Freescale Semiconductor, Inc.

CodeWarrior™ Development Studio

mobileGT™ Processor Edition, Version 8.1

Targeting Manual

Revised 2004/06/23

metrowerks

For More Information: www.freescale.com

Metrowerks and the Metrowerks logo are registered trademarks of Metrowerks Corporation in the United States and/or other countries. CodeWarrior is a trademark or registered trademark of Metrowerks Corporation in the United States and/or other countries. All other trade names and trademarks are the property of their respective owners.

Copyright © 1999-2003 Metrowerks Corporation. ALL RIGHTS RESERVED.

No portion of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, without prior written permission from Metrowerks. Use of this document and related materials is governed by the license agreement that accompanied the product to which this manual pertains. This document may be printed for non-commercial personal use only in accordance with the aforementioned license agreement. If you do not have a copy of the license agreement, contact your Metrowerks representative or call 1-800-377-5416 (if outside the U.S., call +1-512-996-5300).

Metrowerks reserves the right to make changes to any product described or referred to in this document without further notice. Metrowerks makes no warranty, representation or guarantee regarding the merchantability or fitness of its products for any particular purpose, nor does Metrowerks assume any liability arising out of the application or use of any product described herein and specifically disclaims any and all liability. **Metrowerks software is not authorized for and has not been designed, tested, manufactured, or intended for use in developing applications where the failure, malfunction, or any inaccuracy of the application carries a risk of death, serious bodily injury, or damage to tangible property, including, but not limited to, use in factory control systems, medical devices or facilities, nuclear facilities, aircraft navigation or communication, emergency systems, or other applications with a similar degree of potential hazard**.

How to Contact Metrowerks

Corporate Headquarters	Metrowerks Corporation 7700 West Parmer Lane Austin, TX 78729 U.S.A.
World Wide Web	`http://www.metrowerks.com`
Sales	United States Voice: 800-377-5416 United States Fax: 512-996-4910 International Voice: +1-512-996-5300 Email: `sales@metrowerks.com`
Technical Support	United States Voice: 800-377-5416 International Voice: +1-512-996-5300 Email: `support@metrowerks.com`

Table of Contents

1 Introduction 9

Read the Release Notes. 9

Related Documentation. 9

CodeWarrior™ Information 10

Embedded PowerPC Programming Information 10

PowerTAP® Pro and WireTAP Information 11

2 Getting Started 13

System Requirements 13

Supported Evaluation Boards 14

Installing Your CodeWarrior™ Product 14

Registering Your CodeWarrior™ Product 15

Overview: The CodeWarrior™ Software Development Tools 17

CodeWarrior™ IDE 17

Project Manager 18

C/C++ Compiler 20

Standalone Assembler 20

Linker 21

Debugger 21

Metrowerks Standard Libraries 22

Overview: The CodeWarrior™ Software Development Process 22

Project Files 23

Editing Code. 23

Compiling. 23

Linking 24

Debugging 24

Viewing Preprocessor Output 24

3 Tutorial 25

Creating a Project 25

Building and Debugging a Project 31

For More Information: www.freescale.com

Freescale Semiconductor, Inc.

4 Creating a CodeWarrior™ Project 35

Types of Projects 35
Using PowerPC EABI Templates 36
Using the Makefile Importer Wizard 36
Project Targets 38

5 Target Settings 39

Target Settings Overview 39
Embedded PowerPC-Specific Target Settings Panels 40
Target Settings 40
EPPC Target 44
EPPC Assembler 50
Global Optimizations 52
EPPC Processor 54
EPPC Disassembler 63
EPPC Linker. 65
Debugger PIC Settings 73
EPPC Debugger Settings 74
Source Folder Mapping 76
System Call Service Settings 79
PC-lint Target Settings Panels 81
PCLint Main Settings 82
PCLint Options. 84

6 Embedded PowerPC Debugging 87

Supported Remote Connections 87
Abatron Remote Connections 89
MetroTRK 91
MSI BDM Raven/MSI Wiggler 92
P&E BDM 94
PowerTAP PRO CCS 95
WireTAP 8xx 97
WireTAP CCS 98

For More Information: www.freescale.com

Special Debugger Features 100
Displaying Registers 100
EPPC-Specific Debugger Features 101
Register Details 106
Using MetroTRK 108
MetroTRK Overview 108
Connecting to the MetroTRK Debug Monitor. 109
MetroTRK Memory Configuration 110
Using MetroTRK for Debugging 111
Using MetroTRK with the Lite5200 Board 112
Debugging ELF Files 120
Preparing to Debug an ELF File 120
Customizing the Default XML Project File. 122
Debugging an ELF File 123
ELF File Debugging: Additional Considerations. 124

7 C/C++ Compiler and Linker 125

Integer and Floating-Point Formats 126
Embedded PowerPC Integer Formats 126
Embedded PowerPC Floating-Point Formats 127
Data Addressing 127
Register Variables 129
Register Coloring Optimization 130
Pragmas. 131
opt_full_unroll_limit 132
opt_findoptimalunrollfactor 132
opt_unroll_count 133
opt_unrollpostloop 133
opt_unroll_instr_count 133
inline_max_auto_size 133
ppc_no_fp_blockmove 133
force_active 134
function_align 134

For More Information: www.freescale.com

Freescale Semiconductor, Inc.

incompatible_return_small_structs 134
incompatible_sfpe_double_params 135
interrupt 135
pack 136
pooled_data 137
section 137
EPPC Linker Issues 143
Additional Small Data Sections 143
Linker Generated Symbols 147
Deadstripping Unused Code and Data 148
Link Order 148
Linker Command Files. 149
Using __attribute__ ((aligned(?))) 159
Variable Declaration Examples 159
Struct Definition Examples 160
Typedef Declaration Examples 160
Struct Member Examples. 161

8 Inline Assembler 163

Working With Assembly Language 163
Assembly Language Syntax. 164
Special Embedded PowerPC Instructions 166
Creating Statement Labels 167
Using Comments 168
Using the Preprocessor in Embedded PowerPC Assembly 168
Using Local Variables and Arguments. 168
Creating a Stack Frame 169
Specifying Operands 170
Assembler Directives 174
entry 174
fralloc 175
frfree. 175
machine 175

For More Information: www.freescale.com

nofralloc 176
opword 176
Intrinsic Functions 176
Low-Level Processor Synchronization. 177
Absolute Value Functions 177
Byte-Reversing Functions 178
Setting the Floating-Point Environment 178
Manipulating a Variable or Register. 178
Data Cache Manipulation. 179
Math Functions. 179
Buffer Manipulation. 180

9 Support Libraries and Code 181

Metrowerks Standard Libraries. 181
Using the Metrowerks Standard Libraries 181
Using Console I/O 182
Allocating Memory and Heaps 183
Runtime Libraries 184
Library Naming Conventions 184
Required Libraries and Source Code Files 185
Board Initialization Code 186

10 Hardware Tools 187

Flash Programmer. 187
Hardware Diagnostics 189
Logic Analyzer 189
Logic Analyzer Menu 190
Logic Analyzer Tutorial 191

A Debug Initialization Files 199

Using Debug Initialization Files 199
Debug Initialization File Commands 200
Debug Initialization File Command Syntax. 200
Descriptions and Examples of Commands 200

B Memory Configuration Files 205

Command Syntax 205
Memory Configuration File Commands 205
range 206
reserved 206
reservedchar 206

C Command-Line Tool Options 207

Compiler/Linker Options 207
Disassembler Options 211

D Using the Dhrystone Benchmark Software with the Lite5200 215

Building the Dhrystone Example Project 215
Running the Dhrystone Program 216

Index 221

1 Introduction

This manual explains how to install and use the CodeWarrior™ Development Studio, mobileGT™ Processor Edition product.

The sections of this chapter are:

- Read the Release Notes
- Related Documentation

Read the Release Notes

The release notes contain information about new features, bug fixes, and incompatibilities that is not in the documentation due to release deadlines.

The release notes are in this directory:

`installDir\Release_Notes`

where *installDir* is a placeholder for the path in which you installed your CodeWarrior product.

Related Documentation

This section lists documentation related to the CodeWarrior IDE and Embedded PowerPC development.

- CodeWarrior™ Information
- Embedded PowerPC Programming Information
- PowerTAP® Pro and WireTAP Information

CodeWarrior™ Information

- Your CodeWarrior product includes example projects that show you how to do such things as use MetroTRK and use the Dhrystone benchmark software.

  The example projects are in subdirectories of this directory:

  `installDir\(CodeWarrior_Examples)`
- For general information about the CodeWarrior IDE and the CodeWarrior debugger, refer to the *IDE User Guide*. This manual is here:

  `installDir\Help\PDF`
- For information specific to the CodeWarrior C/C++ compiler, see the *C Compilers Reference*. This manual is here:

  `installDir\Help\PDF`
- For information about Metrowerks' standard C/C++ libraries, see the *MSL C Reference* and the *MSL C++ Reference* in the `installDir\Help\PDF` directory. These manuals are here:

  `installDir\Help\PDF`
- For general information about MetroTRK and instructions that explain how to customize MetroTRK to work with any target board, see the *MetroTRK Reference*. This manual is here:

  `installDir\Help\PDF`

Embedded PowerPC Programming Information

To learn more about the Embedded PowerPC Application Binary Interface (PowerPC EABI), refer to these documents:

- *System V Application Binary Interface, Third Edition*, published by UNIX System Laboratories, 1994 (ISBN 0-13-100439-5).
- *System V Application Binary Interface, PowerPC Processor Supplement*, published by Sun Microsystems and IBM (1995). This document is available at this web address:

  `http://www.cloudcaptech.com/downloads.htm`
- *PowerPC Embedded Binary Interface, 32-Bit Implementation*, published by Motorola, Inc. This document is available at this web address:

  `http://e-www.motorola.com/`
  `files/32bit/doc/app_note/PPCEABI.pdf`

The PowerPC EABI specifies data structure alignment, calling conventions, and other information about how high-level languages can be implemented for a Embedded

PowerPC processor. The code generated by the CodeWarrior C/C++ compiler and the CodeWarrior Assembler conforms to the PowerPC EABI.

The PowerPC EABI also specifies the required object and symbol file format. The specification requires ELF (Executable and Linker Format) for the output file format and DWARF (Debug With Arbitrary Record Format) for the symbol file format. For more information about these file formats, refer to these documents:

- *Executable and Linker Format, Version 1.1*, published by UNIX System Laboratories.
- *DWARF Debugging Information Format, Revision: Version 1.1.0*, published by UNIX International, Programming Languages SIG, October 6, 1992.

  This document is available at this web address:

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

  This document is available at this web address:

  `http://www.nondot.org/sabre/os/files/Executables/Dwarf.pdf`

PowerTAP® Pro and WireTAP Information

The manuals listed below provide information about the PowerTAP PRO and WireTAP debug tools.

- Emulator installation guide for PowerTAP PRO JTAG

  installDir\Help\PDF\
  PowerTAP_PRO_ICE_JTAG_Emulator_Guide.pdf
- Quick Start guide for PowerTAP PRO JTAG

  installDir\PowerTAP_PRO_ICE_JTAG_QuickStart.pdf
- Emulator installation guide for PowerTAP PRO DPI

  installDir\Help\PDF\
  PowerTAP_PRO_ICE_DPI_Emulator_Guide_cd.pdf
- Quick Start guide for PowerTAP PRO DPI

  installDir\PowerTAP_PRO_ICE_DPI_Quick_Start.pdf
- Emulator installation guide for WireTAP

  installDir\Help\PDF\
  Wiretap_Installation_Guide.pdf

For More Information: www.freescale.com

2

Getting Started

This chapter shows you how to install the CodeWarrior™ Development Studio, mobileGT™ Processor Edition tools. In addition, the chapter provides an overview of the tools included in this CodeWarrior product and the development process you follow as you use these tools.

The sections are:

- System Requirements
- Supported Evaluation Boards
- Installing Your CodeWarrior™ Product
- Registering Your CodeWarrior™ Product
- Overview: The CodeWarrior™ Software Development Tools
- Overview: The CodeWarrior™ Software Development Process

System Requirements

The system requirements for the CodeWarrior™ Development Studio, mobileGT Edition product are:

- Hardware:
  - PC with a 500 MHz Intel® Pentium® class processor (minimum)
  - 128 MB RAM (minimum)
  - 230 MB free hard disk space (minimum)
  - CD-ROM drive
  - Serial port, parallel port, and Ethernet port.
- Software: Microsoft® Windows 2000/XP® or Windows NT® Workstation 4.0.

Supported Evaluation Boards

Table 2.1 lists the evaluation boards supported by the mobileGT Processor Edition.

Table 2.1 Supported Evaluation Boards

Manufacturer	Boards
Motorola	Lite5200, rev. I
	Lite5200, rev. G
	823 FADS

Installing Your CodeWarrior™ Product

To install your CodeWarrior product, follow these steps:

1. Put the product installation CD in the CD drive.

   The CodeWarrior installation menu appears.

NOTE If auto-install is disabled, run `Launch.exe` manually. This program is in the root directory of the installation CD.

2. Click **Launch the installer**

   The installation wizard starts and displays a welcome screen.

3. Follow the on-screen instructions to install the software.
4. When prompted to check for CodeWarrior software updates, click **Yes**

   The **CodeWarrior Updater** window appears.

NOTE If the CodeWarrior updater already has the correct Internet connection settings, proceed directly to step 8.

5. Click **Settings**

   The **Internet Properties** dialog box appears.

6. Use this dialog box to modify your Internet connection settings (if necessary).

7. Click **OK**

   The **Internet Properties** dialog box closes.

8. In the **CodeWarrior Updater** window, click **Next**

   The updater checks for newer versions of the CodeWarrior products installed on your PC.

9. Follow the on-screen instructions to download CodeWarrior product updates to your PC.

10. When the updater displays the message *Update Check Complete!*, click **Finish**.

    The wizard displays a readme file in Notepad.

11. When finished with the readme file, press **ALT-F4**

    Notepad exits, and the wizard displays its "installation complete" page.

12. Select **Yes, I want to restart my computer now**, and click **Finish**

    Your PC restarts; CodeWarrior installation is complete.

Registering Your CodeWarrior™ Product

To register your CodeWarrior product, follow these steps:

1. Select **Start > Programs > Metrowerks CodeWarrior > CodeWarrior for mobileGT V8.1 > CodeWarrior IDE**

   The IDE starts and displays the registration dialog box. (See Figure 2.1.)

Figure 2.1 Registration Dialog Box

2. Click **Register Now**

   The registration dialog box closes. Your web browser starts and displays the Metrowerks registration web page.

3. Complete the registration form on the registration web page.

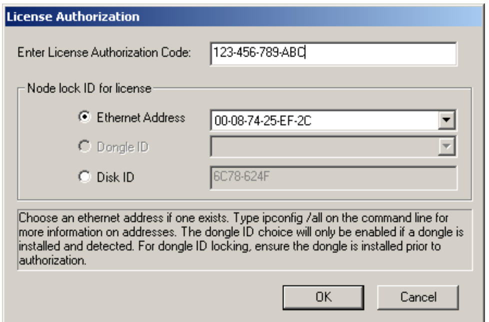
   Metrowerks emails a license authorization code to you (typically within 15 minutes).

4. Open the email message containing the license registration code.

5. From the IDE's menu bar, select **Help > License Authorization**

   The **License Authorization** dialog box appears. (See Figure 2.2.)

Figure 2.2 License Authorization Dialog Box



6. Copy and paste the license authorization code from the email message to the Enter License Authorization Code text box of the **License Authorization** dialog box.

7. Click **OK**

   The **License Authorization** dialog box closes; Your CodeWarrior software is ready to use.

Overview: The CodeWarrior™ Software Development Tools

Programming for mobileGT processor family is much like programming for any other CodeWarrior target. If you have never used the CodeWarrior IDE before, the tools with which you must become familiar are:

- CodeWarrior™ IDE
- Project Manager
- C/C++ Compiler
- Standalone Assembler
- Linker
- Debugger
- Metrowerks Standard Libraries

NOTE If you have used other CodeWarrior products, you already know how to use the IDE and the debugger because they are the same for all CodeWarrior versions.
However, the linker and the Metrowerks Standard Libraries (MSL) differ significantly from one CodeWarrior product to another; as a result, you must familiarize yourself with the mobileGT versions of these tools before you start to write software.

CodeWarrior™ IDE

The CodeWarrior IDE (Integrated Development Environment) is a program that provides a set of tools that you can use to develop software for the mobileGT processor family.

The IDE has a graphical user interface (GUI). You use the GUI to control the tools included in the mobileGT processor family product.

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

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

For More Information: www.freescale.com

Project Manager

A CodeWarrior project is a collection of build targets. A build target is a set of related files and configuration settings that the CodeWarrior IDE uses to generate a final output file, such as an application or a library.

The project manager displays a project in a window called the project window. This window displays the files in each build target in a project. (See Figure 2.3.)

NOTE Those who are more familiar with command-line development tools than an IDE may find the project manager a new concept. The project manager organizes all files related to your project. This lets you to see your project at a glance and eases the organization of and navigation between your source files.

Figure 2.3 Example Project Window

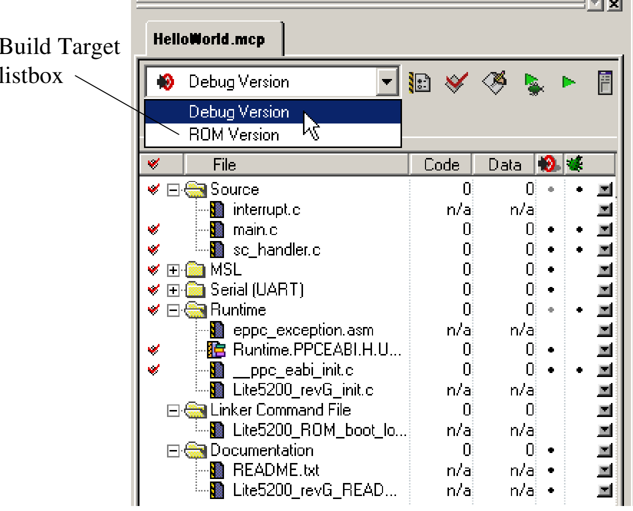


Table 2.2 defines several project-related terms.

Table 2.2 Project-related Terms

Term	Definition
Host	The system on which you run the CodeWarrior IDE to develop software for one or more platform targets.
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 lets you define different builds for different purposes.
Platform target	The operating system or evaluation board for which you develop software. The platform target can be different from the host.

The project manager keeps track of dependencies between files in your project. As a result, when 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.

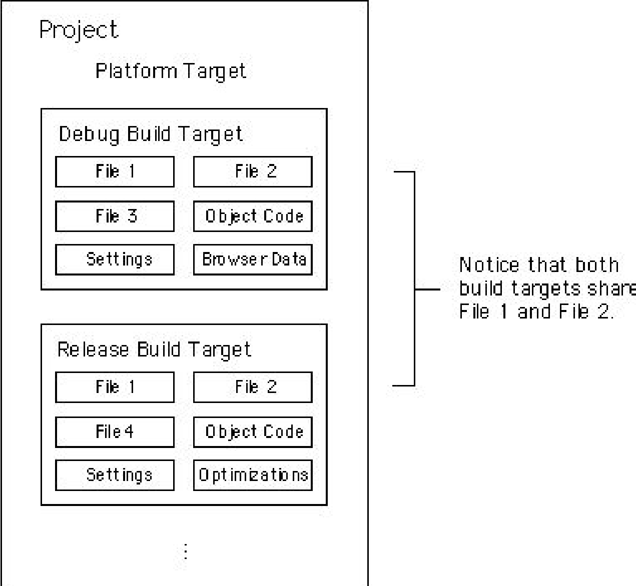
The project manager lets you define one or more build targets for the same project. A build target is a named set of project settings and files that the IDE uses to build a final output file.

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.

Figure 2.4 shows a project containing a debug build target and a release build target.

Figure 2.4 A Project with Multiple Build Targets



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

C/C++ Compiler

The CodeWarrior EPPC compiler is an ANSI-compliant C/C++ compiler. This compiler employs the same architecture as all other all other CodeWarrior C/C++ compilers. You can generate Embedded PowerPC applications and libraries that conform to the PowerPC EABI by using the CodeWarrior C/C++ compiler in conjunction with the CodeWarrior EPPC linker.

For information about the CodeWarrior compiler family's C/C++ language implementation, refer to the *C Compilers Reference*.

Standalone Assembler

The CodeWarrior EPPC assembler is a standalone, command-line assembler that provides an easy-to-use assembly language syntax. The CodeWarrior assemblers for other platform targets use the same syntax as the EPPC assembler.

For More Information: www.freescale.com

For more information about the standalone CodeWarrior assembler, refer to the *Assembler Guide*.

Linker

The CodeWarrior EPPC linker generates output in Executable and Linkable (ELF) format. Among other features, the linker, lets you:

- Assign absolute addresses to objects using linker command file directives.
- Define multiple user-defined sections.
- Generate S-Record files.
- Define additional small data sections.
- Use Position Independent Code/Position Independent Data (PIC/PID).

  For more information about PIC/PID support, refer to this release notice:

  `installDir\Release_Notes\PowerPC_EABI\`
  `CW_Tools\Compiler_Notes\CW Common PPC Notes 3.0.x.txt`

  where *`installDir`* is a placeholder for the path in which you installed your CodeWarrior product.

