checkbox to have the disassembler include ELF code sections in the results of the disassembly.

Checking the Show Code Modules checkbox enables these checkboxes:

- Use Extended Mnemonics
- Show Source Code
- Only Show Operands and Mnemonics

Use Extended Mnemonics

Check the Use Extended Mnemonics checkbox to have the disassembler include the extended mnemonics for each instruction in the module being disassembled in the results of the disassembly.

Show Source Code

Check the Show Source Code checkbox to have the disassembler include the source code used to build the module being disassembled in the results of the disassembly. The source code is interleaved with the mnemonics of the disassembled instructions.

Only Show Operands and Mnemonics

Check the Only Show Operands and Mnemonics checkbox to have the disassembler exclude all information other than operands and mnemonics for each code section in the results of the disassembly.

Show Data Modules

Check the Show Data Modules checkbox to have the disassembler include ELF data sections (such as `.rodata` and `.bss`) in the results of the disassembly.

Checking the Show Data Modules checkbox enables the Disassemble Exception Tables checkbox.

Disassemble Exception Tables

Check the Disassemble Exception Tables checkbox to have the disassembler include C++ exception tables in the module being disassembled in the results of the disassembly.

Show DWARF Info

Check the Show DWARF Info checkbox to have the disassembler include DWARF debugging information in the results of the disassembly.

Checking the Show DWARF Info checkbox enables the Relocate DWARF Info checkbox.

Relocate DWARF Info

Check the Relocate DWARF Info checkbox to have object and function addresses appear in the debug sections of the module being disassembled in the results of the disassembly.

NOTE This option affects modules containing DWARF v1 debug information only.

Verbose Info

Check the Verbose Info checkbox to instruct the disassembler to include additional data for certain categories of data in the module being disassembled in the results of the disassembly.

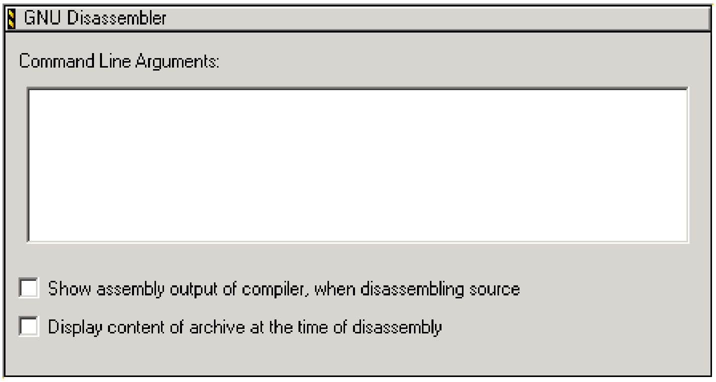
For the `.symtab` section, some of the descriptive constants are shown with their numeric equivalents. The `.line`, `.debug`, `extab`, and `extabindex` sections are also shown in an unstructured hexadecimal dump form.

GNU Disassembler

Use the **GNU Disassembler** target settings to specify command-line arguments to be passed to the GNU disassembler.

Figure 3.12 shows the **GNU Disassembler** target settings panel.

Figure 3.12 GNU Disassembler Panel



Command Line Arguments

In the Command Line Arguments text box, type the command-line arguments to be passed to the GNU disassembler.

Show Assembly Output of Compiler When Disassembling Source

Check this checkbox to instruct the IDE to use the GNU compiler to disassemble files.

If unchecked, the IDE uses the disassembler utility specified in the GNU Tools panel to disassemble binary files.

Display Content of Archive at Time of Disassembly

Check this checkbox to instruct the IDE to use the archiver tool specified in the GNU GNU Tools panel to disassemble binary files. Using this archiver, you can view the list of objects within libraries.

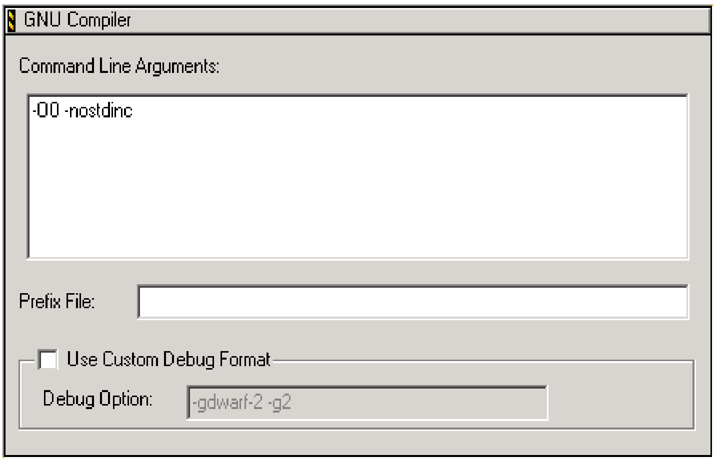
If this box is unchecked, the IDE uses the disassembler utility specified in the GNU Tools panel to disassemble binary files.

GNU Compiler

Use the **GNU Compiler** target settings panel to specify command-line arguments to be passed to the GNU compiler, a prefix file for the compiler to include at the start of each source code file, and the format of the debugging information the compiler places in the object code it generates.

Figure 3.13 shows the **GNU Compiler** target settings panel.

Figure 3.13 GNU Compiler Panel



Command Line Arguments

In the Command Line Arguments text box, type the command-line arguments to be passed to the GNU compiler.

Prefix File

In Prefix File text box, type the path to the prefix file for the GNU compiler to include at the start of each implementation (`.c`, `.cpp`) file.

Use Custom Debug Format

Check the Use Custom Debug Format checkbox to instruct the compiler to generate debugging information in the format specified in the Debug Option text box.

Uncheck this box to instruct the compiler to generate debugging information in the default format.

Debug Option

If you check the Use Custom Debug Format box, the Debug Option text box activates.

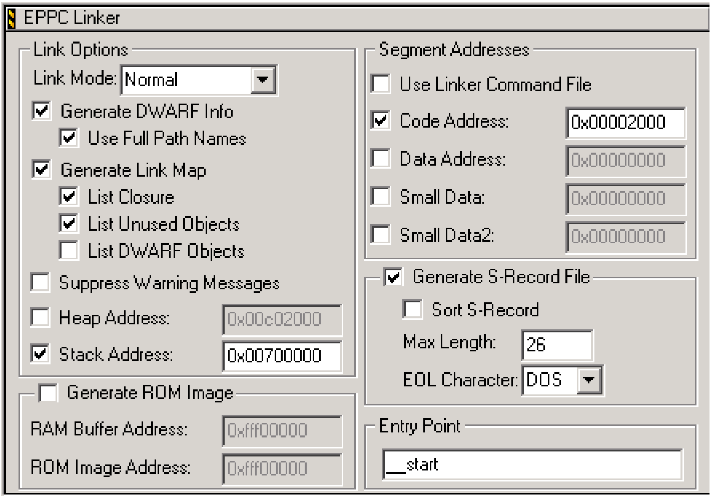
In this text box, type the command-line argument that tells the GNU compiler what debugging information format to use.

EPPC Linker

Use the **EPPC Linker** target settings panel to select options related to linking object code into its final form.

Figure 3.14 shows the **EPPC Linker** target settings panel.

Figure 3.14 EPPC Linker Panel



Link Mode

Link mode lets you control how much memory the linker uses as it writes the output file to the hard disk. Linking requires enough RAM to hold all of the input files and the numerous structures that the linker uses for housekeeping. The housekeeping allocations occur before the linker writes the output file to the disk.

Use the Link Mode dropdown menu to select the link mode. The options are:

- Use Less RAM

  In this link mode, the linker writes the output file directly to disk without using a buffer.

- Normal

  In this link mode, the linker writes to a 512-byte buffer and then writes the buffer to disk. For most projects, this link mode is the best choice.

- Use More RAM

  In this link mode, the linker writes each segment to its own buffer. When all segments have been written to their buffers, the buffers are flushed to the disk. This link mode is best suited for small projects.

Generate DWARF Info

Check the Generate DWARF Info checkbox to instruct the linker to generate debugging information in Debug With Arbitrary Record Format (DWARF) format. DWARF information is included within the linked ELF file. Checking this box does not cause the linker to generate a separate file.

If you check the Generate DWARF Info checkbox, the Use Full Path Names checkbox becomes available.

Use Full Path Names

Use the Use Full Path Names checkbox to control the type of source file code paths the linker embeds in the ELF file the linker generates.

If the Use Full Path Names checkbox is checked, the linker embeds full paths as well as root source file names within the linked ELF file (see the note that follows). If this checkbox is clear, the linker saves just the root file names of the source code file from which the ELF was generated.

NOTE If you build your programs on one machine and debug it on another, clear the Use Full Path Names checkbox. Clearing this box makes it easier for the debugger to find the source code files associated with a binary.

Generate Link Map

Check the Generate Link Map checkbox to instruct the linker to generate a link map.

The linker adds the extension `.MAP` to the file name specified in the File Name text box of the EPPC Target settings panel. The file is saved in the same folder as the output file.

The link map shows which file provided the definition for every object and function in the output file. The map also displays the address assigned to each object and function, a memory map of where each section resides in memory, and the value of each linker generated symbol.

Although the linker aggressively strips unused code and data from relocatable files generated by the CodeWarrior compiler, the linker never deadstrips relocatable files generated by the assembler or relocatable files built with other compilers.

If a relocatable file was *not* built with the CodeWarrior C/C++ compiler, the link map lists all the unused but unstripped symbols. You can use this information to remove the symbol definitions from your source code, thereby making the final image smaller.

List Closure

Check the List Closure checkbox to have all the functions called by the starting point of the program listed in the link map. See the Entry Point topic for details.

This List Closure box is available only if you check the Generate Link Map checkbox.

List Unused Objects

Check the List Unused Objects checkbox to instruct the linker to include unused objects in the link map. This setting helps you find objects that you think are being used, but are really not.

The List Unused Objects checkbox is available only if you check the Generate Link Map checkbox.

List DWARF Objects

Check the List DWARF Objects checkbox to instruct the linker to list all DWARF debugging objects in the section area of the link map. The DWARF debugging objects are also listed in the closure area if you check the List Closure checkbox.

The List DWARF Objects checkbox is available only if you check the Generate Link Map checkbox.

Suppress Warning Messages

Check the Suppress Warning Messages checkbox to instruct the linker to not display warnings in the CodeWarrior **Errors and Warnings** window.

Heap Address

Use the Heap Address text box to define the memory location at which the linker places the heap. The heap is used if your program calls `malloc` or `new`.

To specify a heap address, check this Heap Address checkbox and then type an address in related text box. You must specify the address in hexadecimal (e.g, `0x00c02000`).

The address specified is the bottom of the heap and (if necessary) is changed to align with the nearest 8-byte boundary.

The top of the heap is Heap Size (k) kilobytes above the Heap Address (where Heap Size is defined in the EPPC Target panel). The possible addresses depend on your target board and how this board's memory is mapped. The heap must reside in RAM.

If you do not specify a heap address, the top of the heap is equal to the bottom of the stack, and the following statements are true:

```
_stack_end  = _stack_addr - (Stack Size * 1024);
_heap_end   = _stack_end;
_heap_addr  = _heap_end - (Heap Size * 1024);
```

The MSL memory allocation routines do not require that the heap be below the stack: You can set the heap address to any place in RAM that does not overlap other sections. MSL also lets you have multiple memory pools, which can increase the total size of the heap.

Clear the Heap Address checkbox if your code does not use a heap. If you are using MSL, your program may implicitly use a heap.

NOTE If there is not enough free memory available in your program, `malloc` returns zero. If you do not call `malloc` or `new`, consider setting Heap Size (k) to 0 to maximize the memory available for code, data, and stack.

Stack Address

Use the Stack Address text box to define the memory location at which the linker places the stack.

To specify a stack address, check the Stack Address checkbox and type an address related text box. The address must be in hexadecimal (e.g, `0x007f0000`).

The address you specify is the top of the stack. If necessary this address is changed to align to the nearest 8-byte boundary.

The stack extends downward from the specified address by the number of kilobytes specified in the Stack Size text box of the EPPC Target panel.

The possible address for the stack depend on your target board and the way its memory is mapped. The stack must reside in RAM.

NOTE Alternatively, you can specify the stack address by entering a value for the symbol `_stack_addr` in a linker command file.

If you do not specify an explicit stack address, the linker uses the address `0x003DFFF0`. However, this address may not be suitable for boards with a small amount of RAM. For such boards, see the stationery projects for examples with suitable addresses.

NOTE Because the stack grows downward in memory, it is common to place the stack as high in memory as possible. If you have a board that has CodeWarrior TRK installed, this program puts its data in high memory. The default (factory) stack address reflects the memory requirements of CodeWarrior TRK and places the stack address at `0x003DFFF0`. CodeWarrior TRK also uses memory from `0x00000100` to `0x00002000` for exception vectors.

Generate ROM Image

Check the Generate ROM Image box to instruct the linker to create a ROM image. A ROM image is a file that a flash programmer can write to flash ROM.

RAM Buffer Address

Use the RAM Buffer Address text box to specify the address of a RAM buffer for a flash programmer to use.

Many flash programmers (such as the MPC8BUG programmer) use the RAM buffer you specify to load all segments in your binary to consecutive addresses in flash ROM. Note, however, that at runtime, these segments are loaded at the addresses you specify in your linker command file or in the fields of the Segment Addresses group box.

For example, the MPC8BUG flash programmer requires a RAM Buffer Address of `0x02800000`. This programmer makes a copy of your program starting at address `0xFFE00000`. If `0xFFE00000` is where you want your `.text` section, then you must enter `0xFFE00000` in the Code Address text box of the Segment Addresses group. If you specify a different code address, you must copy the code to this address from address `0xFFE00000`.

NOTE To perform address calculations like that in the example above, you may find the symbols the linker generates for ROM and execution addresses helpful. For more information about the linker-generated symbols related to these addresses, see this file:
`installDir``\PowerPC_EABI_Support\`
`Runtime\Include\__ppc_eabi_linker.h`

NOTE The CodeWarrior flash programmer does not use a separate RAM buffer. As a result, if you use the CodeWarrior flash programmer (or any other flash programmer that does not use a RAM buffer), the RAM Buffer Address *must* be equal to the ROM Image Address.

ROM Image Address

Use the ROM Image Address text box to specify the address at which you want your binary written to flash ROM.

The address you enter must be in hexadecimal (for example, `0xfff00000`).

Segment Addresses

Use the checkboxes in the Segment Addresses group box to indicate whether you want the segment addresses defined by a linker command file or directly in this settings panel.

Use Linker Command File

Check the Use Linker Command File checkbox to use a linker command file to define segment addresses. If the linker does not find a command file, it issues an error message.

Leave this checkbox clear if you want to specify the segment addresses directly in the segment address text boxes: Code Address, Data Address, Small Data, and Small Data2.

NOTE If you have a linker command file in your project and the Use Linker Command File checkbox is clear, the linker ignores this file.

Code Address

Use the Code Address text box to define the memory location at which the linker places a build target's executable code.

To specify a code segment address, check the Code Address checkbox and type an address in the related text box. You must specify the address in hexadecimal notation (for example, `0x00002000`). Possible code segment addresses depend on your target board and how its memory is mapped.

If you clear the checkbox, the default code segment address is `0x00010000`. This default address may not be suitable for boards with a small amount of RAM. For such boards, see the stationery projects for examples with suitable addresses.

Data Address

Use the Data Address text box to define the memory location at which the linker places a build target's global data.

To specify a data segment address, check the Data Address checkbox and type an address in the related text box. You must specify the address in hexadecimal notation (for example, `0x000A0000`). Possible data segment addresses depend on your target board and how its memory is mapped. Data must reside in RAM.

If you clear the Data Address checkbox, the linker sets places the data segment immediately following the read-only code and data segments (`.text`, `.rodata`, `extab`, and `extabindex`).

Small Data

The Small Data checkbox and related text box let you define the memory location at which the linker places the first small data section mandated by the PowerPC EABI specification.

If you uncheck the Small Data checkbox, the linker places the first small data section immediately after the `.data` section.

If you check the Small Data checkbox, the related text box enables. In this text box, type the address at which you want the linker to place the first small data section. The address entered must be in hexadecimal format (for example, `0xABCD1000`). Further, the address entered must be supported by your target board and must not conflict with the memory map of this board. Finally, all types of data must reside in RAM.

Small Data2

The Small Data2 checkbox and related text box let you define the memory location at which the linker places the second small data section mandated by the PowerPC EABI specification.

If you uncheck the Small Data2 checkbox, the linker places the second small data section immediately after the `.sbss` section.

If you check the Small Data2 checkbox, the related text box enables. In this text box, type the address at which you want the linker to place the second small data section. The address entered must be in hexadecimal format (for example, `0x1000ABCD`). Further, the address entered must be supported by your target board and must not conflict with the memory map of this board. Finally, all types of data must reside in RAM.

NOTE The CodeWarrior development tools create the three small data sections required by the PowerPC EABI specification.
Further, the CodeWarrior tools let you define additional small data sections. See the *Power Architecture Build Tools Reference* for instructions that explain how to do this.

Generate S-Record File

Check the Generate S-Record File checkbox to instruct the linker to generate an S-Record file based on the application object image. This file has the same name as the executable file, but with a `.mot` extension. The linker generates S3 type S-Records.

Sort S-Record

Check the Sort S-Record checkbox to have the generated S-Record file sorted in the ascending order by address.

This checkbox is available only if you check the Generate S-Record File checkbox.

Max Length

Use the Max Length text box to define the maximum length of the S-Records generated by the linker. The maximum value allowed for an S-Record length is 256 bytes.

This text box is available only if you check the Generate S-Record File checkbox.

NOTE Most programs that load embedded software have a maximum S-Record length. The CodeWarrior debugger can handle S-Records up to 256 bytes long. If you are using something other than the CodeWarrior debugger to load your embedded application, you must find out what the maximum allowed length is.

EOL Character

Use the EOL Character dropdown menu to select the end-of-line character for the S-Record file. The end of line character options are:

- `<cr><lf>` for DOS
- `<lf>` for Unix
- `<cr>` for Mac

This menu is available only if the Generate S-Record File checkbox is checked.

Entry Point

Use the Entry Point text box to specify the function that the linker uses first when the program launches. This is the starting point of the program.

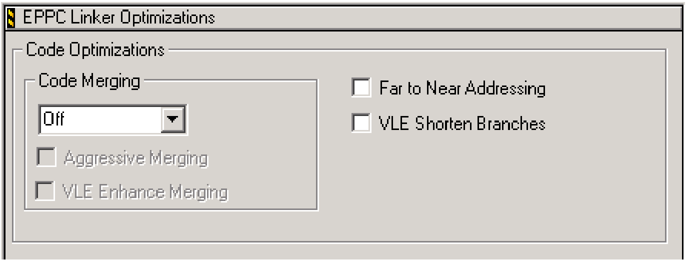
The default `__start` function is bootstrap (or glue) code that sets up the PowerPC EABI environment before your code executes. This function is in the `__start.c` file. The final task performed by `__start` is to call your `main()` function.

EPPC Linker Optimizations

Use the **EPPC Linker Optimizations** target settings panel to configure the EPPC linker's code merging feature.

Figure 3.15 shows the **EPPC Linker Optimizations** target settings panel.

Figure 3.15 EPPC Linker Optimizations Panel



Code Merging

Use the Code Merging dropdown menu to select the type of code merging you want the linker to perform. Code merging is a size optimization that removes duplicated functions.

The options are:

- Off

  The linker performs no code merging.

- Safe Functions

  The linker removes only those functions that are weakly duplicated.

- All Functions

  The linker removes all duplicated functions.

Aggressive Merging

Check to have the linker perform aggressive merging for the selected code merge type.

When performing an aggressive merge, the linker removes a duplicated function even if the function's program uses the function's address.

NOTE Aggressive merging is not ANSI-compliant.

When performing a non-aggressive merge, the linker does not remove a duplicated function if the function's program uses that function's address; instead, the linker replaces the function with a single instruction — a branch to its duplicate function.

Consider the code shown in Listing 3.3.

Listing 3.3 Source Code that Uses the Addresses of Functions

```
pf1 = &func_1;
...
pf2 = &func_2;
...
if (pf1 != pf2)
   return 0;
else
   return 1;
//where pf1 and pf2 are pointers to functions
```

In the code shown above, you probably must preserve distinct objects for `func_1` and `func_2`, even if the functions contain identical code. As a result, you must choose a non-aggressive merge.

However, if your code just takes function addresses to initialize function pointers and does *not* do address comparisons (as in the above example), you can use an aggressive merge.

VLE Enhance Merging

Check to have the linker perform enhanced merging for the selected code merge type.

Checking this option removes duplicated functions that are called by functions that use VLE instructions to reduce object code size.

When applying the code merging optimization, this linker optimization ensures that function calls that use VLE (Variable Length Encoding) instructions are able to reach a function that has been removed. This optimization replaces the 16-bit se_bl instruction with a 32-bit e_bl instruction.

When this option is not used, the linker does not merge functions that are called by functions that use VLE instructions. This optimization requires that the target processor has the Variable Length Encoding (VLE) extension. This optimization has no effect when the linker is not applying the code merging optimization.

NOTE The linker does not apply this optimization to functions that are declared with the `__declspec(no_linker_opts)` directive.

Far to Near Addressing

Check this option to simplify address computations by reducing object code size and improving performance.

This linker optimization simplifies address computations in object code. If an address value is within the range that can be stored in the immediate field of the load immediate instruction, the linker replaces the address's two-instruction computation with a single instruction. An address value that is outside this range still requires two instructions to compute.

The ranges of values that may be stored in the immediate field is `-0x7fff` to `0x8000` for the regular `li` instruction and `-0x7ffff` to `0x80000` for `e_li`, the VLE (Variable Length Encoding) instruction.

NOTE The linker does not apply this optimization to functions that are declared with the `__declspec(no_linker_opts)` directive.

VLE Shorten Branches

Check this option to replace branch instructions to reduce object code size.

This linker optimization replaces each 32-bit `e_bl` instruction with a 16-bit `se_bl` instruction for a function call when the span of memory between the calling function and called function is sufficiently close.

This optimization requires that the target processor has the Variable Length Encoding (VLE) extension.

NOTE The linker does not apply this optimization to functions that have been declared with the `__declspec(no_linker_opts)` directive.

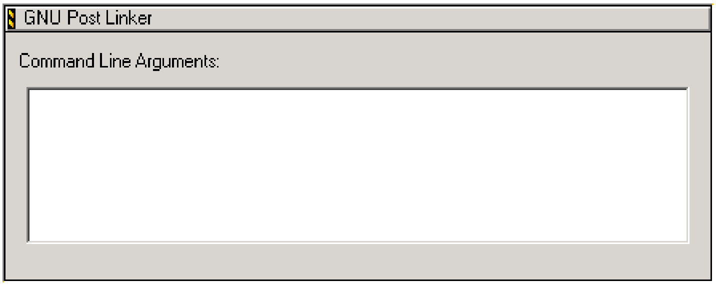
GNU Post Linker

Use the **GNU Post Linker** target settings panel to specify command-line arguments to be passed to the GNU post-linker utility. This utility is specified in the Post Linker text box of the GNU Tools target settings panel.

NOTE The GNU Post Linker panel appears in the panel list of the **Target Settings** window only if you select EPPC GNU Post-linker or EPPC Linux GNU Post-linker from the Post-linker menu of the Target Settings panel.

Figure 3.16 shows the **GNU Post Linker** target settings panel.

Figure 3.16 GNU Post Linker Panel



Command Line Arguments

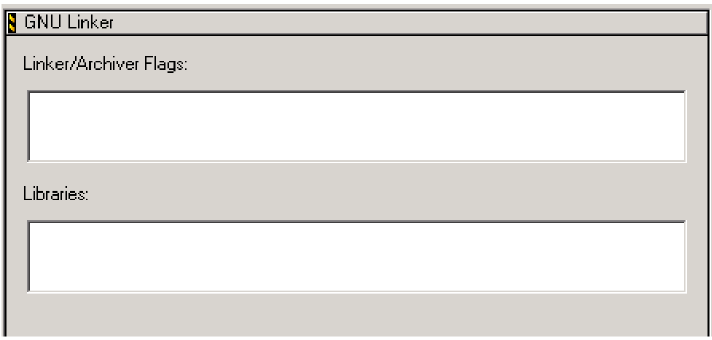
In the Command Line Arguments text box, type the command-line arguments to be passed to the post-linker specified in the GNU Tools target settings panel.

GNU Linker

Use the **GNU Linker** target settings panel to specify command-line arguments to be passed to the GNU linker.

Figure 3.17 shows the **GNU Linker** target settings panel.

Figure 3.17 GNU Linker Panel



Linker/Archiver Flags

In the Linker/Archiver text box, type the command-line arguments to be passed to the GNU linker specified in the Target Settings panel.

Libraries

In the Linker/Archiver text box, type the names of the libraries with which to link the binary generated by the current build target.

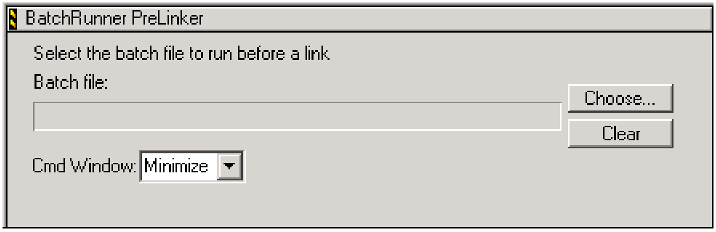
BatchRunner PreLinker

Use the **BatchRunner PreLinker** target settings panel to specify the batch file that the BatchRunner PreLinker runs. The pre-linker runs this batch file immediately before the linker is invoked.

Figure 3.18 shows the **BatchRunner PreLinker** target settings panel.

NOTE This panel is available only if you select BatchRunner PreLinker from the **Target Settings** panel's Pre-linker menu.

Figure 3.18 BatchRunner PreLinker Panel



Batch file

Click the **Choose** button to display the **Select Batch/Command File** dialog box. Use this dialog box to select a batch file for the IDE to run prior to invoking the linker.

Cmd Window

Select an option from the Cmd Window dropdown menu to define the behavior of the Windows operating system's command window (`cmd.exe`) while the specified batch file executes.

The options are:

- Minimize

  Select this option if you want the command window to be minimized to the Windows task bar while the specified batch file executes.

- Show

  Select this option if you want the command window to be visible while the specified batch file executes.

- Hide

  Select this option if you want the command window to be invisible while the specified batch file executes.

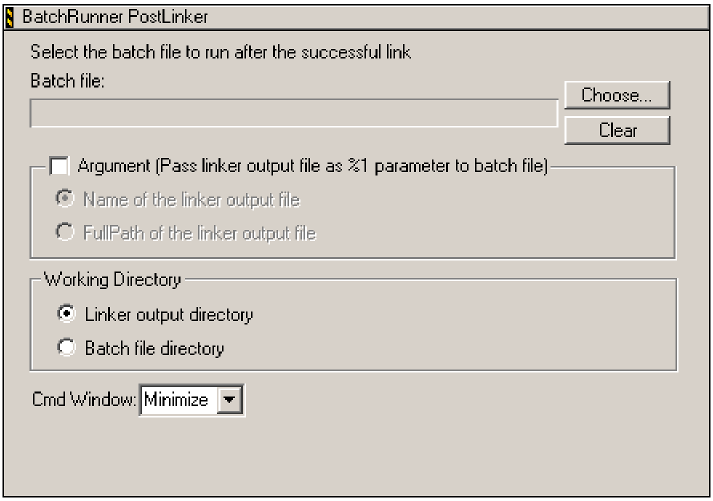
BatchRunner PostLinker

Use the **BatchRunner PostLinker** target settings panel to specify the batch file that the BatchRunner PostLinker runs. The post-linker runs this batch file immediately after the linker terminates.

NOTE This panel is available only if you select BatchRunner PostLinker from the **Target Settings** panel's Post-linker menu.

Figure 3.19 shows the **BatchRunner PostLinker** target settings panel.

Figure 3.19 BatchRunner PostLinker Panel



Batch File

Use the Batch File text box to define the batch file that the BatchRunner Post-Linker runs.

Because this text box is read-only, you cannot type the batch file name into this box. Instead, click **Choose** to display the **Select Batch/Command** file dialog box, and use this dialog box to select the batch file for the BatchRunner Post-Linker to run.

Argument (Pass linker output file as %1 parameter to batch file)

The Argument option buttons let you select the value that the BatchRunner Post-Linker passes as the `%1` parameter to the batch file this post-linker runs.

The options are:

- Name of linker output file

  Select this option to instruct the Batch File Post-Linker to pass the root file name of the linker output file as the `%1` parameter of the batch file the post-linker runs.

  This file name is specified in the File Name text box of the EPPC Target panel.

- FullPath of the linker output file

  Select this option to instruct the Batch File Post-Linker to pass the full path of the directory to which the linker writes its output the `%1` parameter of the batch file the post-linker runs.

  This linker output directory is specified in the Output Directory text box of the Target Settings panel.

Working Directory

The Working Directory option buttons let you select the working directory for the batch file the Batch File Post-Linker runs.

The options are:

- Linker output directory

  Select this option to make the directory to which the linker writes a build target's output file the batch file's working directory. This directory is defined in the Target Settings panel.

- Batch file directory

  Select this option to make the directory in which the batch file resides the batch file's working directory.

Cmd Window

Select an option from the Cmd Window dropdown menu to define the behavior of the Windows operating system's command window (`cmd.exe`) while the specified batch file executes.

The options are:

- Minimize

  Select this option if you want the command window to be minimized to the Windows task bar while the specified batch file executes.

- Show

  Select this option if you want the command window to be visible while the specified batch file executes.

- Hide

  Select this option if you want the command window to be invisible while the specified batch file executes.

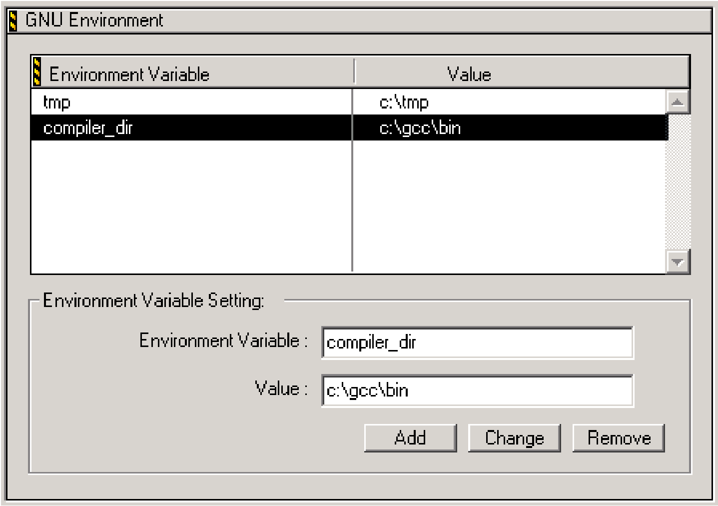
GNU Environment

Use the **GNU Environment** target settings panel to define environment variables that the GNU compiler, linker, assembler, and other build tools can reference.

NOTE If you add environment variables to this panel, the IDE will use the copy of `cygwin1.dll` in the *`installDir`*`\Cross_Tools\` directory to invoke the GNU build tools. If you have Cygwin installed elsewhere on your system and want the IDE to use the copy of `cygwin1.dll` in this installation, do not add any environment variables to this panel.

Figure 3.20 shows the **GNU Environment** target settings panel.

Figure 3.20 GNU Environment Panel



Environment Variable/Value List Box

This Environment Variable/Value list box displays each environment variable/value pair currently defined.

If click on an entry in this list box, the environment variable name contained in this entry appears in the Environment Variable text box, and the value contained in this entry appears in the Value text box. You can then modify these strings using these text boxes.

Environment Variable

Use the Environment Variable text box, to enter the name of the environment variable you want to add, change or remove.

To *add* an environment variable:

1. Type the variable's name in the Environment Variable text box
2. Type the variable's value in the Value text box
3. Click **Add**.

   The new environment variable/value pair appears in the Environment Variable/Value list box.

To *change* the name or value of an environment variable:

1. Click the entry in the Environment Variable/Value list box for the environment variable you want to change.

   The environment variable name contained in this entry appears in the Environment Variable text box, and the value contained in this entry appears in the Value text box.
2. Optionally, modify the name in the Environment Variable text box.
3. Optionally, modify the value in the Value text box.
4. Click **Change**.

   The modified environment variable name and/or value appears in the Environment Variable/Value list box.

To *remove* an environment variable:

1. Click the entry in the Environment Variable/Value list box for the environment variable you want to remove.

   The environment variable name contained in this entry appears in the Environment Variable text box and the value contained in this entry appears in the Value text box.
2. Click **Remove**.

   The selected environment variable is removed from in the Environment Variable/Value list box.

Value

Use the Value text box, to enter the name of the environment variable you want to add, change, or remove.

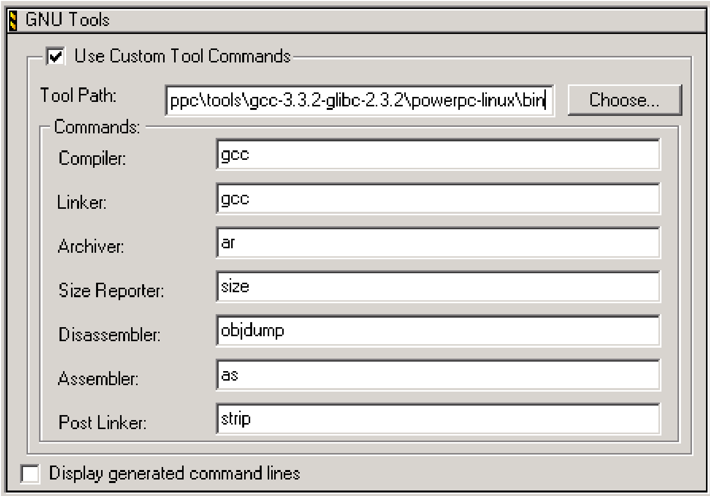
See the Environment Variable topic for instructions that explain how to add, change, and remove an environment variable.

GNU Tools

Use the **GNU Tools** settings panel to specify the path to the GNU build tools and to define the particular tool within a tool class (compiler, archiver, etc.) the IDE uses.

Figure 3.21 shows the **GNU Tools** target settings panel.

Figure 3.21 GNU Tools Panel



Use Custom Tool Commands

Check the Use Custom Tool Commands checkbox if you want to specify your own tools path or tools executables.

Tool Path

Use the Tool Path text box to enter the path to the cross-compiler tools on your system.

Commands

Use the text boxes in the Commands group box to enter the names of the GNU tools you want to use.

The Commands group box has these text boxes:

- Compiler

  Enter the name of the compiler that you want to use.

- Linker

  Enter the linker that you want to use.

- Archiver

  Enter the archiver that you want to use.

- Size Reporter

  Enter the size reporter utility that you want to use.
- Disassembler

  Enter the disassembler utility that you want to use.
- Assembler

  Enter the assembler utility that you want to use.
- Post Linker

  Enter the post linker that you want to use.

NOTE The specified post linker runs only if you select EPPC Linux GNU Post-linker from the Post-linker dropdown menu of the Target Settings panel.

Display generated command lines

Check the Display generated command lines checkbox if you want the IDE to display each command line it passes to the GNU build tools during the build process.

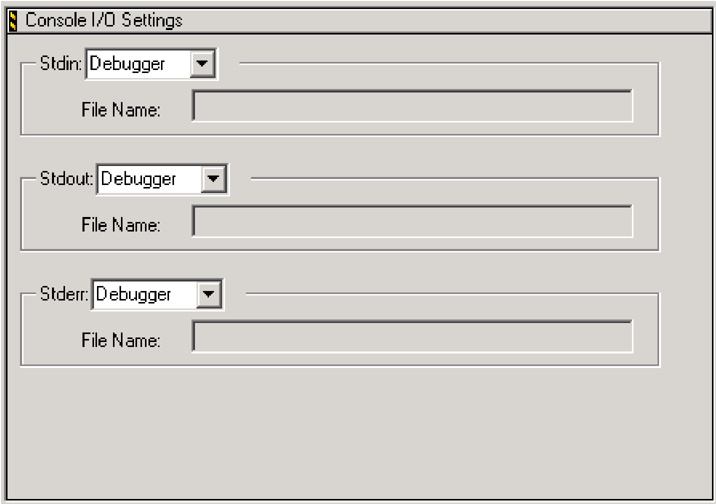
Console I/O Settings

Use the **Console I/O Setting** target settings panel to define the locations to which `stdin`, `stdout`, and `stderr` are redirected when a Linux application is run under control of the debugger.

NOTE The **Console I/O Settings** panel is not present in the panel list of the **Target Settings** window unless the build target's remote connection is EPPC Linux CodeWarrior TRK.
For instructions that explain how to assign a remote connection to a build target, see Working with Remote Connections.

Figure 3.22 shows the **Console I/O Setting** target settings panel.

Figure 3.22 Console I/O Settings Panel



You can redirect `stdin`, `stdout`, and `stderr` to:

- A file on the target system
- The debugger's console window
- The console window from which you launched CodeWarrior TRK (also known as)

In most cases, to redirect `stdin`, `stdout`, or `stderr` to a file on the target system, you must specify the full target-side path of the file as well as the file name. However, if the target-side location of the file to which you want to redirect input or output is the same as the directory in which CodeWarrior TRK resides, you must only supply the root file name.

Stdin

Use the **Stdin** dropdown menu to define the place from which data an application reads from `stdin` comes while the application is running under control of the debugger.

The options are:

- File

  Select File if you want the data the application being debugged reads from `stdin` to come from the specified target-side file.

- Debugger

  Select Debugger if you want the data the application being debugged reads from `stdin` to come from the debugger's console window.

- Console I/O

Select Console I/O if you want the data the application being debugged reads from `stdin` to come from the console window from which CodeWarrior TRK was launched.

Stdout

Use the **Stdout** dropdown menu to define the place at which data an application writes to `stdout` appears while the application is running under control of the debugger.