Debugger

The CodeWarrior debugger controls the execution of your program and allows you to see what is happening internally as your 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 *IDE User Guide*.

The CodeWarrior debugger for EPPC debugs software as it is running on the target board. The debugger communicates with the target board through a monitor program, such as MetroTRK, or through a hardware protocol, such as BDM or JTAG.

Hardware protocols require additional hardware to communicate with the target board, such as Abatron, PowerTAP Pro, WireTAP, P&E BDM, or an MSI Wiggler.

For More Information: www.freescale.com

Metrowerks Standard Libraries

The Metrowerks Standard Libraries (MSL) are ANSI compliant standard C and C++ libraries. These libraries are used to develop applications for Embedded PowerPC. The CodeWarrior CD contains the source code of these libraries. These are the same libraries that are used for all CodeWarrior build targets. However, the libraries have been customized and the runtime has been adapted for use in Embedded PowerPC development.

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

Overview: The CodeWarrior™ Software Development Process

While working with the CodeWarrior IDE, you will proceed through the development stages familiar to all programmers: writing code, compiling and linking, and debugging. See the *IDE User 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
- Editing Code
- Compiling
- Linking
- Debugging
- Viewing Preprocessor Output

Project Files

The CodeWarrior IDE is analogous to a set of makefiles because you can have multiple build targets in a single project. For example, you can have one project that has both a debug version and a release version of your project. 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 source code.

Editing Code

The CodeWarrior IDE has an integrated 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 file name in the project window to open the file.

The editor window has excellent navigational features that let you switch between related files, find a specific function, mark any location within a file, go to a specific line of code, and much more.

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**.

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**.

For More Information: www.freescale.com

Linking

Select **Project > Make** 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** to debug your project. This tells the compiler and linker to generate debugging information for all items in your project.

If you want to only generate debug information on a file-by-file basis, click in the debug column for that file. The debug column is located in the project window, to the right of the data column.

Viewing Preprocessor Output

To view preprocessor output, select the file in the project window and select **Project > Preprocess.** A new window appears that shows you what your file looks like after going through the preprocessor.

You can use this feature to track down bugs caused by macro expansion or other subtleties of the preprocessor.

The preprocessor feature is also useful for submitting bug reports for compiler problems. Instead of sending an entire source tree to technical support, you can preprocess the file causing problems and send it along with the relevant project settings through e-mail.

For More Information: www.freescale.com

Freescale Semiconductor, Inc.

3 Tutorial

This chapter presents a tutorial that shows you how to use the CodeWarrior™ Development Studio, mobileGT™ Processor Edition software development tools.

The tutorial does not teach you have to program; instead, it shows you how to use the CodeWarrior tools to create, build, and debug a project for the mobileGT™ processor family.

The sections are:

- Creating a Project
- Building and Debugging a Project

Creating a Project

You can create a mobileGT project using any of these tools:

- EPPC New Project Wizard
- PowerPC Embedded Application Binary Interface (EABI) templates
- Makefile Importer Wizard

See the Creating a CodeWarrior™ Project chapter to learn how to create a project using the Makefile Importer Wizard and PowerPC EABI templates.

This section explains how to create a mobileGT processor project using the EPPC New Project Wizard.

NOTE	You can change all choices you make while creating a project with the EPPC New Project Wizard manually after the project creation.

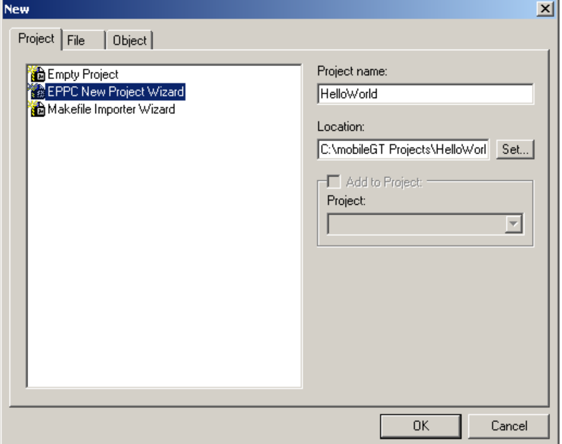
1. Create the project.

    a. Chose **File > New**

        The **New** dialog box appears. (See Figure 3.1.)

For More Information: www.freescale.com

Figure 3.1 New Dialog Box



b. From left pane of **New** dialog box, select `EPPC New Project Wizard`.

c. In the Project Name text box, type the name of your project.

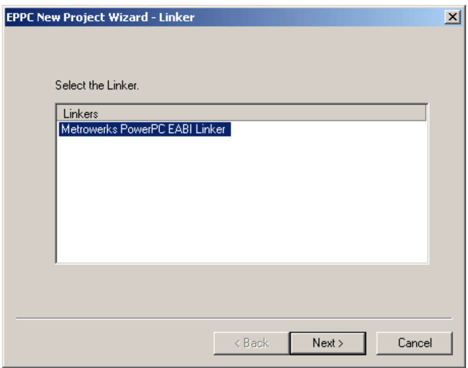
For example, type `HelloWorld`.

d. In the Location text box, type the directory in which to create the project.

e. Click **OK**

The wizard starts and displays the **EPPC New Project Wizard — Linker** dialog box. (See Figure 3.3.)

Figure 3.2 EPPC New Project Wizard — Linker Dialog Box

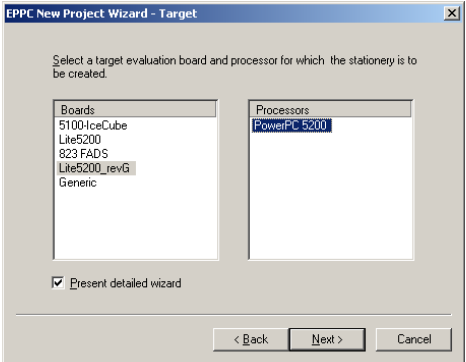


f. From the Linkers list, select the linker for the new project.

g. Click **Next**

The wizard displays the **EPPC New Project Wizard — Target** dialog box. (See Figure 3.3.)

Figure 3.3 EPPC New Project Wizard — Target Dialog Box



2. Select the target board and processor for the new project.

   a. From the Boards listbox, select the evaluation board your are using.

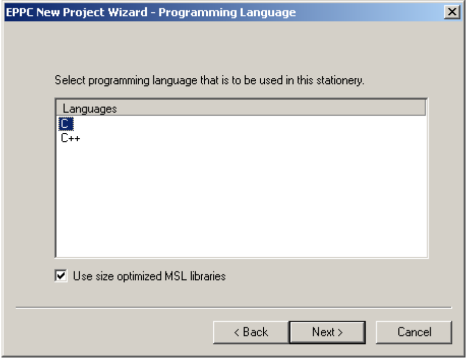
   b. From the Processors listbox, select the processor on the board you are using.

   c. Check the Present detailed wizard checkbox.

   d. Click **Next**

   The **EPPC New Project Wizard — Programming Language** dialog box appears. (See Figure 3.4.)

Figure 3.4 EPPC New Project Wizard — Programming Language Dialog Box



Freescale Semiconductor, Inc.

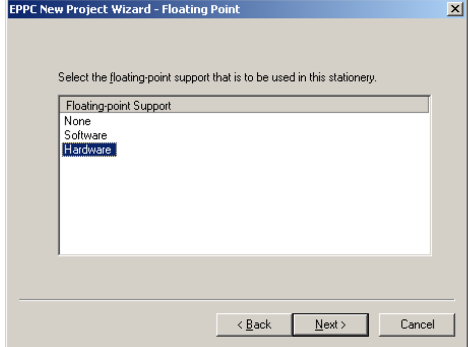
3. Select the programming language for the new project.

NOTE	The programming language selected defines the libraries the wizard includes in your project and the language used in the main source code file. That said, if you select C++, you can still include C language source files in your project and vice versa.

4. Optionally, check the Use size optimized libraries checkbox to use a compact version of the runtime support library in the new project
5. Click **Next**

   The wizard displays the **EPPC New Project Wizard — Floating Point** dialog box. (See Figure 3.5.)

Figure 3.5 EPPC New Project Wizard — Floating Point Dialog Box

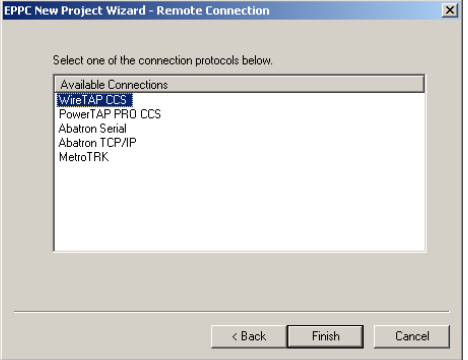


6. Select the floating-point support the new project requires.
7. Click **Next**

   The wizard displays the **EPPC New Project Wizard — Remote Connection** dialog box. (See Figure 3.6.)

For More Information: www.freescale.com

Figure 3.6 EPPC New Project Wizard — Remote Connection Dialog Box



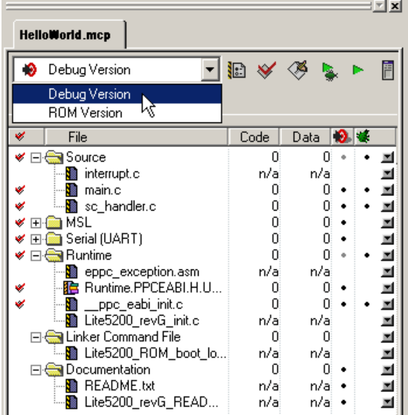
8. Select the remote connection for the debug interface you are using.

9. Click **Finish**

   The wizard creates your project according to your specifications and displays the results in a project window.

   The project window is docked to the left side of the IDE. (See Figure 3.7.)

Figure 3.7 Project Window—HelloWorld Project



10. Specify your remote connection preferences.

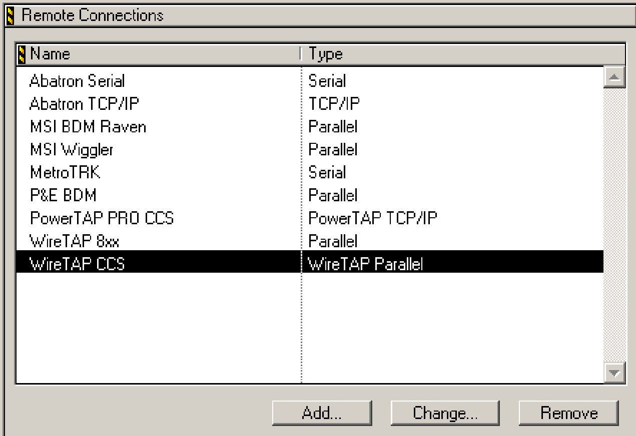
    a. Select **Edit > Preferences**

       The **IDE Preferences** window appears.

b. From the IDE Preference Panel listbox, select `Remote Connections`.

The **Remote Connections** preference panel appears on the right side of the IDE Preferences window. (See **Figure 3.8**.)

Figure 3.8 Remote Connections Preference Panel



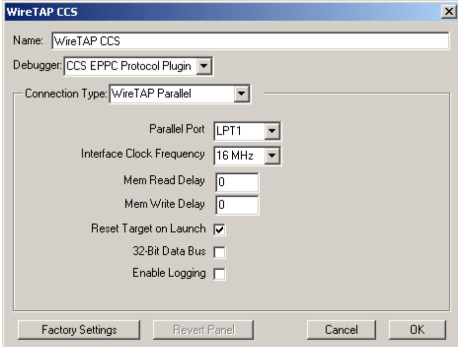
c. From the **Remote Connections** panel, select the remote connection name you selected when you created the project using the wizard.

For this tutorial, select `WireTAP CCS`.

d. Click **Change**

The **WireTAP CCS** dialog box appears. (See **Figure 3.9**.)

Figure 3.9 WireTAP CCS Dialog Box



e. From the Parallel Port dropdown list, select the parallel port of your PC to which the WireTAP CCS run-control tool is connected.

f. Leave the default values for all other settings.

g. Click **OK**

The IDE saves your settings and closes the **WireTAP CCS** dialog box.

h. Click **OK**

The IDE close the **IDE Preferences** window.

Building and Debugging a Project

This section explains how to edit source code then build and debug a project.

1. Edit the source code.

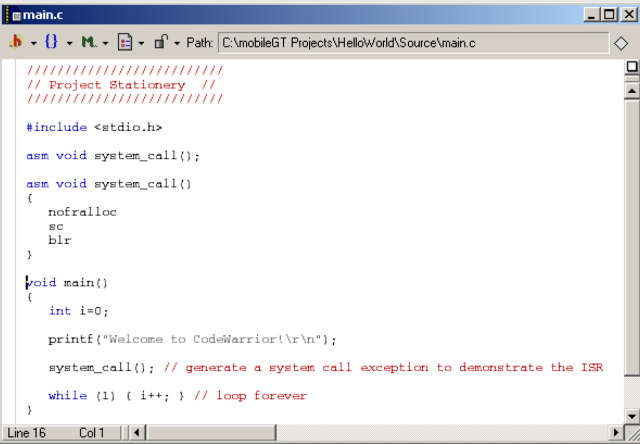
   a. In the project window, expand the Source control tree.

   b. Double-click the file named `main.c`.

   An editor window containing the contents of `main.c` appears. (See Figure 3.10.)

   You can edit `main.c` by typing in this window.

Figure 3.10 main.c



c. Press **CTRL-F4**

The editor window closes.

2. Select **Project > Make**

The IDE builds the project and stores the generated executable file in the project directory. Note that the red check marks next to the filenames disappear. This indicates that the files no longer need to be built.

NOTE Before starting to debug, ensure that your debug hardware is connected to the target board. Also ensure that a serial cable appropriate for the target board is connected between the COM A port of the board and the serial port of your PC.

3. Start a terminal emulation program.

a. Select **Start > Programs > Accessories > Communications > HyperTerminal**

The **Connection Description** dialog box appears.

b. In the Name text box, type a connection name.

c. Click **OK**

The **Connection Description** dialog box closes. The **Connect To** dialog box appears.

d. From the Connect using listbox, select the serial port of your PC to which the target board is connected.

e. Click **OK**

The **Connect To** dialog box closes. The **COM1 Properties** dialog box appears.

f. Specify these serial port settings:

- Bits per second — `57600`
- Data bits — `8`
- Parity — `None`
- Stop bits — `1`
- Flow control — `None`

g. Click **OK**

The **COM1 Properties** dialog box closes.

h. Select **File > Properties**

The **Properties** dialog box appears.

i. Click the **Settings** tab of the **Properties** dialog box.

The **Settings** page of the **Properties** dialog box appears.

j. Click **ASCII Setup**

The **ASCII Setup** dialog box appears.

k. Check the Send line ends with line feeds checkbox.

l. Click **OK**

The **ASCII Setup** dialog box closes.

m. Click **OK**

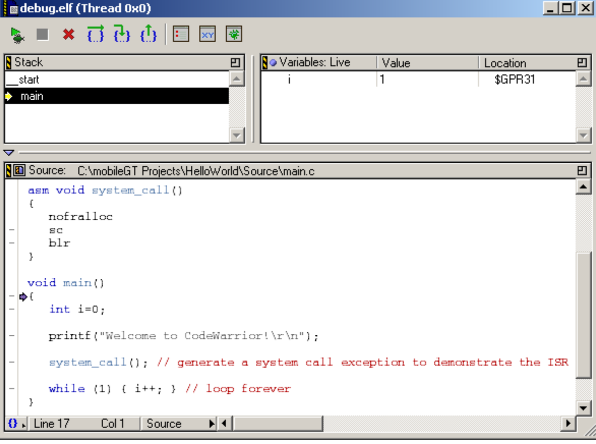
The **Properties** dialog box closes. Terminal emulation begins.

4. Debug the project.

a. Select **Project > Debug**

The IDE launches the debugger. The debugger window appears with the program counter at the `main` function. (See Figure 3.11)

Figure 3.11 Debugger Window



Freescale Semiconductor, Inc.

b. Set a breakpoint by clicking in the gutter to the left of this statement:
`printf("Welcome to CodeWarrior!\r\n")`

c. Click the Run icon.

The program executes up to the breakpoint you set.

d. Click the Step Into icon.

The debugger displays the `printf` function and halts execution at the first statement of this function.

e. Click the Step Over icon.

The debugger executes the first statement of the `printf` function and halts at the next statement.

f. Click the Step Out icon.

The debugger executes the remaining statements of the `printf` function, returns to the `main` function, and halts at the first statement after the invocation of the `printf` function.

5. Click the Kill Thread icon

The debug session terminates, and the debugger window closes.

For More Information: www.freescale.com

4

Creating a CodeWarrior™ Project

This chapter provides an overview of the steps required to generate code that runs on mobileGT™ processors.

The sections are:

- Types of Projects
- Using PowerPC EABI Templates
- Using the Makefile Importer Wizard
- Project Targets

Types of Projects

The CodeWarrior IDE for Embedded PowerPC generates binary files in the ELF format. You can create three different kinds of projects: *application* projects, *library* projects, and *partial linking* projects.

The only difference between the application projects and library projects is that an application project has associated stack and heap sizes; a library does not. A partial linking project allows you to generate an output file that the linker can use as input.

You can create an Embedded PowerPC project by using the:

- EPPC New Project Wizard
- PowerPC EABI templates
- Makefile Importer Wizard

“Creating a Project” on page 25 explains how to create a project by using the EPPC new project wizard.

For More Information: www.freescale.com

Freescale Semiconductor, Inc.

Using PowerPC EABI Templates

The CodeWarrior software includes PowerPC Embedded Application Binary Interface (EABI) templates for Embedded PowerPC projects. Project templates help you get started quickly: You must only create an empty project and add the template sources to this project.

The EABI template source files are in this directory:

installDir\Templates\PowerPC_EABI\Sources

where *installDir* is a placeholder for the path in which you installed your CodeWarrior product.

The PowerPC EABI template directories are organized according to target board name.

Most template source files are placeholders only. You must replace them with your own files.

Using the Makefile Importer Wizard

Use the Makefile Importer wizard to convert a GNU makefile into a CodeWarrior project. The Makefile Importer wizard lets you:

- parse the makefile to determine source files and build targets.
- create a project.
- add the source files and build targets determined during parsing.
- match makefile information, such as output name, output directory, and access paths, with the newly created build targets.
- select a project linker.

To convert makefiles to a CodeWarrior project:

1. Specify the project settings.

   a. Select **File > New**

      The **New** dialog box appears.

   b. Select **Makefile Importer Wizard**

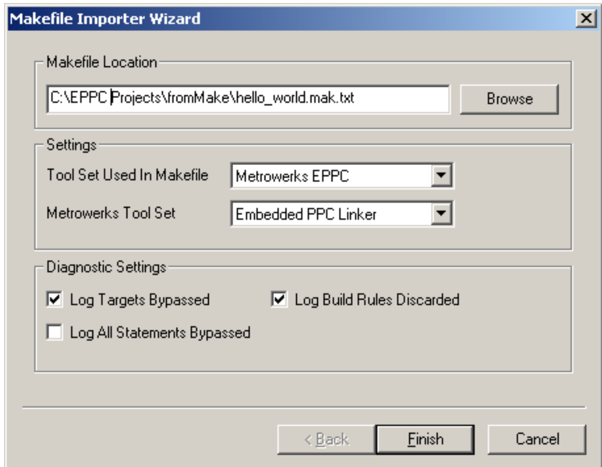
   c. In the Project Name text box, type the project name (including the `.mcp` extension).

For More Information: www.freescale.com

d. Click **OK**

The **Makefile Importer Wizard** dialog box appears. (See Figure 4.1.)

Figure 4.1 Makefile Importer Wizard Dialog Box



2. Specify the path to the makefile.

   Type the path to the makefile in the Makefile Location text box. Alternatively, click **Browse** to display a dialog box you can use to find and select the makefile.

3. Select the makefile conversion tool and the linker.

   a. Use the Tool Set Used In Makefile listbox to select the tool set that was used to generate the selected make file.

      Currently, just Metrowerks EPPC makefiles are supported.

   b. Use the Metrowerks Tool Set listbox to select the Metrowerks tools to use in the project created from the selected make file.

      Currently, just the Embedded PPC Linker is supported.

4. Select the desired diagnostics:

   – Check the Log Targets Bypassed box to generate a log file containing information about make file build targets that the conversion tool fails to convert to project build targets.

   – Check the Log Build Rules Discarded box to generate a log file that contains information about make file rules that the conversion tool discards during conversion.

Freescale Semiconductor, Inc.

- Check the Log All Statements Bypassed box to generate a log file containing information about the targets bypassed, build rules discarded, and other makefile items that the conversion tool fails to convert.

5. Click **Finish**

   The Makefile Importer wizard creates a CodeWarrior project using the information in the specified make file.

Project Targets

The CodeWarrior stationery includes multiple targets with different purposes. Using the project stationery, you can add your own code to an existing stationery project, quickly set up the code so that it is appropriate to place in ROM, and burn the code into ROM.

The available targets are:

- Debug Version

  This is the default target setting when you create the project. This target includes only the user code and the standard and runtime libraries. This target does not perform any hardware initialization or set up any exception vectors. You can continue using only this target until you need ISRs or to flash your code to the ROM.

- ROM Version

  Select this target to generate an s-record final output file for programming your code into the ROM. This target builds an image for ROM that includes all exception vectors, a sample ISR, and the hardware initialization. You can use the s-record that this target generates with any standard flash programmer to burn your program into ROM, or you can use the third target (Flash to ROM version) to burn your program into ROM.

For More Information: www.freescale.com

5

Target Settings

Target settings control the behavior of the development tools used by a build target of a CodeWarrior™ project.

This chapter defines the mobileGT™-specific target settings from which you can select. See the *CodeWarrior IDE User Guide* for information about the target settings available for all CodeWarrior products.

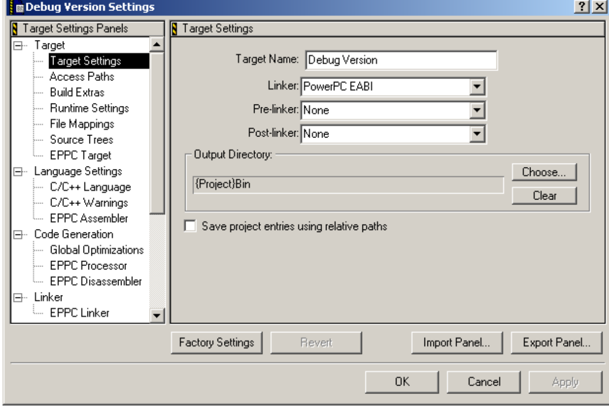
The sections are:

- Target Settings Overview
- Embedded PowerPC-Specific Target Settings Panels
- PC-lint Target Settings Panels

Target Settings Overview

Target settings are organized into panels you can display in the target settings window. To display the **Target Settings** window (Figure 5.1), select **Edit > *Target* Settings** (where ***Target*** is the name of the current build target of your CodeWarrior project).

Figure 5.1 Target Settings Window



Select the panel you want to display from the list in the left pane of the **Target Settings** window.

NOTE If you use the EPPC New Project Wizard to create a project, the wizard assigns default values to all options of all settings panels.

Embedded PowerPC-Specific Target Settings Panels

This section explains the purpose and effect of each setting in the panels specific to Embedded PowerPC development.

The target settings panels covered in this section are:

- Target Settings
- EPPC Target
- EPPC Assembler
- Global Optimizations
- EPPC Processor
- EPPC Disassembler
- EPPC Linker
- Debugger PIC Settings
- EPPC Debugger Settings
- Source Folder Mapping
- System Call Service Settings

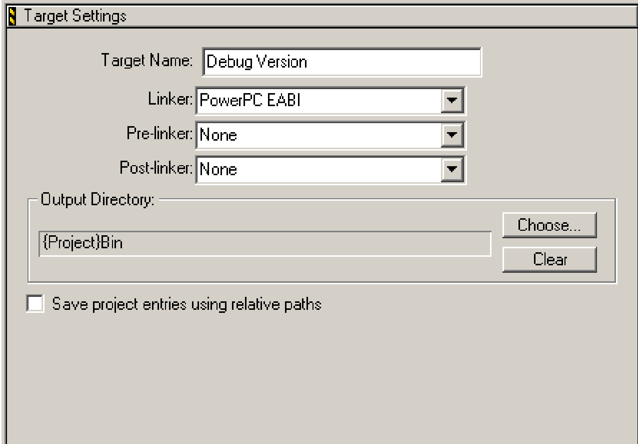
NOTE The **Global Optimizations** and **EPPC Processor** settings panels each provide code optimization options. Use both panels to select the best combination of optimization settings for your application.

Target Settings

The **Target Settings** panel (Figure 5.2) is the most important target settings panel because it lets you select the linker a build target uses.

Linker choice is most important because a linker generates a build target's final output file, thereby determining the operating system and/or microprocessor with which this file can be used. Further, linker choice defines the other target settings panels available in the **Target Settings** window.

Figure 5.2 Target Settings Panel



In addition to linker selection, use the **Target Settings** panel to define the name of the current build target, to select pre- and post-linkers to execute during the build process, and to specify the directory to which the linker writes its output.

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.

Target Name

Type the name of the current project build target in the Target Name text box.

If you display the Targets view of a Project window, the name of each build target in your project is displayed.

NOTE Target Name is *not* the name of your output file generated by a build target; rather, Target Name is the name of your project's current

For More Information: www.freescale.com

build target. You define the name of a build target's output file in the EPPC Target target settings panel.

Linker

From the Linker listbox, select the linker for the current build target to use. Your choices are:

- PowerPC EABI

  Choose this option to configure a build target to generate a file in Executable and Linkable (ELF) format.

- PCLint Linker

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

  PC-lint is a third-party software development tool created 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 Depending on your linker choice, a different set of panel names appears in the left pane of the **Target Settings** window. The sections immediately below document the panels used by both linkers and those specific to the PowerPC EABI linker. See PC-lint Target Settings Panels for documentation of the panels specific to PC-lint.

Pre-linker

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

Use the Pre-linker listbox to select the pre-linker for the current build target to use.

CodeWarrior Development Studio, mobileGT Edition includes just one pre-linker, the 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.

For More Information: www.freescale.com

Post-linker

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

Use the Post-linker listbox to select the post-linker for the current build target to use.

CodeWarrior Development Studio, mobileGT Edition includes just one post-linker, the BatchRunner PostLinker.

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

Output Directory

The output directory is the directory in which the linker places a build target's final output file (application or library file).

The project directory is the default output directory. Click **Choose** to display a dialog box that lets you select the output directory for the current build target.

Save project entries using relative paths

Check the Save project entries using relative paths box to instruct the IDE to save the relative path to each file in a project 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.

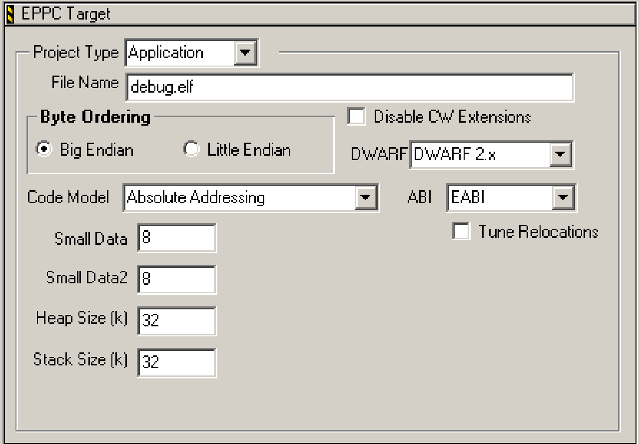
If this box is unchecked, each file in a project must have a unique name. This is so because, when searching for files, the IDE combines the directory names in the **Access Paths** panel with just the root filename of each project file. As a result, the IDE cannot discriminate between two files that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

For More Information: www.freescale.com

EPPC Target

Use the **EPPC Target** settings panel (Figure 5.3) to specify the name of the final output file (application or library file) generated by the linker for the current build target. In addition, use this panel to tell the linker how to setup this file.

Figure 5.3 EPPC Target Settings Panel



Project Type

Use the Project Type listbox 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 (k)**,** Stack Size (k), 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 name of an application ends in `.elf` or `.ELF`, the extension is stripped before the `.mot` and `.MAP` extensions are added (if you have selected the appropriate switches for generating S-Record and map files in the EPPC Linker panel).

Byte Ordering

Use the option buttons in the Byte Ordering area to select either little endian or big endian format to store generated code and data. In big endian format, the most significant byte comes first (B3, B2, B1, B0). In little endian format, the bytes are organized with the least significant byte first (B0, B1, B2, B3). See documentation for the PowerPC processor for details on setting the byte order mode.

Disable CW Extensions

If you are exporting code libraries from CodeWarrior software to other compilers/ linkers, check the Disable CW Extensions checkbox to disable CodeWarrior features that may be incompatible.

The CodeWarrior IDE currently supports one extension: storing alignment information in the `st_other` field of each symbol.

If the Disable CW Extensions checkbox is checked:

- The `st_other` field is always set to 0.

  Certain non-CodeWarrior linkers require that this field have the value 0.

- The CodeWarrior linker cannot deadstrip files.

  To deadstrip, the linker requires that alignment information be stored in each `st_other` field.

Check the Disable CW Extensions checkbox to create a C-language library for use with third-party linkers. However, not all third-party linkers require that you disable CodeWarrior extensions; you may need to try both settings. When building a CodeWarrior linked application, clear the Disable CW Extensions checkbox to avoid generating a larger application. Assembly language files do not need this option; and C++ libraries are not portable to other linkers.

For More Information: www.freescale.com

DWARF

Use the DWARF listbox to select the version of the Debug With Arbitrary Record Format (DWARF) debug format the linker generates. The linker ignores debug information that is not in the format that you select from the DWARF listbox.

ABI

Use the ABI listbox to select the Application Binary Interface (ABI) the compiler uses for function calls and structure layout.

Tune Relocations

The tune relocations option pertains to object relocation and is available for just these application binary interfaces: EABI and SDA PIC/PID.

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

Checking the Tune Relocations checkbox has these effects:

- For the EABI application binary interface, a 14-bit branch relocations is converted to 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 instead of `r0`; absolute code is changed to code references to use the PC relative relocations

For more information about PIC/PID support, see this release notice:

```
installDir\Release_Notes\PowerPC_EABI\
CW_Tools\Compiler_Notes\CW Common PPC Notes 3.0.x.txt
```

where *`installDir`* is a placeholder for the path in which you installed your CodeWarrior product.

Code Model

Use the Code Model listbox to select the Absolute Addressing or SDA PIC/PID addressing mode for the generated application or library.

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 Data2 address space.

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

Heap Size (k)

Use the Heap Size text box to specify the amount of memory (in kilobytes) allocated for the heap. The heap is used when your program calls `malloc` or `new`.

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

Stack Size (k)

Use the **Stack Size** text box to specify the amount of memory (in kilobytes) allocated for the stack.

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

NOTE Allocate stack and heap size based on the amount of memory on your target board. If you allocate lots of memory for the heap and/or stack compared to the total amount of memory available, your program may not run correctly.

Optimize Partial Link

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

Check the Optimize Partial Link checkbox to instruct the linker to directly download the output of your partial link. Enabling this option lets the linker:

- Use a linker command file (LCF).

  The commands in an LCF let you merge the sections of your application into the `.text`, `.data` or `.bss` segment. If you do not use an LCF to perform this merge, the CodeWarrior debugger will probably not be able to 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 tool `munch`.

NOTE It is very important that you do not use `munch` yourself because the linker must put C++ exception handling initialization code in the first constructor. If you see `munch` in your makefile, it is a clue that you need an optimized build.

- Change common symbols to `.bss` symbols. This lets you examine the variable in the debugger.
- Perform a special type of partial link that has no unresolved symbols. This link is the same as that performed by the Diab linker when passed `-r2` argument.

If the Optimize Partial Link checkbox is clear, the build target's output file remains as if you passed the linker the `-r` argument from the command-line.

Deadstrip Unused Symbols

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

Check the Deadstrip Unused Symbols checkbox to instruct the linker deadstrip 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).

Require Resolved Symbols

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

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

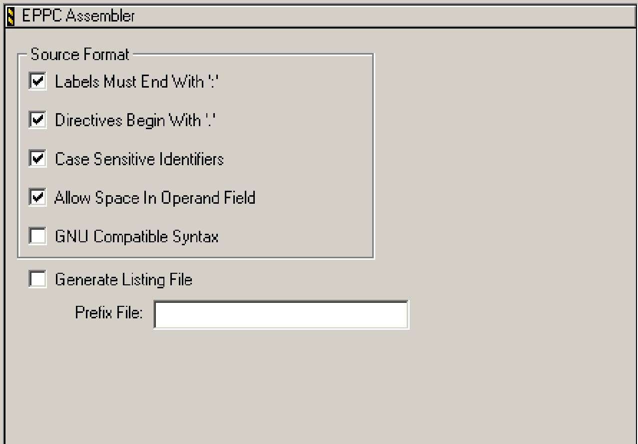
NOTE Some real-time operating systems require that there be no unresolved symbols in a partial link file. In this case, enable this option.

For More Information: www.freescale.com

EPPC Assembler

Use the **EPPC Assembler** target settings panel (Figure 5.4) 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 5.4 EPPC Assembler Target Settings Panel



Source Format

Use the checkboxes in the Source Format area to define certain syntax options for the assembly language source files. For more information on the assembly language syntax for the Embedded PowerPC assembler, read the manual *Assembler Reference*.

GNU Compatible Syntax

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

For More Information: www.freescale.com

- 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 precedence rules for operators
- Implementing GNU compatible numeric local labels from 0 to 9
- Treating numeric constants beginning with 0 as octal
- 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 direct the assembler to generate a listing file when processing a build target.

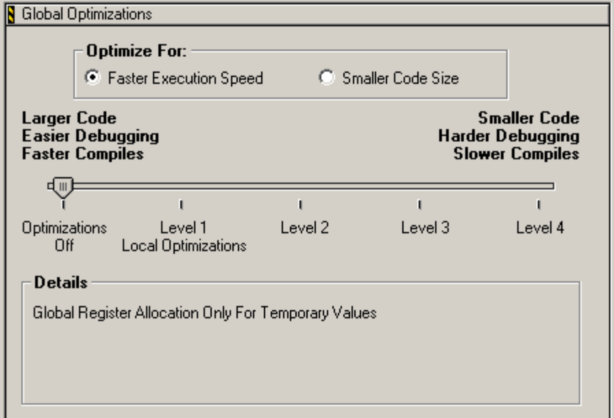
Prefix File

The Prefix File text box specifies a prefix file that is automatically included in all assembly files in the project. This text box lets you include common definitions without including the file in every source file.

Global Optimizations

Use the **Global Optimizations** settings panel (Figure 5.5) to instruct the compiler to rearrange the object code to produce smaller or faster executing object code.

Figure 5.5 Global Optimizations Target Settings Panel



The type of optimization performed depends on the optimization level you select. For example, the compiler may remove redundant operations in a program for a particular optimization level. For other optimization levels, the compiler may analyze how an item is used in a program and attempt to reduce the effect of that item on the performance of the program.

In all cases, the compiler manipulates the instruction stream without affecting the semantics of the program. In other words, an unoptimized program and its optimized counterpart produce the same results.

Optimization may produce unexpected results in syntactically correct but semantically ambiguous code. In the **C/C++ Warnings** target settings panel, check the Extended Error Checking and Possible Errors checkboxes to detect such situations.

The optimization levels are:

- Optimizations Off (level 0)

  The compiler performs global register allocation (register coloring) for compiler-generated (temporary) variables only.

NOTE If creating an application for debugging, select Optimizations Off. Other optimization levels prevent the debugger from showing you the contents of registers accurately.

- Level 1

  The compiler performs dead code elimination and global register allocation.

- Level 2

  The compiler performs the optimizations of Level 1 plus common subexpression elimination and copy propagation.

  Level 2 is the best selection for most build targets.

- Level 3

  The compiler performs the optimizations of Level 2. In addition, the compiler moves invariant expressions out of loops (also called *Code Motion*) and performs strength reduction of induction variables, copy propagation, and loop transformation.

  Level 3 is the best selection for a build target with many loops.

- Level 4

  The compiler performs the optimizations of Level 3, including performing some of them a second time for even greater code efficiency.

  Level 4 can provide the greatest code optimization but requires takes more compilation time than do the other levels.

This **Global Optimizations** target settings panel provides the same options as the `#pragma global_optimizer` and `#pragma optimization_level` pragmas.

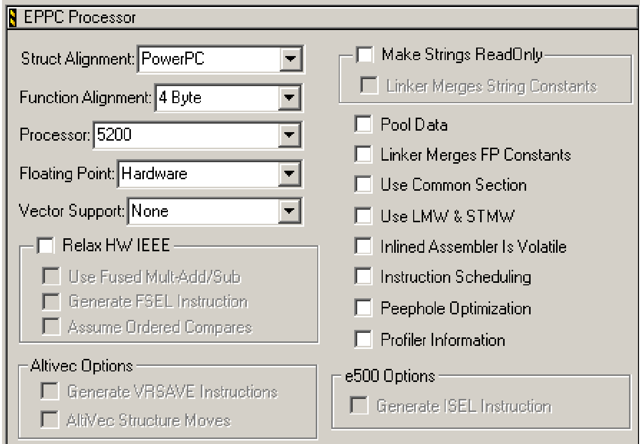
NOTE Use compiler optimizations only after debugging your software. Using a debugger on an optimized code may affect the register values, stack trace, and source code a debugger displays.

For More Information: www.freescale.com

EPPC Processor

Use the **EPPC Processor** settings panel (Figure 5.6) to make processor-dependent code generation settings.

Figure 5.6 EPPC Processor Target Settings Panel



Struct Alignment

Use the Struct Alignment listbox 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 explanation of pragma “pack” on page 136.

NOTE If you choose a Struct Alignment setting other than PowerPC, your code may not work 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 listbox to select alignments from 4 (the default) to

128 bytes. These selections correspond to `#pragma function_align`. For more information, see “function_align” on page 134.

NOTE	The `st_other` field of the `.symtab` (ELF) entries has been overloaded to ensure that dead-stripping of functions does not interfere with the alignment you have chosen. This may result in code that is incompatible with some third-party linkers.

Processor

Use the Processor listbox to specify the 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 Embedded PowerPC processor. Choosing Generic lets you use all optional instructions and the core instructions of the 505, 509, 555, and 56x processors.

Selecting a particular target processor produces these results:

- Instruction scheduling

  If the Instruction Scheduling checkbox (also in the EPPC Processor panel) is selected, the processor selection helps define how scheduling optimizations are made.

- Preprocessor symbol generation

  A preprocessor symbol is defined based on your target processor. It is equivalent to the following definition, where *number* is the three-digit number of the Embedded PowerPC processor being targeted:

  ```
  #define __PPCnumber__ 1
  ```

  For example, for the 555 processor, the symbol would be `__PPC555__`. If you select Generic, the macro `__PPCGENERIC__` is defined to 1.

- Floating-point support

  The None (no floating-point), Software, and Hardware options are available for all processors, even those processors without a floating-point unit. If your target system does not support handling a floating-point exception, you should select the None or Software options. If the Hardware option is not selected, the Use FMADD & FMSUB checkbox is not available.

Floating Point

Use the Floating Point listbox to define how the compiler handles floating-point operations in your code. To specify how the compiler should handle floating-point operations for your project, you need to:

For More Information: www.freescale.com

- Choose an option from the Floating Point listbox.
- Include the corresponding runtime library in your project.

  For example, if you select None, you must also include the library `Runtime.PPCEABI.N.a` in your project.

A description of each option follows.

- None

  Disables floating-point support.
- Software

  Floating-point operations are performed by the runtime library.

NOTE The calls generated by using floating-point emulation are defined in the C runtime library. As a result, selecting software emulation without including the appropriate C runtime library results in link errors. If you use floating-point emulation, you must include the appropriate C runtime file in your project.

- Hardware

  Floating-point operations are performed by the processor's floating-point unit.

NOTE Do not select the Hardware option if the processor you are using has no floating-point unit.

- SPE-EFPU

  Performs single-precision floating-point operations using e500-EFPU hardware instructions. Performs double-precision floating-point operations using the runtime library.

Vector Support

Use the Vector Support listbox to select the type of vector execution unit your target processor has. The CodeWarrior EPPC 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 listbox. If your processor does not have a vector execution unit or you do not want the compiler to emit vector instructions, select None.

NOTE No member of the mobileGT family includes an AltiVec or SPE vector execution unit. As a result, if you are targeting a member of the mobileGT family, select None from the Vector Support listbox.

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

Relax HW IEEE

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

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

Use Fused Multi-Add/Sub

Check this box to instruct the compiler to generate PowerPC 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 kinds 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 listbox.

Altivec Structure Moves

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

Generate VRSAVE Instructions

The VRSAVE register indicates to the operating system which vector registers to save and reload when a context switch happens. The bits of the VRSAVE register that correspond to the number of each affected vector register are set to 1.

When a function call happens, the value of the VRSAVE register is saved as 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 those registers.

Check the Generate VRSAVE Instructions box only if developing for a real-time operating system that supports AltiVec. Checking the Generate VRSAVE Instructions checkbox tells the compiler to generate instructions that save and restore these vector-register-related values.

Make Strings Read Only

Check the Make Strings Read Only checkbox to instruct the compiler to store string constants in the read-only `.rodata` section. Leave this checkbox clear to instruct the compiler to store string constants in the ELF-file data section. The Make Strings Read Only option corresponds to `#pragma readonly_strings`. The default setting of this pragma is `OFF`.

If you check the Make Strings Read Only checkbox, the Linker Merges String Constants checkbox becomes available. Check the Linker Merges String Constants checkbox to have the compiler pool strings together from a given file. If this checkbox is clear, the compiler treats each string as an individual string. The linker can deadstrip unused individual 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`) so 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. The linker is aggressive in stripping unused data and functions from your binaries; 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 the linker finds two or more variables with the same name and at least one of them is in a common section, those variables share the same storage 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 checkbox must be checked 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 project level.

NOTE You must initialize all common variables in each source file that uses these variables; otherwise you get unexpected results.

NOTE It is recommended that you develop with the Use Common Section box clear. Once you have debugged your program, look at the 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 PowerPC instruction that loads a group of registers; `STMW` (Store Multiple Word) is a single PowerPC 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 in little-endian code, 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.