The options are:

- File

  Select File if you want the data the application being debugged writes to `stdout` to appear in the specified target-side file.

- Debugger

  Select Debugger if you want the data the application being debugged writes to `stdout` to appear in the debugger's console window.

- Console I/O

  Select Console I/O if you want the data the application being debugged writes to `stdout` to appear in the console window from which CodeWarrior TRK was launched.

Stderr

Use the **Stderr** dropdown menu to define the place at which data an application writes to `stderr` appears while the application is running under control of the debugger.

The options are:

- File

  Select File if you want the data the application being debugged writes to `stderr` to appear in the specified target-side file.

- Debugger

  Select Debugger if you want the data the application being debugged writes to `stderr` to appear in the debugger's console window.

- Console I/O

  Select Console I/O if you want the data the application being debugged writes to `stderr` to appear in the console window from which CodeWarrior TRK was launched.

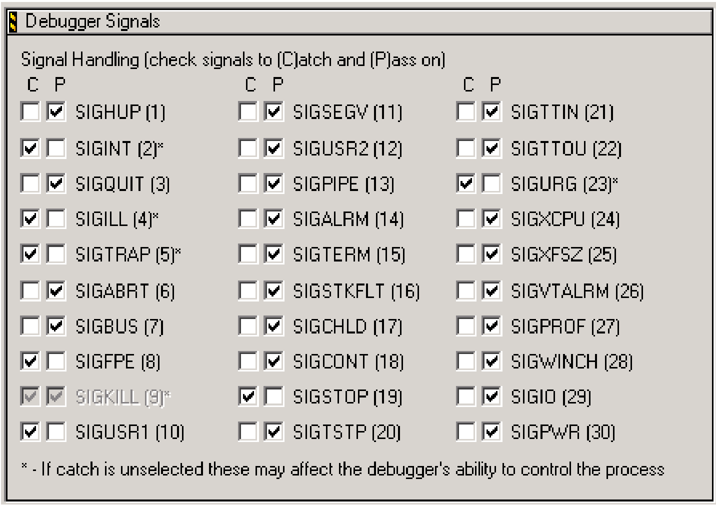
Debugger Signals

Use the **Debugger Signals** target settings panel to define how CodeWarrior TRK (also known as) handles Linux signals on behalf of the debugger.

NOTE The **Debugger Signals** panel is not present in the panel list of the **Target Settings** window unless the build target's remote connection is EPPC Linux CodeWarrior TRK.
For instructions that explain how to assign a remote connection to a build target, see Working with Remote Connections.

Figure 3.23 shows the **Debugger Signals** target settings panel.

Figure 3.23 Debugger Signals Panel



For a given signal:

- If you check just the C (catch) checkbox, when the signal is raised, CodeWarrior TRK sends an event to the debugger. When the debugger user continues the process being debugged, CodeWarrior TRK does *not* pass the signal to this process.
- If you check just the P (pass on) checkbox, when the signal is raised, CodeWarrior TRK passes the signal to the process being debugged. In this case, CodeWarrior TRK does not send an event to the debugger.
- If you check both the C and the P boxes, when the signal is raised, CodeWarrior TRK sends an event to the debugger. Then, when the debugger user continues the process being debugged, CodeWarrior TRK passes the signal to this process.

- If you uncheck *both* the C and P boxes, when the signal is raised, CodeWarrior TRK squelches it. In other words, CodeWarrior TRK does not pass the signal to the debugger or to the process being debugged.

To ensure that the CodeWarrior debugger can control the process being debugged, always check C for these signals:

- SIGINT
- SIGILL
- SIGTRAP
- SIGKILL
- SIGURG

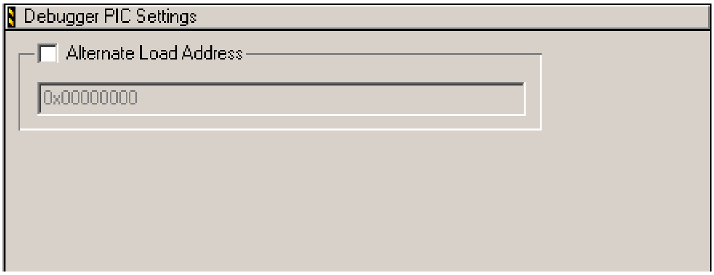
Debugger PIC Settings

Use the **Debugger PIC Settings** target settings panel to specify an alternate address for the debugger to load a PIC module on a target board.

Usually, Position Independent Code (PIC) is linked in such a way that the entire image starts at address `0x00000000`. The **Debugger PIC Settings** panel lets you specify an alternate address at which the debugger will load the PIC module in target memory.

Figure 3.24 shows the **Debugger PIC Settings** target settings panel.

Figure 3.24 Debugger PIC Settings Panel



Alternate Load Address

To specify an alternate load address, check the Alternate Load Address checkbox and then type the alternate address in the associated text box. The debugger will load your binary on the target at the specified address.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.

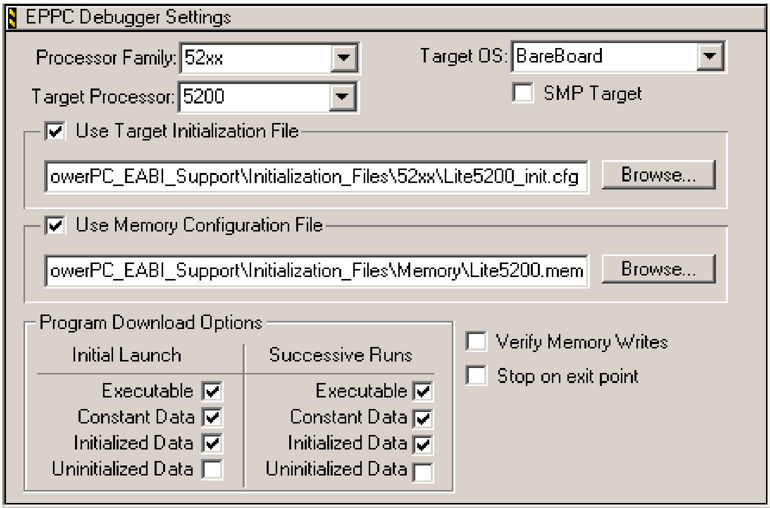
NOTE The debugger does not verify whether your code can execute at the new address. As a result, the PIC generation settings of the compiler, linker and your program's startup routines must correctly set any base registers and perform any required relocations.

EPPC Debugger Settings

Use the **EPPC Debugger Settings** target settings panel to provide information the debugger needs to work with the target and to define how and when the debugger downloads portions of your binary to the target.

Figure 3.25 shows the **EPPC Debugger Settings** target settings panel.

Figure 3.25 EPPC Debugger Settings Panel



Processor Family

Use the Processor Family dropdown menu to select the processor family of the processor on your target board. The family that you select defines the processors that appear in the Target Processor dropdown menu. (See below.)

Target Processor

Use the Target Processor dropdown menu to select the processor on your target board.

Target OS

Use the Target OS dropdown menu to select the operating system running on your board.

The choices are:

- BareBoard

  Enables bare board debugging.

  Select this option if you are not using an operating system.

- OSEK

  Enables OSEK Aware debugging.

  Select this option if your board is running an implementation of the OSEK real-time operating system.

  Selecting OSEK enables OIL (Object Interface Language) support which, in turn, lets the debugger interpret the information in the ORTI (OSEK Run Time Interface) file generated when you built your OSEK image.

SMP Target

Check the SMP Target checkbox if you want to use symmetric multiprocessing (SMP) debugging mode. The SMP debugging mode involves debugging a single executable image that is shared by multiple cores. SMP mode implies a shared memory model. Consequently, there is a single CodeWarrior project for the application that is shared by all the cores.

NOTE SMP mode can be used only for 8641D core #0. In the Remote Debugging Panel, check the Multi-Core Debugging checkbox and select Core Index #0.

Use Target Initialization File

Check the Use Target Initialization File checkbox if you want the current build target to use a target initialization file. Type the full path and name of the initialization file you want, or click **Browse** to display a dialog box with which you can select the required file.

NOTE The New Project Wizard automatically selects the correct target initialization files for the board selected at wizard-time.

Sample target initialization files are in the `PQ1`, `PQ2`, `PQ3`, `Host`, and `52xx` subdirectories of this path:

`installDir\PowerPC_EABI_Support\Initialization_Files\`

See Target Initialization Files for documentation that explains the purpose of these files and the commands that can appear in them.

Use Memory Configuration File

Check the Use Memory Configuration File checkbox if you want to use a memory configuration file. Type the full path and name of the memory configuration file you want, or click **Browse** to display a dialog box with which you can select the required file.

A memory configuration file defines the memory access rules (restrictions, translations) used each time the debugger needs to access memory on the target board.

Sample memory configuration files are in this directory:

`*installDir*\PowerPC_EABI_Support\Initialization_Files\memory\`

If you are using a memory configuration file and you try to read from an invalid address, the debugger fills the memory buffer with a reserved character (defined in the memory configuration file).

If you try to write to an invalid address, the write command is ignored and fails.

You can change a memory configuration file during a debug session.

See Memory Configuration Files for documentation that explains the purpose of these files and the commands that can appear in them.

Program Download Options

Use the options in the Program Downloads group box to define the sections of your program that the debugger downloads to the target board initially and on successive runs.

The initial run is the first time the debugger downloads your program to the target board for execution.

Successive runs are the second through last times the debugger downloads your program to the target board for execution.

The program section download options are:

- Executable

  Executable sections contain your program’s code.

  Check the Executable box in the Initial Launch group to instruct the debugger to download your program’s executable sections to the target board the first time the debugger runs the program.

  Check the Executable box in the Successive Runs group to instruct the debugger to download the executable sections on each successive run of the program.

- Constant Data

  Constant data sections contain your program’s constants.

Check the Constant Data box in the Initial Launch group to instruct the debugger to download your program's constant data sections to the target board the first time the debugger runs the program.

Check the Constant Data box in the Successive Runs group to instruct the debugger to download the constant data sections on each successive run of the program.

- Initialized Data

  Initialized data sections contain your program's modifiable data.

  Check the Initialized Data box in the Initial Launch group to instruct the debugger to download your program's initialized data sections to the target board the first time the debugger runs the program.

  Check the Initialized Data box in the Successive Runs group to instruct the debugger to download the initialized data sections on each successive run of the program.

- Uninitialized Data

  Uninitialized data sections contain your program's uninitialized variables.

  Check the Uninitialized Data box in the Initial Launch group to instruct the debugger to download your program's uninitialized data sections to the target board the first time the debugger runs the program.

  Check the Uninitialized Data box in the Successive Runs group to instruct the debugger to download the uninitialized data sections on each successive run of the program.

NOTE You do not need to download uninitialized data if you are using CodeWarrior runtime code because this code initializes this data for you.

Verify Memory Writes

Check the Verify Memory Writes checkbox to instruct the debugger to verify that each program section is downloaded to the target without error. The debugger then verifies that sections selected to be downloaded are written correctly to the target board's memory. However, the debugger does not check for modified data in other sections.

Stop on exit point

Check the Stop on exit point checkbox to instruct the debugger to set a breakpoint at the exit point of the code. This is valid for CodeWarrior but not for other runtime environments like VxWorks. In case of multi-core projects if you set the checkbox for a project on one core it is set for projects on both the cores.

System Controller

Select the system controller on the target processor. This setting controls which system controller registers the IDE displays in the **Register Details** window during a debug session.

NOTE The System Controller dropdown menu is only available if you select a Host processor (such as 7xx or 74xx) from the Processor dropdown menu.

Table 3.7 lists and describes each item in the System Controller menu.

Table 3.7 System Controller Menu Items

Item	Description
None	Display no system controller registers; show only CPU registers.
107	Display the MPC107 system controller registers.
109	Display the Tundra 109 system controller registers.

NOTE Setting the System Controller option to a system controller other than the one on the target system, and then using the **Register Details** window to view the system controller registers on the target system may cause target instability.

EPPC Exceptions

The **EPPC Exceptions** target settings panel lists each of the EPPC exceptions that the CodeWarrior debugger can catch.

Use this panel to select the EPPC exceptions that you want the debugger to catch.

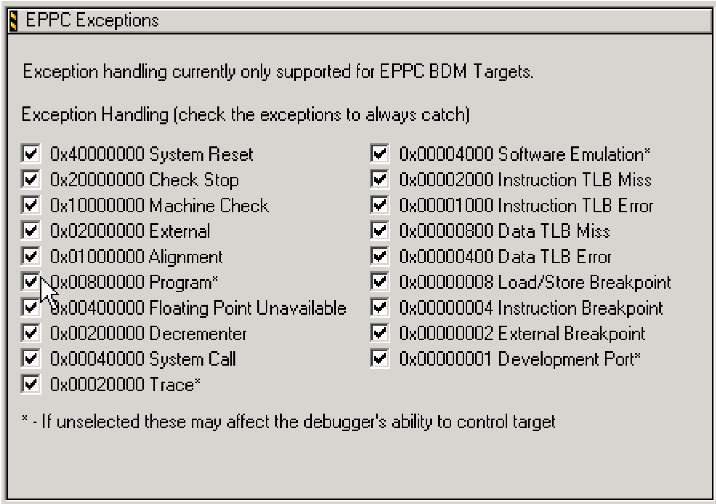
NOTE This panel applies only to processors that have BDM and PQ3 debug module.

EPPC Exceptions for BDM Target

Check all of the checkboxes in this panel if you want the debugger to catch all the listed exceptions. Clear the checkboxes for those exceptions that you would prefer to handle. By default, catching all exceptions is enabled for BDM target.

Figure 3.26 shows the **EPPC Exceptions** target settings panel for BDM target.

Figure 3.26 EPPC Exceptions Panel for BDM Target



The settings in this panel define the value to which the target EPPC processor's Debug Enable Register (DER) is set. The value of the DER register, in turn, defines which exceptions are caught and which are ignored by the processor's Background Debug Module (BDM) on-chip debug interface. Consult your processor's documentation for more information about the DER register and BDM.

To ensure that the CodeWarrior debugger works properly, always check these exceptions:

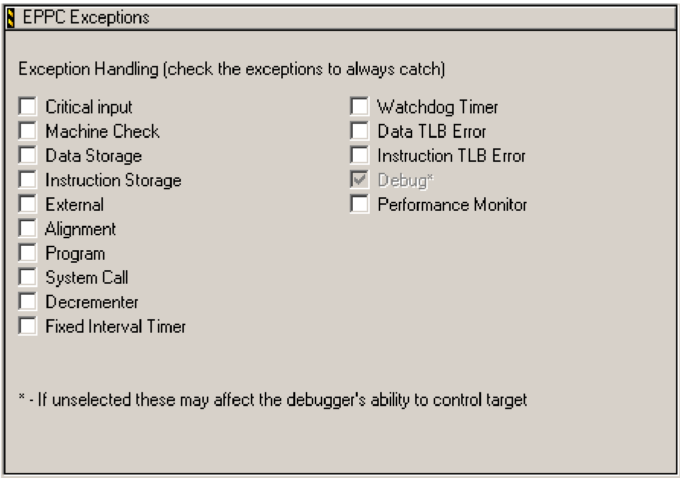
- `0x00800000 Program` — for software breakpoints on some boards
- `0x00020000 Trace` — for single stepping
- `0x00004000 Software Emulation` — for software breakpoints on some boards
- `0x00000001 Development Port` — for halting the target processor.

EPPC Exceptions for PQ3 Target

Check the checkboxes in this panel if you want the debugger to catch the required exceptions. By default, catching all exceptions is disabled for PQ3 target. Only the Debug exception is caught, as the debugger uses this exception for setting breakpoints. Catching the debug exception cannot be unset.

Figure 3.26 shows the **EPPC Exceptions** target settings panel for PQ3 target.

Figure 3.27 EPPC Exceptions Panel for PQ3 Target



Checking any of the checkboxes 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.

To ensure that the CodeWarrior debugger works properly, the debug exception is always set. Catching the selected exceptions works only if the target is debugged. Make sure that you click only the Debug icon.

NOTE Catching PQ3 exceptions is not supported for e500 simulator.

EPPC Trace Buffer

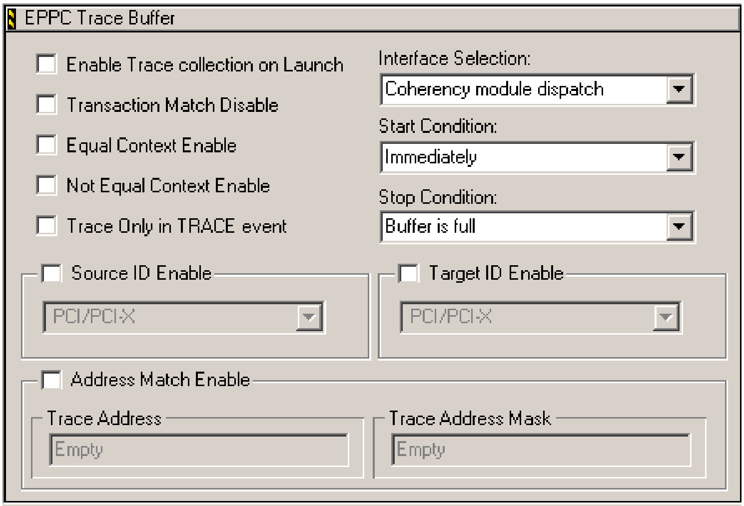
Use the **EPPC Trace Buffer** target settings panel to configure the trace events you want to capture while debugging a target equipped with a trace buffer.

The options in this panel correspond to bits in the trace configuration registers `TBCR0` and `TBCR1`, the address register `TBAR`, the address mask register `TBAMR`, and the transaction mask register `TBTMR`.

NOTE For more information about using the EPPC trace buffer with the debugger, see the EPPC Trace Buffer Support topic.

Figure 3.28 shows the **EPPC Trace Buffer** target settings panel.

Figure 3.28 EPPC Trace Buffer Panel



Enable Trace collection on Launch

Check the Enable Trace collection on Launch checkbox to start trace event collection when you connect to the target board. The monitor configures the trace buffer each time you do a software or hardware reset from the CodeWarrior debugger. The trace buffer can be reconfigured at any time during a debug session.

Transaction Match Disable

Check the Transaction Match Disable checkbox to ignore the transaction type match when the monitor receives a trace buffer event. Clear this checkbox to have the monitor report only transaction types that match the transaction mask from the `TBTMR` register.

NOTE Currently, the debugger reports all possible transaction types for a particular interface.

Equal Context Enable

Check the Equal Context Enable checkbox to record trace events only if the current context (the value of `CCIDR` register) is equal to the programmed context (the value of `PCIDR` register).

NOTE Do not check this checkbox and the Not Equal Context Enable checkbox at the same time. If both checkboxes are checked, the watchpoint monitor will not record trace events.

Not Equal Context Enable

Check the Not Equal Context Enable checkbox to record trace events only if the current context (the value of `CCIDR` register) is not equal to the programmed context (the value of `PCIDR` register).

NOTE Do not check this checkbox and the Equal Context Enable checkbox at the same time. If both checkboxes are checked, the watchpoint monitor will not record trace events.

Trace Only in TRACE Event

Check the Trace Only in TRACE Event checkbox to trace only cycles in which the monitor detects a trace event. Clear this checkbox to have the monitor trace all valid transactions.

NOTE If the trace buffer is not properly configured to specify traceable events, the monitor traces every valid address.

Interface Selection

Select an item from the Interface Selection dropdown menu to specify the interface you want to trace. Selecting an interface activates tracing for all possible transaction types specific to that interface.

For more information see the description of Trace Buffer Transaction Mask Register (`TBTMR`) in the *MPC8560 Reference manual*.

Start Condition

Select an item from the Start Condition dropdown menu to define the event that causes the monitor to start watching for traceable events.

Table 3.8 lists and describes the items in this menu.

Table 3.8 Start Conditions

Item	Description
Immediately	Start tracing immediately after configuring the trace buffer.
Watchpoint event detected	Start tracing when the monitor detects a watchpoint event.
Trace Buffer event detected	Start tracing when the monitor detects a trace buffer event.
Performance monitor overflow	Start tracing when the performance monitor signals that an overflow occurred.
TRIG_IN 0 to 1 transition	Start tracing when the value of the `TRIG_IN` signal changes from 0 to 1.
TRIG_IN 1 to 0 transition	Start tracing when the value of the `TRIG_IN` signal changes from 1 to 0.
Context: Current == programmed	Start tracing when the current context ID is equal to the programmed context ID.
Context: Current != programmed	Start tracing when the current context ID is not equal to the programmed context ID.

Stop Condition

Select an item from the Stop Condition dropdown menu to define the event that causes the monitor to stop watching for traceable events.

Table 3.9 lists and describes the items in this menu.

Table 3.9 Stop Conditions

Item	Description
Buffer is full	Stop tracing once all 256 elements of the trace buffer are recorded.
Watchpoint event detected	Stop tracing once the monitor detects a watchpoint event.
Trace Buffer event detected	Stop tracing once the monitor detects a trace event.

Table 3.9 Stop Conditions (*continued*)

Item	Description
Performance monitor overflow	Stop tracing when the performance monitor signals that an overflow occurred.
TRIG_IN 0 to 1 transition	Start tracing when the value of the `TRIG_IN` signal changes from 0 to 1.
TRIG_IN 1 to 0 transition	Start tracing when the value of the `TRIG_IN` signal changes from 1 to 0.
Context: Current == programmed	Start tracing when the current context ID is equal to the programmed context ID.
Context: Current != programmed	Start tracing when the current context ID is not equal to the programmed context ID.

Source ID Enable

Check the Source ID Enable checkbox and select a block or port from the dropdown menu to record only trace events whose transaction source ID matches the selected block or port.

Table 3.10 lists the valid source IDs.

Table 3.10 Transaction Source Identifiers

PCI1	CPM
PCI2	DMA
PCI Express	SAP
Local Bus	Ethernet 0
Security	Ethernet 1
Config Space	Ethernet 2
Boot Sequencer	Ethernet 3
Rapid IO	Rapid IO Message
Local Space DDR	Rapid IO Doorbell
Local Processor Instruction Fetch	Rapid IO Port Write
Local Processor Data Fetch	

NOTE If you select an invalid block or port, no transaction will target the specified block or port and, as a result, the monitor will not record trace events.

Target ID Enable

Check the Target ID Enable checkbox and select a block or port from the dropdown menu to record only trace events whose transaction target ID matches the selected block or port.

Table 3.11 lists the valid target IDs.

Table 3.11 Transaction Target IDs

PCI1	CPM
PCI2	DMA
PCI Express	SAP
Local Bus	Ethernet 0
Security	Ethernet 1
Config Space	Ethernet 2
Boot Sequencer	Ethernet 3
Rapid IO	Rapid IO Message
Local Space DDR	Rapid IO Doorbell
Local Processor Instruction Fetch	Rapid IO Port Write
Local Processor Data Fetch	

NOTE If you select an invalid block or port, no transaction will target the specified block or port, and as a result, the monitor will not record trace events.

Address Match Enable

Check the Address Match Enable checkbox to record only trace events whose trace address matches the transaction address. Enter an address in the Trace Address text box. Enter a mask in the Trace Address Mask text box. The monitor masks the trace address by excluding the address mask bits before comparison.

Source Folder Mapping

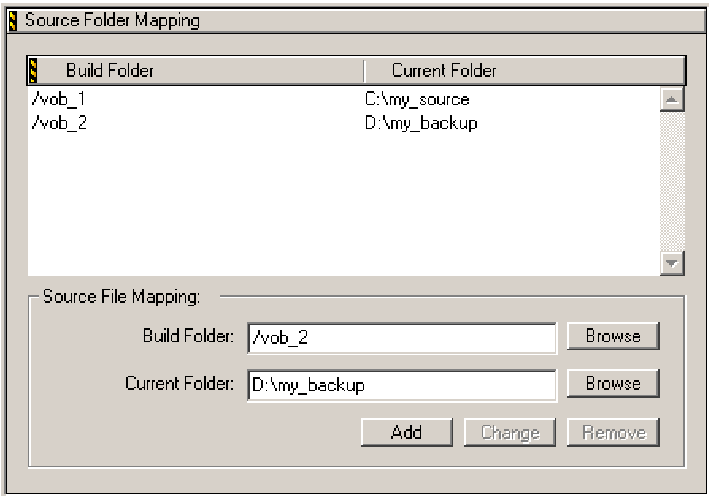
Use the **Source Folder Mapping** target settings panel if you are debugging an ELF binary that was built in one place, but which is being debugged from another.

The mapping information you supply lets the CodeWarrior debugger find and display the binary's source code files even though they are not in the locations specified in the ELF file's debug information.

NOTE If you create a CodeWarrior project by opening an ELF file in the IDE, the IDE automatically creates entries in the **Source Folder Mapping** panel. The IDE creates these entries using the current folder information you provide during the project creation process and the existing folder information in the ELF's debug information.

Figure 3.29 shows the **Source Folder Mapping** target settings panel.

Figure 3.29 Source Folder Mapping Panel



Build Folder

Use the Build Folder text box to enter the path that contained the executable's source files when this executable was originally built. Alternatively, click **Browse** to display a dialog box you can use to select the correct path.

The supplied path can be the root of a source code tree. For example, if your source code files were in the directories

```
/vob/my_project/headers
/vob/my_project/source
```

you can enter `/vob/my_project` in the Build Folder text box.

If the debugger cannot find a file referenced in the executable's debug information, the debugger replaces the string `/vob/my_project` in the missing file's name with the associated Current Folder string and tries again. The debugger repeats this process for each Build Folder/Current Folder pair until it finds the missing file or no more folder pairs remain.

Current Folder

Use the Current Folder text box to enter the path that contains the executable's source files now, that is, at the time of the debug session. Alternatively, click **Browse** to display a dialog box you can use to select the correct path.

The supplied path can be the root of a source code tree. For example, if your source code files are now in the directories

```
C:\my_project\headers
C:\my_project\source
```

you can enter `C:\my_project` in the Current Folder text box.

If the debugger cannot find a file referenced in the executable's debug information, the debugger replaces the Build Folder string in the missing file's name with the string `C:\my_project` and tries again. The debugger repeats this process for each Build Folder/Current Folder pair until it finds the missing file or no more folder pairs remain.

Add

Click the **Add** button to add the current Build Folder/Current Folder association to the Source Folder Mapping list.

Change

Click the **Change** button to change the Build Folder/Current Folder mapping currently selected in the Source Folder Mapping list.

Remove

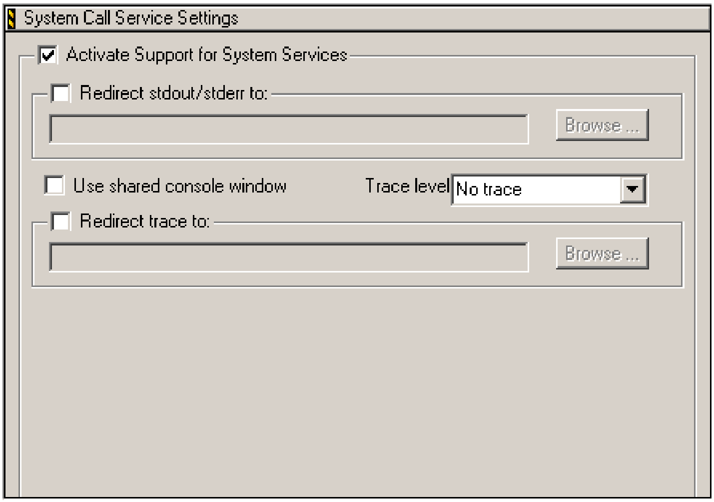
Click the **Remove** button to remove the Build Folder/Current Folder mapping currently selected in the Source Folder Mapping list.

System Call Service Settings

Use the **System Call Service Settings** target settings panel to activate the debugger's support for system calls and to select options the define how the debugger handles system calls.

Figure 3.30 shows the **System Call Service Settings** target settings panel.

Figure 3.30 System Call Service Settings Panel



The CodeWarrior debugger provides system call support over JTAG. System call support lets bare board applications use the functionality of host OS service routines. This feature is useful if you do not have a board support package (BSP) for your target board.

The host debugger implements these services. Therefore, the host OS service routines are available only when you are debugging a program on a target board or simulator.

NOTE The OS service routines provided must comply with an industry-accepted standard. The definitions of the system service functions provided are a subset of Single UNIX Specification (SUS).

Activate Support for System Services

Check the Activate Support for System Services checkbox to enable support for system services. All the other options in the **System Call Service Setting** panel are available only if you check this checkbox.

Redirect stdout/stderr to

The default place at which output written to `stdout` and `stderr` appears in a CodeWarrior IDE “console” window.

To redirect this output to a file, check the Redirect stdout/stderr to checkbox. Click **Browse** to display a dialog box with which you can define the path and name of this file.

Use Shared Console Window

Check the Use shared console window checkbox if you wish to share the same console window between different debug targets. This setting is useful in multi-core or multi-target debugging.

Trace Level

Use the Trace level dropdown menu to specify the system call trace level. The system call trace level options available are:

- No Trace — system calls are not traced
- Summary Trace — the requests for system services are displayed
- Detailed Trace — the requests for system services are displayed along with the arguments/parameters of the request

The place where the traced system service requests are displayed is determined by the Redirect trace to checkbox.

Redirect Trace to

The default place at which traced system service requests appear is in a CodeWarrior IDE “console” window.

To log traced system service requests to a file, check the Redirect trace to checkbox. Click **Browse** to display a dialog box with which you can define the path and name of this file.

NOTE In a a project created by the New Project wizard, use the library syscall.a rather than a UART library. syscall.a is in this directory:
`installDir\PowerPC_EABI_Support\SystemCallSupport\Lib`

PC-lint Target Settings Panels

PC-lint is a third-party software development tool that checks C/C++ source code for bugs, inconsistencies, non-portable constructs, redundant code, and other problems.

CodeWarrior for Power Architecture™ Processor includes target settings panels and plug-ins that let you configure and use PC-lint from within the CodeWarrior IDE. However, the PC-lint software itself is *not* included with your CodeWarrior product. As a result, you must obtain and install a copy of PC-lint before you can use it with the CodeWarrior IDE. Among other places, PC-lint is available from its developers, Gimpel Software (www.gimpel.com).

NOTE The default CodeWarrior PC-lint configuration requires that your PC-lint installation to be in `installDir\Lint`
(where `installDir` is the path to your CodeWarrior product.)
That said, you can install PC-lint anywhere and then adjust the CodeWarrior configuration to match.

Once you have installed PC-lint, you can configure any build target of any CodeWarrior project to use this software. To do this, follow these steps:

1. Open a project and select the build target with which you want to use PC-lint.
2. Display the **Target Settings** window for this build target.
3. Display the Target Settings panel in the **Target Settings** window.
4. In the Target Settings panel, choose PCLint Linker from Linker dropdown menu.

   The PCLint Main Settings and PCLint Options target settings panels appear in the panel list of the **Target Settings** window. In addition, the IDE removes panels that pertain to ELF generation and debugging from the panel list.
5. Choose the PC-lint configuration options appropriate for your build target using the PC-lint target settings panels.

Table 3.12 lists and describes each PC-lint target settings panel.

The sections this table explain each option available in these panels.

Table 3.12 PC-lint Target Settings Panels

Target Settings Panel	Description
PCLint Main Settings	Use this panel to provide the path to the PC-lint executable and to define the compiler option files and prefix file that PC-lint will use.
PCLint Options	Use this panel to define the syntax rules PC-lint uses to validate your C/C++ source code, to define the environment with which PC-lint must ensure your code conforms, and to pass command-line switches to PC-lint.

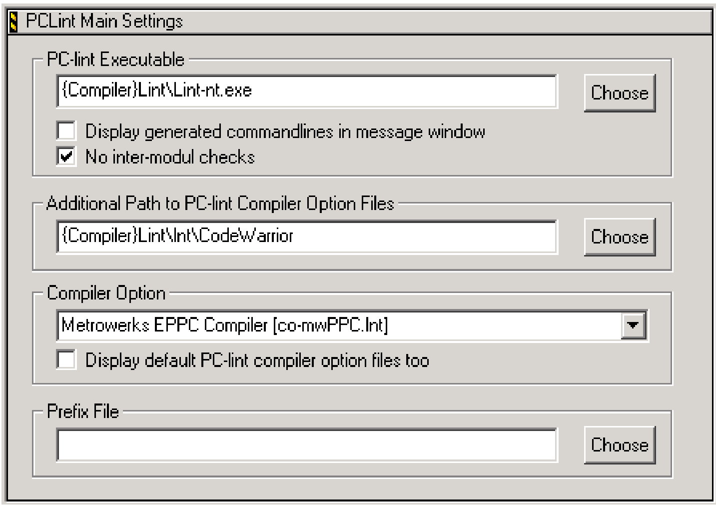
PCLint Main Settings

Use the **PCLint Main Settings** settings panel to provide the path to the PC-lint executable and to define the compiler option files and prefix file that PC-lint uses.

NOTE The PC-lint target settings panels are available only if you first select PCLint Linker in the **Target Settings** panel. (See Figure 3.4.)

Figure 3.31 shows the **PCLint Main Settings** panel.

Figure 3.31 PCLint Main Settings Panel



PC-lint Executable

In the PC-lint Executable text box, type the path to and name of the PC-lint executable. Alternatively, click **Choose** to display a dialog box with which you can select this file.

NOTE The default PC-lint path is `{Compiler}Lint\Lint-nt.exe`. If you installed PC-lint somewhere else, replace this default path with the correct PC-lint executable path.

Display generated command lines in message window

Check this box to instruct the IDE to display the command line it passes to PC-lint in the **Errors & Warnings** window.

No inter-modul checks

Check the No Inter-modul checks box to instruct PC-lint to do *no* inter-module checking.

NOTE If you uncheck this box, PC-lint takes longer to process your build target's source files.

Additional Path to PC-lint Compiler Option Files

The IDE's default behavior is to use any PC-lint compiler option files (`*.lnt`) it finds in the directory `{Compiler}\Lint\lnt`.

To configure a build target to use a PC-lint compiler option file in addition to those in the default directory, enter the path to the directory that contains this file in the Additional Path to PC-lint Compiler Option Files text box. If the specified directory contains any files that end with the suffix `.lnt`, the Compiler Option dropdown menu (see below) enables and displays these files.

The default CodeWarrior installation includes pre-written PC-lint compiler option files. They are in this directory:

`{CodeWarrior}Lint\lnt\CodeWarrior`

Each file in this directory is designed to work with a particular CodeWarrior compiler. Many users enter this path in the Additional Path to PC-lint Compiler Option Files text box and then choose the file for the CodeWarrior compiler they are using from the Compiler Option list.

You can leave this text box empty, if desired.

Compiler Option

Use the Compiler Option dropdown menu to select the PC-lint compiler option file for the CodeWarrior compiler the build target is using.

This menu displays all `.lnt` files in the directory specified in the Additional Path to PC-lint Compiler Option files text box. If this directory contains no `.lnt` files, the Compiler Option dropdown menu is disabled.

Display default PC-lint compiler option files too

Check this box to include the default `.lnt` files (the files in `{Compiler}Lint\lnt`) in the Compiler Option dropdown menu along with those in the directory specified in the Additional Path to PC-lint Compiler Option Files text box.

Prefix File

In the Prefix File text box, type the name of a prefix file to pass to PC-lint. Alternatively, click **Choose** to display a dialog box that lets you navigate to and select this file.

Typically, you use this feature to define macros to required values for a particular PC-lint run or to instruct PC-lint to check certain command-line commands. To do this, define this information in a prefix file.

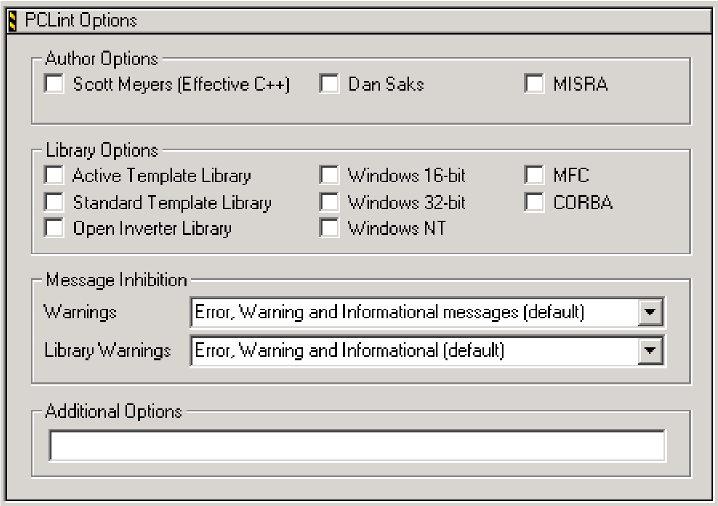
You may leave this text box empty, if desired.

PCLint Options

Use the **PCLint Options** target settings panel to define the syntax rules that PC-lint uses to validate your C/C++ source code, to define the environment (libraries, operating system, remote procedure call standard, etc.) with which PC-lint must ensure your code conforms, and to pass command-line switches to PC-lint.

Figure 3.32 shows the **PCLint Options** panel.

Figure 3.32 PC-lint Options Panel



Author Options

This Author Options group box contains checkboxes that let you select the set of syntax rules that PC-lint uses as it checks your code.

The options are:

- Scott Meyers (Effective C++)

  Check this box to instruct PC-lint to verify that your code adheres to the syntax rules documented in Effective C++.

- Dan Saks

  Check this box to instruct PC-lint to verify that your code adheres to the syntax rules recommended by Dan Saks.

- MISRA

  Check this box to instruct PC-lint to verify that your code adheres to the Motor Industry Software Reliability Association (MISRA) C language guidelines for safety-critical embedded software.

You can check none, some, or all boxes in this group.

Library Options

This Library Options group box contains checkboxes that let you define the environment with which PC-lint must ensure your code conforms.

The options are:

- Active Template Library

  Check this box to instruct PC-lint to validate your Active X Template (ATL) library code.

- Standard Template Library

  Check this box to instruct PC-lint to validate your Standard Template Library (STL) code.

- Open Inverter Library

  Check this box to instruct PC-lint to validate your Open Inverter Library code.

- Windows 16-bit

  Check this box to instruct PC-lint to validate your 16-bit Windows API calls.

- Windows 32-bit

  Check this box to instruct PC-lint to validate your 32-bit Windows API calls.

- Windows NT

  Check this box to instruct PC-lint to validate your Windows NT API calls.

- MFC

  Check this box to instruct PC-lint to validate your Microsoft Foundation Classes (MFC) code.

- CORBA

  Check this box to instruct PC-lint to validate your Common Object Request Broker Architecture (CORBA) code.

Warnings

Use the options in the Warnings dropdown menu to control the warning and error messages that PC-lint emits.

The default setting displays error, warning, and information messages.

Library Warnings

Use the options in the Library Warnings dropdown menu to control the warning and error messages that PC-lint emits for libraries.

The default setting displays error, warning and information messages.

Additional Options

In the Additional Options text box, type the PC-lint command-line arguments to be passed to PC-lint. Refer to your PC-lint manuals for documentation of these arguments.

4

Working with the Debugger

This chapter explains how to use the CodeWarrior™ development tools to debug both bare board and embedded Linux® software for Power Architecture™ processors.

See the appendix Debugger Limitations and Workarounds for documentation of processor-specific debugger limitations and workarounds.

NOTE This chapter documents debugger features that are specific to the CodeWarrior for Power Architecture Processors product. For documentation of debugger features that are in all CodeWarrior products, refer to the *CodeWarrior™ IDE User's Guide*.

The sections of this chapter are:

- Standard Debugger Features
- Debugging Bare Board Software
- Debugging Embedded Linux® Software

Standard Debugger Features

This section presents information that applies to debugging both bare board and embedded Linux software.

The topics are:

- Working with Remote Connections
- Setting the Watchpoint Type
- Attaching to Processes
- Ways to Initiate a Debug Session
- Displaying Register Contents
- Using the Register Details Window
- Viewing and Modifying Cache Contents
- Using CodeWarrior TRK
- Using the Command-Line Debugger

Working with Remote Connections

This section defines what remote connection are and explains how to define and use them.

The topics are:

- What is a Remote Connection?
- Using Remote Connections
- Predefined Remote Connections
- Editing a Remote Connection
- Creating a Remote Connection

What is a Remote Connection?

A *remote connection* is a named collection of configuration settings for a connection between the CodeWarrior debugger and a target board or simulator.

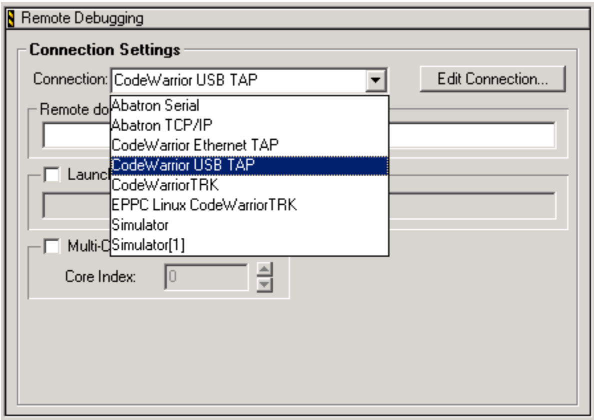
Each remote connection assigns values to parameters that apply to the type of connection between the debugger and the target.

For example, the CodeWarrior USB TAP remote connection defines the interface clock frequency at which the USB TAP runs, while the Abatron Serial remote connection defines the attributes (baud rate, number of data bits, etc.) of the serial link between the debugger and the Abatron probe.

Using Remote Connections

You use a remote connection by assigning it to a build target. To do this, display the **Remote Debugging** target settings panel and select the remote connection you want to use from the Connection dropdown menu. (See Figure 4.1.)

Figure 4.1 The Remote Debugging Target Settings Panel



For example, you might assign the CodeWarrior USB TAP remote connection to a build target. When you debug this build target, the CodeWarrior debugger configures the USB TAP as specified in this remote connection and then uses the USB TAP to connect to and control your target board.

Predefined Remote Connections

Your CodeWarrior product includes several predefined remote connections for the probes that have been tested with the supported target boards. These remote connections define configuration parameters that work for the majority of these boards.

Table 4.1 lists and describes the pre-defined remote connections.

Table 4.1 Predefined Remote Connections

Remote Connection Name	Description
Abatron Serial	Assign this remote connection to a build target if you are using an Abatron probe and this device communicates with your development PC over a serial link.
Abatron TCP/IP	Assign this remote connection to a build target if you are using an Abatron probe and this device communicates with your development PC over a TCP/IP link.
CodeWarrior Ethernet TAP	Assign this remote connection to a build target if you are using an Ethernet TAP probe.

Table 4.1 Predefined Remote Connections

Remote Connection Name	Description
CodeWarrior USB TAP	Assign this remote connection to a build target if you are using a USB TAP probe.
CodeWarriorTRK	Assign this remote connection to a build target if you are using the CodeWarriorTRK software debug monitor to debug a bare board program on your target board.
EPPC Linux CodeWarrior TRK	Assign this remote connection to a build target that generates a binary for execution by the embedded Linux operating system on your target board. EPPC Linux CodeWarrior TRK also needs CodeWarrior TRK to work.
Simulator	Assign this remote connection to a build target if you are using a simulator to debug a bare board program. This remote connection is configured to use the `ccssim2` simulator.
Simulator[1]	Assign this remote connection to a build target if you are using a simulator to debug a bare board program. This remote connection is configured to use the `SimRun` simulator. **NOTE:** `SimRun` is the simulator connection to use with the Linux-hosted CodeWarrior tools. It is also based on `ccssim2`.

Editing a Remote Connection

The predefined remote connections reliably cover typical use cases. That said, you might have to change a setting to get a particular remote connection to work with your target board. This section explains how to edit a remote connection.

NOTE You can also create a new remote connection, so you do not have to change one of the pre-defined ones.

First you select a debugger protocol. A *debugger protocol* is the protocol that the CodeWarrior debugger and a probe use to communicate.

Table 4.2 lists each debugger protocol supported by the CodeWarrior debugger.

Table 4.2 Debugger Protocols

Debugger Protocol	Description
EPPC - Abatron	Select to define a remote connection that uses an Abatron probe.
EPPC - CodeWarriorTRK	Select to define a remote connection that uses the CodeWarrior TRK software debug monitor designed for used with a bare board.
EPPC - Linux CodeWarriorTRK	Select to define a remote connection that uses the CodeWarrior TRK software debug monitor designed for used with embedded Linux.
CCS EPPC Protocol Plugin	Select to define a remote connection that uses CCS (CodeWarrior Connection Server) to control a probe or simulator.

Once you have selected a debugger protocol, you must assign a *connection type* to this protocol. A connection type defines the type and attributes of the communications link between the CodeWarrior debugger and a probe.

Each debugger protocol supports one or more connection type (CCS remote, TCP/IP, USB, or Serial).

Table 4.3 lists the debugger protocols and the connection types each supports.

Table 4.3 Connection Types Supported by each Debugger Protocol

Debugger Protocol	Supported Connection Types
EPPC - Abatron	Serial, TCP/IP
EPPC - CodeWarriorTRK	Serial
EPPC - Linux CodeWarriorTRK	Serial, TCP/IP
CCS EPPC Protocol Plugin	CCS Remote Connection, Ethernet TAP, USBTAP

Based on the selected debugger protocol and connection type, the IDE makes different connections settings available. For example, if you select Serial for connection type, the IDE presents settings for baud rate, stop bits, flow control, and so on.

To configure a remote connection to work with the particular hardware debug probe, software debug probe, or simulator you are using, you must edit the connection settings.

You access the settings with the **Edit Connection** dialog box. You can display this dialog box in one of these ways:

- In the **Remote Connections** preference panel, select a connection from the list, and click **Change**. The **Edit Connection** dialog box appears.
- In the **Remote Connections** preference panel, click **Add** to create a new remote connection. The **New Connection** dialog box appears.
- In the **Remote Debugging** target settings panel, select a connection from the Connection dropdown menu, then click the **Edit Connection** button. The **Edit Connection** dialog box appears.

This section explains the purpose of each setting available for each connection type.

The topics are:

- Serial
- TCP/IP
- CCS Remote Connection
- Ethernet TAP
- USBTAP

Serial

Use this connection type to configure how the debugger uses the serial interface of the host PC to connect to the target system. This connection type is available if the EPPC - Abatron, EPPC - CodeWarriorTRK, or EPPC - Linux CodeWarriorTRK debugger protocol is selected.

Figure 4.2 shows the settings that are available to you when you select Serial from the Connection Type dropdown menu in the **Edit Connection** dialog box.

Figure 4.2 Serial Connection Type — Options

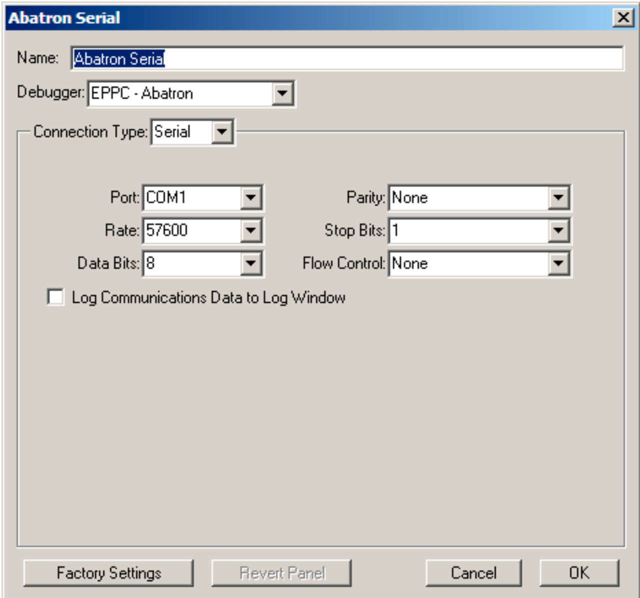


Table 4.4 describes the options in this dialog box.

Table 4.4 Serial Connection Type — Option Descriptions

Option	Description
Name	Enter the name to assign to this remote connection. This name appears in the Remote Connections preference panel.
Debugger	Select EPPC - Abatron or EPPC - CodeWarriorTRK.
Connection Type	Select Serial.
Port	Select the PC serial port to which the Abatron device or CodeWarrior TRK is connected.
Rate	Select the baud rate at which the debugger transmits and receives bits. This rate must match the rate being used by the Abatron probe or by CodeWarrior TRK.

Table 4.4 Serial Connection Type — Option Descriptions (*continued*)

Option	Description
Data Bits	Select the number of data bits in each character the debugger transmits and receives. This setting must match the setting being used by the Abatron probe or by CodeWarrior TRK.
Parity	Select the type of parity the debugger uses in each character it transmits and receives. This setting must match the setting being used by the Abatron probe or by CodeWarrior TRK.
Stop Bits	Select the number of stop bits in each character the debugger transmits and receives. This setting must match the setting being used by the Abatron probe or by CodeWarrior TRK.
Flow Control	Select the type of flow control for the debugger to use. This setting must match the setting being used by the Abatron probe or by CodeWarrior TRK.
Log Communications Data to Log Window	Check to have the debugger display communications data in a log window when the debugger uses this connection.

TCP/IP

Use this connection type to configure how the debugger uses the TCP/IP protocol to connect to the target system. This connection type is available if the EPPC - Abatron debugger protocol is selected.

Figure 4.3 shows the settings that are available when you select TCP/IP from the Connection Type dropdown menu in the **Edit Connection** dialog box.

Figure 4.3 TCP/IP Connection Type — Options

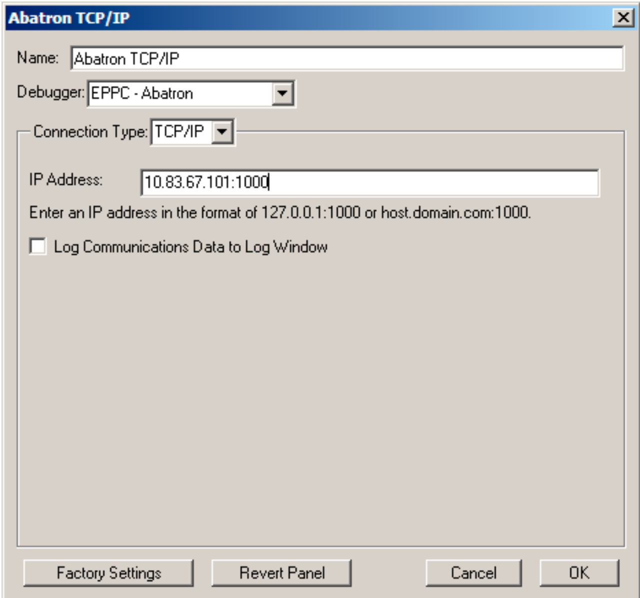


Table 4.5 describes the options in this dialog box.

Table 4.5 TCP/IP Connection Type — Option Descriptions

Option	Description
Name	Enter the name to assign to this remote connection. This name appears in the Remote Connections preference panel.
Debugger	Select EPPC - Abatron.
Connection Type	Select TCP/IP.
IP Address	Enter the Internet Protocol (IP) address assigned to the target system.
Log Communications Data to Log Window	Check to have the debugger display data in a log window.

CCS Remote Connection

Use this connection type to configure how the debugger uses the CodeWarrior Connection Server (CCS) to connect to the remote target system. This connection is usually used when you want to connect to a remote CCS running on another machine, or if you want to connect to the simulator server running either on localhost or on a remote machine. This connection type is available only if the CCS EPPC Protocol Plugin debugger protocol is selected.

Figure 4.4 shows the settings that are available when you select CCS Remote Connection from the Connection Type dropdown menu in the **Edit Connection** dialog box.

Figure 4.4 CCS Remote Connection Connection Type — Options

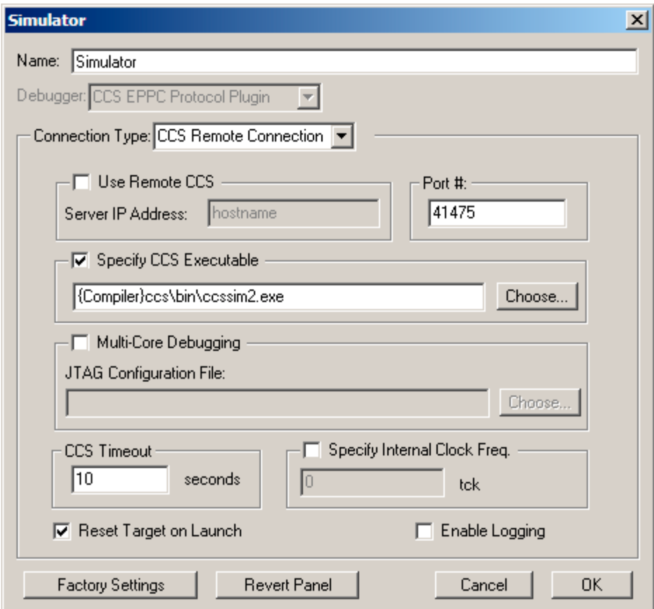


Table 4.6 describes the options in this dialog box.

Table 4.6 CCS Remote Connection Connection Type — Option Descriptions

Option	Description
Name	Enter the name to assign to this remote connection. This name appears in the Remote Connections preference panel.
Debugger	Select CCS EPPC Protocol Plugin.

Table 4.6 CCS Remote Connection Connection Type — Option Descriptions (*continued*)

Option	Description
Connection Type	Select CCS Remote Connection.
Use Remote CCS	Check to debug a board connected to a remote PC. In this scenario, the CodeWarrior debugger on your machine must connect to the CCS instance running on the remote PC.
Server IP Address	Enter the Internet Protocol (IP) address of the remote PC on which a remote CCS instance is running. This text box activates only if Use Remote CCS is checked.
Port #	Enter the port on the target system to which the debugger should connect for CCS operations. The default port number for CCS hardware connections is 41475. Enter 41476 for the CCS Simulator.
Specify CCS Executable	Check to use a CCS executable other than the default CCS executable file in `installDir\ccs\bin\ccs.exe`
Multi-Core Debugging	Check to debug a target system that has multiple cores for which you must specify the JTAG chain for debugging. Click **Choose** to specify the JTAG configuration file. A JTAG configuration file defines the names and order of the boards and/or cores you want to debug.
CCS Timeout	Enter the maximum number of seconds the debugger should wait for a response from CCS. By default, the debugger waits up to 10 seconds for responses.
Specify Internal Clock Freq.	Manually enter the JTAG clock frequency to be used during this connection.
Reset Target on Launch	Check to have the debugger send a reset signal to the target system when you start debugging. Clear to prevent the debugger from resetting the target device when you start debugging.
Enable Logging	Check to display a log of all debugger transactions during the debug session. If this checkbox is checked, a protocol logging window appears when you connect the debugger to the target system.

Ethernet TAP

Use this connection type to define how the debugger configures a CodeWarrior Ethernet TAP probe when the debugger connects to a target system. This connection type is available only if the CCS EPPC Protocol Plugin debugger protocol is selected.

Figure 4.5 shows the settings that are available to you when you select Ethernet TAP from the Connection Type dropdown menu in the **Edit Connection** dialog box.

Figure 4.5 Ethernet TAP Connection Type — Option Descriptions

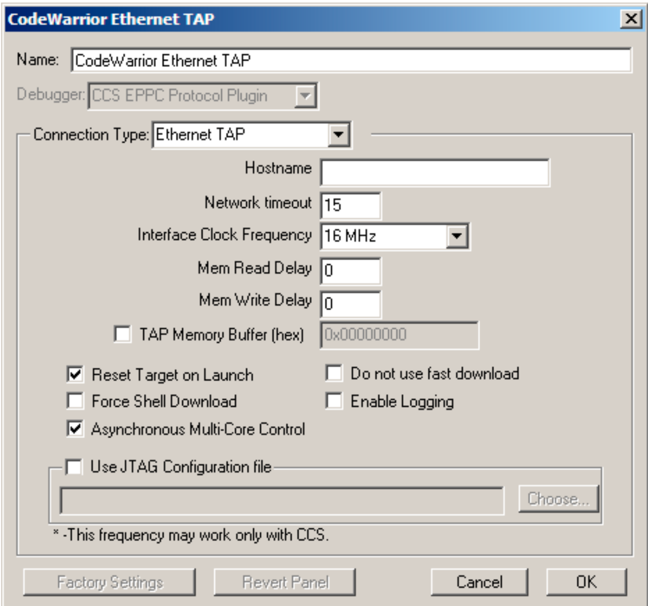


Table 4.7 describes the options in this dialog box.

Table 4.7 Ethernet TAP Connection Type — Option Descriptions

Option	Description
Name	Enter the name to assign to this remote connection. This name appears in the Remote Connections preference panel.
Debugger	Select CCS EPPC Protocol Plugin.
Connection Type	Select Ethernet TAP.

Table 4.7 Ethernet TAP Connection Type — Option Descriptions (*continued*)

Option	Description
Hostname	Enter the host name or IP address that you assigned to the Ethernet TAP device.
Network Timeout	Enter the maximum number of seconds the debugger should wait for a response from the Ethernet TAP device. By default, the debugger waits up to 15 seconds for responses.
Interface Clock Frequency	Select the clock frequency for the Ethernet TAP device. We recommend you to set this to 16 MHz.
Mem Read Delay	Enter the number of additional processor cycles (in the range 0 through 65024) the debugger should insert as a delay for completion of memory read operations. By default, the debugger delays for 350 cycles. Mem Read delay is supported only for 8260, 8280, 8272, and 5200 targets. Changing the value has no effect for other processors. Setting this value to 0 translates in 350 delay cycles. Entering a nominal value other than 0 causes that value to be used. We recommend you to keep the default value for this setting.
Mem Write Delay	Enter the number of additional processor cycles (in the range 0 through 65024) the debugger should insert as a delay for completion of memory write operations. By default, the debugger delays for 350 cycles. Mem Write delay is supported only for 8260, 8280, 8272, and 5200 targets. Changing the value has no effect for other processors. Setting this value to 0 translates in 350 delay cycles. Entering a nominal value other than 0 causes that value to be used. We recommend you to keep the default value for this setting.
TAP Memory Buffer	Enter an available target memory address for the debugger to use as an internal buffer. This is usually used to download the cache flushing routines necessary to maintain cache coherence during asynchronous multi-core debug. The minimum buffer size is 512 bytes.
Reset Target on Launch	Check to have the debugger send a reset and stop signal to the target system when you start debugging. Clear to prevent the debugger from resetting the target device when you start debugging.

Table 4.7 Ethernet TAP Connection Type — Option Descriptions (*continued*)

Option	Description
Force Shell Download	Check to have the debugger re-download the Ethernet TAP's shell when you start debugging. Clear to prevent the debugger from downloading the Ethernet TAP shell each time you start debugging. This option is used to force updating the shell version running on the Ethernet TAP.
Asynchronous Multi-Core Control	Check to have the debugger perform asynchronous debugging on a multi-core system. Uncheck to have the debugger perform synchronous debugging.
Do not use fast download	Check to have the debugger use a standard (slow) procedure to write to memory on the target system. Clear to have the debugger use an optimized (fast) download procedure to write to memory on the target system. The fast download mechanism is used by default when writing target memory. Check this if fast download procedure results in failures.
Enable Logging	Check to have the debugger display a log of all debug transactions during the debug session. If this checkbox is checked, a protocol logging window appears when you connect the debugger to the target system. **Note:** If you define the `CCS_LOG_PARAMETERS` environment variable, the debugger writes log messages to the specified file.
Use JTAG Configuration File	Check to assign a JTAG configuration file to this remote connection. In the related text box, type the path and name of the JTAG configuration file to use, or click **Choose** to display a dialog box with which you can select the required file. Clear if this remote connection does not need a JTAG configuration file. **Note:** For more information, refer to the CodeWarrior Project readme file.

USBTAP

Use this connection type to define how the debugger configures a CodeWarrior USB TAP probe when the debugger connects to a target system. This connection type is available only if the CCS EPPC Protocol Plugin debugger protocol is selected.

Figure 4.6 shows the settings that are available to you when you select USBTAP from the Connection Type dropdown menu in the **Edit Connection** dialog box.

Figure 4.6 USBTAP Connection Type — Options

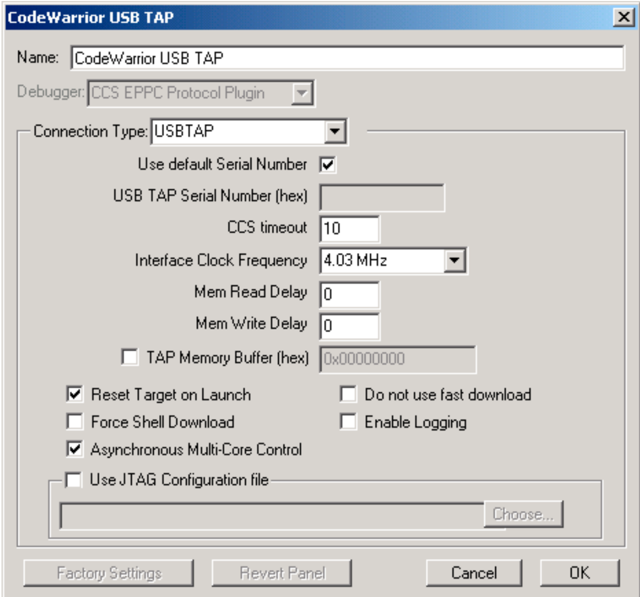


Table 4.8 describes the options in this dialog box.

Table 4.8 USBTAP Connection Type — Option Descriptions

Option	Description
Name	Use this text box to enter the name to assign to this remote connect. This name appears in the Remote Connections preference panel.
Debugger	Select CCS EPPC Protocol Plugin.
Connection Type	Select USBTAP.
Use default serial number	Check if you only have one USB TAP device connected to the host computer. Clear if you have more than one USB TAP device connected to the host computer. When this checkbox is clear, the USB TAP Serial Number text box is available.

Table 4.8 USBTAP Connection Type — Option Descriptions (*continued*)

Option	Description
USB TAP Serial Number	If you have more than one USB TAP connected to the host computer, enter the serial number of the USB TAP you want to use for debugging. **Note:** The USB TAP serial number is on a label on the bottom of the device.
CCS Timeout	Enter the maximum number of seconds the debugger should wait for a response from CCS. By default, the debugger waits up to 10 seconds for responses.
Interface Clock Frequency	Select the clock frequency for the USB TAP device. We recommended you set this to 4 MHz.
Mem Read Delay	Enter the number of additional processor cycles (in the range: 0 through 65024) the debugger should insert as a delay for completion of memory read operations. By default, the debugger delays for 350 cycles. Mem Read delay is supported only for 8260, 8280, 8272, and 5200 targets. Changing the value has no effect for other processors. Setting this value to 0 translates in 350 delay cycles. Entering a nominal value other than 0 causes that value to be used. We recommend you to keep the default value for this setting.
Mem Write Delay	Enter the number of additional processor cycles (in the range: 0 through 65024) the debugger should insert as a delay for completion of memory write operations. By default, the debugger does not delay. Mem Write delay is supported only for 8260, 8280, 8272, and 5200 targets. Changing the value has no effect for other processors. Setting this value to 0 translates in 350 delay cycles. Entering a nominal value other than 0 causes that value to be used. We recommend you to keep the default value for this setting.
TAP Memory Buffer	Enter an available target memory address for the debugger to use as an internal buffer. This is usually used to download the cache flushing routines necessary to maintain cache coherence during asynchronous multi-core debug.

Table 4.8 USBTAP Connection Type — Option Descriptions (*continued*)

Option	Description
Reset Target on Launch	Check to have the debugger send a reset signal to the target system when you start debugging. Clear to prevent the debugger from resetting the target device when you start debugging.
Force Shell Download	Check to have the debugger start the USB TAP shell when you start debugging. Clear to prevent the debugger from starting the USB TAP shell when you start debugging. This option is used to force updating the shell version running on the USB TAP.
Asynchronous Multi-Core Control	Check to have the debugger perform asynchronous debugging on a multi-core system. Uncheck to have the debugger perform synchronous debugging.
Do not use fast download	Check to have the debugger use a standard (slow) procedure to write to memory on the target system. Clear to have the debugger use an optimized (fast) download procedure to write to memory on the target system. The fast download mechanism is used by default when writing target memory. Check this if fast download procedure results in failures.
Enable Logging	Check to have the IDE display a log of all debugger transactions during the debug session. If this box is checked, a protocol logging window appears when you connect the debugger to the target system. **Note:** If you set the `CCS_LOG_PARAMETERS` environment variable, the debugger writes log messages to the specified file.
Use JTAG Configuration File	Check to assign a JTAG configuration file to this remote connection. In the related text box, type the path and name of the JTAG configuration file to use, or click **Choose** to display a dialog box with which you can select the required file. Clear if this remote connection does not need a JTAG configuration file. **Note:** For more information, refer to the CodeWarrior project readme file.

Creating a Remote Connection

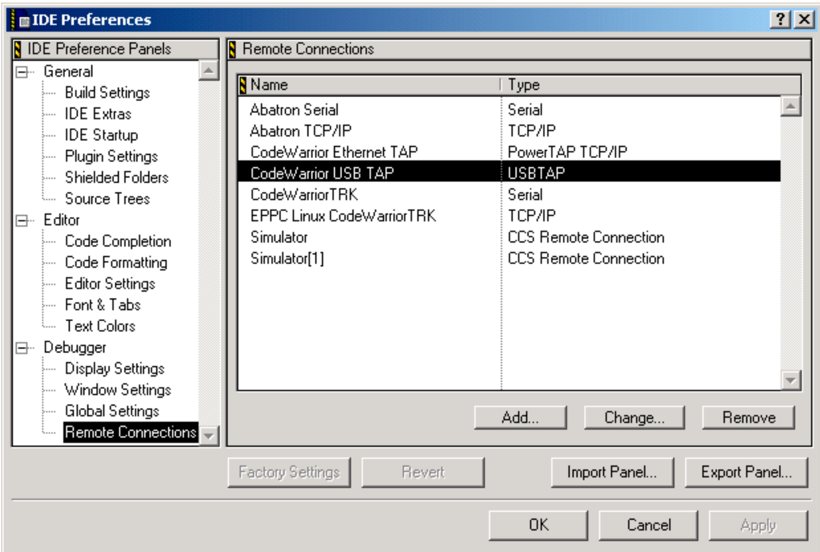
If the pre-defined remote connections do not meet your needs, you can create your own.

To create a remote connection, follow these steps:

1. From the IDE's menu bar, select **Edit > Preferences**.

   The **IDE Preferences** window appears. The Remote Connections list box displays each remote connection currently defined. (See Figure 4.7.)

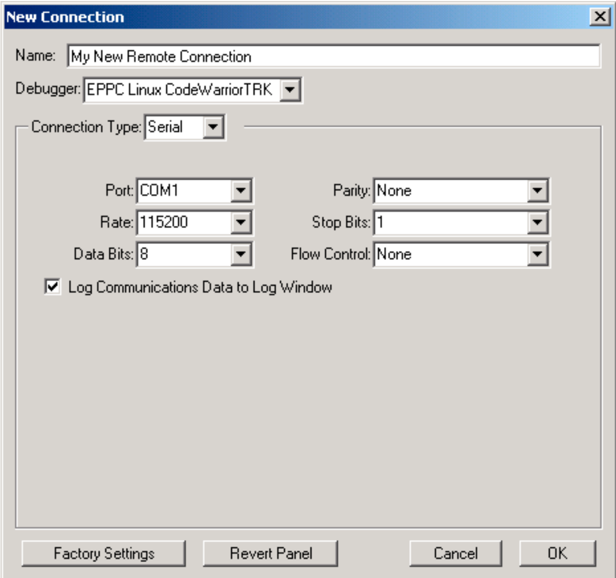
Figure 4.7 Remote Connections Preference Panel



2. Click **Add**.

   The **New Connection** dialog box appears. (See Figure 4.8.)

Figure 4.8 The New Connection Dialog Box



3. In the Name text box, type a mnemonic name for the new remote connection.
4. From the Debugger dropdown menu, select the debugger protocol appropriate for the probe for which the remote connection is being created.

   The Connection Type dropdown menu populates with choice appropriate for the selected Debugger protocol.
5. From the Connection Type dropdown menu, select the type of connection that the new remote connection's probe uses.

   Options appear in the bottom of the New Connection dialog appropriate for the selected connection type.
6. For each connection type option, enter the value appropriate for the new remote connection.

Setting the Watchpoint Type

A watchpoint is another name for a data breakpoint. The debugger halts execution each time the watchpoint location is read, written, or accessed (read *or* written).

Use the **Debug > Set Watchpoint Type** command to set a watchpoint. Setting the watchpoint type defines the conditions under which the debugger halts execution.

The options are:

- Read

  Program execution stops when your program reads from memory at the watchpoint address.

- Write

  Program execution stops when your program writes to memory at the watchpoint address.

- Read/Write

  Program execution stops when your program reads or writes memory at the watchpoint address.

- Prompt when set

  The debugger lets you to select one of the watchpoint types above at the time you set the watchpoint.

NOTE If a C/C++ statement assembles to a machine instruction that reads or writes the same address multiple times, a watchpoint set on this statement will be hit multiple times.

NOTE The Watchpoint Type command is available only if both the selected processor and your probe support it.

TIP You can also set watchpoint types by issuing the `watchpoint` command in the CodeWarrior **Command Window**.

Attaching to Processes

Use the **Debug > Attach to Process** command to attach the debugger to a process running on a target board. The debugger can control any process to which it is attached.

If the target board is running an operating system or multiple processes, you can use the CodeWarrior **System Browser** window to view and attach to these processes.

To use the **System Browser** window, follow these steps:

1. Open a CodeWarrior project.
2. Ensure that a linker is selected in the Target Settings panel in the **Target Settings** window.
3. Ensure that a remote connection is selected in the **Remote Debugging** target settings panel.

4. Build the CodeWarrior project to generate an executable file.
5. Select **View > System > *Connection*** from the IDE's menu bar (where ***Connection*** is the name of the selected remote connection).

   The **System Browser** window appears and displays a list of the processes running on the target board.
6. In the **System Browser** window, select the process to which you want to attach, then click the Attach To Process button ().

NOTE For more information about the **System Browser** window, refer to the *CodeWarrior™ IDE User's Guide*.

If the target board is *not* running an operating system, then there is just a single process. In this case, select **Debug > Attach To Process** to attach to and debug this process.

TIP You can also attach to processes by issuing the `attach` command in the CodeWarrior **Command Window**.

Ways to Initiate a Debug Session

The CodeWarrior debugger has three ways to initiate a debug session:

- Attach to Process

- Connect

- Debug

These commands differ in these ways:

- The **Attach to Process** command assumes that code is already running on the board and therefore does not run a hardware 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 **Attach to Process** function does not reset the target, even if the remote connection 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).

NOTE The debugger assumes that the current build target's generated executable matches the code currently running on the target.

- The **Connect** command runs the target initialization file specified in the EPPC Debugger Settings panel to set up the board before connecting to it. The **Connect** function does not load any symbolic debugging information for the current build target's executable. You therefore do not have access to source-level debugging and variable display. The **Connect** resets the target if the remote connection 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).

- The **Debug** command resets the target if the remote connection specifies the action. Further, the command stops the target, (optionally) runs an initialization script, downloads the specified ELF file, and modifies the PC.

Table 4.9 Effect of Each Different Connection Type

Connection Type	Resets Target on Launch	Stops Target	Runs Init Script	Uses Symbolics	Modifies Entry PC	Downloads Application
Attach	Never	No	No	Yes	No	Never
Connect	Per Remote Connection setting: Usually set to Yes	Only if Reset on Launch	Per debugger Global Setting panel	No	No	Per EPPC Debugger Settings
Debug	Per Remote Connection setting: Usually set to Yes	Only if Reset on Launch and Stop on Application Launch	Per debugger Global Setting panel	Yes	Yes	Per EPPC Debugger Settings

Table 4.10 Connection Type: Use cases

Connection Type	Typical Use Example
Attach	Debug a target system without modifying its state at all initially, but allow use of symbolics during actual debug. Useful for debugging a system that is already up and running.

Table 4.10 Connection Type: Use cases

Connection Type	Typical Use Example
Connect	Raw debug of a board without any software or symbolics. Useful during hardware bring up, and often combined with scripts for checking various aspects of the hardware.
Debug	Develop code that gets downloaded to the system on debugger launch. Useful for bare board code development without a working bootloader.

NOTE The default debugger configuration causes the debugger to cache symbolics between runs. However, the **Debug > Connect** command invalidates this cache. If you must preserve the contents of the symbolics cache, and you plan to use the **Debug > Connect** command, uncheck the **Cache symbolics between runs** checkbox in the **Debugger Settings** panel just before you issue the **Debug > Connect** command.

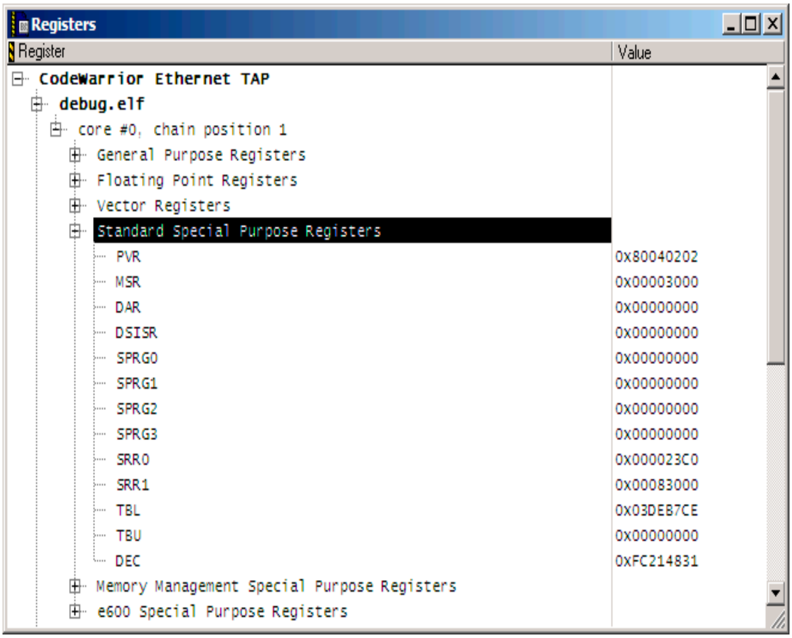
Displaying Register Contents

Use the **Registers** window to view and modify the contents of the registers of the processor on your target board. To display this window, select **View > Registers**.

The **Registers** window 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. Figure 4.9 shows the **Registers** window with the General Purpose Registers tree element expanded.

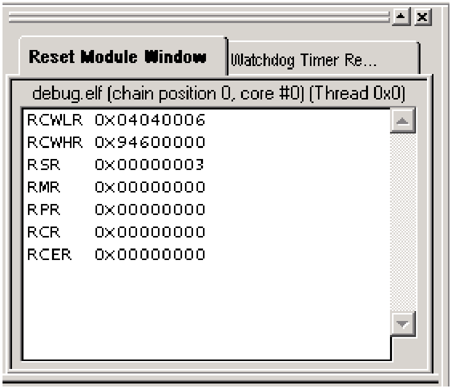
TIP You can also view and modify registers by issuing the `reg`, `change`, or `display` commands in the CodeWarrior **Command Window**.

Figure 4.9 Registers Window



TIP If you want to watch a particular group of registers at all times during a debug session, double-click the name of this register group in the **Registers** window. A new window opens showing the registers in this group. You can even dock this window, if desired. (See Figure 4.10.)