Consult 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 have the compiler treat all `asm` blocks (including inline `asm` blocks) as if the `volatile` keyword was present. This prevents the `asm` block from being optimized.

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

For More Information: www.freescale.com

Instruction Scheduling

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

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). It is sometimes helpful to clear this checkbox when debugging, and then check it again once you have finished the bulk of your debugging.

Peephole Optimization

Check the Peephole Optimization checkbox to have the compiler 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`.

Profiler Information

Check the Profiler Information checkbox to generate special object code during runtime to collect information for a code profiler.

This checkbox corresponds to `#pragma profile`.

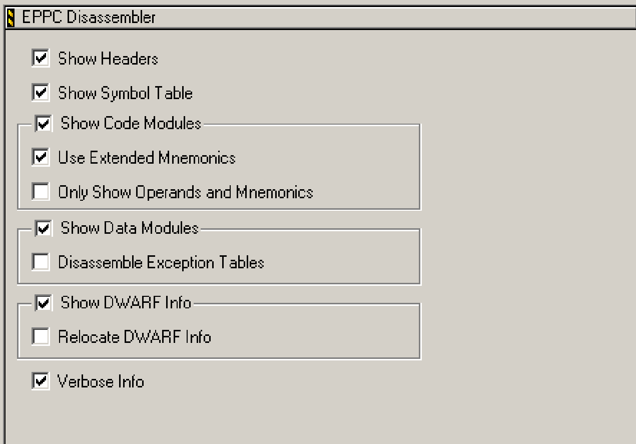
e500 Options

The e500 options do not apply to the CodeWarrior Development Studio, mobileGT Processor Family product.

EPPC Disassembler

Use the **EPPC Disassembler** settings panel (Figure 5.7) to control the information displayed by the CodeWarrior disassembler.

Figure 5.7 EPPC Disassembler Target Settings Panel



See the Compiling and Linking chapter of the *IDE User Guide* for general information about the Disassemble command.

Show Headers

Check the Show Headers checkbox to have the disassembler list any ELF header information in the disassembled output.

Show Symbol Table

Check the Show Symbol Table checkbox to have the disassembler list the symbol table for the disassembled module.

Show Code Modules

Check the Show Code Modules checkbox to have the disassembler provide ELF code sections in the disassembled output for a module.

Checking the Show Code Modules makes these checkboxes available:

- Use Extended Mnemonics

  Check this checkbox to have the disassembler list the extended mnemonics for each instruction for the disassembled module.

- Only Show Operands and Mnemonics

  Check this checkbox to have the disassembler list the offset for any functions in the disassembled module.

Show Data Modules

Check the Show Data Modules checkbox to have the disassembler provide ELF data sections (such as `.rodata` and `.bss`) in the disassembled output for a module.

Checking this checkbox makes the Disassemble Exception Tables checkbox available. Check the Disassemble Exception Tables checkbox to have the disassembler include C++ exception tables in the disassembled output for a module.

Show DWARF Info

Check the Show DWARF Info checkbox to have the disassembler include DWARF symbol information in the disassembled output.

Checking this checkbox makes the Relocate DWARF Info checkbox available. The Relocate DWARF Info checkbox lets you relocate object and function addresses in the DWARF information.

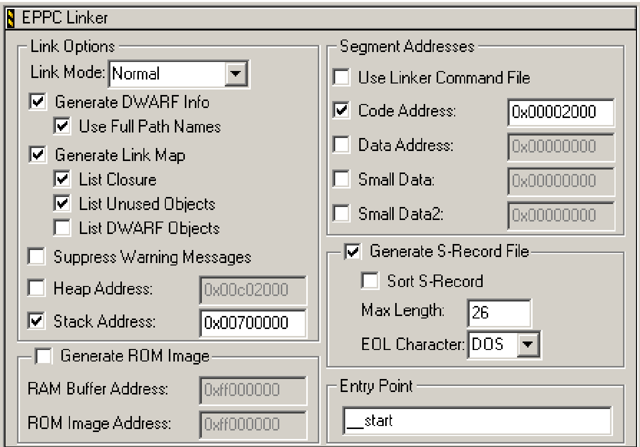
Verbose Info

Check the Verbose Info checkbox to instruct the disassembler to show additional information about certain types of information in the ELF file. 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.

EPPC Linker

Use the **EPPC Linker** target settings panel (Figure 5.8) to select options related to linking your object code into its final form: application or library.

Figure 5.8 EPPC Linker Target Settings Panel



Link Mode

The link mode lets you control how much memory the linker uses while it writes the output file to the hard-disk. Linking requires enough RAM space 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 listbox to select the link mode. The link mode 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.

For More Information: www.freescale.com

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 the Use Full Path Names checkbox to define how the linker includes path information for source files. If the Use Full Path Names checkbox is checked, the linker includes full paths as well as root file names within the linked ELF file (see the note that follows). If this checkbox is cleared, the linker saves just the root file names.

NOTE If you build your programs on one machine and debug it on another, clear the Use Full Path Names checkbox. Clearing this box enables the debugger to more easily find the source code files associated with an application or library.

Generate Link Map

Check the Generate Link Map checkbox to tell 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. It also displays the address given 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 when the relocatable file is compiled with the CodeWarrior compiler, it never deadstrips assembler relocatables or relocatables built with other compilers. If a relocatable was not built with the CodeWarrior C/C++ compiler, the link map lists all the unused but unstripped symbols. You can use that information to remove the symbols from the source and rebuild the relocatable in order to make your final process image smaller.

List Closure

This checkbox is available only if you check the Generate Link Map checkbox. Check the List Closure checkbox to have all the functions called by the starting point of the program listed in the link map. See “Entry Point” on page 73 for details.

List Unused Objects

This checkbox is available only if you check the Generate Link Map checkbox. Check the List Unused Objects checkbox to tell the linker to include unused objects in the link map. This setting is useful in cases where you may discover that an object you expect to be used is not in use.

List DWARF Objects

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

Suppress Warning Messages

Check the Suppress Warning Messages checkbox to tell the linker not to display warnings in the CodeWarrior message 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`.

If you wish to specify a specific heap address, check the checkbox and type an address in the Heap Address text box. You must specify the address in hexadecimal notation. The address you specify is the bottom of the heap. The address is then aligned up to the nearest 8-byte boundary, if necessary. The top of the heap is Heap Size (k) kilobytes above the Heap Address (Heap Size (k) is found in the EPPC Target panel). The possible addresses depend on your target hardware platform and how the memory is mapped. The heap must reside in RAM.

If you clear the checkbox, the top of the heap is equal to the bottom of the stack. In other words:

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

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

For More Information: www.freescale.com

You can clear the Heap Address checkbox if your code does not make use of a heap. If you are using MSL, your program may implicitly use a heap.

NOTE If there is not enough free space 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 the stack.

Stack Address

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

If you want to specify a stack address, check the checkbox and type an address in the Stack Address text box. You must specify the address in hexadecimal format. The address you specify is the top of the stack. The stack extends downward from the specified address the number of kilobytes you specify in the Stack Size (k) text box of the EPPC Target panel. The stack address is aligned to the nearest 8-byte boundary, if necessary. The possible addresses 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 clear this checkbox, 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 MetroTRK installed, this monitor puts its data in high memory. The default (factory) stack address reflects the memory requirements of MetroTRK and places the stack address at `0x003DFFF0`. MetroTRK 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 enter 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 addresses and for execution addresses helpful.
For more information about the linker-generated symbols created 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.

For More Information: www.freescale.com

ROM Image Address

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

Segment Addresses

Use the checkboxes in the Segment Addresses area to specify whether you want the segment address specified in a linker command file or directly in this settings panel.

Use Linker Command File

Check the Use Linker Command File checkbox to have the segment addresses specified in a linker command file. If the linker doesn't find the command file it expects, it issues an error message.

Leave this checkbox clear if you want to specify the segment addresses directly in 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 cleared, the linker ignores the file.

Code Address

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

If you wish to specify a code address, check the checkbox and type an address in the Code Address text box. You must specify the address in hexadecimal notation. The possible addresses depend on your target hardware platform and how the memory is mapped.

If you clear the checkbox, the default code 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.

NOTE To enter a hexadecimal address, use the format `0x12345678`, (where the address is the 8 digits following the character “x”).

Data Address

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

If you wish to specify a data address, check the checkbox and type an address in the Data Address text box. You must specify the address in hexadecimal notation. The possible addresses depend on your target hardware platform and how the memory is mapped. Data must reside in RAM.

If you clear the Data Address checkbox, the linker calculates the data address to begin immediately following the read-only code and data (`.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 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 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.

For More Information: www.freescale.com

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 Additional Small Data Sections for instructions.

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

This checkbox is available only if you check the Generate S-Record File checkbox. Check the Sort S-Record checkbox to have the generated S-Record file sorted in the ascending order by address.

Max Length

The Max Length text box specifies the maximum length of the S-record generated by the linker. This text box is available only if you check the **Generate S-Record File** checkbox. The maximum value allowed for an S-Record length is 256 bytes.

NOTE Most programs that load applications onto embedded systems have a maximum length allowed for the S-Records. The CodeWarrior debugger can handle S-Records of 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 listbox to select the end-of-line character for the S-Record file. This listbox is available only if the Generate S-Record File checkbox is checked. The end of line character options are:

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

Entry Point

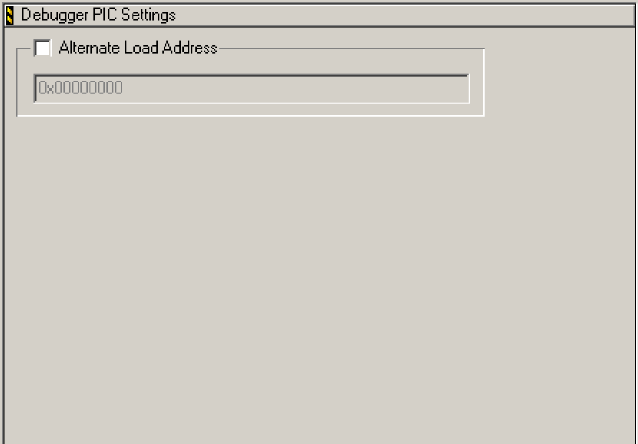
Use the Entry Point text box to specify the function that the linker uses first when the program launches. This is the starting point of the program.

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.

Debugger PIC Settings

Use the **Debugger PIC Settings** panel (Figure 5.9) to specify an alternate address at which you want your ELF image loaded on your target board.

Figure 5.9 Debugger PIC Target Settings Panel



Usually, Position Independent Code (PIC) is linked in such a way so that the entire image starts at address `0x00000000`. The Debugger PIC Settings panel lets you specify the alternate address where you want to load the PIC module on the target.

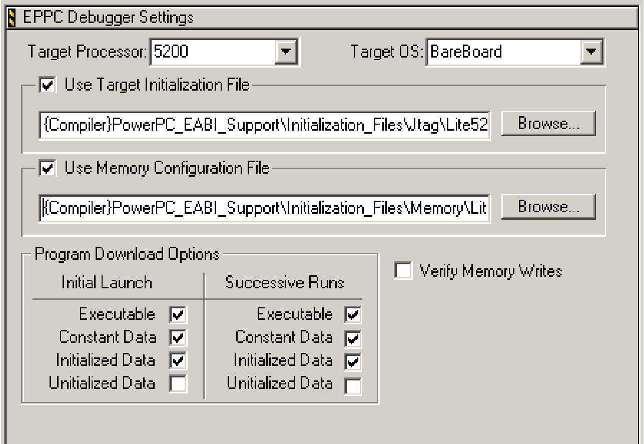
To specify the alternate load address, check the Alternate Load Address checkbox and enter the address in the associated text box. The debugger loads your ELF file on the target at the new address.

The debugger does not verify whether your code can execute at the new address. Instead, correctly setting any base registers and performing any needed relocations are handled by the PIC generation settings of the compiler and linker and the startup routines of your code.

EPPC Debugger Settings

Use the **EPPC Debugger Settings** panel (Figure 5.10) to supply information the CodeWarrior debugger needs to debug programs running on your target device.

Figure 5.10 EPPC Debugger Target Settings Panel



Target Processor

Use the Target Processor listbox to select the processor of your emulator or board.

Target OS

Use the Target OS listbox to enable the type of debugging desired. The choices are:

- Bareboard

  Enables bareboard 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 KOIL (Kernel 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.

For More Information: www.freescale.com

Use Target Initialization File

Check this box if you want the build target to use a target initialization file. Click **Browse** to display a dialog box you can use to find and select the appropriate target initialization file. Prebuilt target initialization files are automatically selected for supported boards.

Sample target initialization files are in the `Jtag` subdirectory of this path:

`installDir``\PowerPC_EABI_Support\Initialization_Files`

Use Memory Configuration File

Check the Use Memory Configuration File checkbox if you want to use a memory configuration file. This file defines the valid accessible areas of memory for your specific board. Click **Browse** to display a dialog box you can use to find and select the appropriate memory configuration file.

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.

For details, see the appendix “Memory Configuration Files” on page 205.

Program Download Options

There are four section types listed in the Program Download Options section of the EPPC Debugger Settings panel:

- Executable

  The executable code and text sections of the program.
- Constant Data

  The constant data sections of the program.
- Initialized Data

  The initialized data sections of the program.
- Uninitialized Data

The uninitialized data sections of the program that are usually initialized by the runtime code included your CodeWarrior product.

If one of these section types is selected, this means that it is to be downloaded when the program is debugged.

NOTE You do not need to download uninitialized data if you are using Metrowerks runtime code.

Verify Memory Writes

Check this checkbox to verify that any or all sections of the program are making it to the target processor successfully, or that they have not been modified by runaway code or the program stack. For example, once you download a text section you might never need to download it again, but you may want to verify that it still exists.

Source Folder Mapping

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

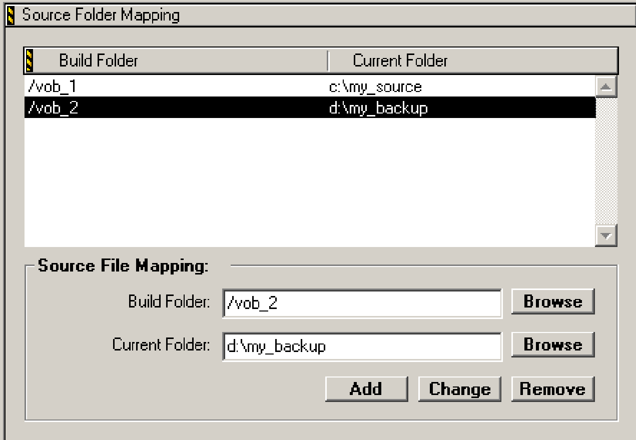
The mapping information you supply lets the CodeWarrior debugger find and display your source code files even though they are not in the locations specified in the executable file's debug information.

NOTE If you create a CodeWarrior project by opening an ELF file in the IDE, the IDE uses the debug information in this file to add the source files used to build the file to the new project.
If the IDE cannot find a particular source file, the IDE displays a dialog box that you use to tell the IDE where the missing file is currently. The IDE uses the current location information in conjunction with the debug information in the ELF file to create entries in the **Source Folder Mapping** panel.

Figure 5.11 shows the **Source Folder Mapping** target settings panel.

For More Information: www.freescale.com

Figure 5.11 Source Folder Mapping Target Settings 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 that 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.

When 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 that 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.

When 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.

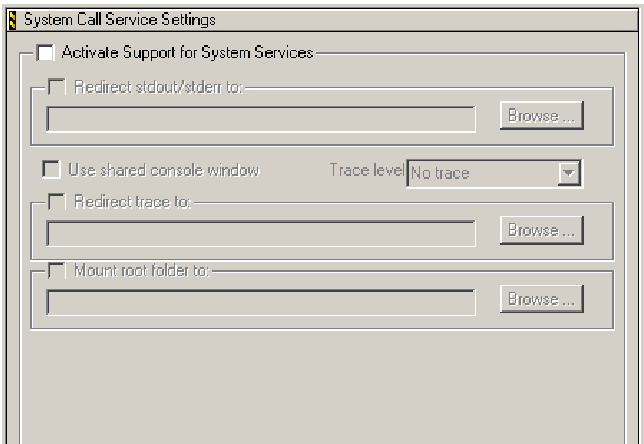
Remove

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 Setting** panel (Figure 5.12) to activate support for system services and to select options for handling requests for system services.

Figure 5.12 System Call Service Target Setting Panel



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

The host debugger implements the services. Therefore, the host OS service routines are available only when you are debugging code on the target.

NOTE The OS service routines provided must be compliant to an industry-accepted standard. The definitions for 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 location for displaying the console output is a separate CodeWarrior IDE window. If you wish to redirect the console output to a file, check the Redirect stdout/stderr to checkbox. Click **Browse** to display a dialog box you can use to find and select the log 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 listbox to specify the system call trace level. The system call trace level options available are:

- No Trace

  System calls are not traced.

- Summary Trace

  Requests for system services are displayed.

- Detailed Trace

  Requests for system services are displayed along with the arguments/parameters of the each request.

The Redirect trace to checkbox defines the place to which traced system service requests are displayed.

Redirect Trace to

The default location for displaying traced system service requests is a separate CodeWarrior IDE window. If you wish to log the traced system service requests in a file, check the Redirect trace to checkbox. Click **Browse** to display a dialog box you can us to find and select a log file.

Mount Root Folder to

The default root folder for file IO services is the parent folder for the loaded ELF file. If you wish to specify the root folder for file IO services, check the Mount root folder

to checkbox. Click **Browse** to display a dialog box that you can use to find and select a root folder.

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 Development Studio, mobileGT Processor Edition 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 expects your PC-lint installation to be in *`installDir`*`\Lint` (where *`installDir`* is the path in which you installed 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 listbox.

   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.

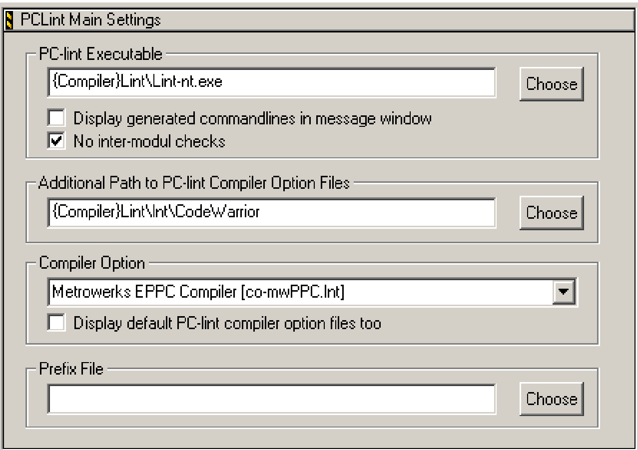
   The sections that follow explain how to use the PC-lint target settings panels.

PCLint Main Settings

Use the **PCLlint Main Settings** panel (Figure 5.13) to provide the path to the PC-lint executable and to define the compiler option files and prefix file that PC-lint will use.

NOTE The IDE displays the PC-lint target settings panels only if you first select PCLint Linker in the **Target Settings** panel (Figure 5.2).

Figure 5.13 PCLint Main Settings Target Settings Panel



PC-lint Executable

Type the path to and name of the PC-lint executable file in this text box. Alternatively, click **Choose** to display a dialog box that lets you navigate to and 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 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 this 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 listbox (see below) enables and displays these files.

The default CodeWarrior installation includes prewritten 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 Metrowerks 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 Metrowerks compiler they are using from the Compiler Option list.

You can leave this text box empty, if desired.

Compiler Option

Select the PC-lint compiler option file for the Metrowerks compiler the build target is using from this listbox.

This listbox 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 listbox 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 listbox along with those in the directory specified in the Additional Path to PC-lint Compiler Option Files text box.

Prefix File

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.

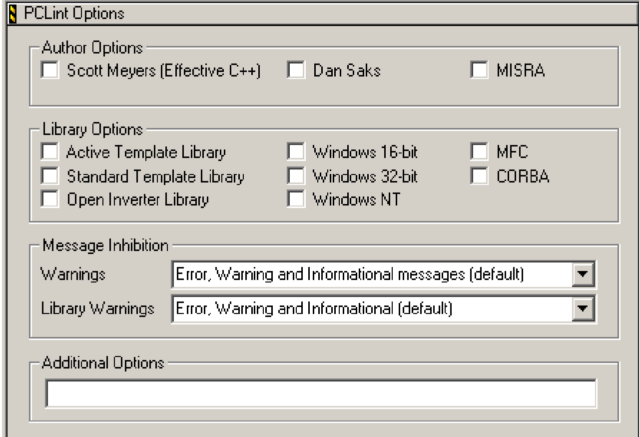
Typically, you use this feature to define macros to required values for a particular PC-lint run or to instruct PC-lint to check certain command-line commands. To do this, define this information in a prefix file.

You can leave this text box empty, if desired.

PCLint Options

Figure 5.14 shows the **PCLint Options** target settings panel. Use this panel to define the syntax rules PC-lint uses to validate your C/C++ source, 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 5.14 PC-lint Options Target Settings Panel



Author Options

This group of checkboxes lets 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 group of checkboxes lets 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.

For More Information: www.freescale.com

Freescale Semiconductor, Inc.

- 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 this listbox to control the warning and error messages that PC-lint emits.