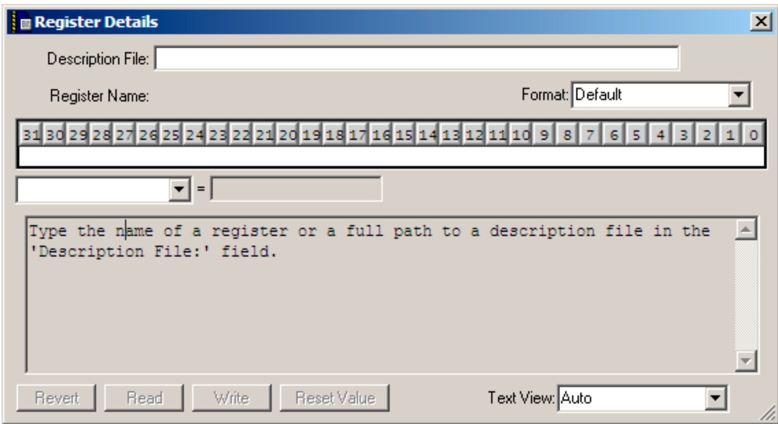
Figure 4.10 The Reset Module Register Group Shown in Separate, Docked Window



Using the Register Details Window

You can use the **Register Details** window to view different EPPC registers by specifying the name of the register. Selecting **View > Register Details** displays the **Register Details** dialog box (Figure 4.11).

Figure 4.11 Register Details Dialog Box



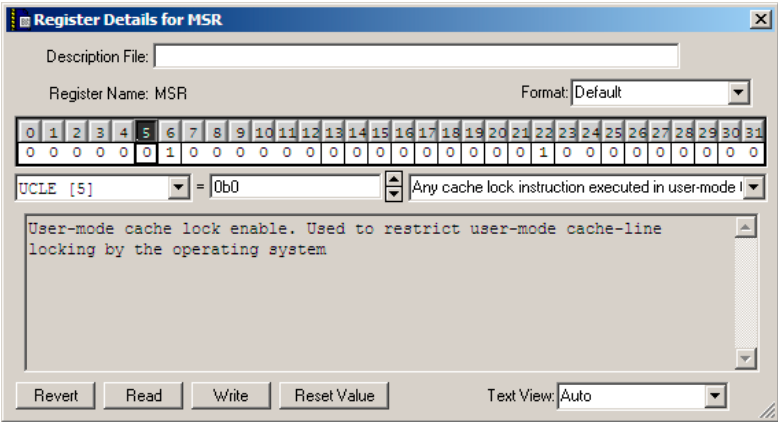
After the CodeWarrior software displays the **Register Details** window, type the name of the register (or the full path to the register description file) in the Description File text box to display the applicable register and its values.

TIP A convenient way to display the **Register Details** window for a particular register is to right-click on the register of interest in the **Registers** window and then select **Register Details** from the context menu that appears.

TIP You can also view register details by issuing the `reg` command in the CodeWarrior **Command Window**.

Figure 4.12 shows the **Register Details** dialog box displaying the MSR register.

Figure 4.12 Register Details Dialog Box Showing the MSR Register



You can change the format in which the CodeWarrior software displays registers using the Format dropdown menu. In addition, if you click on different bit fields of the displayed register, the CodeWarrior software displays a description of the bit or group of bits chosen. You also can change the text information that the CodeWarrior software displays by using the Text View dropdown menu.

NOTE For more information, see *CodeWarrior™ IDE User's Guide*.

Viewing and Modifying Cache Contents

The CodeWarrior debugger lets you view and modify the instruction cache and data cache of the target system during a debug session.

This topics of this section are:

- Displaying Processor Caches
- Cache Window Toolbar Buttons

- Components of the Cache Window
- Using the Command Window to View Caches
- Supported Processor Cache Features

Displaying Processor Caches

To display a processor's caches, start a CodeWarrior debug session on your target board, select **Data > View Cache** from the IDE menu bar, and select the instruction or data cache from the submenu that appears. The **Cache Window** appears.

Figure 4.13 shows the cache window displaying the contents of an instruction cache; Figure 4.14 shows the cache window displaying the contents of a data cache.

NOTE The **Data** menu is present in the IDE menu bar only during a debug session.

Figure 4.13 Cache Window Showing the Contents of an Instruction Cache

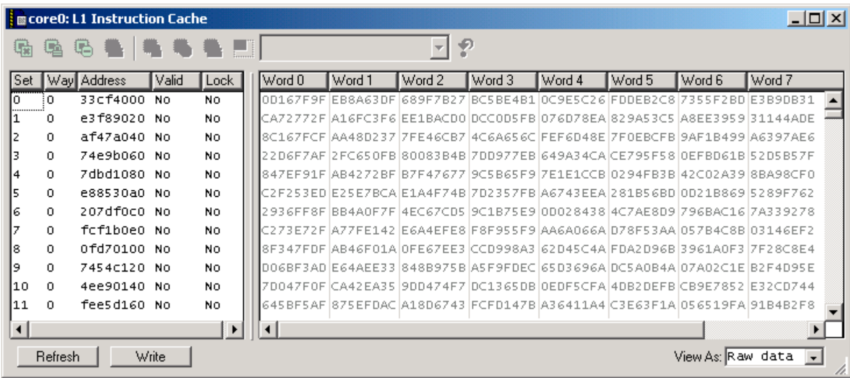
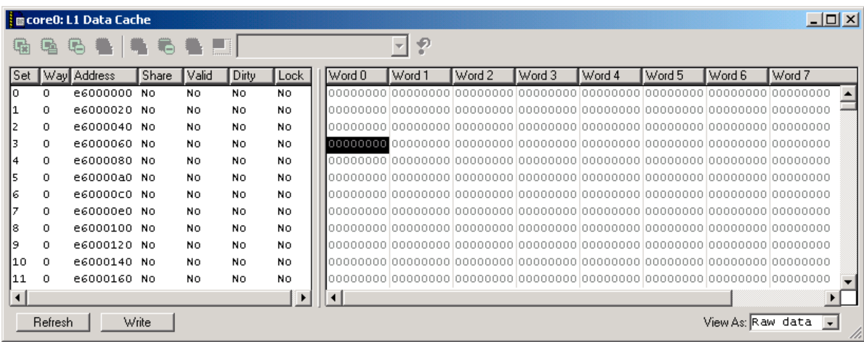


Figure 4.14 Cache Window Showing the Contents of a Data Cache



Cache Window Toolbar Buttons

At the top of the cache window is a toolbar with two groups of buttons separated by a vertical divider line. The buttons to the left of the toolbar divider line control the entire cache. The buttons to the right of the toolbar divider line control only the currently-selected cache lines.

NOTE Certain toolbar buttons are unavailable (grayed out) if the target hardware does not support their corresponding functions, or if the operation could be performed in assembly language and is not supported by the cache viewer.

Table 4.11 lists and describes each cache window toolbar button.

Table 4.11 Cache Window Toolbar Buttons

Button	Function	Description
	Enable/ Disable Cache	Enabled — The cache is on. Disabled — The cache is off.
	Lock/ Unlock Cache	Enabled — Every line of the cache is locked. This state prevents the cache from fetching new lines and from discarding current valid lines. Disabled — Every line of the cache is unlocked. This state allows the cache to fetch new lines and discard current valid lines.
	Invalidate Cache	Click to invalidate all lines in the cache. **Note:** To avoid data loss, flush the data cache before invalidating it.
	Flush Cache	Click to push out all lines marked as modified to the main memory so that the main memory and cache content are completely in sync.
	Lock/ Unlock Line	Enabled — Locks the selected cache lines. Disabled — Unlocks the selected cache lines. **Note:** Feature is not available for all processor variants.

Table 4.11 Cache Window Toolbar Buttons (*continued*)

Button	Function	Description
	Invalidate Line	Click to invalidate the selected cache lines. **Note:** Feature is not available for all processor variants.
	Flush Line	Click to push out the contents of the selected cache lines to the main memory so that the main memory and cache content are completely in sync. **Note:** Feature is not available for all processor variants.

NOTE The **Activate/Deactivate LRU Display** and **Inverse LRU** buttons never activate because their functions do not apply to Power Architecture processors.

Components of the Cache Window

Below the toolbar, there are two panes in the window, separated by another vertical divider line. The pane to the left of the divider line displays the attributes for each displayed cache line. The pane to the right of the divider line displays the actual contents of each displayed cache line. You can modify information in this pane and click the **Write** button to apply those changes to the cache on the target board.

Below the cache line display panes are the Refresh and Write buttons and the View As dropdown menu. Click the **Refresh** button to clear the entire contents of the cache, re-read status information from the target hardware, and update the cache lines display panes. Click the **Write** button to commit cache content changes from this window to the cache memory on the target hardware (if the target hardware supports doing so). Select `Raw Data`, or `Disassembly` from the View As dropdown menu to change the way the IDE displays the data in the cache line contents pane on the right side of the window.

You can perform all cache operations from assembly code in your programs. For details about assembly code, refer to the core documentation for the target processor.

You can also perform cache operations by selecting appropriate items from the **Cache** menu in the CodeWarrior menu bar. The **Cache** menu is available only when the Cache Window is open in the CodeWarrior IDE.

Using the Command Window to View Caches

Another way to manipulate the processor's caches is by using the CodeWarrior **Command Window**.

To display the **Command Window**, select **View > Command Window** from the CodeWarrior menu bar.

To display a list of the commands supported by the **Command Window**, enter this at the command prompt:

```
help -tree
```

Certain cache operations are invisible by default. To make them visible, enter these commands at the command prompt:

```
cmdwin::setvisible on cmdwin::ca
```

```
cmdwin::setvisible on cmdwin::caln
```

For more information about the **Command Window** support of cache commands, (`cmdwin::ca` for global operations and `cmdwin::caln` for cache line operations) enter these commands at the command prompt:

```
help cmdwin::ca
```

```
help cmdwin::caln
```

Supported Processor Cache Features

This section lists the cache features of the processors supported by this product.

Table 4.12 lists cache features supported by PowerQUICC I processors.

Table 4.13 lists cache features supported by PowerQUICC II processors.

Table 4.14 lists cache features supported by PowerQUICC III processors.

Table 4.15 lists cache features supported by e600 processors.

Table 4.12 PowerQUICC I Family — Supported Cache Operations

Cache	Features	Supported Operations
L1D *L1 data cache*	• 8 KB size • 256 sets • 2 ways • 4 words / line	• enable/disable cache • lock/unlock cache • invalidate line • read/modify data
L1I *L1 instruction cache*	• 16 KB size • 256 sets • 4 ways • 4 words / line	• enable/disable cache • lock/unlock cache • invalidate line • read/modify data

Table 4.13 PowerQUICC II Family — Supported Cache Operations

Cache	Features	Supported Operations
L1D *L1 data cache*	• 16 KB size • 128 sets • 4 ways • 8 words / line	• enable/disable cache • lock/unlock cache • invalidate cache • read/modify data
L1I *L1 instruction cache*	• 16 KB size • 128 sets • 4 ways • 8 words / line	• enable/disable cache • lock/unlock cache • invalidate cache • read/modify data

Table 4.14 PowerQUICC III Family — Supported Cache Operations

Cache	Features	Supported Operations
L1D *L1 data cache*	• 32 KB size • 128 sets • 8 ways • 8 words / line	• enable/disable cache • lock/unlock cache • invalidate cache • lock/unlock line • invalidate line • read/modify data
L1I *L1 instruction cache*	• 32 KB size • 128 sets • 8 ways • 8 words / line	• enable/disable cache • lock/unlock cache • invalidate cache • lock/unlock line • invalidate line • read/modify data
L2 *L2 cache (data only, instruction only, unified)*	• 256 KB/512 KB size • 1024/2048 sets • 8 ways • 8 words / line	• enable/disable cache • lock/unlock cache • invalidate cache • read/modify data

Table 4.15 e600 Family — Supported Cache Operations

Cache	Features	Supported Operations
L1D *L1 data cache*	• 32 KB size • 128 sets • 8 ways • 8 words / line	• enable/disable cache • lock/unlock cache • invalidate cache • lock/unlock way • invalidate line • read/modify data
L1I *L1 instruction cache*	• 32 KB size • 128 sets • 8 ways • 8 words / line	• enable/disable cache • lock/unlock cache • invalidate cache • lock/unlock way • invalidate line • read/modify data
L2 *L2 cache (data only, instruction only, unified)*	• 512 KB size • 2048 sets • 8 ways • 8 words / line	• enable/disable cache • invalidate cache • invalidate line • read/modify data

Using CodeWarrior TRK

This section briefly describes CodeWarrior TRK and then presents information that explains how to using CodeWarrior TRK with this product.

This topics are:

- CodeWarrior TRK Overview
- Connecting to CodeWarrior TRK
- CodeWarrior TRK Memory Configuration
- Using CodeWarrior TRK for Debugging

CodeWarrior TRK Overview

CodeWarrior TRK is a software debug monitor that a debugger uses to manipulate the target board and control the execution of software running on this board.

CodeWarrior TRK runs on the target board along with the program you are debugging and provides debug services to a debugger running on your development system. CodeWarrior TRK and the development workstation communicate over a serial communications link.

The CodeWarrior installer installs the source code for CodeWarrior TRK, as well as ROM images and project files for several pre-configured builds of CodeWarrior TRK.

The board-specific directories that contain CodeWarrior TRK source code are here:

installDir\PowerPC_EABI_Tools\
CodeWarriorTRK\Processor\ppc\Board\

If you are using an unsupported board, you may need to customize this source code to get CodeWarrior TRK to work with this board. For instructions, see the *CodeWarrior TRK Reference*.

NOTE You cannot adapt the CodeWarrior TRK source code to work for PowerQUICC III boards.

To modify a version of CodeWarrior TRK, find an existing CodeWarrior TRK project for your target board. You either can make a copy of the project (and its associated source files) or you can directly edit the originals. If you edit the originals, you always can get the original version from your CodeWarrior CD.

Connecting to CodeWarrior TRK

This section explains how to connect to the CodeWarrior TRK debug monitor over a serial connection.

To connect to CodeWarrior TRK, follow these steps:

1. Ensure that your target board has a debug monitor in ROM or in flash memory.

   If CodeWarrior TRK is not installed your the board, burn the program into ROM, or use a flash programmer to write the program to flash memory.

   For some boards, you can use one of the included CodeWarrior TRK projects to write CodeWarrior TRK to flash memory. These projects are here:

   installDir\PowerPC_EABI_Tools\
   CodeWarriorTRK\Processor\ppc\Board

2. Determine whether CodeWarrior TRK is in ROM or flash memory.

   a. Connect the serial cable to the target board.

NOTE Many target boards require a straight-through serial cable. Other boards require a null modem (DTE/DCE) cable. The user manual for your board should indicate which type of serial cable the board requires.

   b. Start a terminal emulator and configure it as shown in Table 4.16.

Table 4.16 Terminal Emulator Configuration Settings

bits per second	57600
data bits	8
parity	none
stop bits	1
hardware flow control	none
software flow control	none

c. Reset the target board.

If CodeWarrior TRK is present, the terminal emulation program displays CodeWarrior TRK initialization messages.

3. If you plan to use console I/O, ensure that your project contains the libraries required for console I/O.

Ensure that your project includes the MSL library and the UART driver library. If needed, add the libraries and rebuild the project. In addition, you must have a free serial port (besides the serial port that connects the target board with the host machine) and be running a terminal emulation program.

NOTE See the `project read me` file regarding CodeWarrior TRK options.

CodeWarrior TRK Memory Configuration

This section presents the default memory locations of the CodeWarrior TRK code and data sections and of your target application.

Locations of CodeWarrior TRK RAM Sections

Several CodeWarrior TRK RAM sections exist. You can reconfigure some of the CodeWarrior TRK RAM sections.

This section contains these topics:

- Exception Vectors
- Data and Code Sections
- The Stack

Exception Vectors

For a ROM-based CodeWarrior TRK, the CodeWarrior TRK initialization process copies the exception vectors from ROM to RAM.

The location of the exception vectors in RAM is a set characteristic of the processor. For Power Architecture processors, the exception vector must start at `0x000100` and span 7936 bytes to end at `0x002000`.

NOTE Do not change the location of the exception vectors because the processor expects the exception vectors to reside at the specific location.

Data and Code Sections

The standard configuration for CodeWarrior TRK uses approximately 29KB of code space as well as 8KB of data space.

In the default ROM-based implementation of CodeWarrior TRK used with most supported target boards, no CodeWarrior TRK code section exists in RAM because the code executes directly from ROM. However, for some Power Architecture target boards, some CodeWarrior TRK code does reside in RAM, usually for one of these reasons:

- Executing from ROM is slow enough to limit the CodeWarrior TRK data transmission rate (baud rate)
- For the 603e and 7xx processors, the main exception handler must reside in cacheable memory if the instruction cache is enabled. On some boards the ROM is not cacheable; consequently, the main exception handler must reside in RAM if the instruction cache is enabled

RAM does contain a CodeWarrior TRK data section. For example, on the Freescale 8xx FADS board, the default address where CodeWarrior TRK data section starts is `0x3F8000` and ends at the address `0x3FA000`.

You can change the location of the data and code sections in your CodeWarrior TRK project using one of these methods:

- By modifying settings in the EPPC Linker settings panel
- By modifying values in the linker command file (the file in your project that has the extension `.lcf`)

NOTE To use a linker command file, you must check the Use Linker Command File checkbox in the EPPC Linker settings panel.

The Stack

In the default implementation, the CodeWarrior TRK stack resides in high memory and grows downward. The default implementation of CodeWarrior TRK requires a maximum of 8KB of stack space.

For example, on the Freescale 8xx ADS and Freescale 8xx MBX boards, the CodeWarrior TRK stack resides between the addresses `0x3F6000` and `0x3F8000`.

You can change the location of the stack section by modifying settings of the EPPC Linker settings panel and rebuilding the CodeWarrior TRK project.

CodeWarrior TRK Memory Map

For more information about the CodeWarrior TRK memory map, see the board-specific information provided with the CodeWarrior TRK source code.

Using CodeWarrior TRK for Debugging

To use CodeWarrior TRK for debugging, you must load it on your target board in flash memory.

CodeWarrior TRK can communicate over serial port A or serial port B, depending on how the software was built. Ensure that you connect your serial cable to the correct port for the version of CodeWarrior TRK that you are using.

After you load CodeWarrior TRK on the target board, you can use the debugger to download and debug your application if the debugger is set to use CodeWarrior TRK.

NOTE Before using CodeWarrior TRK with hardware other than the supported target boards, see the *CodeWarrior TRK Reference*.

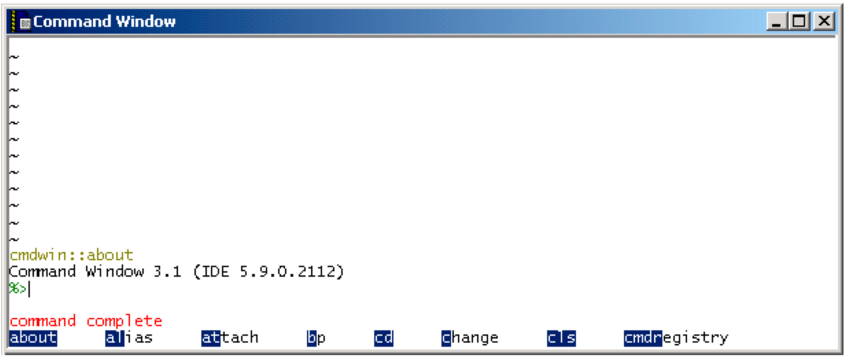
Using the Command-Line Debugger

The CodeWarrior IDE supports a command-line interface to some of its features, including the debugger. 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 a your command-line activity to a log file.

Use the **Command Window** (Figure 4.15) to issue commands to the IDE. For example, enter `debug` to start a debugging session. The IDE's **Command Window** shows the standard output and standard error streams generated by commands issued from the command line.

To display the **Command Window**, select **View > Command Window**.

Figure 4.15 Command Window



To issue a command-line command, make the **Command Window** the active window, type the desired command at the command prompt (`%>`), and press **Enter** or **Return**. The **Command Window** executes the specified command.

If you work with hardware as part of your project, you can use the **Command Window** to issue commands to the IDE while the hardware is running.

NOTE To display a list of the commands the **Command Window** supports, type `help` at the command prompt and press **Enter**. The `help` command lists each supported command along with a brief description of the command.

Refer to the *IDE Automation Guide* for information about using the command-line debugger. This manual presents the definition, shortcut, syntax, and examples of each command-line debugger option.

Debugging Bare Board Software

The topics in this section apply to debugging software on bare board systems, that is, to systems that are not running an operating system.

NOTE The Linux Application Edition of this product does not support debugging bare board software.

The topics are:

- Tutorial: Debugging a Bare Board Application
- Setting the Default Breakpoint Template
- Setting Hardware Breakpoints
- Accessing Translation Look-aside Buffers

- Setting the IMMR Register
- Setting the SCRB Register
- Sending a Hard Reset Signal
- Loading and Saving Memory
- Filling Memory
- Saving and Restoring Registers
- Virtual Address Translation Support
- Debugging ELF Files Created by Other Build Tools
- Debugging Multiple ELF Files Simultaneously
- Debugging a Multi-Core Processor
- Debugging Multiple Processors Connected in a JTAG Chain

Tutorial: Debugging a Bare Board Application

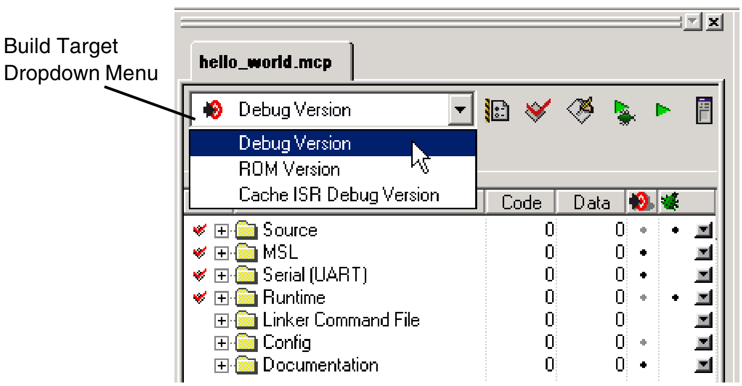
This chapter explains how to use your CodeWarrior tools to build and debug the project created in the Using the Bare Board New Project Wizard section.

To build this project, download and debug the resulting binary on your target board, follow these steps:

1. Start the CodeWarrior IDE.
2. Open the project created in the Using the Bare Board New Project Wizard section.

   The project window appears, docked to the left, top, and bottom of the IDE's main window. (See Figure 4.16.)

Figure 4.16 Project Window — hello_world.mcp



3. From the build target dropdown menu, select `Debug Version`.
4. Select **Project > Make**.

   The IDE compiles/assembles the Debug Version build target's source code files and links the resulting object code into an `.elf` format executable file.
5. Connect your target board to your PC.
   a. Ensure that target board's power switch is in the OFF position.
   b. Connect a power supply to the board.
   c. Connect your run-control hardware to the target board and to your PC.
   d. Connect a serial cable to the target board and to your PC.
   e. Move the board's power switch to the ON position.

      The target board powers up.
6. Start a terminal emulator program (such as HyperTerminal).
7. Configure the terminal emulator as shown in Table 4.17.

Table 4.17 Terminal Emulator Configuration Settings

bits per second	57600
data bits	8
parity	none
stop bits	1
hardware flow control	none
software flow control	none

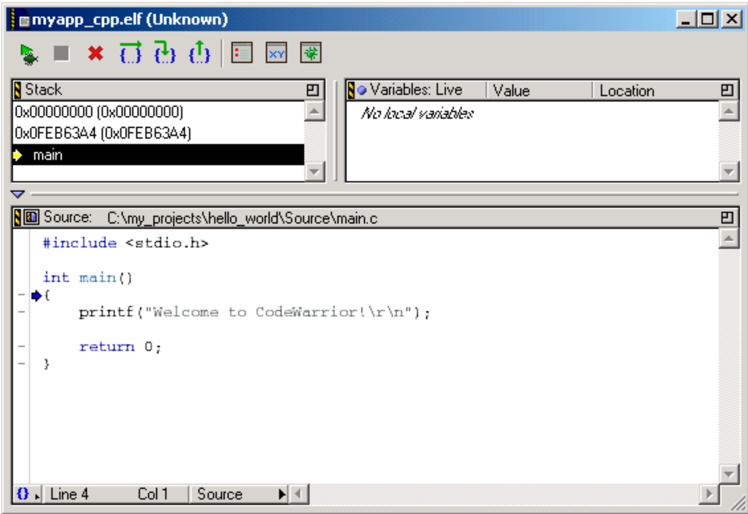
NOTE If you created your project with the New Project Wizard, the project contains a board-specific README file. This file provides information such as the serial connection settings to use, the type of serial cable required, and more.

8. Select **Project > Debug**.

   The debugger downloads the `.elf` format executable to the board, halts execution at the first statement in `main()`, and displays your source code in the debugger window. (See Figure 4.17.)

   The program counter icon points to the current statement (that is, to the next statement to be executed).

Figure 4.17 Debugger Window



9. In the debugger window, click the step over button.

   The processor executes the current statement and halts at the next statement.

10. In the leftmost column of the debugger window, click the dash next to this statement:

```
system_call(); // generate a system call exception to
               // demonstrate the ISR
```

   A breakpoint indicator ● appears next to this statement.

11. In the debugger window, click the run button.

   The processor executes all statements up to but not including the breakpoint statement and then halts at the breakpoint statement.

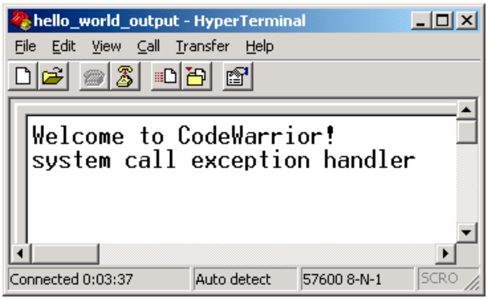
   The terminal emulator displays the string `Welcome to CodeWarrior!` because execution has passed the `printf()` statement.

12. In the debugger window, click the run button again.

   The program enters an infinite loop.

   The terminal emulator displays the string `system call exception handler` because execution has passed the `system_call()` statement. (See Figure 4.18.)

Figure 4.18 Terminal Emulator Window Containing Program Output Message



13. In the debugger window, click the break  button.

    The debugger halts the program at the next statement to be executed.

14. In the debugger window, click the kill  button.

    The debug session ends and the debugger window closes.

15. Exit the terminal emulator.

16. Press **Alt-F4**.

    The CodeWarrior IDE exits.

That's it. You have created a bare board project for your target board, built this project, downloaded the resulting application to the board, and used the CodeWarrior debugger to control this application's execution.

Setting the Default Breakpoint Template

Use the options in the **Debug > Default the Breakpoint Template** submenu to specify the type of breakpoint the debugger sets.

The options are:

- Software

  In this breakpoint mode, the debugger sets breakpoints in target memory. When program execution reaches the breakpoint, execution stops. The breakpoint remains in the target memory until the user removes it.

  A software breakpoint can be set only in writable memory like SRAM or DDR. You cannot use this type of breakpoints in ROM.

- Hardware

  In this breakpoint mode, the debugger uses the internal processor breakpoints. A processor does not provide many hardware breakpoints (sometimes as few as one), but they can be used with any kind of memory because they are implemented using processor registers.

- Auto

  In this breakpoint mode, the debugger first tries to set a software breakpoint. If this fails, the debugger then tries to set a hardware breakpoint.

NOTE The Default Breakpoint Template command is available only if both the selected processor and your probe support it.

TIP You can also set breakpoint types by issuing the `bp` command in the CodeWarrior **Command Window**.

Setting Hardware Breakpoints

To set a hardware breakpoint, follow these steps:

1. Connect to the target board.
2. Select **Debug > Default Breakpoint Template > Hardware**.

NOTE You can also set a hardware breakpoints by right clicking on a code line and then selecting Set Hardware Breakpoint command.

Table 4.18 lists the number of hardware breakpoints that can be set for each Power Architecture processor supported by this CodeWarrior product. Every processor listed in the table supports software breakpoints.

Table 4.18 Hardware BPs Allowed by the Supported Power Architecture™ Processors

Processor	Number of Hardware Breakpoints
8641, 8641D 8245, 8250, 8255, 8260, 8264, 8265, 8266, 8610	1

Table 4.18 Hardware BPs Allowed by the Supported Power Architecture™ Processors

5200 8540, 8541, 8543, 8545, 8547, 8548, 8555,8560, 8572 8313, 8315, 8321, 8323, 8343, 8347, 8349, 8358, 8360, 8379 8247, 8248, 8270, 8271, 8272, 8275, 8280 P1020, P1021, P1022, P2010, P2020, P1011, P1012	2
823, 850, 852, 857, 859 860, 862, 866, 870, 875, 880, 885	4

NOTE For some processors, the debugger must use one hardware breakpoint to implement software breakpoints. For these processors, you will have one less hardware breakpoint than listed in Table 4.18 unless you clear all software breakpoints.

Accessing Translation Look-aside Buffers

This section shows you how to use the CodeWarrior debugger to access PowerQUICC III Level 2 Memory Management Unit (MMU) translation look-aside buffers (TLBs).

PowerQUICC III Level 2 MMUs have two TLBs:

- TLB0 — a 256-element, two-way set associative array supporting a 4K page size, a 512-element for e500v2 (8544, 8548, 8572)
- TLB1 — a 16-element, fully associative array supporting nine or more page sizes

If you are debugging a supported PowerQUICC III board, the **Registers** window displays the TLB0 Registers and TLB1 Registers register groups.

Initializing TLBs

You can use `writereg128` commands in debugger initialization files to set up TLBs at target system startup. For details, read writereg128.

Accessing TLB Registers

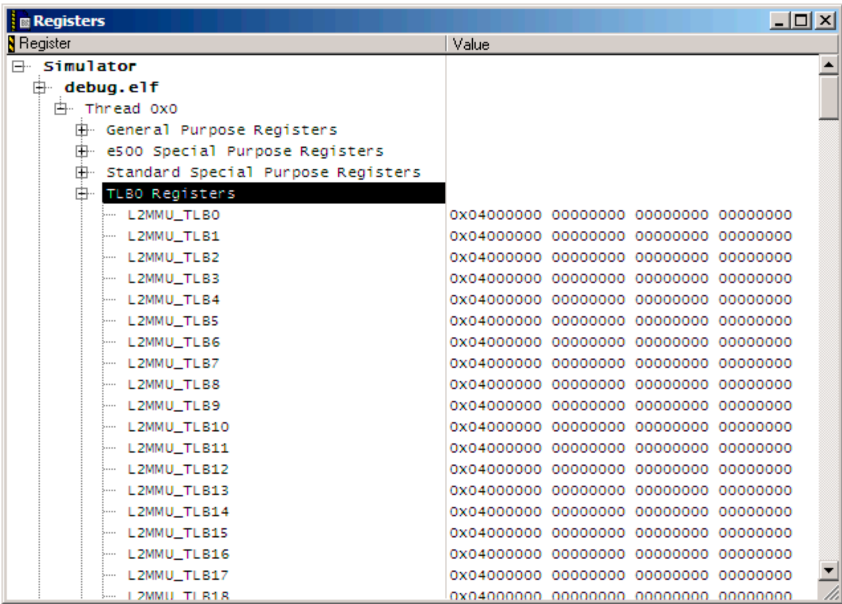
To view the **Registers** window:

1. Start the CodeWarrior IDE.
2. Open (or create) a project that targets the Power Architecture system you want to debug.
3. From the CodeWarrior menu bar, select **Project > Debug**.

   The IDE starts a debug session, connects to the target system, and halts the system at the program entry point.
4. Select **View > Registers**.

   The **Registers** window appears. (See Figure 4.19.)

Figure 4.19 Registers Window — TLB Register Groups Displayed

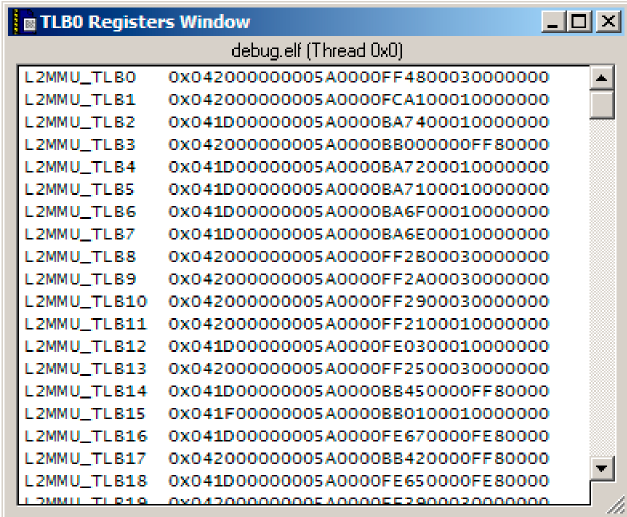


The **Registers** window shows all registers supported by the target system. The window groups all TLB0 registers in the `TLB0 Registers` group and groups all TLB1 registers in the `TLB1 Registers` group. (See Figure 4.19.)

To view all of the elements of a TLB register group, double-click the group you want to view. A window appears that displays all of the elements of the selected TLB.

For example, if you double-click the TLB0 Registers group, the TLB0 Registers window appears. (See Figure 4.20.)

Figure 4.20 TLB0 Registers Window



This window shows all of the TLB registers, and their contents. To modify TLB registers during a CodeWarrior debug session:

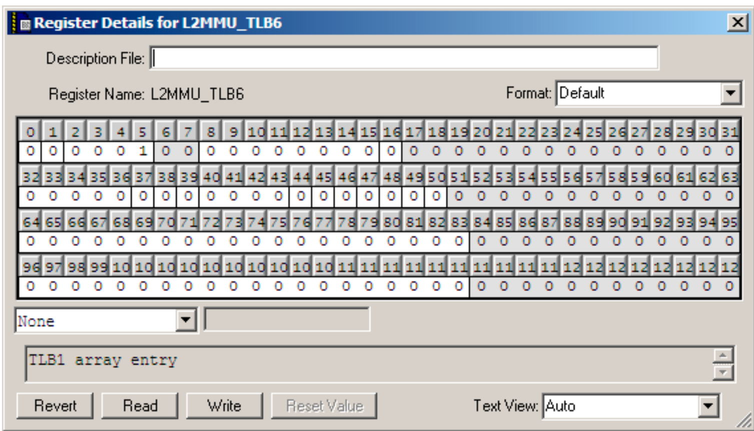
1. In the **Registers** window, select the TLB register you want to modify.

   The IDE highlights your selection.

2. From the CodeWarrior menu bar, select **View > Register Details**.

   The **Register Details** window appears. (See Figure 4.21.)

Figure 4.21 Register Details Window



This window lets you view register contents in different formats, and change portions of the selected register.

Setting the IMMR Register

Use the **Debug > EPPC > Change IMMR** command to define the memory location of the IMMR (Internal Memory Map) register. This information lets the CodeWarrior debugger find the IMMR register at debug-time.

NOTE The Change IMMR command is enabled only if you first select an 825x or 826x processor in the EPPC Debugger Settings target settings panels.

TIP You can also set the IMMR base address by issuing the `cmdwin::eppc::setMMRBaseAddr` command in the **Command Window**.

Setting the SCRB Register

Use **Debug > EPPC > Change SCRB** command to set the System Controller register base value. This information lets the CodeWarrior debugger find the System Controller registers during a debug session.

NOTE This command is disabled unless you select `107` or `109` from the System Controller dropdown menu of the EPPC Debugger Settings panel. The System Controller dropdown menu, in turn, is displayed only if you first select a processor having a 109 system controller (for example, the MPC744x) in the EPPC Debugger Settings target settings panel.

Sending a Hard Reset Signal

Use the **Debug > EPPC > Hard Reset** command to send a hard reset signal to the processor on the target board.

NOTE The Hard Reset command is enabled only if the run-control hardware you are using supports this command.

TIP You can also perform a hard reset by issuing the `reset hard` command from the CodeWarrior **Command Window**.

Loading and Saving Memory

Use the **Debug > EPPC > Load/Save Memory** command to copy data from a file into memory or to save data in memory to a file.

In more detail, the Load/Save Memory command:

- Loads the specified amount of data from a binary or text file on the host and writes this data to the target board's memory starting at the specified address.
- Reads the specified amount of data from the specified address of the target board's memory and saves this data in a binary or text file on the host.

TIP You can also load and save memory by issuing the `restore` and `save` commands from the CodeWarrior **Command Window**.

If you load an S-Record file, the loader behaves as follows:

- The loader uses the offset field to shift the address contained in each S-Record to a lower or higher address. The sign of the offset field determines the direction of the shift.
- The address produced by this shift is the memory address at which the loader starts writing the S-Record data.
- The loader uses the address and size fields as a filter. The loader applies these fields to the initial S-Record (not to its shifted version) to ensure that only the zone defined by these fields is actually written to.

Filling Memory

Use the **Debug > EPPC > Fill Memory** command to assign the specified value to a range of memory locations starting at the specified memory address.

Saving and Restoring Registers

Use the **Debug > EPPC > Save/Restore Registers** command to:

- Copy the contents of the specified registers to a text file

  or

- Load the specified registers from a text file.

The command lets you select the register group to save or restore.

Virtual Address Translation Support

The CodeWarrior debugger supports two types of address translation:

- Static address translation

  To achieve this form of address translation, you add a translate command to your build target's memory configuration file for each block to be translated.

  Each translation block definition includes a virtual and a physical address. The address mapping of a block does not change during a debug session.

  Static address translation works with most remote connections and is not affected by differences in the architecture of a chip's MMU.

  For instructions that explain how to enable static address translation support, refer to the Enabling Address Translations topic.

  Finally, static address translation does not apply to dynamic memory pages.

- Dynamic address translation

  To achieve this form of address translation, you add a range command to your build target's memory configuration file for each block to be translated. To each `range` command, you pass the argument `LogicalData` for the optional *`memorySpace`* parameter.

  At debug-time, the specified address translations are sent to the probe. The probe, in turn, translates virtual addresses to physical ones before requesting data through the JTAG port.

  Use dynamic address translation to access dynamic memory pages.

Refer to the Memory Configuration Files appendix for instructions that explain how to create and use a memory configuration file.