The default setting displays error, warning and information messages.

Library Warnings

Use this listbox to control the warning and error messages that PC-lint emits for libraries.

The default setting displays error, warning and information messages.

Additional Options

Type the PC-lint command-line switches for the IDE to pass to PC-lint in this text box. Refer to your PC-lint manuals for documentation of these switches.

For More Information: www.freescale.com

6 Embedded PowerPC Debugging

This chapter explains how to use the CodeWarrior™ tools to debug programs for the mobileGT™ processor family.

The chapter covers those aspects of debugging that are specific to the mobileGT™ platform. See the *IDE User Guide* for debugger information that applies to all CodeWarrior products.

The sections are:

- Supported Remote Connections
- Special Debugger Features
- Using MetroTRK
- Debugging ELF Files

Supported Remote Connections

A remote connection is used for debugging an application on the remote target system. The EPPC debugger uses a plug-in architecture for communicating to the target.

There are several remote connection types included in the default installation. Before you debug a project, you need to specify the settings for the remote connection you selected while creating the project.

NOTE If you want to debug via another connection, you must first write a plug-in that converts the CodeWarrior API to the API of the new connection that you are using.

To specify the settings for a remote connection:

1. Display the **Remote Connections** panel.

a. Select **Edit > Preferences**

The **IDE Preferences** window appears.

b. Select the **Remote Connections** item in the **IDE Preference Panels** list.

The **Remote Connections** panel (Figure 6.1) appears.

Figure 6.1 Remote Connections Panel

Remote Connections

Name	Type
Abatron Serial	Serial
Abatron TCP/IP	TCP/IP
MSI BDM Raven	Parallel
MSI Wiggler	Parallel
MetroTRK	Serial
P&E BDM	Parallel
PowerTAP PRO CCS	PowerTAP TCP/IP
WireTAP 8xx	Parallel
WireTAP CCS	WireTAP Parallel

Add... Change... Remove

2. Select the remote connection name.

a. Click the remote connection name for which you want to specify settings.

b. Click **Change**

A dialog box in which you can specify connection settings appears.

The Name text box in the connection settings dialog box displays the remote connection name. Additionally, the appropriate debugger interface and the remote connection type are already selected in the Debugger and Connection Type listboxes, respectively.

The other remote connection settings for each remote connection included in the default installation are described in these sections:

- Abatron Remote Connections
- MetroTRK
- MSI BDM Raven/MSI Wiggler
- P&E BDM
- PowerTAP PRO CCS

For More Information: www.freescale.com

- WireTAP 8xx
- WireTAP CCS

Abatron Remote Connections

Figure 6.2 shows the dialog box in which you specify the connection settings for a serial Abatron remote connection.

Figure 6.2 Serial Type Abatron Remote Connection Dialog Box

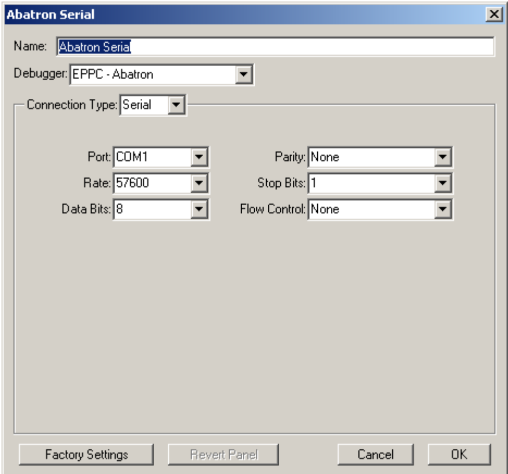
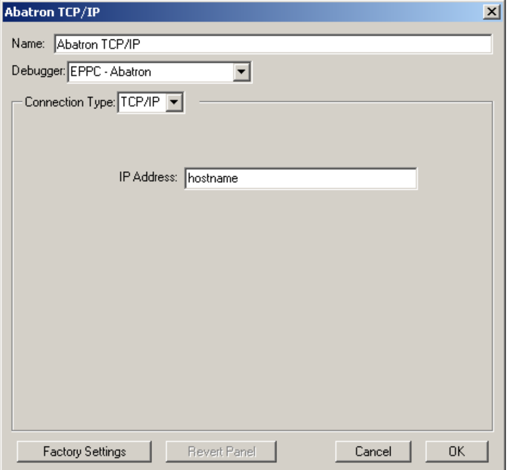


Figure 6.3 shows the dialog box in which you specify the connection settings for a TCP/IP Abatron remote connection.

Freescale Semiconductor, Inc.

Figure 6.3 TCP/IP Type Abatron Remote Connection Dialog Box



IP Address

The IP Address text box specifies the IP address of the Abatron device.

Port

Use the Port listbox to select the serial port on your computer that the debugger uses to communicate with the target hardware.

The options are COM1, COM2, COM3, and COM4.

Rate

Use the Rate listbox to select the serial baud rate for communicating with the target hardware.

Data Bits

Use the Data Bits listbox to select the number of data bits per character. The default value is 8.

For More Information: www.freescale.com

Parity

Use the Parity listbox to select whether you want an odd parity bit, an even parity bit, or none. The default value is none.

Stop Bits

Use the Stop Bits listbox to select the number of stop bits per character. The default value is 1.

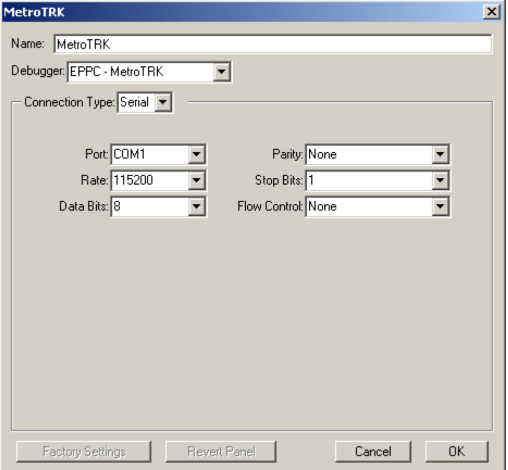
Flow Control

Use the Flow Control listbox to select whether you want hardware flow control, software flow control, or none. The default value is none.

MetroTRK

Figure 6.4 shows the dialog box in which you specify the connection settings for a MetroTRK remote connection.

Figure 6.4 MetroTRK Remote Connection Dialog Box



Freescale Semiconductor, Inc.

Port

Use the Port listbox to select the serial port on your computer that the debugger uses to communicate with the target hardware.

The options are COM1, COM2, COM3, and COM4.

Rate

Use the Rate listbox to select the serial baud rate for communicating with the target hardware.

Data Bits

Use the Data Bits listbox to select the number of data bits per character. The default value is 8.

Parity

Use the Parity listbox to select whether you want an odd parity bit, an even parity bit, or none. The default value is none.

Stop Bits

Use the Stop Bits listbox to specify the number of stop bits per character. The default value is 1.

Flow Control

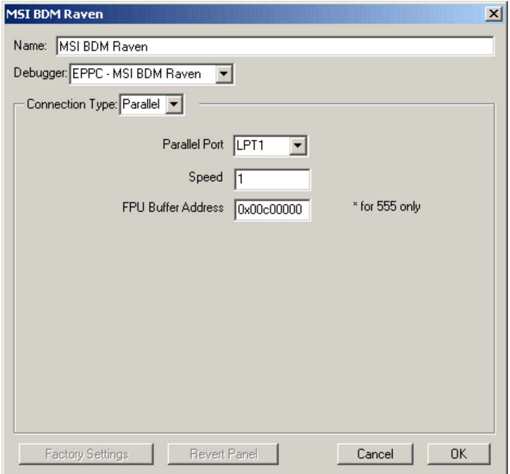
Use the Flow Control listbox to select whether you want hardware flow control, software flow control, or none. The default value is none.

MSI BDM Raven/MSI Wiggler

The remote connection settings for MSI BDM Raven and MSI Wiggler are the same. Figure 6.5 shows the dialog box in which you specify the connection settings for the remote connection settings for these debug devices.

For More Information: www.freescale.com

Figure 6.5 MSI BDM Raven/MSI Wiggler Remote Connection Dialog Box



Parallel Port

From the Parallel Port listbox, select the parallel port of your PC to which your MSI debug device is connected.

Speed

Use the default speed of 1.

FPU Buffer Address

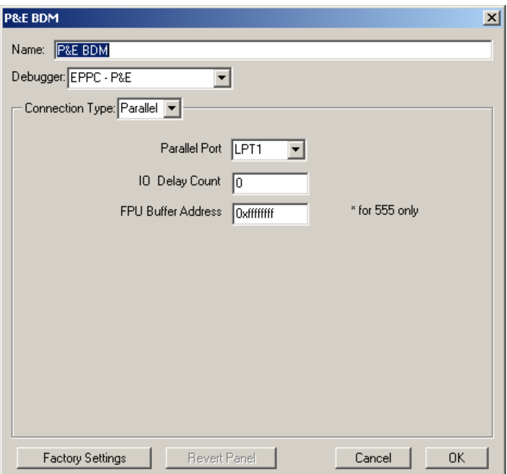
Unused by the mobileGT processor family. Leave default value unchanged.

Freescale Semiconductor, Inc.

P&E BDM

Figure 6.6 shows the dialog box in which you specify the connection settings for a P&E BDM remote connection.

Figure 6.6 P&E BDM Remote Connection Dialog Box



Parallel Port

From the Parallel Port listbox, select the parallel port of your PC to which the P&E BDM is connected.

IO Delay Count

The value specified in the IO Delay Count text box controls the communication speed between the host PC and the target board. The default value displayed in this text box is 0. If the communication speed of the target board is slower than that of the host PC, a higher value allows the shift clock out of the PC to be slowed down and reduce the communication speed.

FPU Buffer Address

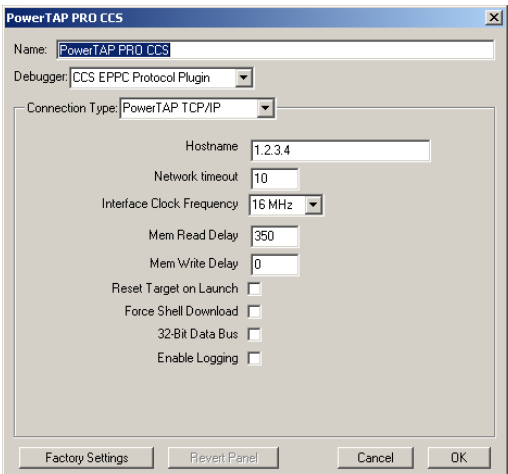
Unused by the mobileGT processor family. Leave default value unchanged.

For More Information: www.freescale.com

PowerTAP PRO CCS

Figure 6.7 shows the dialog box in which you specify the settings for a PowerTAP PRO CCS remote connection.

Figure 6.7 PowerTAP PRO CCS Remote Connection Dialog Box



Hostname

In the Hostname text box, type the host name or IP address that you assigned to the PowerTAP PRO device during the emulator setup.

Network timeout

Use this text box to define the amount of time (in seconds) for the debugger to wait for data from the PowerTAP Pro before issuing an error.

The default value is 10. Increase this value if you receive frequent timeout errors.

Interface Clock Frequency

Use the Interface Clock Frequency listbox to select the clock frequency for the JTAG interface. The default clock frequency is 16 MHz.

For More Information: www.freescale.com

Freescale Semiconductor, Inc.

Mem Read Delay

Specifies the number of additional processor cycles inserted as a delay for completion of read memory operations. The range of values that can be entered is 0-65024 cycles. Use the default delay of 350 cycles.

Mem Write Delay

Specifies the number of additional processor cycles inserted as a delay for memory write operations. The range of values that can be entered is 0-65024 cycles. Use the default delay of 350 cycles.

Reset Target on Launch

Check the Reset Target on Launch checkbox to send a reset signal to the target board when you launch the debugger.

Force Shell Download

Check the Force Shell Download checkbox if you wish to reload the PowerTAP PRO shell at each debugger connection.

32-Bit Data Bus

Check the 32-Bit Data Bus checkbox if you wish to use the 32-bit data bus.

Enable Logging

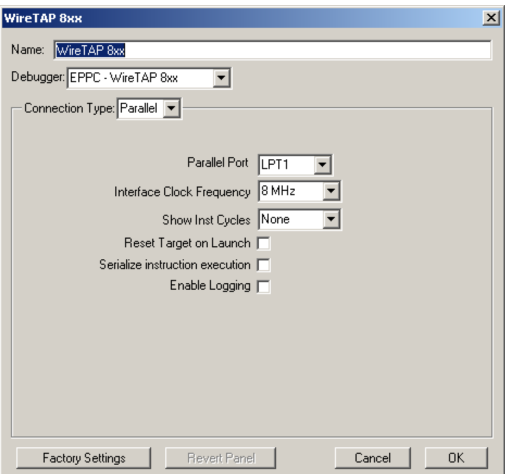
Check the Enable Logging checkbox to have the IDE display a log of all the debug transactions. If this checkbox is checked, a Protocol logging window appears when the debugger connects to the target.

For More Information: www.freescale.com

WireTAP 8xx

Figure 6.8 shows the dialog box in which you specify the settings for a WireTAP 8xx remote connection.

Figure 6.8 WireTAP 8xx Remote Connection Dialog Box



Parallel Port

From the Parallel Port listbox, select the parallel port of your PC to which the WireTAP is connected.

Interface Clock Frequency

Use the Interface Clock Frequency list box to select the clock frequency for the JTAG. The recommended clock frequency is 8 MHz.

Show Inst Cycles

Use the Show Inst Cycles list box to select which show cycles are performed (All, Flow or Indirect, None).

Reset Target on Launch

Check the Reset Target on Launch checkbox to send a reset signal to the target board when you launch the debugger.

Serialize Instruction Execution

Check the Serialize instruction execution checkbox if you wish to serialize instruction execution.

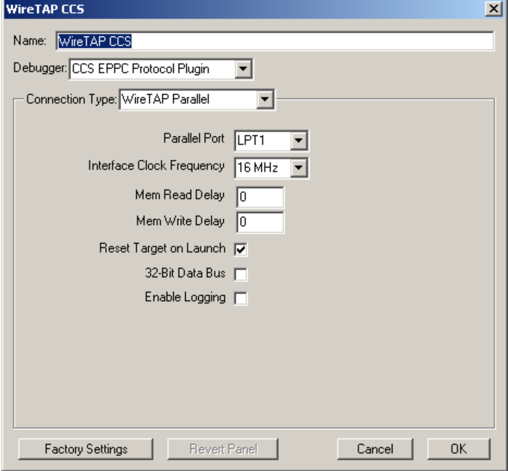
Enable Logging

Check the Enable Logging checkbox to have the IDE display a log of all the debug transactions. If this checkbox is checked, a Protocol logging window appears when the debugger connects to the target.

WireTAP CCS

Figure 6.9 shows the dialog box in which you specify the connection settings for a WireTAP CCS remote connection.

Figure 6.9 WireTAP CCS Remote Connection Dialog Box



Parallel Port

From the Parallel Port listbox, select the parallel port of your PC to which the WireTAP is connected.

Interface Clock Frequency

Use the Interface Clock Frequency listbox to select the clock frequency for the JTAG interface. The default clock frequency is 16 MHz.

Mem Read Delay

Specifies the number of additional processor cycles inserted as a delay for completion of read memory operations. The range of values that can be entered is 0-65024 cycles. Use the default delay of 0 cycles.

Mem Write Delay

Specifies the number of additional processor cycles inserted as a delay for memory write operations. The range of values that can be entered is 0-65024 cycles. Use the default delay of 0 cycles.

Reset Target on Launch

Check the Reset Target on Launch checkbox to send a reset signal to the target board when you launch the debugger.

32-Bit Data Bus

Check the 32-Bit Data Bus checkbox if you wish to use the 32-bit data bus.

Enable Logging

Check the Enable Logging checkbox to have the IDE display a log of all the debug transactions. If this checkbox is checked, a Protocol logging window appears when the debugger connects to the target.

For More Information: www.freescale.com

Freescale Semiconductor, Inc.

Special Debugger Features

This section explains debugger features that are not found in the *IDE User Guide*. These features are unique to this platform target and enhance the debugger especially for Embedded PowerPC development.

- Displaying Registers
- EPPC-Specific Debugger Features
- Register Details

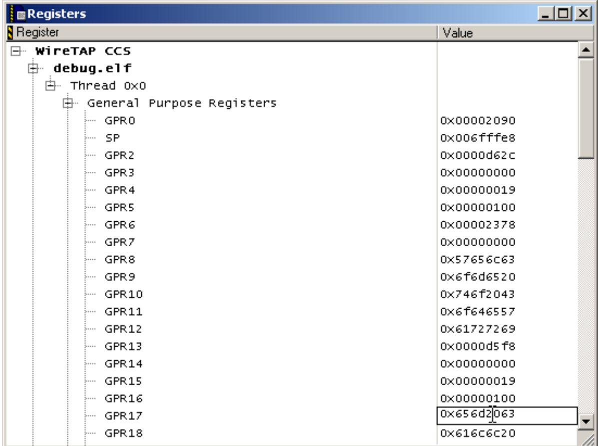
Displaying Registers

Use the **Registers** window to view and update the contents of the registers of the processor on your evaluation 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 6.10 shows the **Registers** window with the General Purpose Registers tree element expanded.

Figure 6.10 Registers Window



For More Information: www.freescale.com

EPPC-Specific Debugger Features

The debugger included with the CodeWarrior Development Studio, mobileGT Processor Edition includes some Embedded PowerPC-specific features. The **Debug > EPPC** menu makes these features available. The sections that follow explain how to use each option of this menu.

- Set Stack Depth
- Change IMMR
- Change MBAR
- Soft Reset
- Hard Reset
- Load/Save Memory
- Fill Memory
- Save/Restore Registers
- Enable Address Translations
- Watchpoint Type
- Breakpoint Type

Set Stack Depth

Select the Set Stack Depth command to set the depth of the stack to read and display. Showing all levels of calls when you are examining function calls several levels deep can sometimes make stepping through code more time-consuming. Therefore, you can use this menu option to reduce the depth of calls that the CodeWarrior IDE displays.

For More Information: www.freescale.com

Freescale Semiconductor, Inc.

Change IMMR

Use the 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.

Change MBAR

Use the Change MBAR command to assign an address to the MBAR (Memory Base Address) register of a member of the MGT5100 processor family.

NOTE The Change MBAR command is enabled only if you first select the 5100 in the EPPC Processor target settings panel.
For other EPPC processors, you change the memory base address by writing a value to the MBAR SPR register (SPR 311).

Soft Reset

Use the Soft Reset command to send a soft reset signal to the target processor.

NOTE The Soft Reset command is enabled only if the debug hardware you are using supports it.

Hard Reset

Use the Hard Reset command to send a hard reset signal to the target processor.

NOTE The Hard Reset command is enabled only if the debug hardware you are using supports it.

Load/Save Memory

The Load/Save Memory command:

- Loads the specified amount of data from a binary file on the host and writes this data to the target board’s memory starting at the specified address.

For More Information: www.freescale.com

- Reads the specified amount of data from the specified address of the target board's memory and saves this data in a binary file on the host.

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.

Fill Memory

Use the Fill Memory command to fill a particular memory location with data of particular size and type. This command lets you write a set of characters to a particular memory location on the target by repeatedly copying the characters until the specified fill size has been reached.

Save/Restore Registers

Use the Save/Restore Registers command to save the values of registers to a text file and set the values of registers from a text file. The command lets you specify the particular group of registers to save or restore.

Enable Address Translations

Use the 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 Address translation commands for a definition of the syntax and effect of an address translation command.

For More Information: www.freescale.com

Freescale Semiconductor, Inc.

To perform MMU debugging, follow these steps:

1. Add the required address translation commands to a 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 created 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.

Address translation commands

The syntax of an address translation command is:

`translate` *`virtual_address physical_address address_range`*

`virtual_address`

Address of the first byte of the virtual address range to translate.

`physical_address`

Address of the first byte of the physical address range to which to translate virtual addresses.

`address_range`

The size (in bytes) of the address range to translate.

For example, consider this `translate` command:

```
translate 0xC0000000   0x00000000   0x100000
```

This command:

- Defines a 1 MB address range (because `0x100000` bytes is 1 MB).

For More Information: www.freescale.com

- 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`.

Auto-enabling address translation

By default, address translations are disabled. However, if you linked your executable with virtual addresses, you must enable address translation *before* downloading the executable to the target device. To auto-enable address translations, add this statement to your memory configuration file:

```
AutoEnableTranslations true
```

Watchpoint Type

Select the Watchpoint Type command to indicate the type of watchpoint to set from among these options:

- Read

  Program execution stops at the watchpoint when your program reads from memory at the watch address.

- Write

  Program execution stops at the watchpoint when your program writes to memory at the watch address.

- Read/Write

  Program execution stops at the watchpoint when your program accesses memory at the watch address.

NOTE The Watchpoint Type command is available only if both the selected processor and your debug hardware support it.

Breakpoint Type

Select the Breakpoint Type command to indicate the type of breakpoint to set from among these options:

- Software

  The CodeWarrior software sets the breakpoint to target memory. When program execution reaches the breakpoint and stops, the breakpoint is removed. The breakpoint can only be set in writable memory.

For More Information: www.freescale.com

Freescale Semiconductor, Inc.