Enabling Address Translations

Use the **Debug > EPPC > Enable Address Translations** command to enable and disable the debugger's virtual-to-physical address translation feature. Typically, you enable this feature to debug programs that use a memory management unit (MMU) that performs block address translations.

If you enable address translations, the debugger uses the address translation commands in your memory configuration file to perform virtual-to-physical address translations. Refer to the translate topic for a definition of the syntax and effect of the address translation command.

To perform MMU debugging, follow these steps:

1. Add required address translation commands to your memory configuration file.

NOTE To create the required address translation commands, you must know how your application maps memory.

2. In the EPPC Debugger Settings target settings panel, check the Use Memory Configuration File checkbox, and specify the memory configuration file described above in the related text box.
3. Select **Project > Debug**.

   The debugger downloads your executable to the target device. The executable enables the MMU of the target device.
4. Select **Debug > EPPC > Enable Address Translations**.

   The debugger performs address translations using the address translation commands it finds in the your memory configuration file.

Automatically Enabling Address Translation

By default, address translations are disabled. However, if you must download an executable to a virtual address, you must enable address translation *before* the download.

To enable address translations before a download, add this statement to your memory configuration file:

```
AutoEnableTranslations true
```

NOTE Typically, when using virtual addressing, you link your executable with virtual addresses and initialize the MMU of your target device from a target initialization file or boot-loader.

Debugging ELF Files Created by Other Build Tools

You can use the CodeWarrior debugger to debug a “foreign” ELF file, that is, an ELF file created by build tools other than the CodeWarrior build tools.

Before you open a foreign ELF file for debugging, you must examine some IDE preferences and change them if needed. In addition, you must customize the default XML project file with appropriate target settings. The CodeWarrior IDE uses the XML file to create a project with the same target settings for any ELF file that you open to debug.

Preparing to Debug an ELF File

Before you debug an ELF file, you need to change certain IDE preferences and modify them if needed.

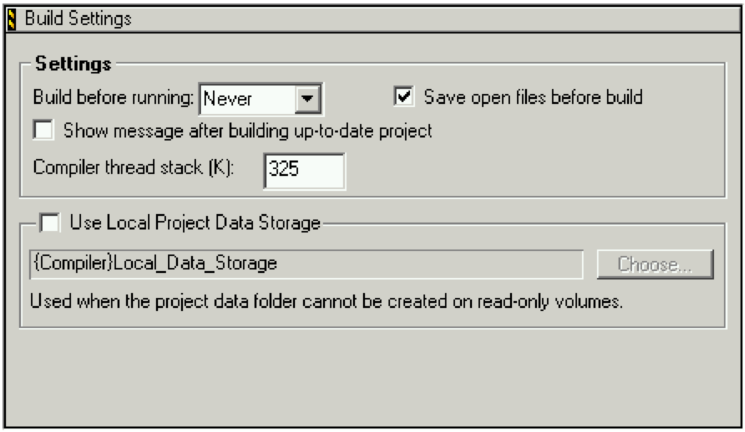
1. Select **Edit > Preferences**.

   The **IDE Preferences** window appears.

2. From the IDE Preference Panels list, select `Build Settings`.

   The **Build Settings** panel appears. (See Figure 4.22.)

Figure 4.22 Build Settings Preference Panel



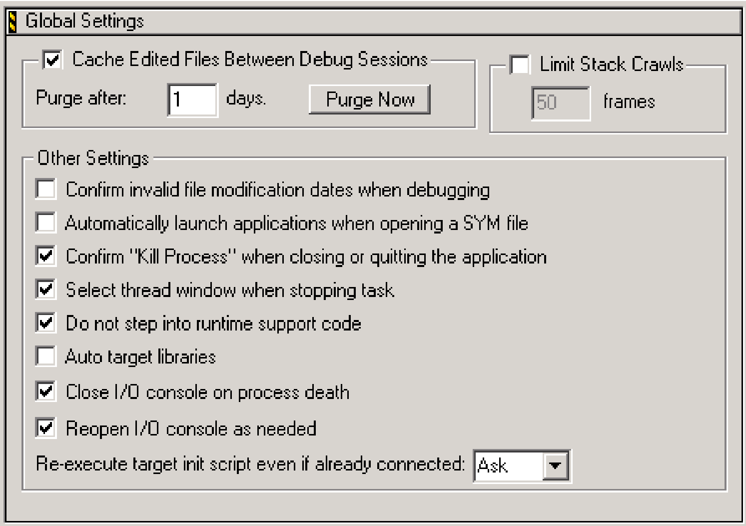
3. Make sure that the Build before running dropdown menu specifies `Never`.

NOTE Selecting `Never` prevents the IDE from building the newly created project, which is useful if you prefer to use a different compiler.

4. In the IDE Preference Panels list, click the `Global Settings` item.

   The **Global Settings** preference panel appears. (See Figure 4.23.)

Figure 4.23 Global Settings Preference Panel



5. Make sure that the Cache Edited Files Between Debug Sessions checkbox is clear.
6. Close the **IDE Preferences** window.

That's it. You have examined the relevant IDE preference settings.

Customizing the Default XML Project File

When you debug an ELF file, the CodeWarrior software uses the default XML project file to create a CodeWarrior project for the ELF file. The path to an name of this file is:

```
installDir\bin\Plugins\
Support\PowerPC_EABI\EPPC_Default_Project.XML
```

You must import the default XML project file, adjust the target settings of the new project, and export the changed project back to the original default XML project file. The CodeWarrior software then uses the changed XML file to create projects for any ELF files that you open to debug.

NOTE The IDE overwrites the `EPPC_Default_Project.XML` file each time you customize it. To preserve a customized version of this file, rename it or save it in another directory.

To customize the default XML project file:

1. Import the default XML project file.
   a. Select **File > Import Project**.
   b. Navigate to this location in the CodeWarrior installation directory: `bin\Plugins\Support\PowerPC_EABI\`
   c. Select the `EPPC_Default_Project.XML` file name.
   d. Click **Open**.
   e. Select the location where you want to save the new project.
   f. In the File name text box, enter the name of the new project file and click **Save**.

      The CodeWarrior software displays a new project based on `EPPC_Default_Project.XML`.
2. Change the target settings of the new project.

   Select **Edit > Target Settings** to display the **Target Settings** window. In this window, you can change the target settings of the new project as per the requirements of your target board and debugging devices.
3. Export the new project with its changed target settings.

   Export the new project back to the original default XML project file (`EPPC_Default_Project.XML`) by selecting **File > Export Project** and saving the new XML file over the old one.

   The new `EPPC_Default_Project.XML` file reflects any target settings changes that you made. Any projects that the CodeWarrior software creates when you open an ELF file to debug use these target settings.

Debugging an ELF File

This section explains how to prepare for debugging an ELF file for the first time.

To debug an ELF file:

1. From the CodeWarrior menu bar, select **File > Open**.
2. Navigate to and select the ELF file (with included debugger symbolic information).

   The CodeWarrior software creates a new project using the previously customized default XML project file. The CodeWarrior software bases the name of the new project on the name of the ELF file. For example, an ELF file named `cw.ELF` results in a project named `cw.ELF.mcp`.

   The symbolics in the ELF file specify the files in the project and their paths. Therefore, the ELF file must include the full path to the files.

   The DWARF information in the ELF file does not contain full path names for assembly (`.s`) files. Therefore, the CodeWarrior software cannot find them when creating the project. However, when you debug the project, the CodeWarrior software finds and uses the assembly files if the files reside in a directory that is an access path in the project. If not, you can add the directory to the project, after which the CodeWarrior software finds the directory whenever you open the project. You can add access paths for any other missing files to the project as well.

3. (Optional) Check whether the target settings in the new project are satisfactory.
4. Begin debugging.

   Select **Project > Debug**.

NOTE For more information on debugging, see *CodeWarrior™ IDE User's Guide*.

After debugging, the ELF file you imported is unlocked. If you choose to build your project in the CodeWarrior software (rather than using another compiler), you can select **Project > Make** to build the project, and the CodeWarrior software saves the new ELF file over the original one.

Additional Considerations

This section presents information that is useful when debugging ELF files.

Deleting Old Access Paths From ELF-Created Projects

After you create a project to allow debugging an ELF file, you can delete old access paths that no longer apply to the ELF file by using these methods:

- Manually remove the access paths from the project in the **Access Paths** target settings panel.
- Delete the existing project for the ELF file and recreate it by dragging the ELF file icon to the IDE.

Removing Files From ELF-Created Projects

After you create a project to allow debugging an ELF file, you may later delete one or more files from the ELF project. However, if you open the project again after rebuilding the ELF file, the CodeWarrior software does not automatically remove the deleted files from the corresponding project. For the project to include only the current files, you must manually delete the files that no longer apply to the ELF file from the project.

Recreating ELF-Created Projects

To recreate a project that you previously created from an ELF file:

1. Close the project if it is open.
2. Delete the project file. The project file has the file extension `.mcp` and resides in the same directory as the ELF file.
3. Drag the ELF file icon to the IDE. The CodeWarrior IDE opens a new project based on the ELF file.

Debugging Multiple ELF Files Simultaneously

This section explains how to use the CodeWarrior IDE to simultaneously debug multiple ELF files on a bare board. This is similar to debugging both an application and an associated shared library.

In order to debug multiple ELF files simultaneously with the CodeWarrior IDE, both ELF files must be available on the host computer, and must have DWARF 1.x, DWARF 2.x, or STABS symbolic information.

In this section, we show you how to debug multiple ELF files simultaneously under these scenarios:

- Debugging a Secondary ELF File Using the Load/Save Memory Option
- Debugging a Secondary PIC/PID ELF File
- Debugging a Secondary ELF File Created by Third-Party Tools

Debugging a Secondary ELF File Using a Serial Connection

In this scenario, you have two CodeWarrior projects that generate ELF files. The first project builds the main application, an application that loads and launches a secondary application. The second project builds the secondary application, which you send to the target system in S-Record format over a serial connection.

NOTE All source files for both projects must be available on the host system.

NOTE You can use two build targets in a single CodeWarrior project to generate both the main and the secondary applications, or you can use separate CodeWarrior projects to build each application. In this example, we use separate projects.

1. In the CodeWarrior IDE, open the main and secondary application projects.

   Ensure that the host computer and the target system are connected by a serial cable.

TIP You can easily create a CodeWarrior project from an existing ELF file by dragging the ELF file from Windows Explorer and dropping it on the CodeWarrior menu bar.

2. Configure both projects to connect to the target system using a CCS connection.

TIP Read Working with Remote Connections for instructions that explain how to configure remote connections.

3. In the main application project, add the secondary ELF file to the **Other Executables** target settings panel.

   The secondary ELF is the file produced by compile/linking the secondary project.

4. In the secondary application project, check the **Generate S-Record** checkbox in the EPPC Linker target settings panel.

5. Build the secondary application.

   The IDE generates the secondary ELF file in the `Bin` subfolder of the project folder.

6. Use a terminal application such as HyperTerminal to connect to the target system.

7. In the terminal, send the secondary ELF over the serial connection to the target system.

8. In the CodeWarrior IDE, open the `main.c` source code file in the main application project.

   Set a breakpoint in the `main.c` source code file where the loading of the S-Record (`.mot`) file is finished, before application launch.

9. Start a debug session of the main application project.

   The debugger stops at the main application entry point.

10. In the terminal window, paste the content of the secondary S-Record (`.mot`) file the CodeWarrior IDE generated when you built the secondary project.

11. From the CodeWarrior menu bar, select **View > Symbolics.**

    The **Symbolics** window appears.

12. In the Executables pane of the Symbolics window, select `secondary.elf`.

13. In the Files pane of the Symbolics window, select `main.c`.

    The **Symbolics** window displays the `main.c` source code file.

14. In the Source pane of the **Symbolics** window, click the breakpoint column (at the left side of the source code listing) to set a breakpoint in the `main.c` file.

15. From the CodeWarrior menu bar, select **Project > Run**.

    The processor executes the main application. The main application loads the secondary application and passes control to it. The debugger halts the secondary application when its execution reaches the breakpoint you set.

Debugging a Secondary ELF File Using the Load/Save Memory Option

In this scenario, you have two CodeWarrior projects that generate ELF files. The first project builds the main application, an application that loads and launches a secondary application. The second project builds the secondary application, which you send to the target system in S-Record format over a serial connection.

NOTE All source files for both projects must be available on the host system.

NOTE You can use two build targets in a single CodeWarrior project to generate both the main and the secondary applications, or you can use separate CodeWarrior projects to build each application. In this example, we use separate projects.

1. Connect a serial cable between the host computer and the target board.
2. In the CodeWarrior IDE, open the main and secondary application projects.

TIP You can easily create a CodeWarrior project from an existing ELF file by dragging the ELF file from Windows Explorer and dropping it on the CodeWarrior menu bar.

3. Configure both projects to connect to the target system using a CCS connection.

TIP Read Working with Remote Connections for detailed instructions that explain how to configure remote connections.

4. In the main application project, add the secondary ELF file to the **Other Executables** target settings panel.

   The secondary ELF is the file produced by compile/linking the secondary project.
5. In the secondary application project, check the **Generate S-Record** checkbox in the EPPC Linker target settings panel.
6. In the CodeWarrior IDE, open the `main.c` source code file in the main application project.
7. Set a breakpoint in the `main.c` source code file where the loading of the S-Record (`.mot`) file is finished, before application launch.

8. Start a debug session for the main application project.

   The debugger stops at the main application's entry point.

9. From the CodeWarrior menu bar, select **Debug > EPPC > Load/Save Memory.**

   The **Load/Save Memory** dialog box appears.

10. Use this dialog box to download the S-Record file (secondary.mot) to the target.
11. From the CodeWarrior menu bar, select **View > Symbolics.**
12. The **Symbolics** window appears.
13. In the Executables pane of the **Symbolics** window, select `secondary.elf`.
14. In the Files pane of the **Symbolics** window, select `main.c`.

    The **Symbolics** window displays the `main.c` source code file.

15. In the Source pane of the **Symbolics** window, click the breakpoint column (at the left side of the source code listing) to set a breakpoint in the `main.c` file.
16. From the CodeWarrior menu bar, select **Project > Run**.

    The processor executes the main application. The main application loads the secondary application and passes control to it. The debugger halts the secondary application when its execution reaches the breakpoint you set.

NOTE For more information about the **Load/Save Memory** function, refer to the Loading and Saving Memory topic.

Debugging a Secondary PIC/PID ELF File

In this scenario, you create a simple CodeWarrior project to build an application that uses PIC/PID addressing.

1. Connect a serial cable between the host computer and the target system.
2. Start the target system.
3. In the CodeWarrior IDE, open the primary and secondary application projects.

TIP You can easily create a CodeWarrior project from an existing ELF file by dragging the ELF file from Windows Explorer and dropping it on the CodeWarrior menu bar.

4. Configure both projects to connect to the target system using a remote connection over the serial cable.

TIP See Working with Remote Connections for detailed instructions that explain how to configure remote connections.

5. In the main application project, check the Generate S-Record checkbox in the EPPC Linker target settings panel.
6. In the secondary application project, check the Generate S-Record checkbox in the EPPC Linker target settings panel.
7. In the secondary application project, set the Code Address text box in the Segment Addresses area of the EPPC Linker target settings panel to an appropriate value.
8. Build the secondary application.

   The IDE generates the secondary ELF file in the `Bin` subfolder of the project folder.
9. In the main application project, add the secondary ELF file to the **Other Executables** target settings panel.
10. Use a terminal application such as HyperTerminal to connect to the target system.
11. Start a debug session of the main application project.
12. From the CodeWarrior menu bar, select **View > Symbolics** to view the **Symbolics** window.
13. In the Executables pane of the **Symbolics** window, select `secondary.elf`.
14. In the Files pane of the **Symbolics** window, select `main.c`.

    The **Symbolics** window displays the `main.c` source code file.
15. In the Source pane of the **Symbolics** window, click the breakpoint column (at the left side of the source code listing) to set a breakpoint somewhere in the main.c file.
16. Start a CodeWarrior debug session for the main application project.
17. From the CodeWarrior menu bar, select **Debug > EPPC > Load/Save Memory** to display the **Load/Save Memory** dialog box.
18. Use the **Load/Save Memory** dialog box to load the secondary ELF at the address you set in the Code Address text box in the EPPC Linker target settings panel.

TIP For instructions that explain how to use the **Load/Save Memory** dialog box, see Loading and Saving Memory.

19. From the CodeWarrior menu bar, select **View > Symbolics** to display the **Symbolics** window.
20. In the Executables pane of the **Symbolics** window, select `secondary.elf`.
21. In the Files pane of the **Symbolics** window, select `main.c`.

    The **Symbolics** window displays the `main.c` source code file.

22. In the Source pane of the **Symbolics** window, drag the program counter indicator (the blue arrow) to a line of source code in the `main.c` file.

    The thread window displays the source code of the `main.c` file. The program counter (blue arrow) appears at the line of source code you specified.

23. Change the view mode of thread window to Mixed.

    The Address line displays the address you set in the Code Address text box in the Segment Addresses area of the EPPC Linker target settings panel.

Debugging a Secondary ELF File Created by Third-Party Tools

In this scenario, you have a CodeWarrior project for the main application and a secondary application built by third-party tools.

NOTE The secondary ELF file to be debugged must contain debugging information in one of these formats: DWARF 1.x, DWARF 2.x, or Stabs.

NOTE All source files for the secondary ELF file must be present on the host system.

To debug a secondary ELF file that was built with third-party tools, follow these steps:

1. Use the third-party tools to build the secondary ELF file.
2. Open the secondary ELF file in the CodeWarrior IDE.

TIP To open the secondary ELF file, you can drag-and-drop it into the client area of the CodeWarrior IDE.

The IDE creates a CodeWarrior project for the secondary ELF file.

3. Configure the settings for the project such as the target processor for the secondary ELF file.
4. Open the CodeWarrior project for the main application.
5. In the main application project, add the secondary ELF file to the **Other Executables** target settings panel.

NOTE Downloading is a part of the launching process. Sections of the elf file to be downloaded are controlled by EPPC Debugger Settings panel settings. The debugger does not download the contents of a secondary ELF file defined in the Other Executables section of the CodeWarrior project. The debugger downloads and launches the specified project, only if there is a mcp file specified in the Other Executables section. In case of an ELF file the debugger loads only the symbolic information for that file.

6. Start a CodeWarrior debug session for the main application project.
7. From the CodeWarrior menu bar, select **Debug > EPPC > Load/Save Memory** to display the **Load/Save Memory** dialog box.
8. Use the **Load/Save Memory** dialog box to load the secondary ELF at the address you set in the Code Address text box of the EPPC Linker target settings panel.

TIP For instructions that explain how to use the **Load/Save Memory** dialog box, see the Loading and Saving Memory topic.

NOTE The **Load/Save Memory** dialog box offers the option of downloading only a SREC or bin file and not an elf file. Therefore, the elf file should be available in one of these formats. The included freescale compiler generates all three formats - elf, bin and mot.

9. Start a debug session of the main application project.
10. From the CodeWarrior menu bar, select **View > Symbolics**.

    The **Symbolics** window appears.
11. In the **Executables** pane of the **Symbolics** window, select `secondary.elf`.
12. In the **Files** pane of the **Symbolics** window, select `main.c`.

    The **Symbolics** window displays the `main.c` source code file.

    The thread window displays the source code of the `main.c` file. The program counter (blue arrow) appears at the line of source code you specified.

Debugging a Multi-Core Processor

This section explains how to create CodeWarrior projects for board that has a multi-core Power Architecture processor (such as the MPC8641D), download the binaries generated by this project to each core of the target processor, and debug the binary on each core.

The topics are:

- Creating Projects for a Multi-Core Processor
- Debugging Multi-Core System
- Other Multi-Core Debugger Features

Creating Projects for a Multi-Core Processor

To create a project for a board that has a multi-core processor, follow these steps:

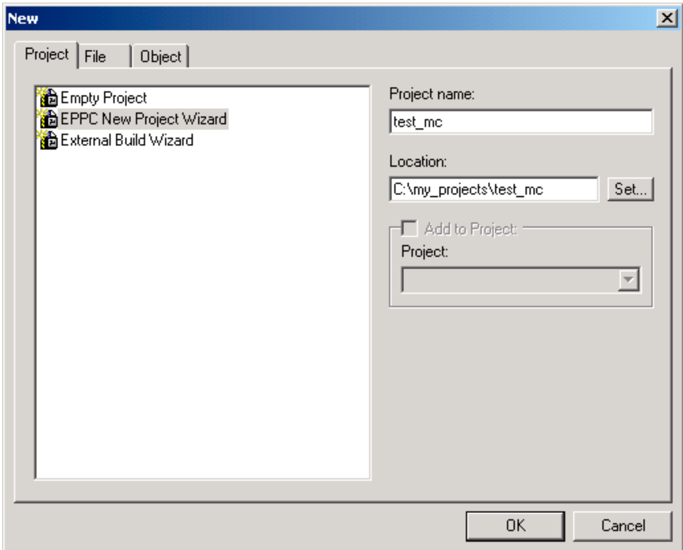
1. Start the CodeWarrior IDE.

   The CodeWarrior IDE starts and displays its main window.

2. From the IDE menu bar, select **File > New**.

   The **New** dialog box appears. (See Figure 4.24.)

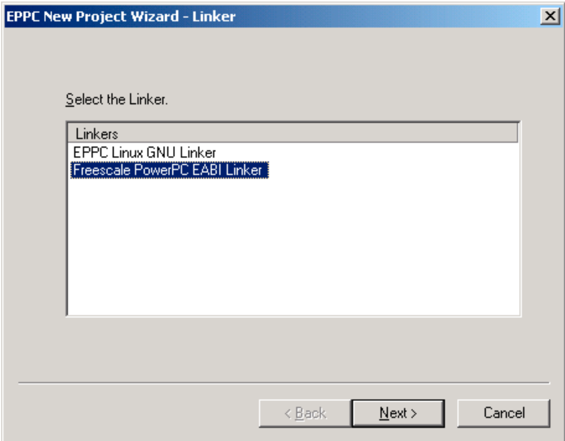
Figure 4.24 New Dialog Box



3. From the Project list box, select `EPPC New Project Wizard`.
4. In the Project Name text box, type `test_mc`.
5. In the Location text box, type the path in which to create this project, or click **Set** to use the **Create New Project** dialog box to find and select this path.
6. Click **OK**.

   The EPPC New Project Wizard starts and displays its **Linker** page. (See Figure 4.25.)

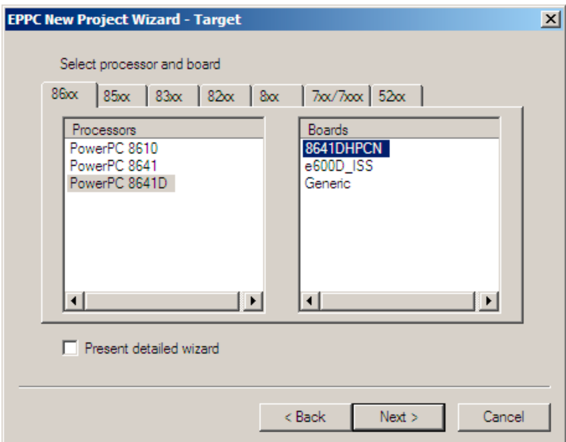
Figure 4.25 EPPC New Project Wizard — Linker Page



7. From the Linkers list box, select `Freescale PowerPC EABI Linker`.
8. Click **OK**.

   The wizard displays its **Target** page. (See Figure 4.26.)

Figure 4.26 EPPC New Project Wizard — Target Page



9. In the **Target** page, click the tab for a processor family that includes a multi-core processor.

   For example, click the **86xx** tab because this processor family includes the MPC8641D multi-core processor.

10. From **Target** page's Processors list box, select a multi-core processor.

    For example, in the Processors list box of the **86xx** tab, select the `PowerPC 8641D` processor because this is a dual-core processor.

11. From the **Target** page's Boards list box, select a board that has a multi-core processor on it.

    For example, in the Boards list box of the **86xx** tab, select the `8641DHPCN` board because this board contains the PowerPC 8641D dual-core processor.

12. Check the Present detailed wizard check box.

13. Click **Next**.

    The wizard displays the **Programming Language** page.

14. From the Languages list box, select the programming language you want to use.

    For example, if you plan to use the C language in your source code files, select `C`.

NOTE The language you select determines the libraries with which the new project's links and the contents of the main source file. If you select the C++ language, you can still add C source files to the project (and vice versa).

15. Check the Use size optimized MSL libraries box.

16. Click the **Next**.

    The wizard displays the **Floating Point** page.

17. From the Floating-point Support list, select the type of floating-point support your project requires.

18. Click **Next**.

    The wizard displays the **Remote Connection** page.

19. From the Available Connections list box, select the remote connection for the run-control hardware you plan to use.

NOTE To debug a multi-core processor, you must use a JTAG probe.

20. Click **Finish**.

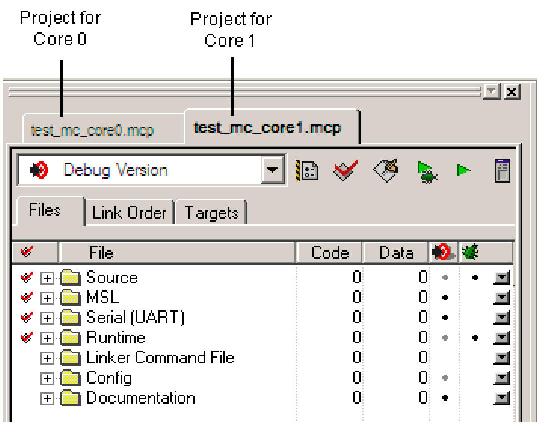
    The wizard creates a project *for each core* on the selected processor and displays each project window docked to the left, top, and bottom of the IDE main window.

    The wizard appends the string `_coreX` (where *X* is a number between 0 and the number of cores on the selected processor) to the each project name. This suffix lets you tell which project is for which core.

    The project with the suffix `_core0` is the *master* project. When you debug this project, the debugger downloads its binary to the processor's first core, and then downloads the binaries generated by each *slave* project to their respective cores.

    Figure 4.27 shows the two projects that the wizard creates for the dual-core PowerPC 8641D processor.

Figure 4.27 Project Windows — test_mc_core0.mcp and test_mc_core1.mcp



21. Click the tab labeled **test_mc_core0.mcp**.

    This project becomes the active project.

22. Press **Alt-F7**.

    The **Target Settings** window for the current build target of the test_mc_core0.mcp project appears.

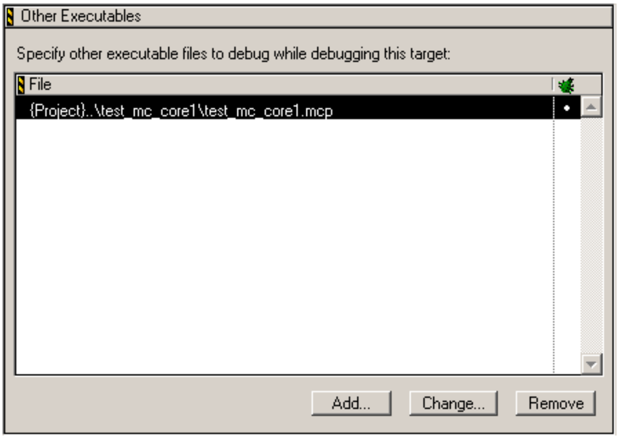
23. In the **Target Settings** window, select `Other Executables` from the Target Settings Panels list.

    The **Other Executables** target settings panel appears. (See Figure 4.28.)

    Notice that the wizard included the path to and name of the slave project (that is, the project that generates a binary for the second MPC8641D core) in this panel.

    This information instructs the debugger to download the binary generated by the slave project after it downloads the binary generated by `test_mc_core0.mcp`. Further, it is this information that makes `test_mc_core0.mcp` the master project.

Figure 4.28 The Other Executables Panel Showing a Slave Project



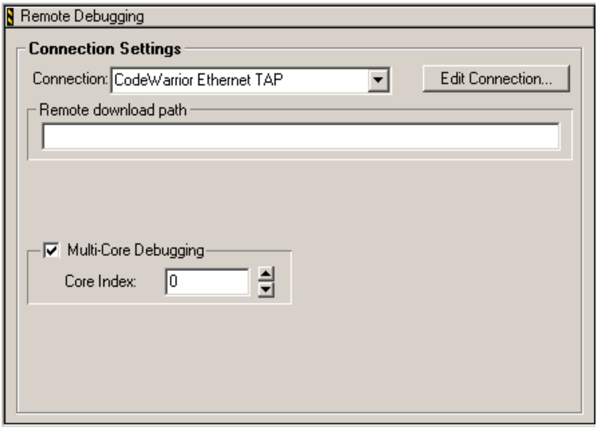
24. In the **Target Settings** window, select `Remote Debugging` from the Target Settings Panels list.

    The **Remote Debugging** target settings panel appears. (See Figure 4.29.)

    Notice that the Multi-Core Debugging box is checked and that the Core Index text box contains a 0. This information tells the debugger that:

    - The current build target is for a multi-core processor.
    - The binary generated by this build target should be downloaded to core 0 of the multi-core processor.

Figure 4.29 The Remote Debugging Panel Showing the Settings Needed to Debug Core 0



25. Click **Edit Connection** in the **Remote Debugging** target settings panel.

    The Selected Connection dialog box appears. (See Figure 4.30.)

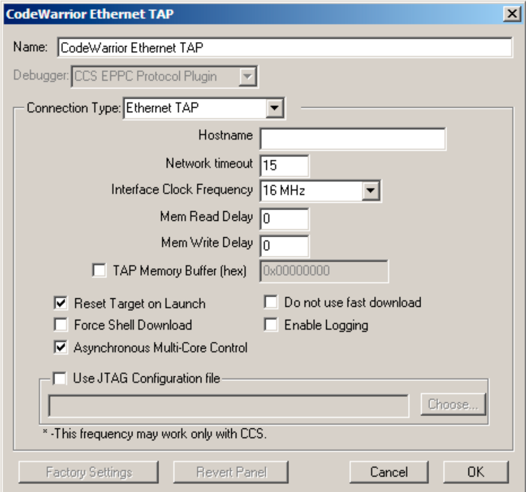
    Make Sure that the Asynchronous Multi-Core Control box is checked.

    In case of asynchronous run control, stop, run, and step commands affect only that particular core.

    In case of synchronous multi-core control, any run or halt command on one core applies for all the cores. All the cores stop when any of the cores hits a breakpoint. Step commands do not affect all cores.

NOTE Synchronous multi-core debugging is not supported for all the systems (currently only 8641D supports it).

Figure 4.30 The Remote Debugging Panel Showing the Asynchronous Multi-core Control Checkbox



26. Close the **Target Settings** window.
27. Select **Project > Make**.

    The IDE compiles/assembles the source code files of the current build target of the `test_mc_core0` project and links the resulting object code into an `.elf` format executable file.
28. Click the tab labeled `test_mc_core1.mcp`.

    This project becomes the active project.
29. Press **Alt-F7**.

    The **Target Settings** window for the current build target of the `test_mc_core1.mcp` project appears.
30. In the **Target Settings** window, select `Other Executables` from the Target Settings Panels list.

    The **Other Executables** target settings panel appears.

    Notice that the panel contains *no* project name. The absence of a slave project list is what makes this project a slave project.
31. In the **Target Settings** window, select `Remote Debugging` from the Target Settings Panels list.

    The **Remote Debugging** target settings panel appears.

    Notice that the Multi-Core Debugging box is checked and that the Core Index text box contains a 1. This information tells the debugger that:

- The current build target is for a multi-core processor.
- The binary generated by this build target should be downloaded to core 1 of the multi-core processor.

32. Close the **Target Settings** window.
33. Select **Project > Make**.

    The IDE compiles/assembles the source code files of the current build target of the `test_mc_core1` project and links the resulting object code into an `.elf` format executable file.

34. Click the tab labeled `test_mc_core0.mcp`.

    This project becomes the active project.

That's it. You have created a project that is configured for multi-core debugging.

Debugging Multi-Core System

The following tutorial uses the MPC8641D example project to show you how to use some of the multi-core debugger's features.

1. Start the CodeWarrior IDE.
2. Open the master project.
3. Select **Project > Debug**.

   The debugger:

   - Opens each slave project and displays its project window.
   - Adds the **Multi-Core Debug** menu item to the IDE's menu bar.
   - For core 0, downloads the binary generated by the master project to core 0, halts execution at the first statement in `main()`, and displays the source code in a debugger window.
   - For core 1, downloads the binary generated by the first slave project to core 1, halts execution at the first statement in `main()`, and displays the source code in a second debugger window.

   ... and so on until the binary for each slave project has been downloaded to the core with which the binary is associated.

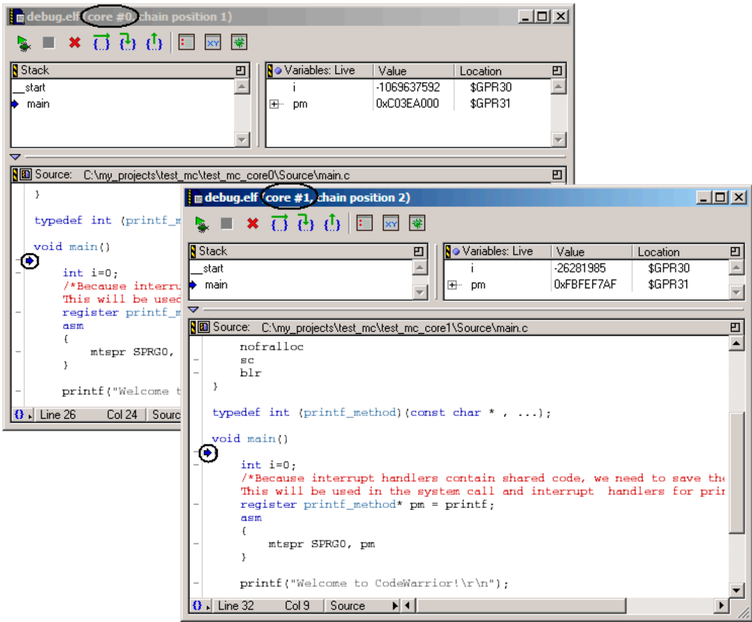
   For example, for the dual-core MPC8641D processor, two debugger windows are displayed. (See Figure 4.31.)

   In each window, the program counter icon points to the current statement (that is, to the next statement to be executed) for each core.

   The string in the title bar of each window tells you which debugger window is for which core.

NOTE In case of Synchronous multi-core control, when core 1 stops at entry point the core 0 has already executed part of the code.

Figure 4.31 Debugger Windows for the Dual-Core MPC8641D Processor — View 1



You use each debugger window the same way as with a single-core processor. In most cases, an action taken in one window affects just the core with which this window is associated — the action does not affect the other core.

NOTE In case of Synchronous multi-core debugging, an action taken in one window affects all the core with which this window is associated. In case of two cores, both the cores run and stop at the same time. When a breakpoint is hit on a core that core will stop resulting in a similar behavior on all other cores. Run/Stop operations have the same behavior as Run All/Stop All, but on the target only one run/stop command is issued by the core on which the command is requested.

That said, if the program counter is in memory shared by a pair of cores, the debugger window for each core shows the same source code and changes to the code in one window are reflected in the other. Further, a breakpoint set in either window is honored by both cores.

NOTE For the following procedure the run control is asynchronous.

4. In core 1's debugger window, click the step over button.

   Core 1 executes the current statement and halts at the next statement.

   Notice that the program counter icon in core 0's debugger window does not move. This is because the debugger controls each core's execution individually (that is, the execution of the cores is not synchronized).

5. In the core 0's debugger window, click the step over button until reaching the `printf()` statement.

   Core 0 executes the current statement, the following statements, and halts at the `printf()` statement.

   Notice that the program counter icon in core 1's debugger window does not change position. Again, this is because the debugger controls each core's execution individually. (See Figure 4.32.)

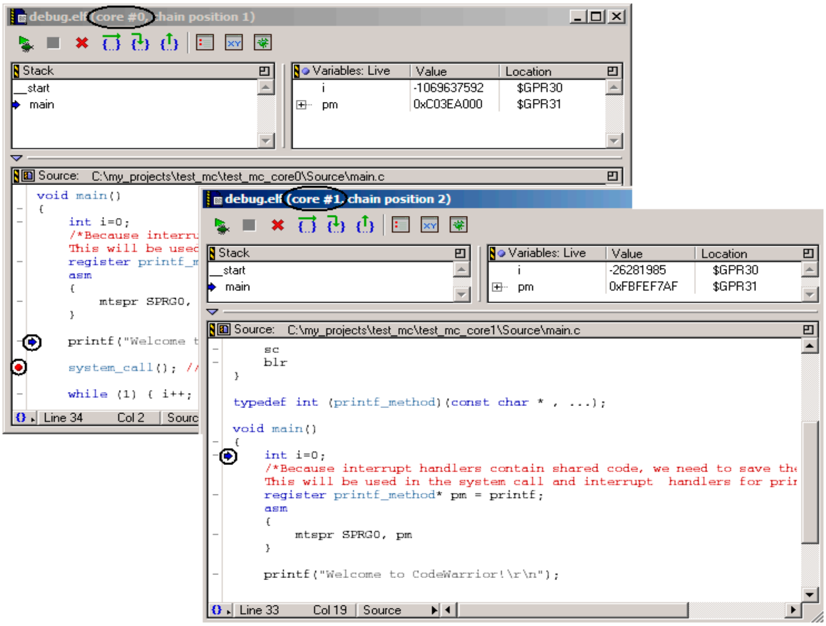
6. In the leftmost column of the core 0's debugger window, click the dash next to this statement:

```
system_call(); // generate a system call exception to
                // demonstrate the ISR
```

   A breakpoint indicator appears next to this statement.

   Notice that a breakpoint indicator does *not* appear in core 1's debugger window. This is because each core has a private copy of the `main()` function. (See Figure 4.32.)

Figure 4.32 Debugger Windows for the Dual-Core MPC8641D Processor — View 2



7. In core 1’s debugger window, click the run  button.

   The program enters an infinite loop.

   The debugger window displays this status message:

   ```
   Program "debug.elf" is executing.
   Choose Break from the Debug menu to stop it.
   ```

   Notice that core 1 does not hit the breakpoint set in core 0’s debugger window.

8. In the core 0’s debugger window, click the run  button.

   Core 0 executes all statements up to but not including the breakpoint statement and then halts at the breakpoint statement.

   Core 1 continues to execute.

9. In the core 0’s debugger window, click the run  button.

   Core 0 enters an infinite loop.

   Core 1 continues to execute.

10. In core 1's debugger window, click the break  button.

    The debugger halts the core 1 at the next statement to be executed.

    Core 0 continues to execute.

11. From the IDE's menu bar, select **Multi-Core Debug > Kill All**.

    Both debugger windows close and the debug session ends.

12. Press **Alt-F4**.

    The CodeWarrior IDE exits.

That's it. You have created a project for a board that has a multi-core processor, built this project, downloaded the resulting binaries to each core on the board, and used the multi-core features of the CodeWarrior debugger to control each binary's execution.

Other Multi-Core Debugger Features

This sections explains how to use the memory window, registers window, symbolics window, and the Multi-Core Debug menu when debugging a multi-core processor.

The topics are:

- Using the Memory Window
- Using the Registers Window
- Using the Symbolics Window
- Using the Cache Window
- Using the Multi-Core Debug Menu

Using the Memory Window

You can open an instance of the memory window for each core in your multi-core system.

Like each debugger window lets you control particular core's execution, so each memory window lets you display and modify a particular core's memory contents. However, if memory shared by a pair of cores changes, the memory windows for each core reflect the change.

The following tutorial uses the MPC8641D project to shows you how to display a memory window for two cores.

1. Start the CodeWarrior IDE.
2. Open the master project.
3. Select **Project > Debug**.

   The debugger downloads a binary to core 0, a binary to core 1, and displays two debugger windows.

4. Activate core 0's debugger window by clicking its title bar.

   Core 0's debugger window becomes topmost.

5. Select **Data > View Memory**.

   A memory window displaying core 0's memory appears.

6. Activate core 1's debugger window by clicking its title bar.

   Core 1's debugger window becomes topmost.

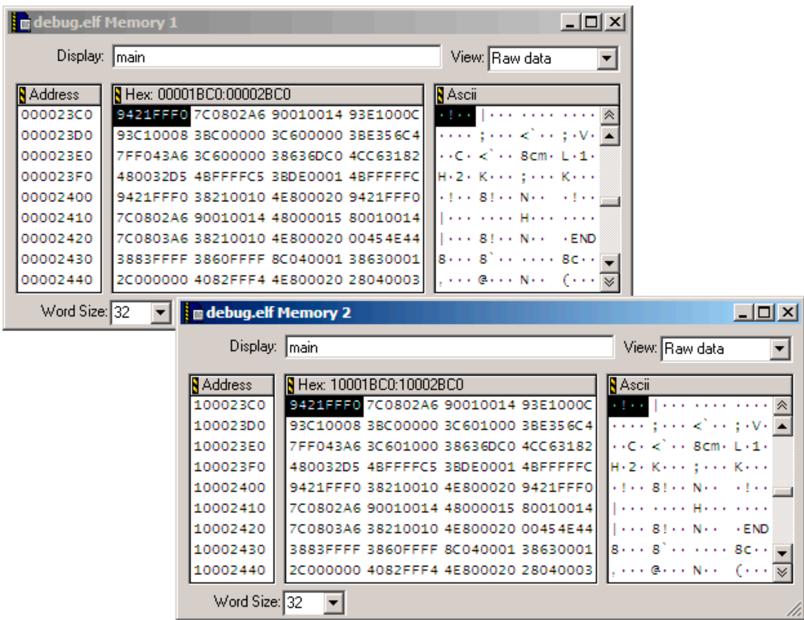
7. Select **Data > View Memory**.

   A memory window displaying core 1's memory appears.

Figure 4.33 shows the memory windows for cores 0 and 1.

Refer to the *CodeWarrior™ IDE User's Guide* for instructions that explain how to use the memory window.

Figure 4.33 Memory Windows for the Dual-Core MPC8641D Processor



Using the Registers Window

You can open just one instance of the **Registers** window for all cores in your multi-core system. This window displays each core's private register set, along with the registers that each core shares.

The following tutorial uses the MPC8641D project to shows you how to display a memory window for two cores.

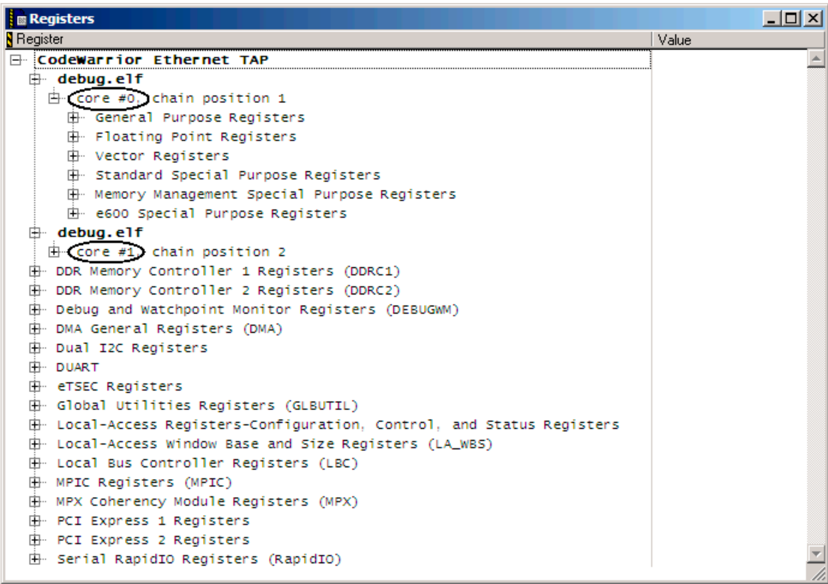
1. Start the CodeWarrior IDE.
2. Open the master project.
3. Select **Project > Debug**.

   The debugger downloads a binary to core 1, a binary to core 2, and displays two debugger windows.
4. Select **View > Registers**.

   The **Registers** window appears. (See Figure 4.34.)

Refer to the *CodeWarrior™ IDE User's Guide* for instructions that explain how to use the **Registers** window.

Figure 4.34 Registers Window Showing Registers for the MPC8641D Dual-Core Chip



Using the Symbolics Window

You can open just one instance of the **Symbolics** window for all cores in your multi-core system.

For the selected core, the **Symbolics** window lists the source code files used to build the binary running on the core, the public symbols defined in each file, and the contents of the currently selected file.

The following tutorial uses the MPC8641D project to shows you how to display the multi-core **Symbolics** window.

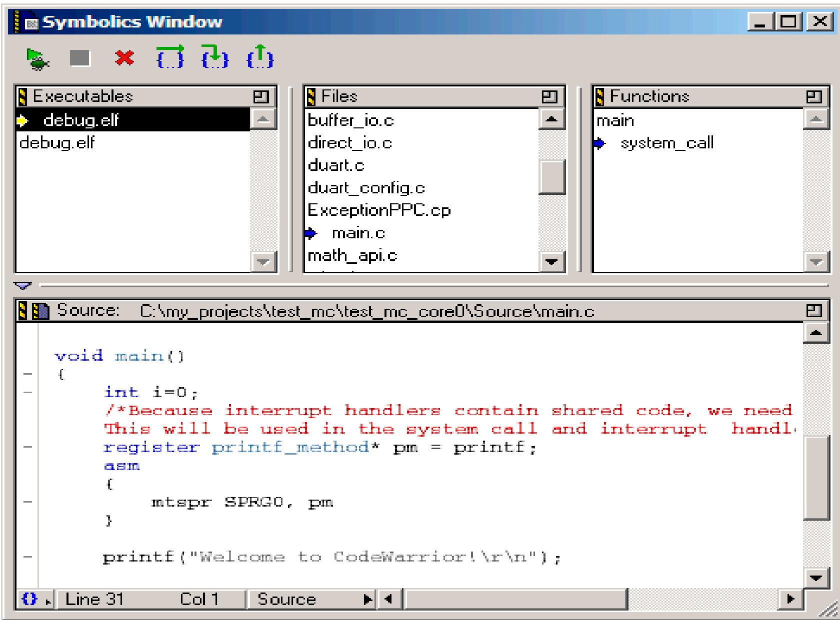
1. Start the CodeWarrior IDE.
2. Open the master project.
3. Select **Project > Debug**.

   The debugger downloads a binary to core 1, a binary to core 2, and displays two debugger windows.
4. Select **View > Symbolics**.

   The **Symbolics** window appears. (See Figure 4.35.)

Refer to the *CodeWarrior™ IDE User's Guide* for instructions that explain how to use the **Symbolics** window.

Figure 4.35 Symbolics Window for the MPC8641D Dual-Core Chip



Using the Cache Window

The cache window can display the contents of any cache of any core of a multi-core system. You can then manipulate this data in the same way as for a single-core processor. (See Viewing and Modifying Cache Contents for instructions that explain how to use the cache window.)

To display a cache of a core of a multi-core system, follow these steps:

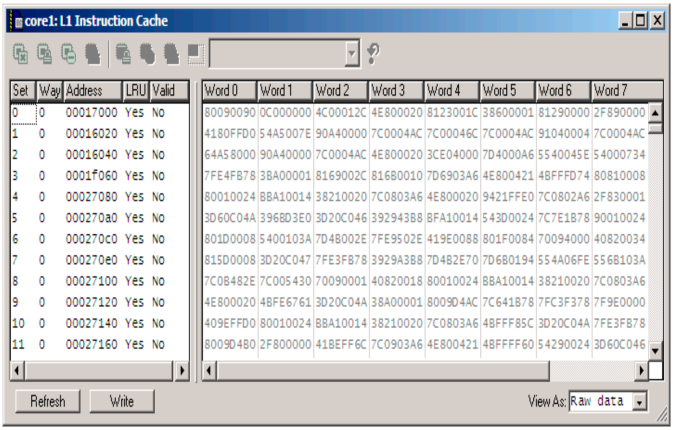
1. Activate the debugger window of the core for which you want to display cache contents.
2. Select **Data > View Cache**.

   A submenu listing the caches of the current core appears.
3. From this submenu, select the cache with which you want to work.

   The cache window appears, showing the contents of the selected cache.

   Figure 4.36 shows the cache window displaying the contents of the L1 instruction cache of core 1 of the MPC8641D dual core processor.

Figure 4.36 Cache View Window for the MPC8641D Dual-Core Chip



Using the Multi-Core Debug Menu

When you start a multi-core debug session, the debugger adds the **Multi-Core Debug** menu to the IDE's menu bar. This menu contains commands that affect all cores simultaneously. Table 4.19 lists and describes each menu option.

Table 4.19 Multi-Core Debug Menu Commands

Command	Descriptions
Run All	Starts all cores of a multi-core system running simultaneously.
Stop All	Stops execution of all cores of a multi-core system simultaneously

Table 4.19 Multi-Core Debug Menu Commands

Command	Descriptions
Kill All	Kills the debug session for all cores of a multi-core system simultaneously.
Restart All	Restarts all the debug sessions for all cores of a multi-core system simultaneously.

Debugging Multiple Processors Connected in a JTAG Chain

This section explains how to configure the CodeWarrior Power Architecture debugger to support targets with more than one device on the JTAG scan chain.

The CodeWarrior debugger debugs the Freescale Power Architecture processors, single-core or multi-core, connected together on one JTAG scan chain. Each single-core or multi-core processor has its own CodeWarrior project and is debugged individually in the CodeWarrior IDE. Debugging multi-core processors is handled automatically by the CodeWarrior debugger. Debugging multiple processors, or debugging processors that have other devices on the JTAG scan chain, requires some configuration provided by the user.

The topics are:

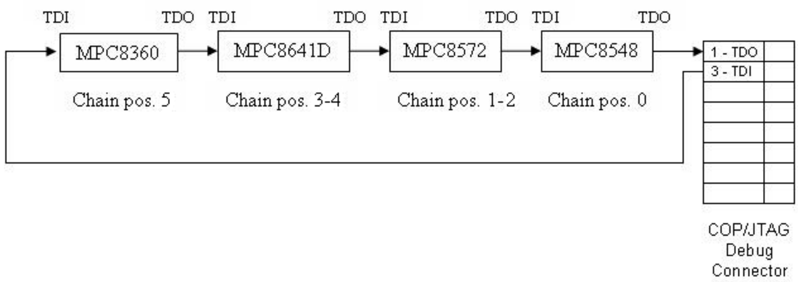
- Creating a JTAG Configuration File
- Assigning the JTAG configuration file to the CodeWarrior Project

Creating a JTAG Configuration File

First step involves creating a JTAG configuration file.

The JTAG configuration file is an ASCII text file that defines all the devices on the scan chain and the order in which they occur. Starting with the device directly connected to the TDO (transmit data out) signal (Pin 1) of the 16-pin COP/JTAG debug connector on the hardware target, list each device on a separate line and conclude with a blank line.

Figure 4.37 A JTAG Scan Chain



In the JTAG scan chain shown in Figure 4.37, the hardware target has four Freescale Power Architecture processors on the JTAG scan chain: MPC8548, MPC8572, MPC8641D, and MPC8360.

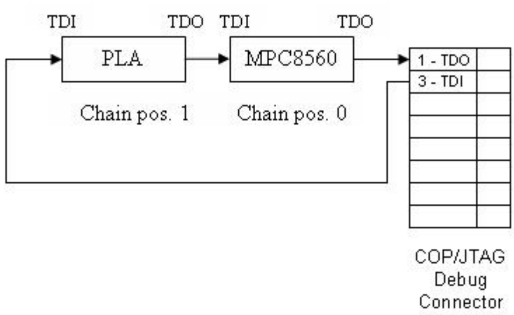
The JTAG Configuration File for this scan chain appears as shown:

```
PQ38
MPC8572
MPC8641D
MPC8360 (1 1) (2 0x84030006) (3 0x8C600000)
```

This list of devices follows the order of the scan chain starting with the device directly connected to TDO of the COP/JTAG debug connector. In this example, the entry for the MPC8360 also includes the Hard Reset Control Word (`HRCW`) data that will overwrite the `HRCW` fetched by the MPC8360 upon power up or Hard Reset. The Hard Reset Control Word parameters are optional.

The CodeWarrior debugger also supports targets with non-Freescale devices on the scan chain. Each non-Freescale device is declared as "generic" and needs three parameters: JTAG Instruction Length; Bypass Command; and Bypass Length. The values for these three parameters are available in the device's data sheet or from its manufacturer.

Figure 4.38 A JTAG Scan Chain



In the JTAG scan chain shown in Figure 4.38, the hardware target has one Freescale MPC8560 and one non-Freescale PLA on the JTAG scan chain. From the PLA's data sheet, the JTAG Instruction Length = 5, the Bypass Command = 1, and the Bypass Length = `0x1F`. The JTAG Configuration File for this scan chain appears as shown:

```
MPC8560
Generic 5 1 0x1F
```

Assigning the JTAG configuration file to the CodeWarrior Project

1. In the CodeWarrior IDE, create a CodeWarrior project for each Freescale processor. For detailed instructions, see Creating Projects for a Multi-Core Processor topic.

   The wizard creates a project *for each core* on the hardware target and displays each project window docked to the left, top, and bottom of the IDE main window.

2. Click the tab of each project to activate it.
3. Press **Alt-F7**.

   The **Target Settings** window for the current build target of the project appears.

4. In the **Target Settings** window, select **Remote Debugging** from the Target Settings Panels list.

   The **Remote Debugging** target settings panel appears. (See Figure 4.39.)

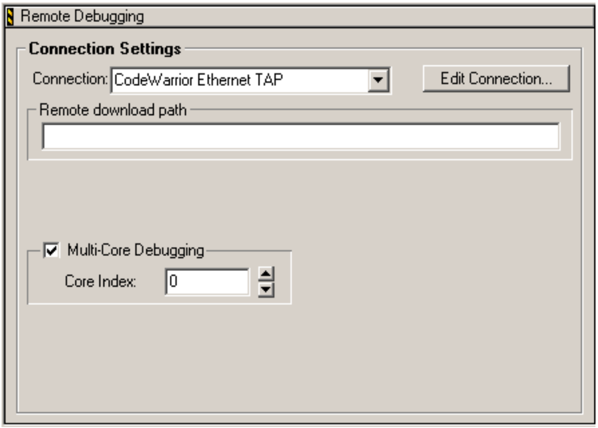
   The Core Index refers to the position of the processor or core on the scan chain, starting with zero. The device directly connected to the TDO signal (Pin 1 of the target's 16-pin COP/JTAG debug connector) is Core Index `0`, the next device is Core Index `1`, and so on.

   In JTAG scan chain shown in Figure 4.37, the hardware target has four Freescale Power Architecture processors on the JTAG scan chain:

- MPC8548 – Core Index 0
- MPC8572 – Core Indices 1-2
- MPC8641D – Core Indices 3-4
- MPC8360 – Core Index 5

For this target the MPC8548 project uses Core Index `0`, core `1` of the MPC8572 uses Core Index `1`, core `2` of the MPC8572 uses Core Index `3`, core `1` of the MPC8641D uses Core Index `3`, core `2` of the MPC8641D uses Core Index `4`, and the MPC8360 project uses Core Index `5`.

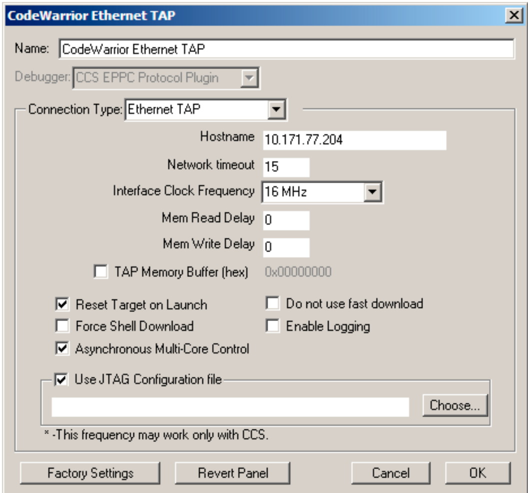
Figure 4.39 The Remote Debugging Panel



5. Click **Edit Connection** in the **Remote Debugging** target settings panel.

   The Selected Connection dialog box appears. (See Figure 4.40.)

Figure 4.40 The Remote Debugging Panel Showing the Use JTAG Configuration file Checkbox



6. Check the Use JTAG Configuration File check box.
7. Click **Choose**.

   Navigate to the location of the JTAG Configuration File created above.
8. Select the file and click **OK**.
9. Close the **Target Settings** window.
10. Similarly, for each CodeWarrior project, enable the Multi-Core Debugging option of the Remote Debugging target settings panel.
11. Similarly, for each CodeWarrior project, specify the correct Core Index value in the Remote Debugging target settings panel.

That's it. You have created a project that is configured for debugging multiple processors connected in a JTAG chain.

Debugging Embedded Linux® Software

This section explains how to use the CodeWarrior debugger to debug an embedded Linux application and to debug the U-Boot bootstrap firmware.

The topics are:

- Tutorial: Debugging an Embedded Linux® Application
- Debugging the U-Boot Bootstrap Firmware

Tutorial: Debugging an Embedded Linux® Application

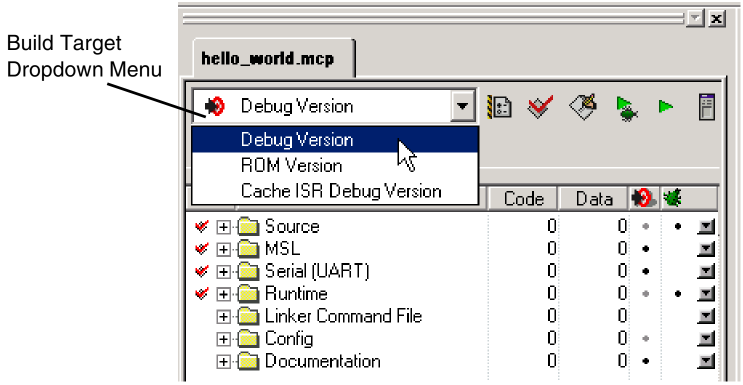
This chapter explains how to use your CodeWarrior tools to build and debug the project created in the Using the Linux® New Project Wizard section.

To build this project, and download and debug the resulting binary on your target board, follow these steps:

1. Start the CodeWarrior IDE.
2. Open the project created in the Using the Linux® New Project Wizard section.

   The project window appears, docked to the left, top, and bottom of the IDE's main window. (See Figure 4.41.)

Figure 4.41 Project Window



3. From the build target dropdown menu, select `Application Debug`.
4. Select **Project > Make**.

   The IDE compiles/assembles the project's source code files and links the resulting object code into an executable file.
5. Connect your target board to your PC.

   a. Ensure that target board's power switch is in the OFF position.

   b. Connect a power supply to the board.

   c. Connect an Ethernet cable to the target board and to your PC.

   d. Connect a serial cable to the target board and to your PC.

   e. On your PC, start a terminal emulator program.

   f. Configure the terminal emulator as shown in Table 4.20.

Table 4.20 Terminal Emulator Configuration Settings

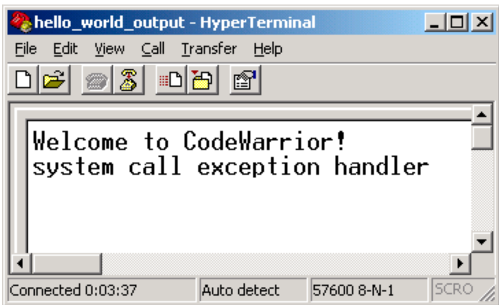
bits per second	115200
data bits	8
parity	none
stop bits	1
hardware flow control	none
software flow control	none

g. Move the board's power switch to the ON position.

The target board powers up.

The terminal emulator displays Linux boot status messages and then displays a login prompt. (See Figure 4.42.)

Figure 4.42 Terminal Emulator Showing Linux® Shell Prompt



h. In the terminal emulator, type `root` at the login prompt and press **Enter**.

The system prompts you for a password.

i. In the terminal emulator, type `root` at the password prompt an press **Enter**.

The system logs you in as user `root` and displays this prompt:
`~ #`

j. At the prompt, execute this command:
`~ # ifconfig eth0 IPAddress netmask Mask`

(where *`IPAddress`* is an available, static IP address on your network, and *`Mask`* is the mask appropriate for your subnet).

The `ifconfig` utility assigns the specified IP address and netmask to Ethernet port 0.

NOTE If you do not have an unused, static IP address, obtain one from your network administrator.

k. At the prompt, execute this command:
`~ # ./apptrk.elf :1000`

The CodeWarrior Target-Resident Kernel (apptrk) runs in the background on the processor board and listens on port `1000` for CodeWarrior debugger connections.

The terminal emulator redisplays the `~ #` prompt.

NOTE If the Linux file system does not contain a copy of `apptrk`, you can generate one by building the project `trk_linux_ppc.mcp`. This file is in this folder:
`installDir``\PowerPC_EABI_Tools\`
`CodeWarriorTRK\Os\unix\linux\ppc`

l. Exit the terminal emulator program.

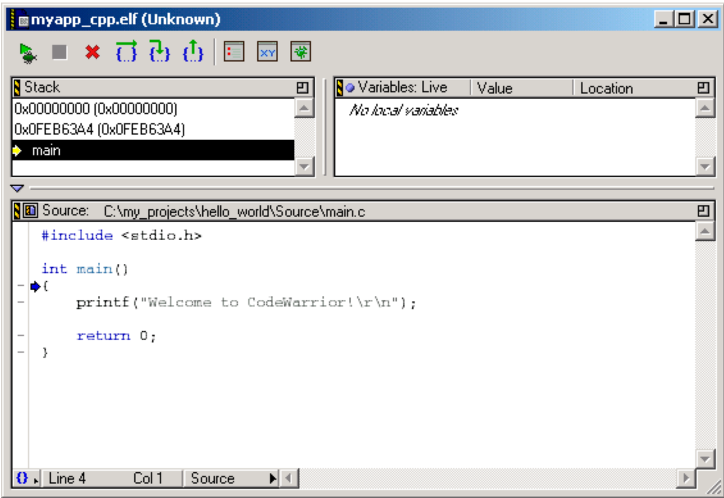
6. Select **Project > Debug**.

The debugger opens a console window, downloads the application to the processor board, halts execution at the first statement in `main()`, and displays your source code in the debugger window. (See Figure 4.43.)

The program counter icon points to the current statement (that is, to the next statement to be executed).

The debugger adds the item **Linux Info** to the IDE's menu bar.

Figure 4.43 Debugger Window



7. Control the application using the debugger.

   a. In the bottom pane of the debugger window, scroll to this statement:
   `printf("Welcome to CodeWarrior!\r\n");`

   b. In the leftmost column of debugger window, click the dash next to this statement.

   A breakpoint indicator • appears next to the statement.

   c. Click the run  button.

   The debugger executes all statements up to but not including the breakpoint statement and then halts at the breakpoint statement.

   d. Click the step over  button.

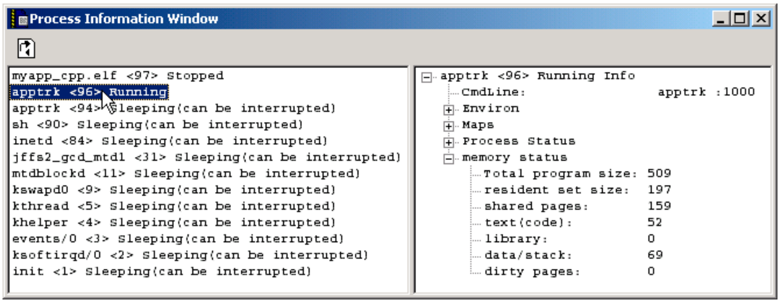
   The debugger executes the `printf()` statement and halts execution at the next statement. The text `Welcome to CodeWarrior!` appears in the console window.

   e. From the IDE's menu bar, select **Linux Info > Process Info**.

   The debugger displays the **Process Information Window**.

   The left side of this window displays the name of each process running on the target board. The right side of the window displays information about the process currently selected in the left side of the window. (See Figure 4.44.)

Figure 4.44 Process Information Window



f. Click the kill thread  button.

The debugger kills your application without letting it complete and closes the debugger window.

8. Press **Alt-F4**.

The CodeWarrior IDE exits.

That's it. You have created a Linux application project for your target board, built this project, downloaded the resulting application to the board, and used the CodeWarrior debugger to control this application's execution.

Debugging the U-Boot Bootstrap Firmware

This section explains how to use the CodeWarrior debugger to debug the U-Boot bootstrap firmware.

U-Boot resides in flash memory on your target board and boots an embedded Linux image developed for the board.

NOTE The Linux Application Edition of this product does not support debugging the U-Boot bootstrap firmware.

The topics in this section are:

- Preparing to Debug U-Boot
- Debugging U-Boot

Preparing to Debug U-Boot

NOTE The first part of this procedure must be performed on a Linux host.

To prepare to debug U-Boot on a target board, follow these steps:

1. Obtain the board support package (BSP) for your target board.

   You can download a BSP for your board from this web page:
   http://www.freescale.com/powerbsp

2. Open a terminal window.
3. Create a directory named `/mnt/iso`
4. Execute the `su` command to obtain superuser privileges.
5. Mount the `iso` file containing the BSP by executing this command:
   `# mount -o loop` *`bspFileName`* `/mnt/iso`

   (where *`bspFileName`* is a placeholder for the name of your BSP's `.iso` file).

6. Exit superuser mode.
7. Install the BSP.

   Refer to the documentation included with in the BSP for instructions.

8. Use the tools included with the BSP to build an ELF format U-Boot file that includes debugging information.

NOTE If the used BSP does not offer the Codewarrior debug support, go to the `CodeWarriorIDE/CodeWarrior/PowerPC_EABI_Tools/KernelAndUboot_patches` directory, apply one of the U-Boot patches and rebuild the U-Boot. If your U-Boot does not support any of the provided patches, manually apply the changes from the correct core version of one patch to your U-boot source tree. If you encounter problems during U-Boot debug, make sure you have completed steps (a,b,c, and d) below, before building the bootloader.

   a. Make these changes to `u-boot/config.mk`:

      - `DBGFLAGS = -g2 -gdwarf-2`
      - `AFLAGS_DEBUG = -Wa,-gdwarf2`
      - `OPTFLAGS = -O1`

NOTE If you are using an LTIB BSP, you may need to change the optimization flag of the U-Boot `CFLAGS` argument in this file:
`install``/ltib/config/platform/`*`boardName`*`/.config file`.

   b. In `u-boot/lib_ppc/board.c`, change the token `debug` to the token `printf` in the statement that includes the string `now running in ram`.

c. Build U-Boot.

You now have an ELF format U-Boot file that contains debugging information. In addition, you have a U-Boot raw binary that you can write to flash memory on the target board.

NOTE The following procedure must be performed using the Professional Edition of this CodeWarrior product.

9. Start the CodeWarrior IDE.
10. Use the CodeWarrior flash programmer to write the raw binary U-Boot file (not the ELF format file) to the flash memory of your target board.

    Refer to the Flashing U-Boot section of your board's BSP User's Guide for instructions that explain how to flash U-Boot to your board. This document is in this folder of the BSP directory tree: `help/software/`

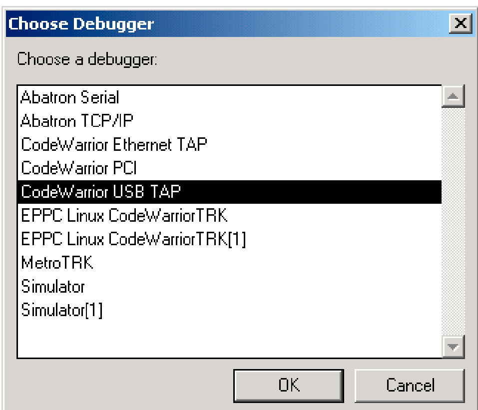
NOTE Do *not* write the ELF format U-Boot file to flash memory; you must use the raw binary U-Boot file.

11. From the IDE's menu bar, select **File > Open**.

    The standard Windows® **Open** dialog box appears.
12. Use this dialog box to find and open the ELF format U-Boot file.
13. Click **OK**.

    The IDE displays the **Choose Debugger** dialog box. (See Figure 4.45.)

Figure 4.45 Choose Debugger Dialog Box



14. From this dialog box, select one of these remote connections:

- CodeWarrior Ethernet TAP—if you are using an Ethernet TAP probe.
- CodeWarrior USB TAP—if you are using a USB TAP probe.

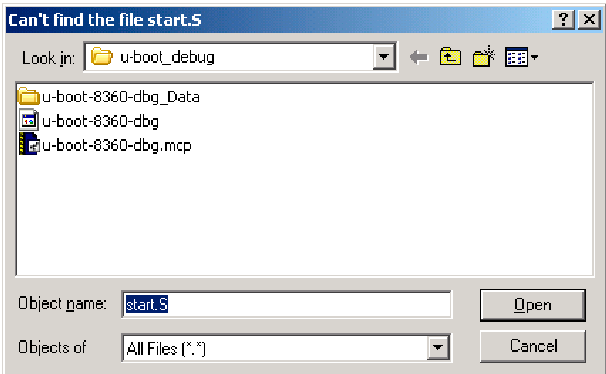
NOTE You must use a JTAG probe to debug U-Boot. The CodeWarrior USB TAP and the CodeWarrior Ethernet TAP are both JTAG devices.

15. Click **OK**.

In the directory containing the ELF format U-Boot file, the IDE creates a CodeWarrior project containing the source code files used to build the U-Boot file. As the IDE creates this project, it displays a progress bar that indicates project-creation progress.

For each U-Boot source code file that the IDE *cannot* find, it displays a dialog box with which you can navigate to and select the missing file. (See Figure 4.46.)

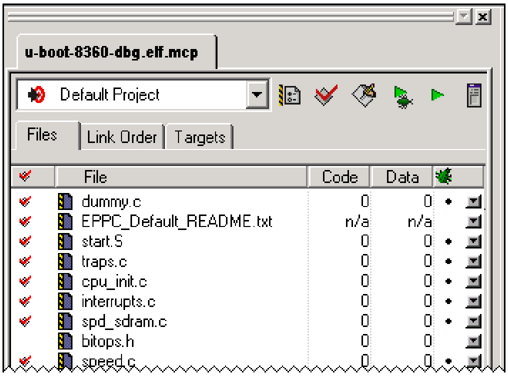
Figure 4.46 Missing Source File Dialog Box



NOTE For the IDE to create a complete U-Boot project file, you must have all source code files used to build the ELF format U-Boot file.

For each source code file that cannot be found, the IDE logs a message to the **Project Creator Log** window. Once project creation is complete, the IDE displays the project in a project window. (See Figure 4.47.)

Figure 4.47 CodeWarrior™ Project Window for U-Boot



That's it. You now have CodeWarrior project with which you can debug the U-Boot bootstrap firmware just written to the target board's flash memory.

NOTE While debugging U-Boot on 86xx, if the Address Translations option has not been enabled and you set a breakpoint in a part of code after the address translation is done, this breakpoint will not be hit. Breakpoints can be used until enable address translation is done. You can use step into to debug through the address translation section (breakpoints / step over / run to cursor cannot be used). After the translation is enabled, you can start using again the hardware breakpoints. A breakpoint set in the c) part of code while debugging in the a) part of code will not be hit.

Debugging U-Boot

On power-up, the processor starts executing the U-Boot image in flash memory. First, the code executed from flash enables the processor's MMU. Next, the code executed from flash copies the main part of the U-Boot image to RAM. Finally, execution jumps to the U-Boot code in RAM.

Because the target settings required to debug U-Boot before the MMU is enabled, after the MMU is enabled, and after execution from RAM starts are different, you must debug U-Boot in three stages.

These sections explain how to debug U-Boot at each stage of its execution:

- Debugging U-Boot before the MMU is Enabled
- Debugging U-Boot after the MMU is Enabled
- Debugging the U-Boot Section in RAM

Debugging U-Boot before the MMU is Enabled

To debug the U-Boot section in flash memory before the chip's MMU has been enabled, follow these steps:

1. Start the CodeWarrior IDE.
2. Open the U-Boot project.
3. Press **Alt-F7**.

   The **Target Settings** window appears.
4. Select `Debugger Settings` from the Target Settings Panels list of the **Target Settings** window.

   The **Debugger Settings** panel appears.

NOTE See the *IDE User's Guide* for a definition of each option in this panel.

5. In this panel, make these settings:
   - Check the Stop on Application Launch box.
   - Select the Program entry point option button.
6. Select `Remote Debugging` from the Target Settings Panels list of the **Target Settings** window.

   The **Remote Debugging** target settings panel appears.

NOTE See the *IDE User's Guide* for a definition of each option in this panel.

7. In this panel, ensure that one of the remote connection names listed below appears in the Connection dropdown menu.
   - CodeWarrior Ethernet TAP
   - CodeWarrior USB TAP
8. In this panel, click the **Edit Connection** button.

   The **Edit Connections** dialog box appears and displays the configuration for the selected remote connection.
9. In this dialog box, check the Reset Target on Launch checkbox.
10. Click **OK**.

    The **Edit Connections** dialog box closes.
11. Select `Debugger PIC Settings` from the Target Settings Panels list of the **Target Settings** window.

    The Debugger PIC Settings target settings panel appears.
12. In this panel, make these selections:

- Check the Alternate Load Address checkbox.
- In the Alternate Load Address text box, enter the address at which the U-Boot image was written to flash memory.

13. Select `EPPC Debugger Settings` from the Target Settings Panels list of the **Target Settings** window.

    The EPPC Debugger Settings panel appears.

14. In this panel, make these selections:

    a. From the Target Processor dropdown menu, select the processor on your target board.

    b. From the Target OS dropdown menu, select `BareBoard`.

    c. If your board needs to be initialized prior to being debugged:

    - Check the Use Target Initialization File checkbox.
    - Click **Browse** to display a dialog box with which you can choose the U-Boot target initialization file for your board.

NOTE If the U-Boot initialization file for the used target is not there, you may use the ROM initialization file for that target.

    d. In the Program Download Options group box, clear all the checkboxes in the Initial Launch and Successive Runs boxes.

15. In the **Target Settings** window, click **OK**.

    The IDE saves your settings and closes the **Target Settings** window.

16. On your PC, start a terminal emulator program.
17. Configure the terminal emulator as shown in Table 4.21.

Table 4.21 Terminal Emulator Configuration Settings

bits per second	115200
data bits	8
parity	none
stop bits	1
hardware flow control	none
software flow control	none

18. Move the board's power switch to the ON position.

    The board powers up.

    In the terminal emulator, U-Boot displays status messages and then displays this message:
    `Hit any key to stop autoboot: N`

    (where *N* is the number of seconds left until autoboot starts).

19. Before *N* reaches zero, press **Enter**.

    U-Boot displays this prompt: `-->`

NOTE If during its initialization, U-Boot requests a MAC address, enter a dummy MAC address (such as, `00:01:03:00:01:04`), and press **Enter**.

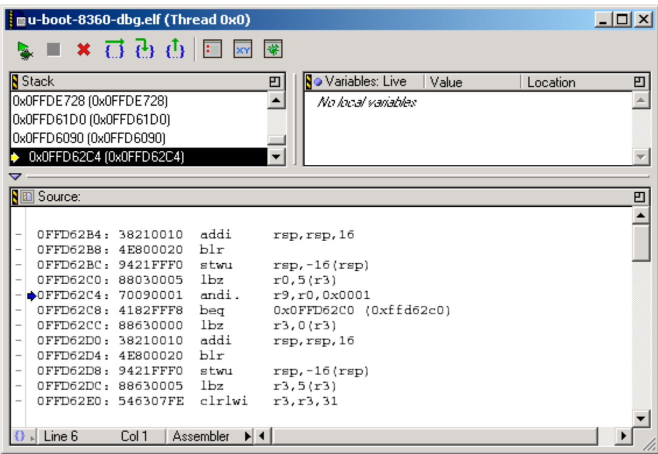
20. In the CodeWarrior IDE, select **Debug > Attach to Process**.