- Hardware

  Selecting the Hardware menu option sets a processor-dependent breakpoint. Hardware breakpoints use registers.

- Auto

  Selecting the Auto menu option causes the CodeWarrior tools to try to set a software breakpoint and, if that fails, to try to set a hardware breakpoint.

NOTE The Breakpoint Type command is available only if both the selected processor and your debug hardware support it.

Setting Hardware Breakpoints

To set a hardware breakpoint:

1. Connect to the target board.
2. Select **Debug > EPPC > Breakpoint Type > Hardware**
3. Set a breakpoint.

Table 6.1 lists the number of breakpoints that can be set for various mobileGT processors. All the processors listed in the table support software breakpoints.

Table 6.1 mobileGT™ Processors—Number of Hardware Breakpoints

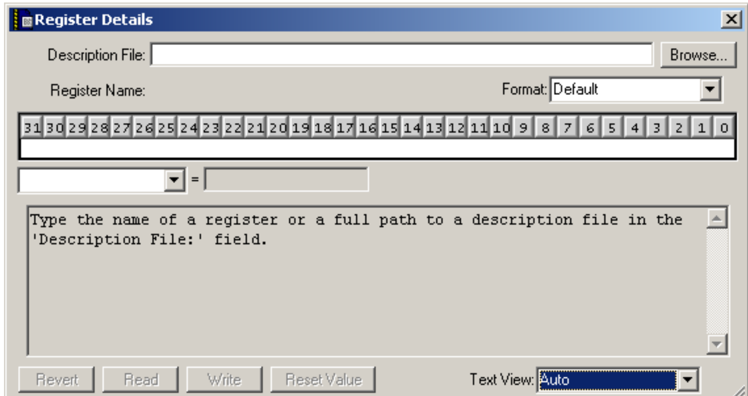
Processor	Number of Hardware Breakpoints
MGT5200	2
MPC823	4

Register Details

You can use the **Register Details** dialog box to view different mobileGT registers by specifying the name of the register description file. Selecting **View > Register Details** displays the **Register Details** dialog box (Figure 6.11).

For More Information: www.freescale.com

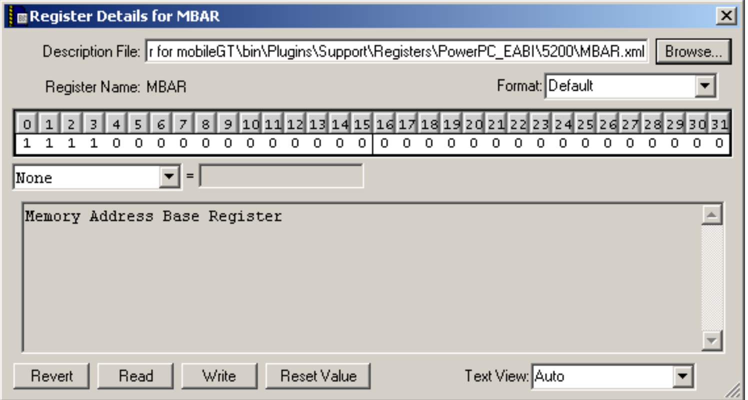
Figure 6.11 Register Details Dialog Box



After the CodeWarrior software displays the **Register Details** dialog box, type the name of the register description file in the Description File text box to display the applicable register and its values. (Alternatively, click **Browse** to display a dialog box you can use to find and select the required register description file.)

Figure 6.12 shows the **Register Details** dialog box displaying the MBAR (memory base address) register.

Figure 6.12 Register Details Dialog Box Showing the MBAR Register



You can change the format in which the CodeWarrior software displays the register using the Format listbox. In addition, when you click on different bit fields of the displayed register, the CodeWarrior software displays an appropriate description, depending on which bit or group of bits you choose. You also can change the text information that the CodeWarrior software displays by using the Text View listbox.

Freescale Semiconductor, Inc.

NOTE For more information, see the *CodeWarrior™ IDE User Guide*.

Using MetroTRK

This section briefly describes MetroTRK and provides information related to using MetroTRK with your CodeWarrior product. This section has these topics:

- MetroTRK Overview
- Connecting to the MetroTRK Debug Monitor
- MetroTRK Memory Configuration
- Using MetroTRK for Debugging
- Using MetroTRK with the Lite5200 Board

MetroTRK Overview

MetroTRK is a software debug monitor. MetroTRK resides on the target board with the program you are debugging and provides debug services to the host debugger. MetroTRK connects to the host computer through a serial port.

You use MetroTRK to download and debug applications built with CodeWarrior for Embedded PowerPC.

The CodeWarrior software installs the source code for MetroTRK, as well as ROM images and project files for several pre-configured builds of MetroTRK.

The board-specific directories that contain the MetroTRK source code are here:

```
installDir\PowerPC_EABI_Tools\
MetroTRK\Processor\ppc\Board\board_mfr_name\board_name
```

where `board_mfr_name` is a placeholder for the name of a manufacturer of a supported evaluation board and `board_name` is a placeholder for the name of a particular board.

If you are using a board other than a supported board, you may need to customize the MetroTRK source code for your board configuration. For more information, see the *MetroTRK Reference*.

To modify a version of MetroTRK, find an existing MetroTRK project for your supported 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 revert back to the original version on your CodeWarrior CD.

For More Information: www.freescale.com

Connecting to the MetroTRK Debug Monitor

This section presents high-level steps for connecting to a debug monitor on the target board by using a serial port.

The type of serial cable connection that you can use depends on your target board. Table 6.2 lists the type of serial cable connection required for various embedded PowerPC target boards.

Table 6.2 Serial Cable Connection Type for Target Boards

EPPC Board	Serial Cable Connection Type
Motorola Lite5200, rev. I	Null modem
Motorola Lite5200, rev. G	Null modem
Motorola MPC 823 FADS	Straight serial

To connect to the debug monitor on the target board:

1. Ensure that your target board has a debug monitor.

   If your debug monitor has not been previously installed on the target board, burn the debug monitor to ROM or use another method, such as the flash programmer, to place MetroTRK or another debug monitor in flash memory.

2. Check whether the debug monitor is in flash memory or ROM.

   a. Connect the serial cable to the target board.

   b. Use a terminal emulation program to verify that the serial connection is working. Set the baud rate in the terminal emulation program to the correct baud rate and set the serial port to 8 data bits, one stop bit, and no parity.

   c. Reset the target board. When you reset the target board, the terminal emulation program displays a message that provides the version of the program and several strings that describe MetroTRK.

3. If you plan to use console I/O, ensure that your project contains appropriate libraries 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.

For More Information: www.freescale.com

Freescale Semiconductor, Inc.

NOTE See the `project read me` file regarding MetroTRK options.

MetroTRK Memory Configuration

This section explains the default memory locations of the MetroTRK code and data sections and of your target application.

This section contains these topics:

- Locations of MetroTRK RAM Sections
- MetroTRK Memory Map

Locations of MetroTRK RAM Sections

Several MetroTRK RAM sections exist. You can reconfigure some of the MetroTRK RAM sections.

This section contains these topics:

- Exception Vectors
- Data and Code Sections
- The Stack

Exception Vectors

For a ROM-based MetroTRK, the MetroTRK 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 PowerPC, the exception vector must start at `0x000100` (which is in low memory) and spans 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 set location.

Data and Code Sections

The standard configuration for MetroTRK uses approximately 29 KB of code space as well as 8 KB of data space.

For More Information: www.freescale.com

In the default ROM-based implementation of MetroTRK used with most supported target boards, no MetroTRK code section exists in RAM because the code executes directly from ROM. However, for some PowerPC target boards, some MetroTRK code does reside in RAM, usually for one of these reasons:

- Executing from ROM is slow enough to limit the MetroTRK data transmission rate (baud rate)
- For some 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 MetroTRK data section.

You can change the location of the data and code sections in your MetroTRK 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 MetroTRK stack resides in high memory and grows downward. The default implementation of MetroTRK requires a maximum of 8KB of stack space.

You can change the location of the stack section by modifying settings of the **EPPC Linker** settings panel and rebuilding the MetroTRK project.

MetroTRK Memory Map

For more information on the MetroTRK memory map, see the board specific information provided with the MetroTRK source code.

Using MetroTRK for Debugging

To use MetroTRK for debugging, you must load it on your target board in ROM.

For More Information: www.freescale.com

Freescale Semiconductor, Inc.

MetroTRK 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 MetroTRK that you are using.

After you load MetroTRK on the target board, you can use the debugger to upload and debug your application if the debugger is set to use MetroTRK.

NOTE Before using MetroTRK with hardware other than the supported reference boards, see the *MetroTRK Reference*.

Using MetroTRK with the Lite5200 Board

The sections listed below explain how to build and use a MetroTRK image with the Motorola Lite5200 evaluation board.

- Creating a MetroTRK Image for the Lite5200 Board
- Writing the MetroTRK Image to Flash Memory
- Starting MetroTRK on the Lite5200 Board
- Using MetroTRK to Debug on the Lite5200

Creating a MetroTRK Image for the Lite5200 Board

To create a MetroTRK image for the Lite5200 evaluation board, follow these steps:

1. Select **Start > Programs > Metrowerks CodeWarrior > CodeWarrior for mobileGT V8.1 > CodeWarrior IDE**

   The CodeWarrior IDE starts and displays its main window.

2. Open the CodeWarrior project file `trk_Lite5200.mcp`.

   This file is here:
   `installDir\PowerPC_EABI_Tools\`
   `MetroTRK\Processor\ppc\Board\motorola\Lite5200`

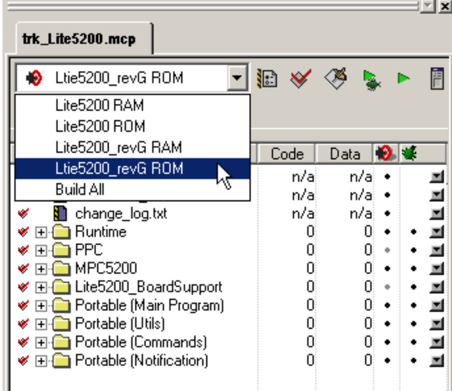
   where `installDir` is a placeholder for the path in which you installed your CodeWarrior product.

3. Select the build target named Lite5200 ROM or Lite5200_revG ROM (depending on the revision of the Lite5200 board you have).

   See Figure 6.13.

For More Information: www.freescale.com

Figure 6.13 The Lite5200 MetroTRK Project Window



4. Select **Project > Make**

   The CodeWarrior IDE builds a MetroTRK S-Record image named `TRK_Lite5200_ROM.mot` or `TRK_Lite5200_revG_ROM.mot` (depending on the build target you selected) and puts it in this directory:

   `installDir``\PowerPC_EABI_Tools\`
   `MetroTRK\Processor\ppc\Board\motorola\Lite5200\Binary`

NOTE The MetroTRK image built by the `trk_Lite5200.mcp` project writes to the serial port at 115200 bits per second. To use a different speed, you must change the value of the constant `TRK_BAUD_RATE` in the file `target.h` from 115200 to the desired value. In addition, you must select the same value in the Rate listbox of the MetroTRK remote connection dialog box.

Writing the MetroTRK Image to Flash Memory

To write the Lite5200 MetroTRK S-Record file to the Lite5200 board's flash memory, follow these steps:

1. Connect your debug hardware to the Lite5200 board and to your PC.
2. Start the CodeWarrior IDE.
3. From the IDE's menu bar, select **Tools > Flash Programmer**

   The IDE displays the **Flash Programmer** window.

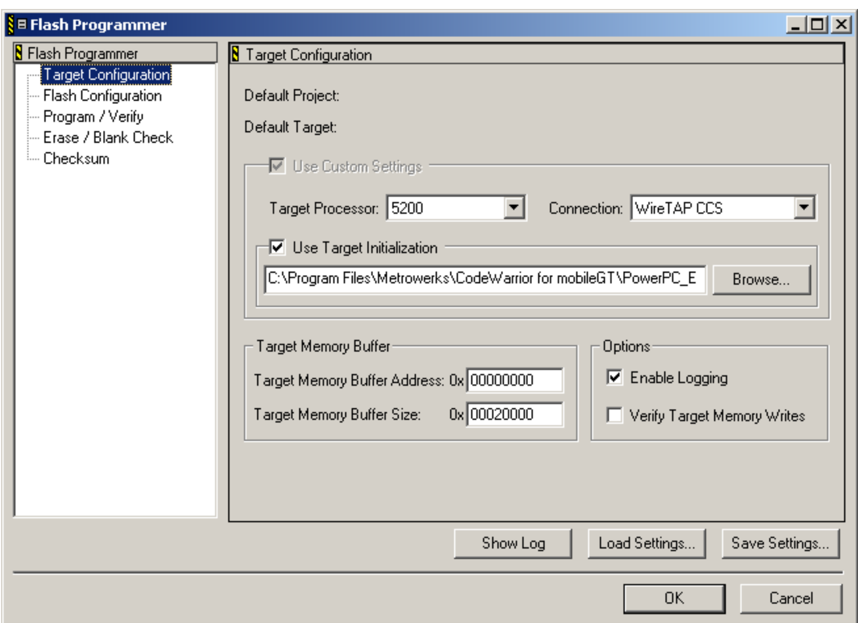
For More Information: www.freescale.com

Freescale Semiconductor, Inc.

4. From the list on the left side of the **Flash Programmer** window, select Target Configuration

   The Target Configuration panel appears on the right side of the **Flash Programmer** window. (See Figure 6.14.)

Figure 6.14 The Target Configuration Panel of the Flash Programmer Window



5. Click **Load Settings**

   The **Load Settings** dialog box appears.

6. Use the **Load Settings** dialog box to select the flash programmer configuration file for the Lite5200 board.

   The configuration file for the Lite5200, rev. I (which has 8 MB of flash) is:

   `*installDir*\bin\Plugins\Support\`
   `Flash_Programmer\EPPC\Lite5200_8MB_flash.xml`

   The configuration file for the Lite5200, rev. G (which has 16 MB of flash) is:

   `*installDir*\bin\Plugins\Support\`
   `Flash_Programmer\EPPC\Lite5200_16MB_flash.xml`

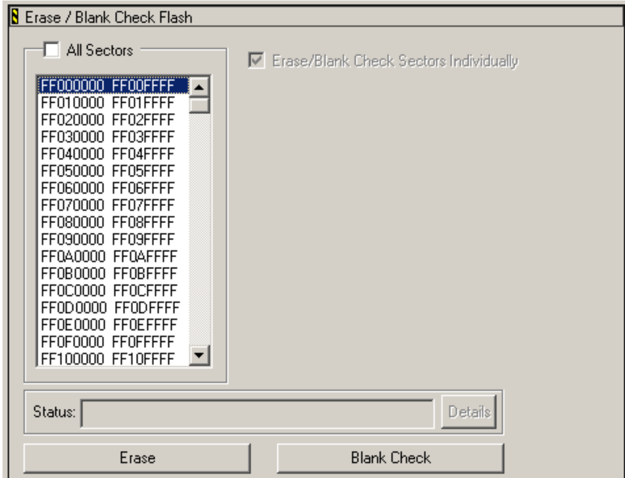
7. From the Connection listbox, select the debug hardware you are using.

For More Information: www.freescale.com

8. From the list on the left side of the **Flash Programmer** window, select Erase / Blank Check

   The Erase / Blank Check panel appears on the right side of the **Flash Programmer** window. (See Figure 6.15.)

Figure 6.15 The Erase / Blank Check Panel of the Flash Programmer Window



9. Select the first sector in the listbox on the left of the Erase / Blank Check panel.
10. Click **Erase**

    The Flash Programmer erases the specified sector of flash memory on your Lite5200 board.
11. Click **Blank Check**

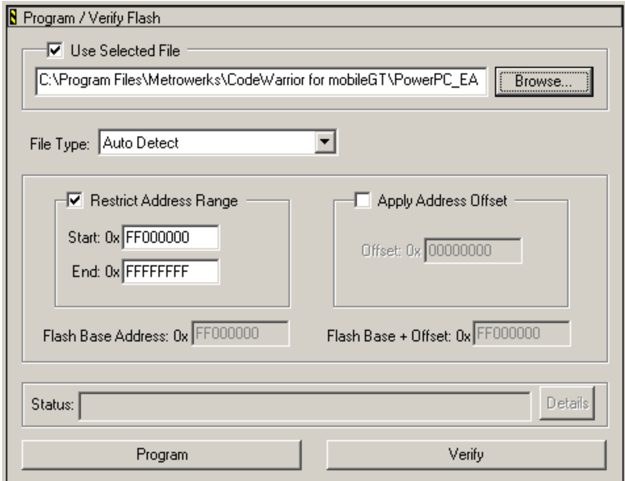
    The Flash Programmer verifies that the specified sector was erased.
12. From the list on the left side of the **Flash Programmer** window, select Program / Verify

    The Program / Verify panel appears on the right side of the **Flash Programmer** window. (See Figure 6.16.)

For More Information: www.freescale.com

Freescale Semiconductor, Inc.

Figure 6.16 The Program / Verify Panel of the Flash Programmer Window



13. Check the Use Selected File checkbox.

    The Use Selected File text box enables.

14. Click the **Browse** button on the right side Use Selected File text box.

    The **Select File to Program** dialog box appears.

15. Use this dialog box to select the MetroTRK S-Record file created previously (`TRK_Lite5200_ROM.mot` or `TRK_Lite5200_revG_ROM.mot`).

    These files are in this directory:

    `installDir``\PowerPC_EABI_Tools\`
    `MetroTRK\Processor\ppc\Board\motorola\Lite5200\Binary\`

16. Click **Program**

    The Flash Programmer writes the selected MetroTRK S-Record file to flash memory of your Lite5200 board.

17. When the Flash Programmer completes the write operation, click **Verify**

    The Flash Programmer verifies that the selected MetroTRK S-record file was written to flash memory without error.

18. Click **OK**

    The **Flash Programmer** window closes.

For More Information: www.freescale.com

Starting MetroTRK on the Lite5200 Board

1. On your Lite5200 board, put the **B H/L** (boot high/low) jumper in the **L** (low) position.

NOTE The **B H/L** jumper must be in the **L** (low) position or the MetroTRK image will not execute.

2. Connect a null modem serial cable to the Lite5200 board and to your PC.
3. Disconnect the JTAG connector of your debug hardware from the Lite5200 board.
4. Start a terminal emulation program and configure it as follows:
   - Bits per second — `115200`
   - Data bits — `8`
   - Parity — `None`
   - Stop bits — `1`
   - Flow control — `None`
5. On the Lite5200, press the **Reset** button to reboot the board.

   This message appears in the terminal emulator window:

   ```
   Welcome to Metrowerks Target Resident Kernel for mot_Lite5200
   Version 3.37 implementing MetroTRK API version 1.10
   ```
6. Exit the terminal emulator program.

   MetroTRK is now ready for use with the CodeWarrior debugger.

NOTE MetroTRK is a debug monitor and therefore runs on a board. As a result, you must ensure that your application does not overwrite the memory areas (code, data, exception handlers) used by MetroTRK or use the serial port used by MetroTRK.
These RAM locations are used by MetroTRK:
data: `0x03fe2000`, length `0x3000`
code: `0x00003000`, length `0x5200`
exception handlers: `0x00000000`, length `0x2000`

Freescale Semiconductor, Inc.

Using MetroTRK to Debug on the Lite5200

To debug an application on the Lite5200 board using MetroTRK, follow these steps:

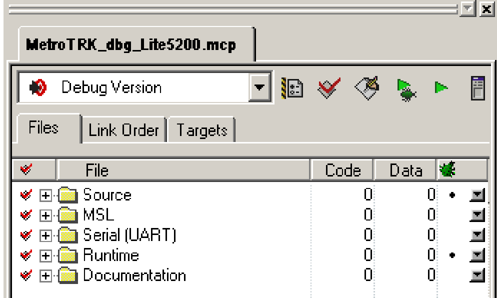
1. Open the example debug example project `MetroTRK_dbg_Lite5200.mcp`.

   This project file is here:
   `installDir\(CodeWarrior_Examples)\`
   `PowerPC_EABI\MetroTRK_debug\5200`

   This project uses the MetroTRK debug protocol to write a welcome message to the terminal I/O window of the CodeWarrior IDE.

   Figure 6.17 shows the project window of the MetroTRK example debug project.

Figure 6.17 Project Window—MetroTRK Example Debug Project



2. Select **Project > Make**

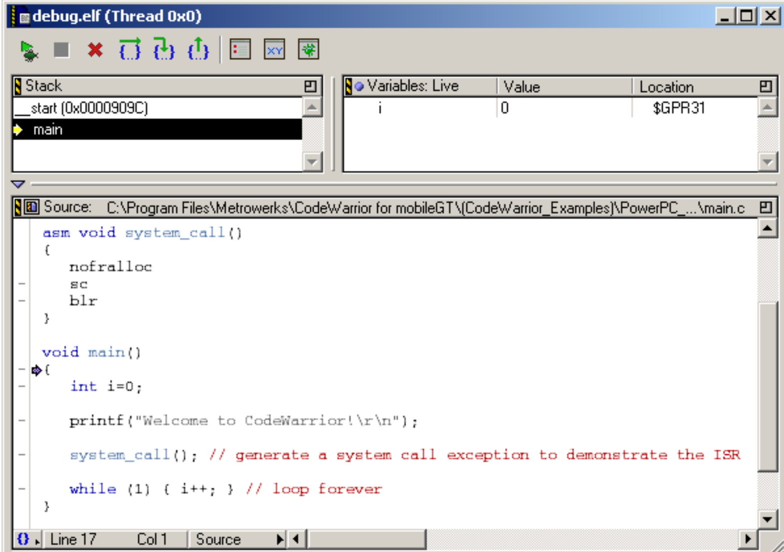
   The IDE builds the project and generates an executable named `debug.elf`.