    The debugger connects to the target board and displays the debugger window.

21. In the debugger window, click the break  button.

    The debugger halts U-Boot's execution and displays disassembled code in the Source pane of the debugger window. (See Figure 4.48.)

Figure 4.48 Debugger Window Showing Disassembled U-Boot Code



22. Select **Debug > EPPC > Hard Reset**.

    The debugger sends a hard reset signal to the board. The debugger window displays the `__start` section. You can debug from this point up to the first `blr` instruction in `start.S`.

Debugging U-Boot after the MMU is Enabled

To debug the U-Boot section in flash memory after the chip's MMU has been enabled, follow these steps:

1. Start the CodeWarrior IDE.
2. Open the U-Boot project.
3. Press **Alt-F7**.

   The **Target Settings** window appears.
4. Select `Debugger Settings` from the Target Settings Panels list of the **Target Settings** window.

   The **Debugger Settings** panel appears.

NOTE See the *IDE User's Guide* for a definition of each option in this panel.

5. In this panel, make these settings:
   - Check the Stop on Application Launch box.
   - Select the Program entry point option button.
6. Select `Remote Debugging` from the Target Settings Panels list of the **Target Settings** window.

   The **Remote Debugging** target settings panel appears.

NOTE See the *IDE User's Guide* for a definition of each option in this panel.

7. In this panel, ensure that one of the remote connection names listed below appears in the Connection dropdown menu.
   - CodeWarrior Ethernet TAP
   - CodeWarrior USB TAP
8. In this panel, click the **Edit Connection** button.

   The **Edit Connections** dialog box appears and displays the configuration for the selected remote connection.
9. In this dialog box, check the Reset Target on Launch checkbox.
10. Click **OK**.

    The **Edit Connections** dialog box closes.
11. Select `Debugger PIC Settings` from the Target Settings Panels list of the **Target Settings** window.

    The Debugger PIC Settings target settings panel appears.
12. In this panel, uncheck the Alternate Load Address checkbox.

13. Select `EPPC Debugger Settings` from the Target Settings Panels list of the **Target Settings** window.

    The EPPC Debugger Settings panel appears.

14. In this panel, make these selections:

    a. From the Target Processor dropdown menu, select the processor on your target board.

    b. From the Target OS dropdown menu, select `BareBoard`.

    c. If your board needs to be initialized prior to being debugged:

    - Check the Use Target Initialization File checkbox.
    - Click **Browse** to display a dialog box with which you can choose the target initialization file for your board.

    d. In the Program Download Options group box, clear all the checkboxes in the Initial Launch and Successive Runs boxes.

15. In the **Target Settings** window, click **OK**.

    The IDE saves your settings and closes the **Target Settings** window.

16. On your PC, start a terminal emulator program.

17. Configure the terminal emulator as shown in Table 4.22.

Table 4.22 Terminal Emulator Configuration Settings

bits per second	115200
data bits	8
parity	none
stop bits	1
hardware flow control	none
software flow control	none

18. Move the board's power switch to the ON position.

    The board powers up.

    In the terminal emulator, U-Boot displays status messages and then displays this message:
    `Hit any key to stop autoboot: N`

    (where *N* is the number of seconds left until autoboot starts).

19. Before *N* reaches zero, press **Enter**.

    U-Boot displays this prompt: `-->`

NOTE If during its initialization, U-Boot requests a MAC address, enter a dummy MAC address (such as, `00:01:03:00:01:04`), and press **Enter**.

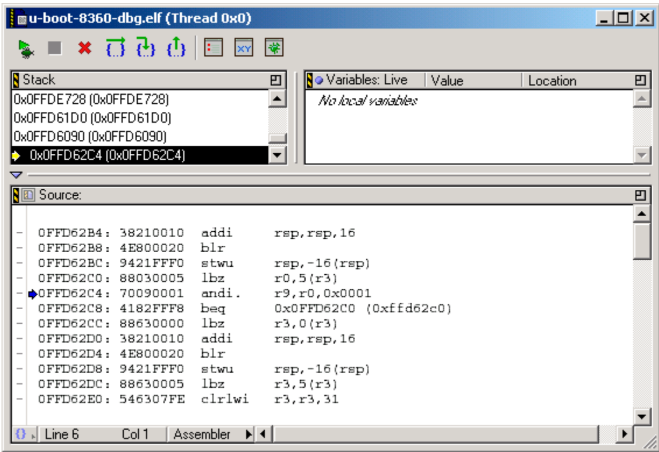
20. In the CodeWarrior IDE, select **Debug > Attach to Process**.

    The debugger connects to the target board and displays the debugger window.

21. In the debugger window, click the break button.

    The debugger halts U-Boot's execution and displays disassembled code in the Source pane of the debugger window. (See Figure 4.49.)

Figure 4.49 Debugger Window Showing Disassembled U-Boot Code

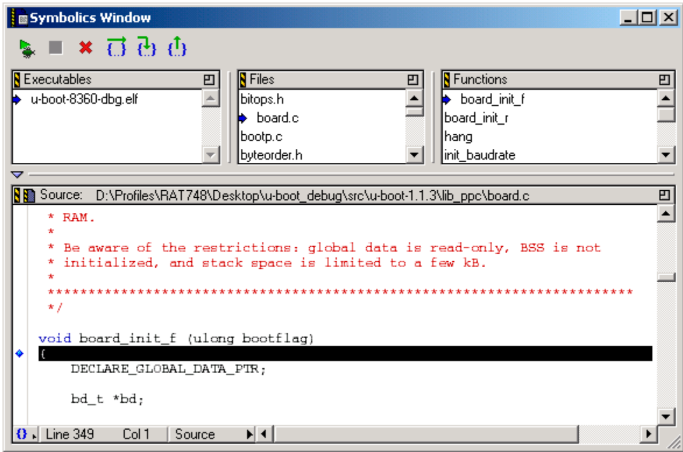


22. Set a hardware breakpoint at `board_init_f`.

    a. In the debugger window, click the symbolics button.

       The **Symbolics** window appears. (See Figure 4.50.)

Figure 4.50 Symbolics Window for U-Boot ELF File Showing Function board_init_f



b. In the Executables pane of the **Symbolics** window, select the U-Boot ELF file.

   The Files pane populates with the names of the source code files used to build the ELF file.

c. In the Files pane of the **Symbolics** window, select `board.c`.

   The Functions pane populates with the functions defined in `board.c`.

d. In the Functions pane of the **Symbolics** window, select `board_init_f`.

   The `board_init_f` function's source code appears in the Source pane of the **Symbolics** window.

e. Move the mouse cursor to the tic mark next to the entry point of the `board_init_f` function and right-click.

   A context menu appears.

f. From this context menu, select Set Hardware Breakpoint.

   A hardware breakpoint indicator ◆ appears at the selected tick mark.

g. Close the **Symbolics** window.

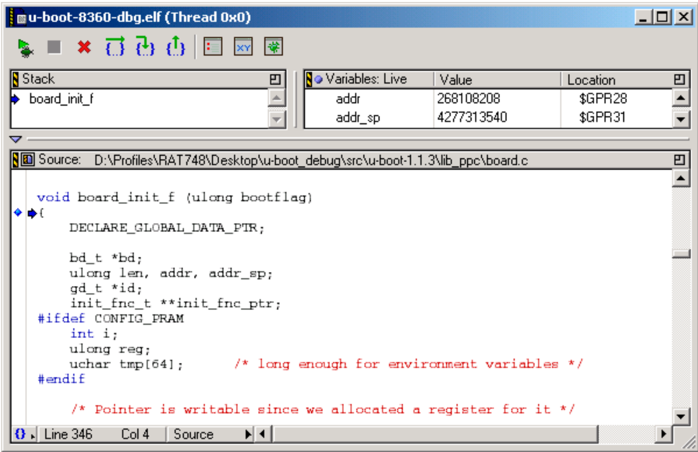
23. Select **Debug > EPPC > Hard Reset**.

    The debugger sends a hard reset signal to the target board.

24. In the debugger window, click the run button.

    U-Boot executes until it reaches the hardware breakpoint set previously. The debugger then halts execution and displays the `board_init_f` function. (See Figure 4.51.)

Figure 4.51 Debugger Window After Hitting the Hardware Breakpoint in board_init_f



25. In the debugger window, click the step over  button.

    The debugger steps from one C-language statement to the next.

That's it. You can now debug the U-Boot section in flash memory.

Debugging the U-Boot Section in RAM

To debug the U-Boot section in RAM, follow these steps:

1. On your PC, start a terminal emulator program.
2. Configure the terminal emulator as shown in Table 4.23.

Table 4.23 Terminal Emulator Configuration Settings

bits per second	115200
data bits	8
parity	none
stop bits	1
hardware flow control	none
software flow control	none

3. Move the board's power switch to the ON position.

   The board powers up.

   The terminal emulator displays U-Boot startup messages and then and then displays this message:
   `Hit any key to stop autoboot: N`

   (where *N* is the number of seconds left until autoboot starts).

4. Press **Enter**.

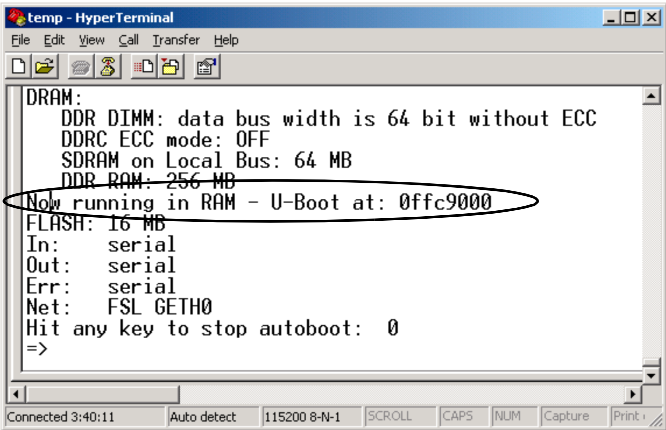
   U-Boot displays this prompt: `-->`

5. Write down the memory address displayed by the terminal emulator in this string:
   `Now running in RAM - U-Boot at: memory_address`

   (where *memory_address* is a placeholder for the real address at which U-Boot resides in RAM).

   Figure 4.52 shows the string that contains the U-Boot RAM address.

Figure 4.52 Terminal Emulator Showing U-Boot RAM Memory Address



6. Start the CodeWarrior IDE.

7. Open the U-Boot project.

8. Press **Alt-F7**.

   The **Target Settings** window appears

9. Select `Debugger Settings` from the Target Settings Panels list of the **Target Settings** window.

   The **Debugger Settings** panel appears.

NOTE See the *IDE User's Guide* for a definition of each option in this panel.

10. In this panel, make these settings:
    - Check the Stop on Application Launch box.
    - Select the Program entry point option button.
11. Select `Remote Debugging` from the Target Settings Panels list of the **Target Settings** window.

    The **Remote Debugging** target settings panel appears.

NOTE See the *IDE User's Guide* for a definition of each option in this panel.

12. In this panel, ensure that one of the remote connection names listed below appears in the Connection dropdown menu.
    - CodeWarrior Ethernet TAP
    - CodeWarrior USB TAP
13. In this panel, click the **Edit Connection** button.

    The **Edit Connections** dialog box appears and displays the configuration for the selected remote connection.
14. In this dialog box, check the Reset Target on Launch checkbox.
15. Click **OK**.

    The **Edit Connections** dialog box closes.
16. Select `Debugger PIC Settings` from the Target Settings Panels list of the **Target Settings** window.

    The Debugger PIC Settings target settings panel appears.
17. In this panel, make these settings:
    - Check the Alternate Load Address checkbox.
    - In the Alternate Load Address text box, enter the U-Boot RAM address you wrote down previously.

NOTE If you specify an alternate load address, the debugger can display source code for sections in RAM only. This is because an alternate load address value causes the debugger to assume that all sections have been relocated to RAM. For the same reason, if no alternate load address is specified, the debugger can display source code for sections in flash memory only.

18. Select `EPPC Debugger Settings` from the Target Settings Panels list of the **Target Settings** window.

    The EPPC Debugger Settings panel appears.

19. In this panel, make these selections:

    a. From the Target Processor dropdown menu, select the processor on your target board.

    b. From the Target OS dropdown menu, select `BareBoard`.

    c. Uncheck the Use Target Initialization File checkbox.

    d. In the Program Download Options group box, clear all the checkboxes in the Initial Launch and Successive Runs boxes.

20. In the **Target Settings** window, click **OK**.

    The IDE saves your settings and closes the **Target Settings** window.

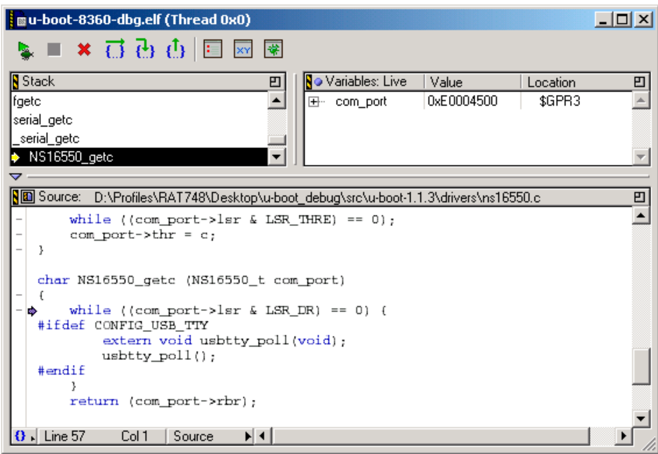
21. Select **Debug > Attach to Process**.

    The debugger connects to the target board and displays the debugger window.

22. In the debugger window, click the break  button.

    The debugger halts U-Boot's execution and displays disassembled code in the Source pane of the debugger window. (See Figure 4.53.)

Figure 4.53 Debugger Window Source Code for the U-Boot RAM Section

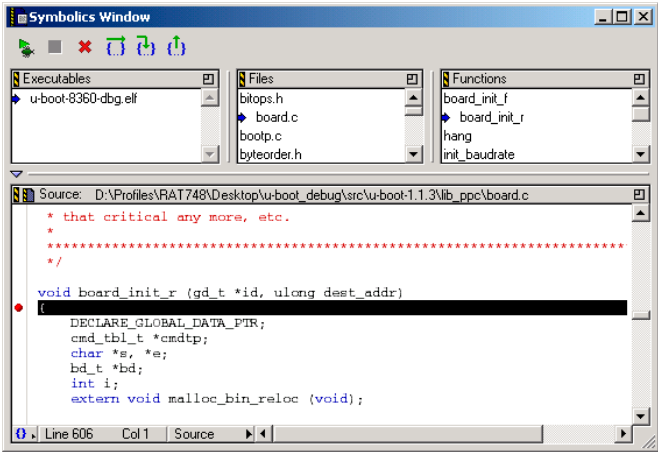


23. Set a software breakpoint at `board_init_r`.

    a. In the debugger window, click the symbolics  button.

       The **Symbolics** window appears. (See Figure 4.54.)

Figure 4.54 Symbolics Window for U-Boot ELF File Showing Function board_init_r



b. In the Executables pane of the **Symbolics** window, select the U-Boot ELF file.

   The Files pane populates with the names of the source code files used to build the ELF file.

c. In the Files pane of the **Symbolics** window, select `board.c`.

   The Functions pane populates with the functions defined in `board.c`.

d. In the Functions pane of the **Symbolics** window, select `board_init_r`.

   The `board_init_r` function's source code appears in the Source pane of the **Symbolics** window.

e. Move the mouse cursor to the tic mark next to the entry point of the `board_init_r` function and right-click.

   A context menu appears.

f. From this context menu, select Set Software Breakpoint.

   A software breakpoint indicator ● appears at the selected tick mark.

g. Close the **Symbolics** window.

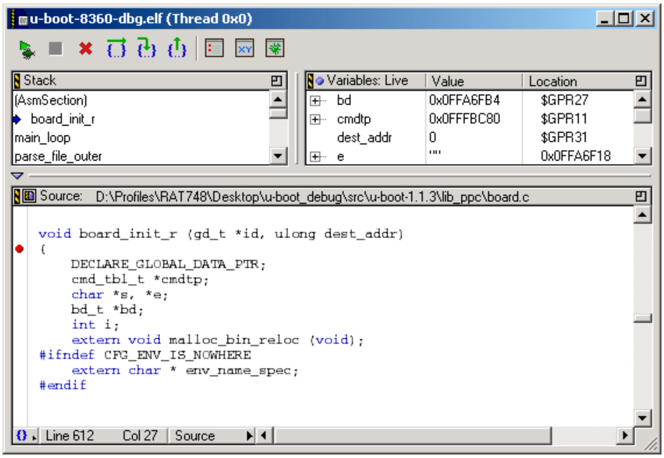
24. Select **Debug > EPPC > Hard Reset**.

    The debugger sends a hard reset signal to the target board.

25. In the debugger window, click the run button.

    U-Boot executes until it reaches the software breakpoint set previously. The debugger then halts execution and displays the `board_init_r` function. (See Figure 4.55.)

Figure 4.55 Debugger Window After Hitting the Software Breakpoint in board_init_r



26. In the debugger window, click the step over  button.

    The debugger steps from one C-language statement to the next.

That's it. You can now debug the U-Boot section in RAM.

5

Working with the Hardware Tools

This chapter explains how to use the CodeWarrior hardware tools. Use these tools for board bring-up, test, and analysis.

The sections of this chapter are:

- Flash Programmer
- Hardware Diagnostics Tool
- EPPC Trace Buffer Support

NOTE The flash programmer, hardware diagnostics tool, and support for the EPPC trace buffer are not included in the Linux® Application Edition of this product.

Flash Programmer

The CodeWarrior flash programmer lets you manipulate the flash memory of any supported Power Architecture board from within the CodeWarrior IDE. Specifically, the flash programmer can perform these functions:

- Program
- Erase
- Blank Check
- Verify
- Checksum

NOTE The debugger provides common flash-programmer features (such as view/modify memory, view/modify registers, and save memory to a file). As a result, the CodeWarrior flash programmer does not include these features.

CodeWarrior for Power Architecture Processors includes a flash programmer settings file for each supported target board. These files are in this directory:

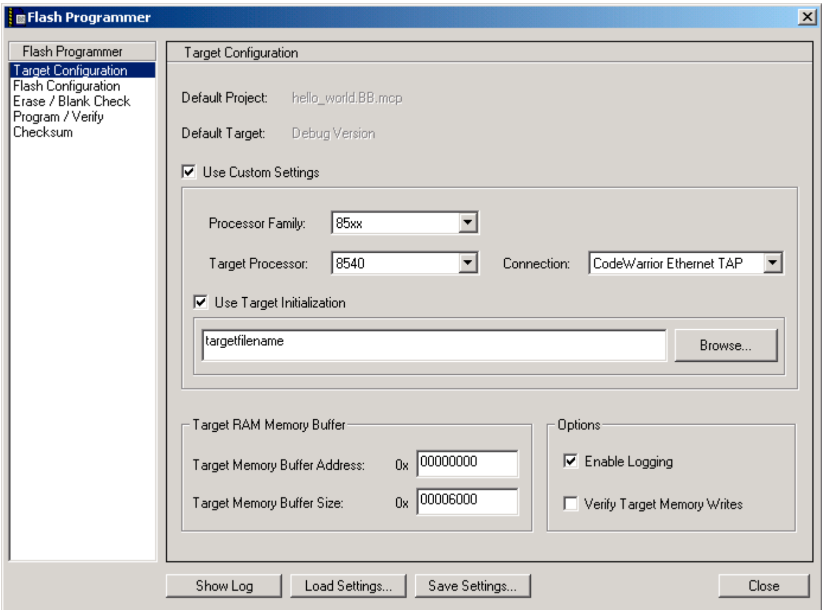
```
installDir\bin\Plugins\Support\Flash_Programmer\EPPC
```

To configure the flash programmer so it works with your target board, follow these steps:

1. From the IDE's menu bar, select **Tools > Flash Programmer**.

   The **Flash Programmer** window appears. (See Figure 5.1.)

Figure 5.1 Flash Programmer Window



2. Select `Target Configuration` from the pane on the left side of the **Flash Programmer** window.

   The **Target Configuration** panel appears on the right side of the **Flash Programmer** window.

3. Optionally, configure the **Flash Programmer** window using a CodeWarrior project.

   a. Open a CodeWarrior project and select the build target that has the target settings you want to use.

   b. Clear the Use Custom Settings checkbox.

      The appearance of the Default Project and Default Target text strings changes from dim to normal.

   c. Select `Flash Configuration` from the pane on the left side of the **Flash Programmer** window.

      The **Flash Device Configuration** panel appears on the right side of the **Flash Programmer** window.

d. For the options listed below, select values appropriate for the flash memory device on your board.

- Flash Memory Base Address
- Device
- Organization
- Sector Address Map

NOTE See your target board's User Manual for the values to specify for these options.

e. Skip step 4.

4. Optionally, configure the **Flash Programmer** window using a flash programmer settings file.

a. Check the Use Custom Settings checkbox.

The appearance of the Default Project and Default Target text strings changes from normal to dim.

b. Click **Load Settings**.

A standard "open file" dialog box appears.

c. Use this dialog box to select the flash programmer settings file appropriate for your target board.

d. Click **Open**.

The dialog box closes.

The options on each panel of the **Flash Programmer** window are set using values from the selected settings file.

5. From the Connection dropdown menu, select the probe you are using.

That's it. The CodeWarrior flash programmer is now configured to work with your board.

See the *CodeWarrior™ IDE User's Guide* for instructions that explain how to use the **Flash Programmer** window.

Hardware Diagnostics Tool

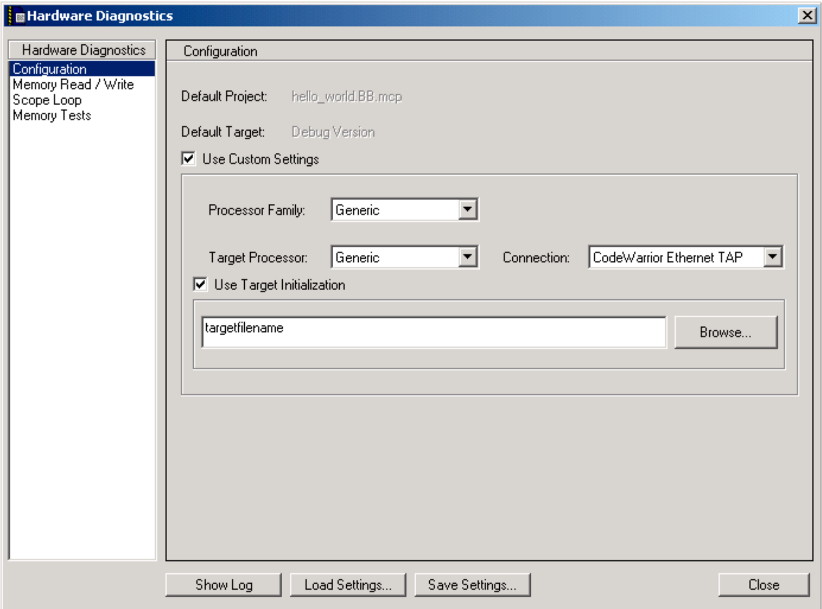
The CodeWarrior hardware diagnostics tool lets you test your target board's hardware.

To configure the hardware diagnostics tool so it works with your target board, follow these steps:

1. From the IDE's menu bar, select **Tools > Hardware Diagnostics**.

   The **Hardware Diagnostics** window appears. (See Figure 5.2.)

Figure 5.2 Hardware Diagnostics Window



2. Select `Configuration` from the pane on the left side of the **Hardware Diagnostics** window.

   The **Configuration** panel appears on the right side of the **Hardware Diagnostics** window.

3. Optionally, configure the **Hardware Diagnostics** window using a CodeWarrior project.

   a. Open a CodeWarrior project and select the build target that has the target settings you want to use.

   b. Clear the Use Custom Settings checkbox.

      The appearance of the Default Project and Default Target text strings changes from dim to normal.

   c. Skip step 4.

4. Optionally, configure the **Hardware Diagnostics** window using a hardware diagnostics settings file.

To use a hardware diagnostics settings file to configure the **Hardware Diagnostics** window, follow these steps:

a. Check the Use Custom Settings checkbox.

   The appearance of the Default Project and Default Target text strings changes from normal to dim.

b. Click **Load Settings**.

   A standard "open file" dialog box appears.

c. Use this dialog box to select the hardware diagnostics settings file appropriate for your target board.

d. Click **Open**.

   The dialog box closes.

   The options on each panel of the **Hardware Diagnostics** window are set using values from the selected settings file.

5. From the Connection dropdown menu, select the probe you are using.

That's it. The CodeWarrior hardware diagnostics tool is now set up to work with your target board.

See the *CodeWarrior™ IDE User's Guide* for instructions that explain how to use the **Hardware Diagnostics** window.

EPPC Trace Buffer Support

The EPPC trace buffer is a 256- x 64-bit buffer that can capture information related to the internal processing of transactions with the processing interfaces. This visibility into internal device behavior is useful for debugging application software through inverse assembly and reconstruction of the fetch stream.

On some Power Architecture processors, trace buffer support is implemented in hardware; as a result, the trace buffer does not affect application performance.

You can configure the trace buffer to trace the dispatch bus from any of these interfaces:

- e500 coherency module (ECM)
- Outbound host interface to the RapidIO controller
- Outbound host interface to the PCI controller
- Host interface to the DDR controller.

NOTE You can trace only one interface at a time.

As transactions come into the ECM, the ECM arbitrates common resources and dispatches the transactions to target ports. You can capture information such as transaction types, source ID, and other attributes for any of the selected interfaces.

Trace events hold this information:

- Transaction type — the type of transaction (for example, write with local processor snoop, or read with unlock)
- Source of the transaction — the source block or port of the transaction (for example, the local processor for data fetches).
- Target of the transaction — the target block or port of the transaction (typically slave ports in a transaction, such as local memory)
- The size of the transaction, in bytes

NOTE Transaction target is meaningful only if monitoring the ECM dispatch bus.

You can configure the trace buffer to record all transactions or to record only:

- transactions with a specified source ID
- transactions with a specified target ID
- transactions whose address matches a specified masked address
- transactions whose current context ID (the value of CCIDR register) matches or does not match the programmed context ID (the value of PCIDR register).

You can combine any of these conditions.

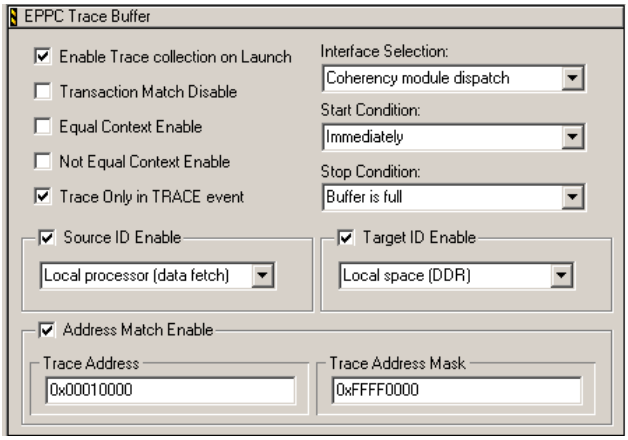
You use the **EPPC Trace Buffer** target settings panel to configure the trace buffer for each build target in a CodeWarrior project.

For example, you could set up the **EPPC Trace Buffer** panel such that the trace buffer records just transactions that meet these criteria:

- Dispatched by the ECM
- Source ID is “Local Processor Data Fetch”
- Target ID is “Local Space DDR”
- Address in the range `0x00010000` to `0x0001FFFF`

Figure 5.3 shows the **EPPC Trace Buffer** set up this way.

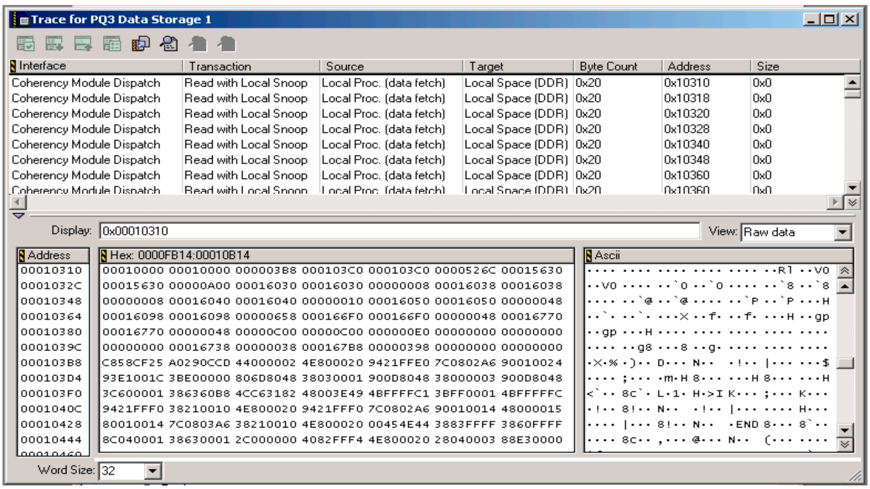
Figure 5.3 EPPC Trace Buffer Panel Showing Example Trace Buffer Configuration



To see the events captured in the trace buffer during a debug session, display the trace window. To do this, select **Data > View Trace** from the IDE's menu bar.

Figure 5.4 show the trace window after a debug session run using the trace buffer configuration shown in Figure 5.3.

Figure 5.4 Trace Buffer Window Showing Captured Events



For a documentation of each option in the EPPC trace buffer target settings panel, see the EPPC Trace Buffer topic.

NOTE This CodeWarrior product includes an example project that shows you how to use the debugger's EPPC trace buffer visibility feature. The project file is named TraceBuffer_8560ADS_REVA and is in this directory: `*installDir*\(CodeWarrior_Examples)\PowerPC_EABI\TraceBuffer_8560ADS_REVA`.

A

Debugger Limitations and Workarounds

This appendix documents processor-specific CodeWarrior debugger limitations and workarounds.

The sections of this appendix are:

PowerQUICC I Processors

The PowerQUICC I family includes the 8xx series of processors.

Working With Watchpoints

The 8xx processor implements two load and store address comparator registers. The CodeWarrior debugger uses both these registers to enable placing a single watchpoint on any variable or memory range. The watchpoint is 1-byte aligned.

Working with Hardware Breakpoints

The 8xx processor implements four address instruction breakpoints (hardware breakpoints) that can be used during a debug session.

PowerQUICC II Processors

The PowerQUICC II family includes these processors:

- G2: 8240/1/5, 825x, 826x
- G2 LE: 8247/8, 827x, 828x, 5200

Working with Watchpoints

G2 Cores

G2 cores do not support watchpoints.

G2 LE Cores

G2 LE cores implement two data address registers. The CodeWarrior debugger uses these registers to place a single watchpoint on a variable or memory range.

A watchpoint set on a variable or memory address is equivalent to a watchpoint set on an aligned address and a range of 64-bit multiple.

Working with Hardware Breakpoints

G2 Cores

G2 cores implement one address instruction breakpoint (hardware breakpoint) that can be used in a debug session.

G2 LE Cores

G2 LE cores implement two address instruction breakpoints (hardware breakpoints) that can be used in a debug session.

Working with Memory Mapped Registers

G2 Cores

For G2 cores, you must provide the internal memory map base address before the CodeWarrior debugger can access the internal memory-mapped registers (MMR). There are three ways to provide this address:

- Use the setMMRBaseAddr command in a target initialization file.
- During a debug session, select **Debug > EPPC** and enter the required address in the **Change IMMR** dialog box that appears.
- During a debug session, display the **Command Window** and issue this command: `cmdwin::eppc::setMMRBaseAddr`

G2 LE Cores

G2 LE cores have an internal memory-mapped registers base address register (IMMRBAR). This is a memory-mapped register that relocates with the whole internal memory map.

Further, the debugger uses the special purpose memory base address register (MBAR) to store the base address of the internal memory-mapped registers.

Each time the location of the internal memory map changes, you must maintain the correspondence between the IMMRBAR and MBAR registers.

PowerQUICC II Pro Processors

The PowerQUICC II Pro family includes these processors:

- e300c1: 834x, 835x, 836x
- e300c2: 832x, e300c3: 831x
- e300c4: 837x, 5121e

Debugging interrupt handlers

If a target takes an exception and is stopped at the beginning of an interrupt handler, the program counter (PC) often shows the previous address instead of the correct address. For example, the PC would show `0x6FC` instead of `0x700` or `0x10FC` instead of `0x1100`.

To overcome this problem, a workaround has been implemented that automatically adds 4 to the PC if the target is stopped at a `0x…FC` address in the interrupt vector address range.

You can enable or disable this workaround for the current debug session or for all subsequent debug sessions by issuing the `cmdwin::eppc::e300_adjust_pc` command in the CodeWarrior **Command Window.**

For more information about this command, refer to the **Command Window** online help. To do this, issue the command `help cmdwin::eppc::e300_adjust_pc` in the CodeWarrior **Command Window**.

Cache Coherence (e300c1 Core Only)

While debugging an e300c1 target, when the core stops due to a breakpoint or due to a request, the core goes into stop mode. After this, the CodeWarrior Connection Server (CCS) moves the core from stop mode to iJam mode. However, while in stop mode, the core does not maintain cache coherency.

To solve this problem, a workaround has been implemented that uses the processor's power management facilities to prevent external masters from generating new memory transactions. To achieve this, CCS tries to keep the PMCCR register with defined values (`PMCCR[SLPEN] == b'1` and `PMCCR[DLPEN] == b'0`).

You can enable or disable this workaround for the current debug session or for all subsequent debug session (after the download phase) by issuing the `cmdwin::eppc::e300c1_cache_coherence` command in the CodeWarrior **Command Window.**

For more information about this command, refer to the **Command Window** online help. To do this, issue the `help cmdwin::eppc::e300c1_cache_coherence` command in the CodeWarrior **Command Window.**

Working with Watchpoints

Resuming Execution after a Watchpoint is Hit

When a target is under the debugger's control and a watchpoint (data breakpoint) condition is met, the core stops execution at the instruction that generated the data access. This instruction is called the watchpoint hit instruction.

Unfortunately, when an e300 core hits a watchpoint, the debugger cannot determine the circumstances under which the target stopped because these cores (except for the e300c1) do not update the necessary status registers. As a result, it is impossible to resume (run or step) the target after a watchpoint has been hit because the debugger cannot temporarily disable the watchpoint generated by the hit instruction.

As a result, for an e300 core, you must manually disable a watchpoint before you can resume execution from the watchpoint hit instruction.

NOTE For e300c1 cores, the watchpoint mechanism works as expected.

64-bit Alignment

The e300 core implements two data address registers. The CodeWarrior debugger uses both registers to place a single watchpoint on a variable or memory range.

Any watchpoint set on a variable or memory address is equivalent to a watchpoint set on an aligned address and a range of 64-bit multiple. This limitation stems from the e300 cores's data breakpoints implementation.

Working with Hardware Breakpoints

The e300 core implements two address instruction breakpoints (hardware breakpoints) that can be used in a debug session.

Working with Memory Mapped Registers

e300 cores have an internal memory-mapped registers base address register (IMMRBAR). This is a memory-mapped register that relocates with the whole internal memory map.

Further, the debugger uses the special purpose memory base address register (MBAR) to store the base address of the internal memory-mapped registers.

Each time the location of the internal memory map changes, you must maintain the correspondence between the IMMRBAR and MBAR registers.

PowerQUICC III Processors

The PowerQUICC III family includes e500: 85xx processors.

MMU Configuration Through JTAG

For e500 cores, the debugger is able to read and write the L2 MMU TLBs registers without using dedicated processor instructions. You can access these registers from the debugger's **Registers** window or with commands in a target initialization file.

For more information on the TLB register structure, refer to the `README.txt` file that includes in the default CodeWarrior project for each supported target board.

Reset Workaround

To put the e500 core in debug mode at reset, you must ensure that the core is running. The target initialization file sets a hardware breakpoint at the reset address. The core is stopped at the reset address to be put in the debug mode.

Working with Software Breakpoints

For e500 cores, the debugger implements software breakpoints by using debug exceptions and the corresponding interrupt handler. When a debug exception is encountered, the target is expected to stop at the debug exception handler pointed by `IVPR+IVOR15`.

However, for e500 cores, there is a chance that the first few instructions of the debug exception handler are fetched and even executed before processor halts.

As a result, the core must be able to fetch and execute valid instructions from the interrupt handler location pointed by `IVPR+IVOR15` without raising a TBL miss exception or any other exception. Also, the first few instructions of the debug interrupt handler must not perform any Load or Store operations that would corrupt the application's context if executed. If any of these conditions is not satisfied, the software breakpoint will not work.

Working with Watchpoints

The e500 core implements two data address compare registers. The CodeWarrior debugger uses both these registers to place a single watchpoint on any variable or memory range. The variable or memory range is 1-byte aligned.

Working with Hardware Breakpoints

The e500 core implements two address instruction breakpoints (hardware breakpoints) that can be used in a debug session.

Host Processors

The Host processor family includes:

- G3: 7xx
- G4: 74xx
- e600: 7448, 86xx

Working with Breakpoints

The debugger implements software breakpoints for the G3, G4 and e600 cores by using an illegal opcode, which generates a program exception. When this exception is encountered, the target is expected to stop at the program exception handler (`0x700` or `0xFFF00700`).

For G3, G4 and e600 cores, a silicon issue causes the processor to execute the first instruction of the exception handler instead of halting immediately. The debugger works around this problem by using the only hardware breakpoint available in the core. The hardware breakpoint is set to the program exception handler location (`0x700` or `0xFFF00700`) to prevent further execution.