3. Select **Project > Debug**

   The CodeWarrior debugger downloads `debug.elf` to the Lite5200, halts execution at the first instruction of `main()`, and displays the debugger window. (See Figure 6.18.)

For More Information: www.freescale.com

Figure 6.18 Debugger Window



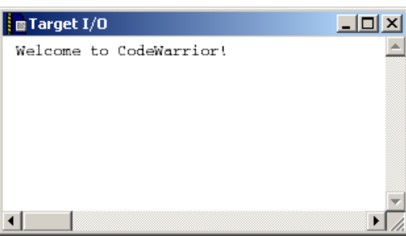
4. In the debugger window, click the Step Over icon.

   The debugger executes the current statement and halts at the next statement.

5. In the debugger window, click the Run icon.

   The program writes a message to the **Target I/O** window (Figure 6.19) and generates a system call exception.

Figure 6.19 Target I/O Window of Debugger



6. In the debugger window, click the Run icon again.

   The program enters an infinite loop.

Freescale Semiconductor, Inc.

7. In the debugger window, click the Break icon.

   The debugger halts program execution at the next statement to execute.

8. In the debugger window, click the Kill Thread icon.

   The debugger window closes; the debug session is complete.

Debugging ELF Files

You can use the CodeWarrior debugger to debug an ELF file that you previously created and compiled in a different environment than the CodeWarrior IDE. Before you open the 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.

This section contains these topics:

- Preparing to Debug an ELF File
- Customizing the Default XML Project File
- Debugging an ELF File
- ELF File Debugging: Additional Considerations

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**.

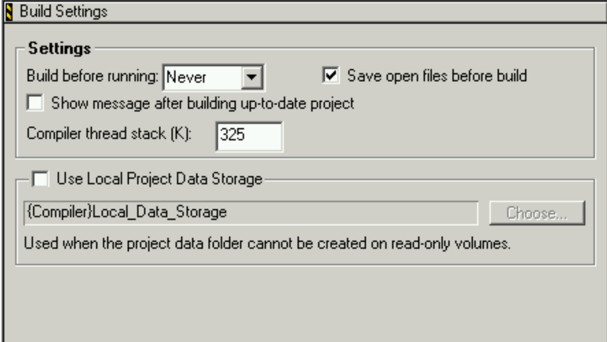
   The **IDE Preferences** window appears.

2. In the **IDE Preference Panels** list, click the Build Settings item.

   The **Build Settings** panel (Figure 6.20) appears.

For More Information: www.freescale.com

Figure 6.20 Build Settings Panel

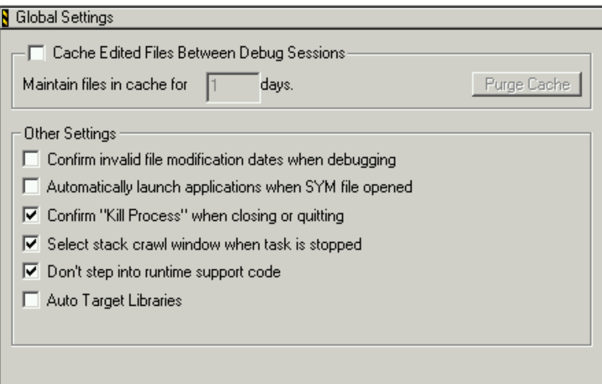


3. From the Build before running listbox, select Never.

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** panel (Figure 6.21) appears.

Figure 6.21 Global Settings Panel



5. Clear the Cache Edited Files Between Debug Sessions checkbox.

6. Close the **IDE Preferences** window.

That's it. You have successfully examined the relevant IDE preference settings and modified them as required.

Freescale Semiconductor, Inc.

Customizing the Default XML Project File

When you debug an ELF file, the CodeWarrior software uses the following default XML project file to create a CodeWarrior project for the ELF file.

`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 CodeWarrior software overwrites the existing `EPPC_Default_Project.XML` file if you customize it again for a different target board or debugging setup. If you want to preserve the file that you originally customized for later use, 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 **OK**

   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.

For More Information: www.freescale.com

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 those 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. Drag the ELF file icon (with symbolics) to the IDE.

   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.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.

2. (Optional) Check whether the target settings in the new project are satisfactory.

3. Begin debugging.

   Select **Project > Debug**

NOTE For more information on debugging, see *IDE User 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.

Freescale Semiconductor, Inc.

ELF File Debugging: Additional Considerations

This section, which explains information that is useful when debugging ELF files, contains these topics:

- Deleting old access paths from an ELF-created project
- Removing files from an ELF-created project
- Recreating an ELF-created project

Deleting old access paths from an ELF-created project

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** 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 an ELF-created project

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 an ELF-created project

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.

For More Information: www.freescale.com

7 C/C++ Compiler and Linker

This chapter explains how to use the CodeWarrior™ Embedded PowerPC C/C++ compiler and linker.

The *back-end* of the compiler refers to the module that generates code for the target processor. *Front-end* refers to the module that parses and interprets source code.

The sections are:

- Integer and Floating-Point Formats
- Data Addressing
- Register Variables
- Register Coloring Optimization
- Pragmas
- EPPC Linker Issues
- Using __attribute__ ((aligned(?)))

NOTE This chapter contains references to Appendix A of the “Reference Manual,” of *The C Programming Language, Second Edition* (Prentice Hall) by Kernighan and Ritchie.
Table 7.1 lists other useful compiler and linker documentation.

Table 7.1 Other Compiler and Linker Documentation

For this topic...	Refer to...
How the CodeWarrior IDE implements the C/C++ language	*C Compilers Reference*
Using C/C++ Language and C/C++ Warnings settings panels	*C Compilers Reference*, “Setting C/C++ Compiler Options” chapter
Controlling the size of C++ code	*C Compilers Reference*, “C++ and Embedded Systems” chapter
Using compiler pragmas	*C Compilers Reference*, “Pragmas and Symbols” chapter

For More Information: www.freescale.com

Table 7.1 Other Compiler and Linker Documentation (*continued*)

Initiating a build, controlling which files are compiled, handling error reports	*IDE User Guide*, "Compiling and Linking" chapter
Information about a particular error	*Error Reference*, which is available online
Embedded PowerPC assembler	*Assembler Guide*
PowerPC EABI calling conventions	*System V Application Binary Interface, 3rd Edition*, published by UNIX System Laboratories, 1994 (ISBN 0-13-100439-5) *System V Application Binary Interface, PowerPC Processor Supplement*, published by Sun Microsystems and IBM, 1995

Integer and Floating-Point Formats

This section describes how the CodeWarrior C/C++ compilers implement integer and floating-point types for Embedded PowerPC processors. You also can read `limits.h` for more information on integer types, and `float.h` for more information on floating-point types.

The topics in this section are:

- Embedded PowerPC Integer Formats
- Embedded PowerPC Floating-Point Formats

Embedded PowerPC Integer Formats

Table 7.2 shows the size and range of the integer types for the Embedded PowerPC compiler.

Table 7.2 PowerPC Integer Types

For this type	Option setting	Size	Range
bool	n/a	8 bits	true or false
char	Use Unsigned Chars is off (see language preferences panel in the "C Compilers Guide.")	8 bits	-128 to 127
	Use Unsigned Chars is on	8 bits	0 to 255
signed char	n/a	8 bits	-128 to 127

For More Information: www.freescale.com

Table 7.2 PowerPC Integer Types

For this type	Option setting	Size	Range
unsigned char	n/a	8 bits	0 to 255
short	n/a	16 bits	-32,768 to 32,767
unsigned short	n/a	16 bits	0 to 65,535
int	n/a	32 bits	-2,147,483,648 to 2,147,483,647
unsigned int	n/a	32 bits	0 to 4,294,967,295
long	n/a	32 bits	-2,147,483,648 to 2,147,483,647
unsigned long	n/a	32 bits	0 to 4,294,967,295
long long	n/a	64 bits	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long long	n/a	64 bits	0 to 18,446,744,073,709,551,615

Embedded PowerPC Floating-Point Formats

Table 7.3 shows the sizes and ranges of the floating-point types for the embedded PowerPC compiler.

Table 7.3 PowerPC Floating-Point Data Types

Type	Size	Range
float	32 bits	1.17549e-38 to 3.40282e+38
double	64 bits	2.22507e-308 to 1.79769e+308
long double	64 bits	2.22507e-308 to 1.79769e+308

Data Addressing

You can increase the speed of your application by selecting different EPPC Processor and EPPC Target settings that affect what the compiler does with data fetches.

In absolute addressing, the compiler generates two instructions to fetch the address of a variable. For example:

```
int red;
int redsky;
```


For More Information: www.freescale.com

Freescale Semiconductor, Inc.

```
void sky()
{
  red = 1;
  redsky = 2;
}
```

becomes something similar to:

```
li   r3,1
lis  r4,red@ha
addi r4,r4,red@l
stw  r3,0(r4)
li   r5,2
lis  r6,redsky@ha
addi r6,r6,redsky@l
stw  r5,0(r6)
```

Each variable access takes two instructions and a total of four bytes to make a simple assignment. If we set the small data threshold in the **EPPC Target** panel to be at least the size of an `int`, we can fetch the variables with one instruction.

```
li   r3,1
stw  r3,red
li   r4,2
stw  r4,redsky
```

Because small data sections are limited in size you might not be able to put all of your application data into the small data and small data2 sections. We recommend that you make the threshold as high as possible until the linker reports that you have exceeded the size of the section.

If you do exceed the available small data space, consider using pooled data.

Because the linker can not deadstrip unused pooled data, you should:

1. Check the Generate Link Map and List Unused Objects checkboxes in the **EPPC Linker** panel.
2. Link and examine the map for data objects that are reported unused.
3. Delete or comment out those used definitions in your source.
4. Check the Pool Data checkbox.

The following example has a zero small data threshold.

```
lis   r3,...bss.0@ha
addi  r3,r3,...bss.0@l
li    r0,1
```


For More Information: www.freescale.com

```
stw   r0,0(r3)
li    r0,2
stw   r0,4(r3)
```

When pooled data is implemented, the first used variable of either the `.data`, `.bss` or `.rodata` section gets a two-instruction fetch of the first variable in that section. Subsequent fetches in that function use the register containing the already-loaded section address with a calculated offset.

NOTE You can access small data in assembly files with the two-instruction fetch used with large data, because any data on your board can be accessed as if it were large data. The opposite is not true; large data can never be accessed with small data relocations (the linker issues an error if you try to do so). Extern declarations of empty arrays (for example, `extern int red [];`) are always treated as if they were large data. If you know that the size of the array fits into a small data section, specify the size in the brackets.

Register Variables

The PowerPC compiler back-end automatically allocates local variables and parameters to registers based on to how frequently they are used and how many registers are available. If you are optimizing for speed, the compiler gives preference to variables used in loops.

The Embedded PowerPC back-end compiler also gives preference to variables declared to be `register`, but does not automatically assign them to registers. For example, the compiler is more likely to place a variable from an inner loop in a register than a variable declared register. See also, K&R, §A4.1, §A8.1

For information on which registers the compiler can use for register variables, see these documents:

- *System V Application Binary Interface, Third Edition*, published by UNIX System Laboratories, 1994 (ISBN 0-13-100439-5)
- *System V Application Binary Interface, PowerPC Processor Supplement*, published by Sun Microsystems and IBM, 1995
- *PowerPC Embedded Binary Interface, 32-Bit Implementation*. This document can be obtained at:

  `ftp://ftp.linuxppc.org/linuxppc/docs/EABI_Version_1.0.ps`

For More Information: www.freescale.com

Freescale Semiconductor, Inc.

Register Coloring Optimization

The PowerPC back-end compiler can perform a register optimization called *register coloring*. In this optimization, the compiler assigns different variables or parameters to the same register if you do not use the variables at the same time. In Listing 7.1, the compiler could place `i` and `j` in the same register:

Listing 7.1 Register coloring example

```
short i;
int j;
for (i=0; i<100;  i++) {   MyFunc(i);  }
for (j=0; j<1000; j++) {   OurFunc(j); }
```

However, if a line, such as the one below, appears anywhere in the function, the compiler recognizes that you are using `i` and `j` at the same time, so it places them in different registers:

```
int k = i + j;
```

The default register optimization performed by PowerPC compiler is register coloring.

If the **Global Optimizations** settings panel specifies the optimization level of 1 or greater, the compiler assigns all variables that fit into registers to virtual registers. The compiler then maps the virtual registers into physical registers by using register coloring. As previously stated, this method allows two virtual registers to exist in the same physical register.

When you debug a project, the variables sharing a register may appear ambiguous. In Listing 7.1, `i` and `j` would always have the same value. When `i` changes, `j` changes in the same way. When `j` changes, `i` changes in the same way.

To avoid confusion while debugging, use the **Global Optimizations** settings panel to set the optimization level to 0. This setting causes the compiler to allocate user-defined variables only to physical registers or place them on the stack. The compiler still uses register coloring to allocate compiler-generated variables.

Alternatively, you can declare the variables you want to watch as volatile.

NOTE The optimization level option in the **Global Optimizations** settings panel corresponds to the `global_optimizer` pragma. For more information, see the *C Compilers Reference*.

For More Information: www.freescale.com

Pragmas

This section lists pragmas supported by all Metrowerks PowerPC compilers and those supported by just Metrowerks PowerPC compilers for embedded PowerPC systems.

Table 7.4 lists the pragmas documented in the *C Compilers Reference* that are supported by all Metrowerks PowerPC C/C++ compilers.

Table 7.4 Pragmas Supported by All PPC Compilers—See *C Compilers Reference*

align	align_array_members
ANSI_strict	ARM_conform
auto_inline	bool
check_header_flags	cplusplus
cpp_extensions	dont_inline
dont_reuse_strings	enumsalwaysints
exceptions	extended_errorcheck
fp_contract	global_optimizer
has8bytebitfields	ignore_oldstyle
longlong	longlong_enums
mark	no_register_save_helpers
once	only_std_keywords
optimize_for_size	optimizewithasm
peephole	pop
precompile_target	push
readonly_strings	require_prototypes
RTTI	scheduling
static_inlines	syspath_once
trigraphs	unsigned_char
unused	warning_errors
warn_emptydecl	warn_extracomma
warn_hidevirtual	warn_illpragma
warn_implicitconv	warn_possunwant

For More Information: www.freescale.com

Table 7.4 Pragmas Supported by All PPC Compilers—See *C Compilers Reference*

warn_unusedarg	warn_unusedvar
wchar_type	

Table 7.5 lists the pragmas documented in this manual that are supported by all Metrowerks PowerPC C/C++ compilers.

Table 7.5 Pragmas Supported by All PPC Compilers—Documented in this Manual

opt_full_unroll_limit	opt_findoptimalunrollfactor
opt_unroll_count	opt_unrollpostloop
opt_unroll_instr_count	inline_max_auto_size
ppc_no_fp_blockmove	

opt_full_unroll_limit

```
#pragma opt_full_unroll_limit n|reset (n: 0..127, default: 8)
```

This pragma controls whether a loop is completely unrolled. A particular loop is completely unrolled if its number of iterations is less than or equal to `n`.

This pragma is ignored if the unroll loops optimization is disabled. Further, this pragma takes precedence over the pragmas `opt_findoptimalunrollfactor` and `opt_unroll_count`.

opt_findoptimalunrollfactor

```
#pragma opt_findoptimalunrollfactor on|off|reset (default: on)
```

This pragma instructs the optimizer to calculate the optimal unroll factor. The optimal unroll factor is the value that results in the fewest leftover iterations for the loops within a compilation unit.

The optimal unroll factor is bound by the current default unroll count. In other words, the optimal unroll factor calculated by the optimizer will be less than or equal to the value defined using the `opt_unroll_count` pragma.

The `opt_findoptimalunrollfactor` pragma is ignored if the unroll loops optimization is disabled. Further, this pragma takes precedence over the `opt_unroll_count` pragma.

For More Information: www.freescale.com

opt_unroll_count

```
#pragma opt_unroll_count n|reset (n: 0..127, default: 8)
```

This pragma defines the default loop unroll factor for the optimizer to use. If you turn off the pragma `opt_findoptimalunrollfactor`, the optimizer uses the unroll factor defined using the `opt_unroll_count` pragma.

This pragma is ignored if the unroll loops optimization is disabled.

opt_unrollpostloop

```
#pragma opt_unrollpostloop on|off|reset (default: on)
```

This pragma controls whether iterations that remain after a loop has been unrolled should be linearized.

This pragma is ignored if the unroll loops optimization is disabled.

opt_unroll_instr_count

```
#pragma opt_unroll_instr_count n|reset (n: 0..127 default: 100)
```

This pragma defines the size of the loop that the optimizer will unroll. A loop with with a number of nodes greater than `n` will *not* be unrolled.

This pragma is ignored if the unroll loops optimization is disabled.

inline_max_auto_size

```
#pragma inline_max_auto_size (n) (default: 800)
```

This pragma defines the maximum size of functions that are auto-inlined. The value of `n` corresponds roughly to the number of instructions in a function—functions that contain more than `n` instructions are not inlined.

This pragma is ignored if auto-inlining is disabled.

ppc_no_fp_blockmove

```
#pragma ppc_no_fp_blockmove on|off|reset (default: off)
```


For More Information: www.freescale.com

The default compiler behavior is to try to use any available floating-point registers to move data structures. Turn this pragma on to suppress this behavior.

Table 7.6 lists the pragmas supported by just Metrowerks C/C++ compilers for embedded PowerPC systems.

Table 7.6 Pragmas Supported by Just Embedded PowerPC Compilers

force_active	function_align
incompatible_return_small_structs	incompatible_sfpe_double_params
interrupt	pack
pooled_data	section

force_active

```
#pragma force_active on|off|reset
```

This pragma inhibits the linker from dead-stripping any variables or functions defined while the dead-stripping option is in effect. It should be used for interrupt routines and any other data structures which are not directly referenced from the program entry point, but which must be linked into the executable program for correct operation.

NOTE You cannot use the `force_active` pragma with uninitialized variables due to language restrictions related to tentative objects.

function_align

```
#pragma function_align 4 | 8 | 16 | 32 | 64 | 128
```

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

With the pragma `function_align`, you can select alignments from 4 (the default) to 128 bytes.

This pragma corresponds to **Function Alignment** listbox in the EPPC Processor settings panel.

incompatible_return_small_structs

```
#pragma incompatible_return_small_structs on|off|reset
```

This pragma makes object files generated by CodeWarrior compilers more compatible with object files generated by GNU Compiler Collection (GCC) compilers.

As per PowerPC EABI settings, structures that are up to 8 bytes in size must be returned in registers `R3` and `R4`, while larger structures are returned by accessing a hidden argument in `R3`. GCC always uses the hidden argument method regardless of structure size.

The CodeWarrior Linker checks to see if you are including objects in your project that have incompatible EABI settings. If you do, a warning is issued.

NOTE Different versions of GCC may fix these incompatibilities, so you should check your version if you will be mixing GCC and CodeWarrior objects.

incompatible_sfpe_double_params

```
#pragma incompatible_sfpe_double_params on|off|reset
```

This pragma makes object files generated by CodeWarrior compilers more compatible with object files generated by GNU Compiler Collection (GCC) compilers.

The PowerPC EABI states that software floating-point `double` parameters always begin on an odd register. In other words, if you have a function

```
void red (long a, double b)
```

`a` is passed in register `R3`, and `b` is passed in registers `R5` and `R6` (effectively skipping `R4`). GCC does not skip registers when doubles are passed (although it does skip them for long longs).

The CodeWarrior Linker checks to see if you are including objects in your project that have incompatible EABI settings. If you do, a warning is issued.

NOTE Different versions of GCC may fix these incompatibilities, so you should check your version if you will be mixing GCC and CodeWarrior objects.

interrupt

```
#pragma interrupt
  [SRR DAR DSISR fprs vrs enable nowarn] on | off | reset
```

This pragma lets you write interrupt handlers in C and C++. For example:

```
#pragma interrupt on
void MyHandler(void)
{
  my_real_handler();
}
#pragma interrupt off
```

The PowerPC architecture lets an interrupt service routine be up to 256 bytes long. The compiler warns you if your routine exceeds 256 bytes. Use the `nowarn` option to eliminate the warning.

If your routine must be larger than 256 bytes, add a `#pragma interupt_routine` statement at the site of the interrupt vector. Your interrupt service routine can then be any size.

`#pragma interrupt` saves all volatile general purpose registers that are used, as well as the `CTR`, `XER`, `LR` and condition fields. These registers and condition fields are restored before the `RFI`. Optionally, you can save certain special purpose registers (such as `SRR0` and `SRR1`, `DAR`, `DSISR`), floating-point registers (`fprs`), and re-enable interrupts within your interrupt service routine.

pack

```
#pragma pack(n)
```

Where `n` is one of these integer values: `1,2,4,8,`or `16.` This pragma creates data that is *not* aligned according to the EABI. The EABI alignment provides the best alignment for performance.