This workaround has these consequences:

- You cannot use this hardware breakpoint simultaneously with other software breakpoints. If you try, the debugger displays the “not enough resources” message, because the hardware breakpoint is already in use. To use the hardware breakpoint, you must remove all software breakpoints currently set.
- The CodeWarrior Flash Programmer uses this workaround to control the execution of the flash algorithm. The target initialization file used by the Flash Programmer manually sets a hardware breakpoint to the program exception handler.

Working with Watchpoints

The G3, G4, and e600 cores implement one data address breakpoint register. The granularity of the data address breakpoint compare is a double word. For AltiVec quad-word loads and stores (e600 cores only), the granularity is quad-word.

Working with Hardware Breakpoints

The G3, G4, and e600 cores implement one address instruction breakpoint (hardware breakpoint), and it is used by the debugger’s software breakpoint implementation. Consequently, you cannot to use this hardware breakpoint if you have any software breakpoints set.

For more information, see Working with Breakpoints.

Generic Processors

Working with Uninitialized Stack

Debugging while the stack is not initialized can cause uninitialized memory accesses errors. This situation occurs when the debugger tries to construct the stack trace.

To avoid this problem, stop the debugger from constructing a stack trace by adding a command to your target initialization file that sets the stack pointer (SP) register to an unaligned address.

For example, you could put this command in your target initialization file:

```
writereg SP 0x0x0000000F
```

B

Target Initialization Files

A target initialization file is a file that contains commands that initialize registers, memory locations, etc. on a target board.

If necessary, you can have the CodeWarrior™ debugger execute a target initialization file immediately before the debugger downloads a bare board binary to a target board. The commands in a target initialization file put a board in the state required to debug a bare board program.

NOTE Assign a target initialization file to bare board build targets only. A board that boots embedded Linux® is already set up properly for debugging. 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).

The sections of this appendix are:

Using Target Initialization Files

A target initialization file is a command file that the CodeWarrior debugger executes each time the build target to which the initialization file is assigned is debugged.

Often, you must use a target initialization file for build targets that use a BDM or JTAG probe. The commands in the file initialize target memory as required and set any registers involved in debugging to the required values.

NOTE You do not need to use an initialization file if you debug using the CodeWarrior TRK debug monitor.

To instruct the CodeWarrior debugger to use a target initialization file, follow these steps:

1. Start the CodeWarrior IDE.
2. Open a bare board project.
3. Select one of this project's build targets.

4. Display the EPPC Debugger Settings target settings panel.
5. Check the Use Target Initialization File box of this panel and then type the path and name of the initialization file you want in the related text box.

   Alternatively, click **Browse** to display a dialog box with which you can select the target initialization file you want.

Your CodeWarrior product includes example target initialization files for the supported target boards. These files are in board-specific subdirectories of this path:

`installDir\PowerPC_EABI_Support\Initialization_Files\`

You can also write your own target initialization files. The next section documents the commands that can appear in such files.

Target Initialization File Commands

This section documents each command that can appear in a target initialization file and defines the syntax rules that these commands follow.

Command Syntax

The syntax of target initialization file commands follows these rules:

- Spaces and tabs (white space) are ignored
- Character case is ignored
- Unless otherwise noted, values may be specified in hexidecimal, octal, or decimal:
  - Hexidecimal values are preceded by `0x` (for example, `0xDEADBEEF`)
  - Octal values are preceded by `0` (for example, `01234567`)
  - Decimal values start with a non-zero numeric character (for example, `1234`)
- Comments start with a semicolon (`;`) or pound sign (`#`), and continue to the end of the line

Table of Commands

Table B.1 lists each command that can appear in a target initialization file.

Table B.1 Target Initialization Commands

alternatePC	ANDmem.l
AND	IncorMMR

Table B.1 Target Initialization Commands (*continued*)

ORmem.l	reset
run	setMMRBaseAddr
sleep	stop
writemem.b	writemem.w
writemem.l	writemem.r
writemmr	writereg
writereg128	writespr
writeupma	writeupmb

Access to Named Registers from within Scripts

Some commands described in the Command Reference section (below) allow access to memory-mapped register by name as well as address. Based on the processor selection in the debugger settings, these commands will accept the register names shown a part's Freescale User's Manual. There are also commands to access built-in registers of a processor core, for example, 'writereg'. The names of these registers follow the architectural description for the respective processor core for general purpose and special purpose registers. Note that these names (for example, GPR5) might be different from names used in assembly language (for example, r5).

NOTE To ensure correct access to named registers, read the description of the setMMRBaseAddr command and ensure it is used when necessary.

Command Reference

The section documents each target initialization file command.

For each command, the section provides a brief statement of what the command does, the command's syntax, a definition of each argument that can be passed to the command, and examples showing how to use the command.

alternatePC

Sets the program counter (PC) register to the specified value.

Syntax

`alternatePC` *`address`*

Arguments

`address`

The address to assign to the program counter register.

This address may be specified in hexidecimal (for example, `0xABCD0000`), octal (for example, `025363200000`), or decimal (for example, `2882338816`).

Example

This command assigns the address `0xc28737a4` to the program counter register:

```
alternatePC 0xc28737a4
```

ANDmem.l

Performs a bitwise AND using the 32-bit value at the specified memory address and the supplied 32-bit mask and writes the result back to the specified address.

No read/write verify is performed.

Syntax

`ANDmem.l` *`address mask`*

Arguments

`address`

The address of the 32-bit value upon which to perform the bitwise AND operation.

This address may be specified in hexidecimal (for example, `0xABCD0000`), octal (for example, `025363200000`), or decimal (for example, `2882338816`).

`mask`

32-bit mask to use in the bitwise AND operation.

Example

The command below performs a bitwise AND operation using the 32-bit value at memory location `0xC30A0004` and the 32-bit mask `0xFFFFFFFF`. The command then writes the result back to memory location `0xC30A0004`.

```
ANDmem.l 0xC30A0004 0xFFFFFEFF
```

AND

Performs a bitwise AND of the contents of the specified memory-mapped register (MMR) and the supplied 32-bit mask, and writes the result back to the specified register.

Syntax

`ANDmmr` *`regName mask`*

Arguments

`regName`

The name of the memory-mapped register upon which to perform a bitwise AND.

NOTE For more information on the memory-mapped register names accepted by this command see Access to Named Registers from within Scripts.

`mask`

32-bit mask to use in the bitwise AND operation.

Example

This command bitwise ANDs the contents of the `ACFG` register with the value `0x00002000`:

```
ANDmmr ACFG 0x00002000
```

IncorMMR

Performs a bitwise OR using the contents of the specified memory-mapped register (MMR) and the supplied 32-bit mask and writes the result back to the specified register.

Syntax

`incorMMR` *`regName mask`*

Arguments

regName

The name of the memory-mapped register (MMR) upon which to perform a bitwise OR.

NOTE For more information on the memory-mapped register names accepted by this command see Access to Named Registers from within Scripts.

mask

32-bit mask to use in the bitwise inclusive OR operation.

Example

This command bitwise ORs the contents of the `ACFG` register with the value `0x00002000`:

```
incorMMR ACFG 0x00002000
```

ORmem.l

Performs a bitwise OR using the 32-bit value at the specified memory address and the supplied 32-bit mask and writes the result back to the specified address.

No read/write verify is performed.

Syntax

`ORmem.l` *`address mask`*

Arguments

address

The address of the 32-bit value upon which to perform the bitwise OR operation.

This address may be specified in hexidecimal (for example, `0xABCD0000`), octal (for example, `025363200000`), or decimal (for example, `2882338816`).

mask

32-bit mask to use in the bitwise OR operation.

Example

The command below performs a bitwise OR operation using the 32-bit value at memory location `0xC30A0008` and the 32-bit mask `0x01000800`. The command then writes the result back to memory location `0xC30A0004`.

```
ORmem.l 0xC30A0008 0x01000800
```

reset

Resets the processor on the target board.

Syntax

`reset` *`code`*

Arguments

`code`

Number that defines what the debugger does after it resets the processor on the target board.

Use one of the values in Table B.2.

Table B.2 Post Reset Actions

Value	Description
`0`	reset the target processor, then run
`1`	reset the target processor, then stop

run

Starts program execution at the current program counter (PC) address.

Syntax

`run`

setMMRBaseAddr

Provide the debugger with the base address of a processor's memory-mapped registers (MMR). Upon execution of this command, the debugger can read, write, and display a processor's memory mapped registers.

The `setMMRBaseAddr` command must appear before any `writemmr` commands in the target initialization file.

NOTE This command is not needed in target initialization files for members of the PowerQUICC III processor family.

NOTE The debugger requires the base address of the memory-mapped registers for 825x/826x processors only. As a result, this command must appear in *all* target initialization files for 825x/826x processors.

Syntax

`setMMRBaseAddr` *`baseAddress`*

Arguments

`baseAddress`

The base address (in hexidecimal) of the memory-mapped registers.

The specified address must be in hexadecimal (for example, `0xABCD1234`).

NOTE For more information on the memory-mapped register names accepted by this command see Access to Named Registers from within Scripts.

Example

This command makes the memory-mapped register base address `0x0f00000`:

```
setMMRBaseAddr 0x0f00000
```

sleep

Causes script execution to pause the specified number of milliseconds before executing the next instruction.

Syntax

`sleep` *`milliseconds`*

Arguments

`milliseconds`

The number of milliseconds (in decimal) to pause the debugger.

Example

This command pauses the debugger for 10 milliseconds:

```
sleep 10
```

stop

Stops program execution and halts the processor on the target board.

Syntax

```
stop
```

writemem.b

Writes a byte (8 bits) of data to the specified memory address.

Syntax

`writemem.b` *`address value`*

Arguments

`address`

The memory address to which to assign the supplied 8-bit value.

This address may be specified in hexidecimal (for example, `0xABCD`), octal ((for example, `0125715`), or decimal (`43981`).

`value`

The 8-bit value to write to the specified memory address.

This value may be specified in hexidecimal (for example, `0xFF`), octal (for example, `0377`), or decimal (for example, `255`).

Example

This command writes the byte `0x1A` to the memory location `0x0001FF00`:

```
writemem.b 0x0001FF00 0x1A
```

writemem.w

Writes a word (16 bits) of data to the specified memory address.

Syntax

`writemem.w` *`address value`*

Arguments

`address`

The memory address to which to assign the supplied 16-bit value.

This address may be specified in hexidecimal (for example, `0xABCD0000`), octal (for example, `025363200000`), or decimal (for example, `2882338816`).

`value`

The 16-bit value to write to the specified memory address.

This value may be specified in hexidecimal (for example, `0xFFFF`), octal (for example, `0177777`), or decimal (for example, `65535`).

Example

This command writes the word `0x1234` to memory location `0x0001FF00`:

```
writemem.w 0x0001FF00 0x1234
```

writemem.l

Writes a long integer (32 bits) of data to the specified memory location.

Syntax

`writemem.l` *`address value`*

Arguments

`address`

The memory address to which to assign the supplied 32-bit value.

This address may be specified in hexidecimal (for example, `0xABCD0000`), octal (for example, `025363200000`), or decimal (for example, `2882338816`).

`value`

The 32-bit value to write to the specified memory address.

This value may be specified in hexidecimal (for example, `0xFFFFABCD`), octal (for example, `037777725715`), or decimal (for example, `4294945741`).

Example

This command writes the long integer `0x12345678` to the memory location `0x0001FF00`:

```
writemem.w 0x0001FF00 0x12345678
```

writemem.r

Writes a value to the specified register.

Syntax

`writemem.r` *`regName value`*

Arguments

`regName`

The name of the register to which to assign the supplied value.

`value`

The value to write to the specified register.

This value may be specified in hexidecimal (for example, `0xFFFFABCD`), octal (for example, `037777725715`), or decimal (for example, `4294945741`).

Example

This command writes the value `0xffffffc3` to the `SYPCR` register:

```
writemem.r SYPCR 0xffffffc3
```

writemmr

Writes a value to the specified memory-mapped register (MMR).

Syntax

`writemmr` *`regName value`*

Arguments

`regName`

The name of the memory mapped register to which to assign the supplied value.

NOTE This command accepts most Power Architecture processor memory-mapped register names. If the command rejects a memory mapped register name, use writemem.r instead. For more information on the memory mapped register names accepted by this command see Access to Named Registers from within Scripts.

`value`

The value to write to the specified memory-mapped register.

This value may be specified in hexidecimal (for example, `0xFFFFABCD`), octal (for example, `037777725715`), or decimal (for example, `4294945741`).

Example

This command writes the value `0xffffffc3` to the `SYPCR` register:

```
writemmr SYPCR 0xffffffc3
```

This command writes the value `0x0001` to the `RMR` register:

```
writemmr RMR 0x0001
```

This command writes the value `0x3200` to the `MPTPR` register:

```
writemmr MPTPR 0x3200
```

writereg

Writes the supplied data to the specified register.

Syntax

`writereg` *`regName value`*

Parameters

`regName`

The name of the register to which to assign the supplied value.

value

The value to write to the specified register.

This value may be specified in hexidecimal (for example, `0xFFFFABCD`), octal (for example, `037777725715`), or decimal (for example, `4294945741`).

Example

This command writes the value `0x00001002` to the `MSR` register:

```
writereg MSR 0x00001002
```

writereg128

Writes the supplied 32-bit values to the specified TLB register.

NOTE This command is applicable only to Book E cores like the e500 or e200 variants.

Syntax

`writereg128` *`regName value1 value2 value3 value4`*

Arguments

regName

The name (or number) of the TLB register to which to assign the specified values.

TIP Valid TLB0 register names range from `L2MMU_TLB0` through `L2MMU_TLB255`, and TLB511 for e500v2.

TIP Valid TLB1 register names range from `L2MMU_CAM0` through `L2MMU_CAM15`.

value1, value2, value3, value4

The four 32-bit values that together make up the 128-bit value to assign to the specified TLB register.

Each value must be specified in hexidecimal (for example, `0xFFFFABCD`).

Example

This command writes the values `0xA1002`, `0xB1003`, `0xC1004`, and `0xD1005` to the `L2MMU_CAM0` TLB register:

```
writereg128 L2MMU_CAM0 0xA1002 0xB1003 0xC1004 0xD1005
```

writespr

Writes the specified value to the specified special-purpose register (SPR).

NOTE This command is similar to the `writereg SPRxxx` command, except that `writespr` lets you specify the SPR register to modify by number (in hexidecimal, octal, or decimal).

Syntax

```
writespr regNumber value
```

Arguments

regNumber

The number of the SPR register to which to assign the supplied value.

This value may be specified in hexidecimal (for example, `0x27E`), octal (for example, `01176`), or decimal (for example, `638`).

value

The value to write to the specified SPR register.

This value may be specified in hexidecimal (for example, `0xFFFFABCD`), octal (for example, `037777725715`), or decimal (for example, `4294945741`).

Example

This command writes the value `0x0220000` to SPR register 638:

```
writespr 638 0x02200000
```

writeupma

Writes the supplied RAM word to the specified offset of user-programmable machine (UPM) A’s RAM array.

Each offset in UPM A’s RAM array corresponds to a type of memory transaction.

The RAM word at a RAM array offset (and the words immediately following the first RAM word) are instructions that control the behavior of UPM A.

For more information about programming UPM A, refer to the Memory Controller section of the hardware manual for the Power Architecture processor you are using.

NOTE This command applies to just PQ1 MPC8xx type devices.

Syntax

`writeupma` *`offset ramWord`*

Arguments

`offset`

Offset into UPM A's RAM array at which to write the supplied RAM word.

This offset must fall within the range `0` through `0x3F` inclusive. Each offset is interpreted by UPM A as a particular memory transaction type.

For more information about UPM transaction types, refer to the UPM Transaction Type table in the Memory Controller section of the hardware manual for the Power Architecture processor you are using.

`ramWord`

The RAM word to assign to the specified offset of UPM A's RAM array.

Example

This command assigns the RAM word `0xAAAA1100` to the `0x18` position of UPM A's RAM array:

```
writeupmb 0x18 0xAAAA1100
```

writeupmb

Writes the supplied RAM word to the specified offset of user-programmable machine (UPM) B's RAM array.

Each offset in UPM B's RAM array corresponds to a type of memory transaction.

The RAM word at a RAM array offset (and the words immediately following the first RAM word) are instructions that control the behavior of UPM B.

For more information about programming UPM B, refer to the Memory Controller section of the hardware manual for the Power Architecture processor you are using.

NOTE This command applies to just PQ1 MPC8xx type devices.

Syntax

`writeupmb` *`offset ramWord`*

Arguments

`offset`

Offset into UPM B's RAM array at which to write the supplied RAM word.

This offset must fall within the range `0` through `0x3F` inclusive. Each offset is interpreted by UPM B as a particular memory transaction type.

For more information about UPM transaction types, refer to the UPM Transaction Type table in the Memory Controller section of the hardware manual for the Power Architecture processor you are using.

`ramWord`

The RAM word to assign to the specified offset of UPM B's RAM array.

Example

This command assigns the RAM word `0xffffcc24` to the `0x08` position of UPM B's RAM array:

```
writeupmb 0x08 0xffffcc24
```

C

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 target's memory map, which has already been established. The actual memory map is initialized either by a target-resident boot loader or by a target initialization file, as described in Target Initialization Files.

If necessary, you can have the CodeWarrior debugger execute a memory configuration file immediately before the debugger downloads a bare board 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 bare board 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.

The sections of this appendix are:

Using Memory Configuration Files

A memory configuration file is a command file that the CodeWarrior debugger executes each time the build target to which the configuration file is assigned is debugged.

To instruct the CodeWarrior debugger to use a memory configuration file, follow these steps:

1. Start the CodeWarrior IDE.
2. Open a bare board project.

3. Select one of this project's build targets.
4. Display the EPPC Debugger Settings target settings panel.
5. Check the Use Memory Configuration File box of this panel and then type the path and name of the configuration file you want to use in the related text box.

   Alternatively, click **Browse** to display a dialog box with which you can select the memory configuration file you want.

Your CodeWarrior product includes example memory configuration files for the supported target boards. These files are in this directory:

`installDir\PowerPC_EABI_Support\Initialization_Files\Memory\`

You can also write your own memory configuration files. The next section documents the commands that can appear in such files.

Memory Configuration File Commands

This section documents each command that can appear in a memory configuration file and defines the syntax rules that these commands follow.

Command Syntax

In general, the syntax of memory configuration file commands follows these rules:

- Spaces and tabs (white space) are ignored
- Character case is ignored
- Unless otherwise noted, values may be specified in hexidecimal, octal, or decimal:
  - hexidecimal values are preceded by `0x` (for example, `0xABCDFFFF`)
  - octal values are preceded by `0` (for example, `01234567`)
  - decimal values start with a non-zero numeric character (for example, `1234`)
- Comments start with standard C and C++ comment characters, and continue to the end of the line

Table of Commands

Table C.1 lists each command that can appear in a memory configuration file.

Table C.1 Target Initialization Commands

autoEnableAddressTranslations	range

Table C.1 Target Initialization Commands (*continued*)

reserved	reservedchar
translate	

Command Reference

This section documents each memory configuration command.

For each command, the section provides a brief statement of what the command does, the command's syntax, a definition of each argument that can be passed to the command, and examples showing how to use the command.

autoEnableAddressTranslations

The autoEnableAddressTranslations command enables the memory management unit (MMU) before the download of the binary to be debugged.

Syntax

`autoEnableAddressTranslations` *`enableFlag`*

Arguments

`enableFlag`

Pass true to instruct the debugger to enable the MMU before downloading the binary to be debugged; otherwise, pass false.

If this command is not present in a memory configuration file, the MMU is not enabled prior to the download of the executable to be debugged.

Examples

This command enables a processor's MMU before the debugger downloads the binary to be debugged:

```
AutoEnableTranslations true
```

range

The range command assigns the specified attributes to the specified range of memory locations. These attributes tell the CodeWarrior debugger how to treat the specified memory range.

The attributes the range command supports are access type (for example, read-only), access size (for example, 2 bytes per memory access), and whether the range consists of physical or virtual addresses.

Syntax

```
range loAddr hiAddr (accessSize | any) accessType
     [memSpaceType]
```

Arguments

loAddr

Defines the start address of the memory block.

hiAddr

Defines the end address of the memory block.

accessSize | any

Defines the size (in bytes) of the memory accesses that the debugger can perform on the specified memory block.

Pass the token `any` if the debugger is to perform dynamic virtual address translations.

accessType

Defines the type of access the debugger has to the specified memory block. Must be one of:

- `Read`
- `Write`
- `ReadWrite`

memSpaceType

Defines the type of the memory block. Must be one of:

- `Physical`

  This attribute tells the debugger that each address in the specified range is a physical memory address.

- `LogicalData`

  This attribute tells the debugger that each address in the specified range is a virtual address and that each address can be accessed as code *or* as data.

  Assign this attribute to memory ranges for which the MMU is configured so that there is a corresponding range of data addresses for the specified code address range. This is the typical MMU configuration.

- `LogicalCode`

  This attribute tells the debugger that each address in the specified range is a virtual address and that each address can be accessed as code *only*.

  Assign this attribute to memory ranges for which the MMU is configured so that there is *not* a corresponding data address range for the specified code address range.

The *`memSpaceType`* parameter is optional, and its default value is `Physical`. Therefore, if you pass no *`memSpaceType`* argument to a range command, the command defines a physical memory block.

Examples

This command makes the memory locations from `0xFF000000` through `0xFF0000FF` read-only, with an access size of 4 bytes:

```
range 0xFF000000 0xFF0000FF 4 Read
```

This command makes the memory locations from `0xFF000100` through `0xFF0001FF` write-only, with an access size of 2 bytes:

```
range 0xFF000100 0xFF0001FF 2 Write
```

This command makes the memory locations from `0xFF000200` through `0xFFFFFFFF` readable and writable, with an access size of 1 byte:

```
range 0xFF000200 0xFFFFFFFF 1 ReadWrite
```

This command instructs the debugger that addresses in the range `0x0` through `0x0FFFFFFC` are virtual addresses and to request that the probe translate addresses using the current TLB entries:

```
range 0x0 0x0FFFFFFC any ReadWrite LogicalData
```

reserved

The reserved command makes the specified range of memory locations inaccessible to the debugger.

If the debugger tries to read reserved memory, the debugger's buffer is filled with the reserved character. If the debugger attempts to write to reserved memory, no write occurs.

NOTE Refer to the reservedchar topic for instructions that explain how to set the reserved character.

Syntax

`reserved` *`loAddress hiAddress`*

Arguments

`loAddress`

The start address of the range of memory locations to reserve.

`hiAddress`

The end address of the range of memory locations to reserve.

Examples

This command reserves the memory locations from `0xFF000024` to `0xFF00002F`:

```
reserved 0xFF000024 0xFF00002F
```

reservedchar

This reservedchar command defines the character the debugger puts in its buffer when it the debugger attempts to read a reserved or invalid memory location.

Syntax

`reservedchar` *`rChar`*

Arguments

rChar

The character the debugger uses to fill its buffer when it attempts to read reserved or invalid memory.

Example

This command makes the character `'x'` the reserved character:

```
reservedchar 0x78
```

translate

This command lets you configure how the debugger performs virtual-to-physical memory address translations. Typically, you use address translations to debug programs that use a memory management unit (MMU) to perform block address translations.

Syntax

`translate` *`virtualAddress physicalAddress numBytes`*

Arguments

virtualAddress

The address of the first byte of the virtual address range to translate.

physicalAddress

The address of the first byte of the physical address range to which the debugger translates virtual addresses.

numBytes

The size (in bytes) of the address range to translate.

Example

This command below:

- Defines a one-megabyte address range (`0x100000` bytes is one megabyte).
- Instructs the debugger to convert a virtual address in the range `0xC0000000` to `0xC0100000` to the corresponding physical address in the range `0x00000000` to `0x00100000`.

```
translate 0xC0000000 0x00000000 0x100000
```

D

Using the Dhrystone Benchmark Software

Dhrystone is a general-performance benchmark test originally developed in 1984. This benchmark is used to measure and compare the performance of different computers or the efficiency of the code generated for the same computer by different compilers. The test reports general performance in Dhrystone-per-second.

Like most benchmark programs, Dhrystone consists of standard code and concentrates on string handling. It uses no floating-point operations. It is heavily influenced by hardware and software design, compiler and linker options, code optimization, cache memory, wait states, and integer data types.

This appendix explains how to use the Dhrystone benchmark example program included with your CodeWarrior product. This example works with a Freescale Lite5200 board. You can use the example as the basis for your own Dhrystone benchmark programs.

NOTE The Dhrystone benchmark software in not included in the Linux® Application Edition of this product.

The sections of this appendix are:

- Building the Dhrystone Example Project
- Running the Dhrystone Program

Building the Dhrystone Example Project

To build the Dhrystone example program, follow these steps:

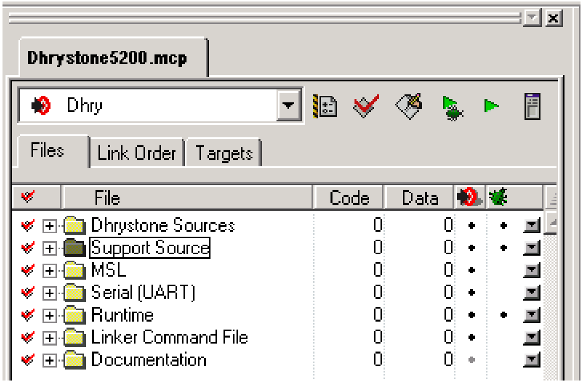
1. Start the CodeWarrior IDE.
2. Open the CodeWarrior project file named `Dhrystone5200.mcp`.

   This project file is here:

   `*installDir*\(CodeWarrior_Examples)\PowerPC_EABI\Dhrystone\`

   The Dhrystone project window appears. (See Figure D.1.)

Figure D.1 Dhrystone Example Project — Project Window



3. Select **Project > Make**.

   The IDE builds the project and generates an executable that you can run on a Freescale Lite5200 target board.

Running the Dhrystone Program

To run the Dhrystone example program on a Lite5200 board, follow these steps:

1. Connect your debug hardware to the Lite5200 and to your PC.

   For example, connect a USB TAP run-control tool to the JTAG port of the Lite5200 and to a USB port of your PC.

2. Start the CodeWarrior IDE.

3. Open the CodeWarrior project file named `Dhrystone5200.mcp`

   This project file is here:

   `installDir``\(CodeWarrior_Examples)\PowerPC_EABI\Dhrystone\`

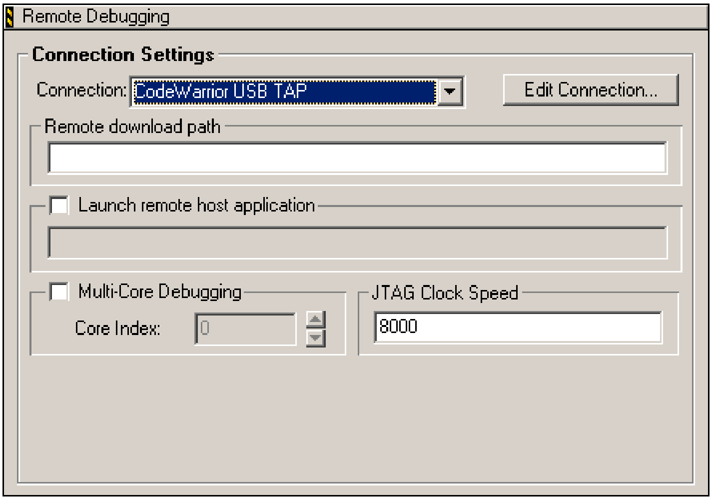
4. From the CodeWarrior menu bar, select **Edit >** ***TargetName*** **Settings**.

   The IDE displays the **Target Settings** window.

5. In the left pane of the **Target Settings** window, select **Remote Debugging**.

   The **Remote Debugging** target settings panel appears in the right side of the **Target Settings** window. (See Figure D.2.)

Figure D.2 The Remote Debugging Target Settings Panel



6. From the **Connection** dropdown menu, select the remote connection appropriate for your debug hardware.
7. Click **Edit Connection**

   The **Edit Connection** dialog box appears. Use this dialog box to configure your debug hardware.

   See Working with Remote Connections for a definition of each option for each available remote connection.
8. Click **OK**.

   The remote connection dialog box closes.
9. Click **OK**.

   The **Target Settings** window closes.
10. Connect a null modem serial cable between port COM1 of the Lite5200 and a free serial port of your PC.
11. Start a terminal emulation program and configure it as shown in Table D.1.

Table D.1 Terminal Emulator Configuration Settings

bits per second	57600
data bits	8
parity	none

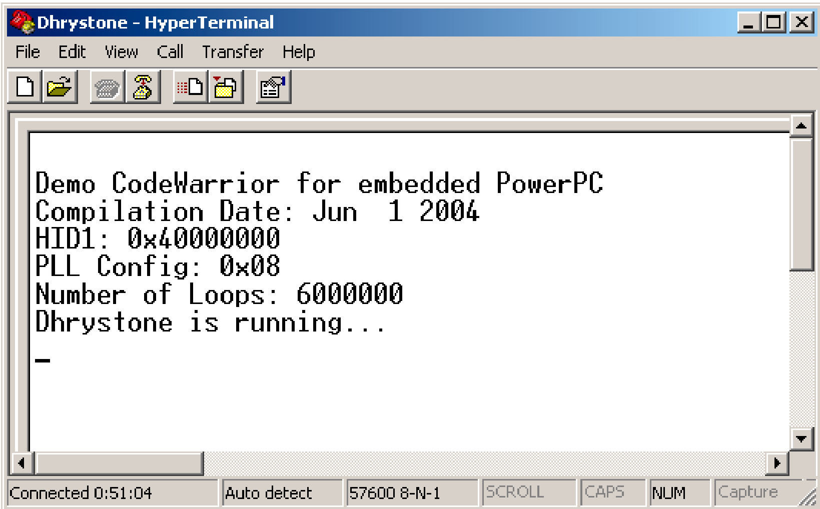
Table D.1 Terminal Emulator Configuration Settings (*continued*)

stop bits	1
hardware flow control	none
software flow control	none

12. From the menu bar of the IDE, select **Project > Run**.

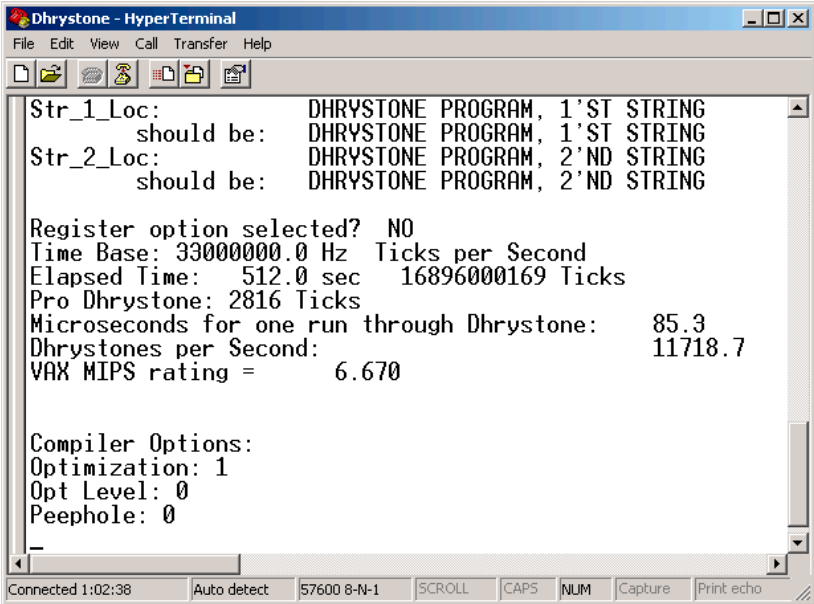
    The debugger downloads the example program to the Lite5200 board. The program writes the “start” information shown in Figure D.3 to the terminal emulator window and then executes 6,000,000 loops. (Depending on the speed of your board’s processor clock, this test can take up to 15 minutes to finish.)

Figure D.3 Terminal Emulator Showing Test “Start” Information



13. Upon completion, the Dhrystone example program displays the results of its tests in the terminal emulator window. (See Figure D.4.)

Figure D.4 Terminal Emulator Showing Test Results



That's it. If you want to write your own Dhrystone benchmark program, you can use this example program as a starting point.

E

Using the Linux-hosted Simulators

While working on a Windows-hosted e500/e600 project, you can configure a remote connection to communicate over the network with the simulator running on the Linux machine.

This appendix explains how to use the Linux-hosted simulators for a Windows-hosted e500/e600 project. The sections of this appendix are:

Creating and Configuring a Windows-hosted e500/e600 Simulator Project

To create and configure an e500/e600 project for remote connectivity, follow these steps:

1. Start the CodeWarrior IDE.
2. Create a new e500/e600 project with **EPPC New Project** Wizard.
3. From the **Linkers** list box, select `Freescale PowerPC EABI Linker.`
4. Click **Next**.
5. From the Target Page:
   - For an e500 simulator project:
     - Click the **85xx** tab.
     - Select any of the 85xx processors from the left box.
     - Select **e500v2_ISS** from the right box**.**
   - For an e600 simulator project:
     - Click the **86xx** tab.
     - Select the **8641** processor from the left box.

NOTE The e600 simulator only supports 8641 boards.

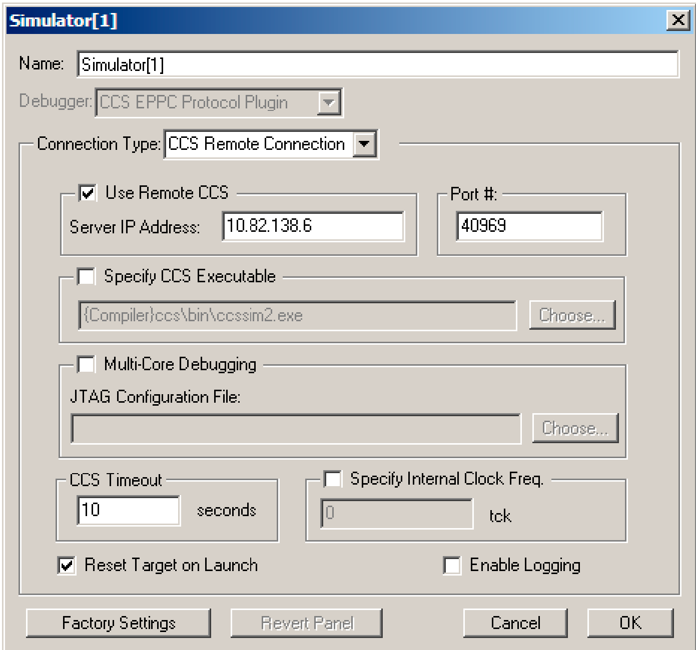
- Select **e600_ISS** from the right box.

6. Click **Next**.
7. In Programming Language page, select **C.**
8. Click **Next**.
9. In Remote Connection page, select the **Simulator1** connection.

After creating the project, you must configure it for remote communication with a Linux-hosted simulator. Follow the steps below to configure the project for remote debugging:

1. Open the **Remote Debugging** panel from the **Debug Version Settings** window.
2. Make sure that the **Simulator1** is selected in the **Connection** box.
3. Click **Edit Connection** button.
4. Check the **Use Remote CSS** checkbox.
5. Enter the IP address of your Linux machine in the **Server IP Address** text box. (See Figure E.1).

Figure E.1 Edit Connection Window



6. Click **OK**.

Configuring the Linux Machine

To run the Windows-hosted e500/e600 project on the Linux-hosted simulator, perform the following steps:

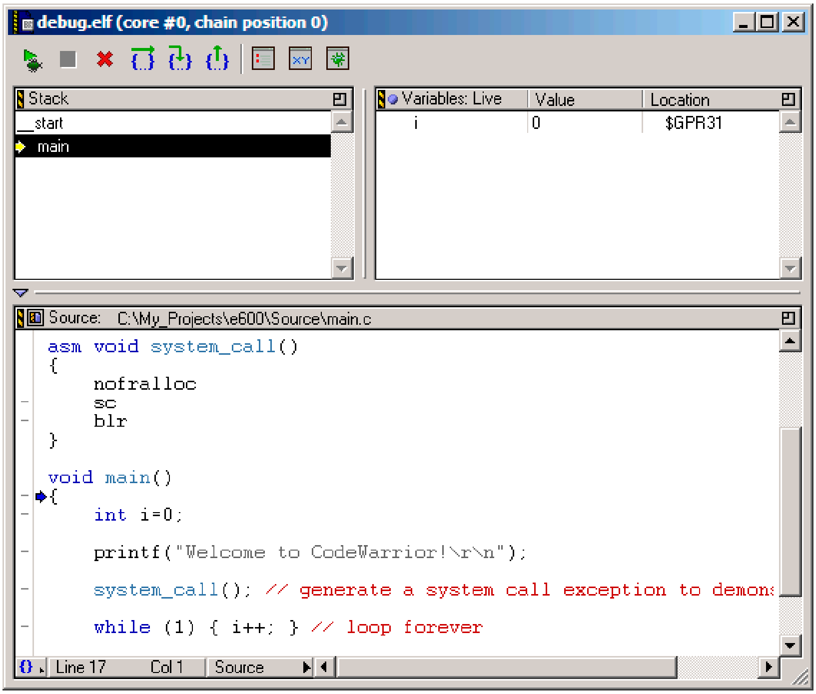
1. Copy the Linux Simulator files from the `*installDir*\ccs\bin\Linux_simulators` folder to your Linux machine. These files include the `ccssim2` file along with the library files for the e500/e600 simulators.
2. Copy the `SimRun` Linux script file from `*installDir*\ccs\bin` along with the `ccssim2` file on your Linux machine.
3. Edit the `SimRun` Linux script file and replace `BASE=/usr/local/Freescale/` with the correct path of the simulator files on your Linux machine.
4. Run the `SimRun` Linux script file to start the simulator.

Debugging the Project

Follow these steps to debug the project:

1. Open the e500/e600 project.
2. Select **Project > Debug**. Figure E.2 appears:

Figure E.2 Debug Window



Index

N

O

P

R

S

T

U

V

W