Not all processors support misaligned accesses, which could cause a crash or incorrect results. Even on processors which don't crash, your performance suffers since the processor has code to handle the misalignments for you. You may have better performance if you treat the packed structure as a byte stream and pack and unpack them yourself a byte at a time.

If your structure has bit fields and the PowerPC alignment does not give you as small a structure as you desire, double-check that you are specifying the smallest integer size for your bit fields.

For example,

```
typedef struct red {
  unsigned a: 1;
  unsigned b: 1;
  unsigned c: 1;
} red;
```

would be smaller if rewritten as:

```
typedef struct red {
  unsigned char a: 1;
  unsigned char b: 1;
  unsigned char c: 1;
} red;
```

NOTE Pragma pack is implemented somewhat differently by most compiler vendors, especially with bit fields. If you need portability, you are probably better off using shifts and masks instead of bit fields.

pooled_data

`#pragma pooled_data on | off | reset`

This pragma changes the state of pooled data.

NOTE Pooled data is only saves code when more than two variables from the same section are used in a specific function. If pooled data is selected, the compiler only pools the data if it saves code. This feature has the added benefit of typically reducing the data size and allowing deadstripping of unpooled sections.

section

`#pragma section [` *objecttype* `|` *permission* `] [`*iname*`] [`*uname*`]`
`[data_mode=`*datamode*`] [code_mode=`*codemode*`]`

This sophisticated and powerful pragma lets you arrange compiled object code into predefined sections and sections you define. This topic is organized into these parts:

- Parameters

For More Information: www.freescale.com

- Section access permissions
- Predefined sections and default sections
- Forms for #pragma section
- Forcing individual objects into specific sections
- Using #pragma section with #pragma push and #pragma pop

Parameters

The optional *objecttype* parameter specifies where types of object data are stored. It may be one or more of these values:

- `code_type` — executable object code
- `data_type` — non-constant data of a size greater than the size specified in the small data threshold option in the **EPPC Target** settings panel
- `sdata_type` — non-constant data of a size less than or equal to the size specified in the small data threshold option in the **EPPC Target** settings panel
- `const_type` — constant data of a size greater than the size specified in the small const data threshold option in the **EPPC Target** settings panel
- `sconst_type` — constant data of a size less than or equal to the size specified in the small const data threshold option in the **EPPC Target** settings panel
- `all_types` — all code and data

Specify one or more of these object types without quotes separated by spaces.

The CodeWarrior C/C++ compiler generates some of its own data, such as exception and static initializer objects, which are not affected by `#pragma section`.

NOTE To classify character strings, the CodeWarrior C/C++ compiler uses the setting of the Make Strings Read Only checkbox in the **EPPC Processor** settings panel. If the checkbox is checked, character strings are stored in the same section as data of type `const_type`. If the checkbox is clear, strings are stored in the same section as data for `data_type`.

The optional *permission* parameter specifies access permission. It may be one or more of these values:

- `R` — read only permission
- `W` — write permission

For More Information: www.freescale.com

- `X` — execute permission

For information on access permission, see “Section access permissions” on page 140. Specify one or more of these permissions in any order, without quotes, and no spaces.

The optional *iname* parameter is a quoted name that specifies the name of the section where the compiler stores initialized objects. Variables that are initialized at the time they are defined, functions, and character strings are examples of initialized objects. The *iname* parameter may be of the form `.abs.`*`xxxxxxxx`* where *`xxxxxxxx`* is an 8-digit hexadecimal number specifying the address of the section.

The optional *uname* parameter is a quoted name that specifies the name of the section where the compiler stores uninitialized objects. This parameter is required for sections that have data objects. The *uname* parameter value may be a unique name or it may be the name of any previous *iname* or *uname* section. If the *uname* section is also an *iname* section then uninitialized data is stored in the same section as initialized objects.

The special *uname* `COMM` specifies that uninitialized data will be stored in the common section. The linker will put all common section data into the “`.bss`” section. When the Use Common Section checkbox is checked in the **EPPC Processor** panel, `COMM` is the default *uname* for the `.data` section. If the Use Common Section checkbox is clear, `.bss` is the default name of `.data` section.

The *uname* parameter value may be changed. For example, you may want most uninitialized data to go into the `.bss` section while specific variables be stored in the `COMM` section.

Listing 7.2 shows an example where specific uninitialized variables are stored in the `COMM` section.

Listing 7.2 Storing Uninitialized Data in the COMM Section

```
#pragma push // save the current state
#pragma section ".data" "COMM"

int red;
int sky;

#pragma pop // restore the previous state
```

You may not use any of the object types, data modes, or code modes as the names of sections. Also, you may not use pre-defined section names by the PowerPC EABI for your own section names.

The optional `data_mode=`*datamode* parameter tells the compiler what kind of addressing mode to use for referring to data objects for a section.

For More Information: www.freescale.com

The permissible addressing modes for *datamode* are:

- `near_abs` — objects must be within the range -65,536 bytes to 65,536 bytes (16 bits on each side)
- `far_abs` — objects must be within the first 32 bits of RAM
- `sda_rel` — objects must be within a 32K range of the linker-defined small data base address

  The `sda_rel` addressing mode may only be used with the "`.sdata`", "`.sbss`", "`.sdata2`", "`.sbss2`", "`.EMB.PPC.sdata0`", and "`.EMB.PPC.sbss0`" sections.

The default addressing mode for large data sections is `far_abs`. The default addressing mode for the predefined small data sections is `sda_rel`.

Specify one of these addressing modes without quotes.

The optional `code_mode=`*`codemode`* parameter tells the compiler what kind of addressing mode to use for referring to executable routines of a section.

The permissible addressing modes for *codemode* are:

- `pc_rel` — routines must be within plus or minus 24 bits of where `pc_rel` is called from
- `near_abs` — routines must be within the first 24 bits of RAM
- `far_abs` — routines must be within the first 32 bits of RAM

The default addressing mode for executable code sections is `pc_rel`.

Specify one of these addressing modes without quotes.

NOTE All sections have a data addressing mode (`data_mode=`*datamode*) and a code addressing mode (`code_mode=`*codemode*). Although the CodeWarrior C/C++ compiler for PowerPC embedded allows you to store executable code in data sections and data in executable code sections, this practice is not encouraged.

Section access permissions

When you define a section by using `#pragma section`, its default access permission is read only. Changing the definition of the section by associating an object type with it sets the appropriate access permissions for you. The compiler adjusts the access permission to allow the storage of newly-associated object types while continuing to allow objects of previously-allowed object types. For example, associating

`code_type` with a section adds execute permission to that section. Associating `data_type`, `sdata_type`, or `sconst_type` with a section adds write permission to that section.

Occasionally you might create a section without associating it with an object type. You might do so to force an object into a section with the `__declspec` keyword. In this case, the compiler automatically updates the access permission for that section to allow the object to be stored in the section, then issue a warning. To avoid such a warning, make sure to give the section the proper access permissions before storing object code or data into it. As with associating an object type to a section, passing a specific permission adds to the permissions that a section already has.

Predefined sections and default sections

When an object type is associated with the predefined sections, the sections are set as default sections for that object type. After assigning an object type to a non-standard section, you may revert to the default section with one of the forms in “Forms for #pragma section” on page 142.

The compiler predefines the sections in Listing 7.3.

Listing 7.3 Predefined sections

```
#pragma section code_type ".text"  data_mode=far_abs code_mode=pc_rel

#pragma section data_type ".data" ".bss" data_mode=far_abs
code_mode=pc_rel

#pragma section const_type ".rodata" ".rodata" data_mode=far_abs
code_mode=pc_rel

#pragma section sdata_type ".sdata" ".sbss" data_mode=sda_rel
code_mode=pc_rel

#pragma section sconst_type ".sdata2" ".sbss2" data_mode=sda_rel
code_mode=pc_rel

#pragma section ".EMB.PPC.sdata0" ".EMB.PPC.sbss0" data_mode=sda_rel
code_mode=pc_rel

#pragma section RX ".init" ".init" data_mode=far_abs code_mode=pc_rel
```


For More Information: www.freescale.com

NOTE	The `.EMB.PPC.sdata0` and `.EMB.PPC.sbss0` sections are predefined as an alternative to the `sdata_type` object type. The `.init` section is also predefined, but it is not a default section. The `.init` section is used for startup code.

Forms for #pragma section

`#pragma section ".`*name1*`"`

This form simply creates a section called *.name1* if it does not already exist. With this form, the compiler does not store objects in the section without an appropriate, subsequent `#pragma section` statement or an item defined with the `__declspec` keyword. If only one section name is specified, it is considered the name of the initialized object section, *iname*. If the section is already declared, you may also optionally specify the uninitialized object section, *uname*. If you know that the section must have read and write permission, use `#pragma section RW .`*name1* instead, especially if you use the `__declspec` keyword.

`#pragma section` *objecttype* `".`*name2*`"`

With the addition of one or more object types, the compiler stores objects of the types specified in the section *.name2*. If *.name2* does not exist, the compiler creates it with the appropriate access permissions. If only one section name is specified, it is considered the name of the initialized object section, *iname*. If the section is already declared, you may also optionally specify the uninitialized object section, *uname*

`#pragma section` *objecttype*

When there is no *iname* parameter, the compiler resets the section for the object types specified to the default section. Resetting the section for an object type does not reset its addressing modes. You must reset them.

When declaring or setting sections, you also can add an uninitialized section to a section that did not have one originally by specifying a uname parameter. The corresponding uninitialized section of an initialized section may be the same.

Forcing individual objects into specific sections

You may store a specific object of an object type into a section other than the current section for that type without changing the current section. Use the `__declspec` keyword with the name of the target section and put it next to the extern declaration or static definition of the item you want to store in the section.

For More Information: www.freescale.com

Listing 7.4 shows examples.

Listing 7.4 Using __declspec to Force Objects into Specific Sections

```
__declspec(section ".data") extern int myVar;
#pragma section "constants"
__declspec(section "constants") const int myConst = 0x12345678;
```

Using #pragma section with #pragma push and #pragma pop

You can use this pragma with `#pragma push` and `#pragma pop` to ease complex or frequent changes to sections settings.

See Listing 7.2 for an example.

NOTE The `pop` pragma does not restore any changes to the access permissions of sections that exist before or after the corresponding `push` pragma.

EPPC Linker Issues

This section provides background information about the CodeWarrior Embedded PowerPC linker and explains how it works.

The topics in this section are:

- Additional Small Data Sections
- Linker Generated Symbols
- Deadstripping Unused Code and Data
- Link Order
- Linker Command Files

Additional Small Data Sections

The PowerPC EABI specification mandates that compliant build tools predefine three small data sections. The EPPC Linker target settings panel lets you specify the address

For More Information: www.freescale.com

at which the CodeWarrior linker puts two of these sections (if the default locations are unsatisfactory).

CodeWarrior Development Studio, mobileGT Edition lets you create small data sections in addition to those mandated by the PowerPC EABI specification. The CodeWarrior tools let you specify that the contents of a given user-defined section will be accessed by the small data base register selected from the available non-volatile registers. To do this, you use a combination of source code statements and linker command file directives.

To create one additional small data area, follow these steps:

1. Open the CodeWarrior project in which you want to create an additional small data section.
2. Select the build target in which you want to create an additional small data section.
3. Press **ALT-F7**

   The IDE displays the **Target Settings** window.
4. In the left pane of the **Target Settings** window, select C/C++ Language.

   The **C/C++ Language** target settings panel appears in the right side of the **Target Settings** window.
5. Open the prefix file whose name appears in the Prefix File text box in an editor window.
6. Add the statements that define a small data section to the top of the prefix file:
   a. Add a statement that creates a global register variable.

      For example, to create a global register variable for register 14, add this statement to the prefix file:

      ```
      // _dummy does not have to be defined
      extern int _dummy asm("r14");
      ```
   b. Turn off the “unsafe global register variables” warning using this pragma:

      ```
      #pragma unsafe_global_reg_vars off
      ```
   c. Create a user-defined section using the `section` pragma; include the clause `data_mode = sda_rel` so the section can use small data area addressing.

      For example:

      ```
      // you do not have to use the names in this example
      // .red is the initialized part of the section
      ```


For More Information: www.freescale.com

```
// .blue is the uninitialized part of the section
#pragma section RW ".red" ".blue" data_mode = sda_rel
```

NOTE If you want your small data area to be the default section for all small data, use this form of the `section` pragma instead of the one above:

```
#pragma section sdata_type ".red" "blue" data_mode = sda_rel
```

7. Save the prefix file and close the editor window.
8. In each header or source file that declares or defines a global variable that you want to put in a small data section, put the storage-class modifier `__declspec(section "`*`initialized_small_sect_nm`*`")` in front of the definition or declaration.

   For example, the statement:

   ```
   __declspec(section ".red") int x = 5;
   ```

   instructs the compiler to put the global variable `x` into the small data section named `.red`

NOTE Use the name of your *initialized* small data section in the `__declspec(section, "`*`nm`*`")` storage-class modifier. The compiler automatically puts a variable in the uninitialized small data section if appropriate.

NOTE If you want a small data section to be the default section for all small data, do not to add the storage-class modifier `__declspec(section "`*`initialized_small_sect_nm`*`")` to any header or source file.

9. In the left pane of the **Target Settings** window, select EPPC Linker.

   The **EPPC Linker** target settings panel appears.
10. In the Segment Addresses group box, check the Use Linker Command File checkbox.

    The other checkboxes and text boxes in the group become disabled.
11. In the left pane of the **Target Settings** window, select EPPC Target.

    The **EPPC Target** settings panel appears.

For More Information: www.freescale.com

12. From the Code Model listbox, select Absolute Addressing
13. From the ABI listbox, select EABI.
14. Click **OK**

    The IDE saves your settings and closes the **Target Settings** window.
15. Modify the linker command file such that it instructs the linker to use the global register declared above as the base register for your new small data section.

    To do this, follow these steps:

    a. In the linker command file, add two REGISTER directives, one for the initialized part of the small data section and one for uninitialized part.

       For example, to make register 14 the base register, add statements like these:

```
.red    REGISTER(14) : {} > ram
.blue  REGISTER(14) : {} > ram
```

    b. Add the linker command file to each build target in which you want to use the new small data section.
16. Open the CodeWarrior project for the runtime library used by your project.

    The runtime library project is in this directory:

```
installDir\PowerPC_EABI_Support\
Runtime\Project\Runtime.PPCEABI.mcp
```

17. In the build target listbox of the runtime library project window, select the build target of the runtime library that your main project uses.
18. Open this build target's prefix file in a CodeWarrior editor window.
19. Add the same statements to this prefix file that you added to the prefix file of the main project.
20. Save the prefix file and close the editor window.
21. Open `__start.c` in a CodeWarrior editor window.
22. Find the string `__init_registers(void)` and add statements that initialize the small data section base register you are using near the end of this function (immediately above the terminating `blr` instruction).

    For example, to initialize register 14, add these statements:

```
lis  r14, _SDA14_BASE_@ha
addi r14, r14, _SDA14_BASE_@l
```

23. Save `__start.c` and close the editor window.

For More Information: www.freescale.com

24. Open `__ppc_eabi_linker.h` in a CodeWarrior editor window.
25. Find the string `_SDA_BASE_[]` in this file and add this statement after the block of statements that follow this string:

```
// SDAnn_BASE is defined by the linker if
// the REGISTER(nn) directive appears in the .lcf file
__declspec(section ".init") extern char _SDA14_BASE_[];
```

26. Save `__ppc_eabi_linker.h` and close the editor window.
27. Press **F7**

    The IDE builds a new runtime library.
28. Close the runtime library project.
29. Return to your main project.
30. Press **F7**

    The IDE builds your project.

    You can now use the new small data section in this project.

NOTE	You can create more small data segments by following the procedure above. Remember, however, that for each small data section created, the compiler loses one non-volatile register to use for other purposes.

Linker Generated Symbols

You can find a complete list of the linker generated symbols in either the C include file `__ppc_eabi_linker.h` or the assembly include file `__ppc_eabi_linker.i`. The CodeWarrior linker automatically generates symbols for the start address, the end address (the first byte after the last byte of the section), and the start address for the section if it will be burned into ROM. With a few exceptions, all CodeWarrior linker-generated symbols are immediate 32 bit values.

If addresses are declared in your source file as `unsigned char _f_text[];` you can treat `_f_text` just as a C variable even though it is a 32-bit immediate value.

```
unsigned int textsize = _e_text - _f_text;
```

If you do need linker symbols that are not addresses, you can access them from C.

```
unsigned int size = (unsigned int)&_text_size;
```

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. Please see `__init_data` in `__start.c`.

These four symbols are actually not 32-bit immediates but are variables with storage. You access them just as C variables. The startup code now automatically handles initializing all bss type sections and moves all necessary sections from ROM to RAM, even for user defined sections.

Deadstripping Unused Code and Data

The Embedded PowerPC linker deadstrips unused code and data only from files compiled by the CodeWarrior C/C++ compiler. Assembler relocatable files and C/C++ object files built by other compilers are never deadstripped. Deadstripping is particularly useful for C++ programs. Libraries (archives) built with the CodeWarrior C/C++ compiler only contribute the used objects to the linked program. If a library has assembly or other C/C++ compiler built files, only those files that have at least one referenced object contribute to the linked program. Completely unreferenced object files are always ignored.

If the Pool Data checkbox is checked in the EPPC Processor panel, the pooled data is not stripped. However, all small data and code is still subject to deadstripping.

There are, however, situations where there are symbols that you don't want dead-stripped even though they are never used. See “Linker Command Files” on page 149 for information on how to prevent dead-stripping of unused symbols.

Link Order

The **Link Order** tab of the project window lets you define the link order. For general information on setting the link order, see the *IDE User Guide*.

Regardless of the link order you specify, the Embedded PowerPC linker always processes C/C++ files, assembly language source files, and object files (.o) before it processes archive files (.a), which are treated as libraries. Therefore, if a source file defines a symbol, the linker uses that definition in preference to a definition in a library.

For More Information: www.freescale.com

One exception exists. The linker uses a global symbol defined in a library in preference to a source file definition of a weak symbol. You can create a weak symbol with `#pragma overload`. See `__ppc_eabi_init.c` or `__ppc_eabi_init.cpp` for examples.

The Embedded PowerPC linker ignores executable files of the project. You may find it convenient to keep the executable files in the project folder so that you can disassemble it. If a build is successful, a check mark appears in the touch column on the left side of the **Project** window. This indicates that the new file in the project is out of date. If a build is unsuccessful, the IDE is not be able to find the executable file and it stops the build with an appropriate message.

Linker Command Files

Linker command files are an alternative way of specifying segment addresses. The other method of specifying segment addresses is by entering values manually in the Segment Addresses area of the **EPPC Linker** settings panel.

Only one linker command file is supported per target in a project. The linker command filename must end in the `.lcf` extension.

Setting up CodeWarrior IDE to accept LCF files

Projects created with the CodeWarrior IDE version 3 or earlier may not recognize the `.lcf` extension. Therefore, you may not be able to add a filename with the `.lcf` extension to the project. You need to create a file mapping to avoid this.

To add the `.lcf` file mapping to your project:

1. Select **Edit > *Target* Settings**, where *Target* is the name of the current build target.
2. Select the **File Mappings** panel.
3. In the File Type text box, type `TEXT`
4. In the Extension text box, type `.lcf`
5. From the Compiler listbox, select None.
6. Click **Add** to save your settings.

Now, when you add an `.lcf` file to your project, the compiler recognizes the file as a linker command file.

For More Information: www.freescale.com

Linker Command File Directives

The CodeWarrior PPC EABI linker supports these directives:

- EXCLUDEFILES
- EXTERNAL_SYMBOL
- FORCEACTIVE
- FORCEFILES
- GROUP
- INCLUDEDWARF
- INTERNAL_SYMBOL
- MEMORY
- REGISTER
- SECTIONS
- SHORTEN_NAMES_FOR_TOR_101

NOTE You can only use one `SECTIONS`, `MEMORY`, `FORCEACTIVE`, and `FORCEFILES` directive per linker command file.

NOTE If you want to mention a source file such as `main.c` in an `.lcf` file, type `main.o`. The `.lcf` only recognizes object and architecture extensions.

EXCLUDEFILES

The `EXCLUDEFILES` directive is for partial link projects only. It makes your partial link file smaller. The directive has this form.

```
EXCLUDEFILES { executablename.extension }
```

In the example:

```
EXCLUDEFILES { kernel.elf }
```

`kernel.elf` is added to your project. The linker does not add any section from `kernel.elf` to your project. However, it does delete any weak symbol from your partial link that also exists in `kernel.elf`. Weak symbols can come from templates or out-of-line inline functions.

For More Information: www.freescale.com

EXCLUDEFILES can be used independently of INCLUDEDWARF. Unlike INCLUDEDWARF, EXCLUDEFILES can take any number of executable files.

EXTERNAL_SYMBOL

Use the EXTERNAL_SYMBOL and INTERNAL_SYMBOL directives to force the addressing of global symbols. This directive is of the form: XXXL_SYMBOL {sym1, sym2, symN}, where symbols are the link time symbol names (mangled for C++).

FORCEACTIVE

The directives FORCEACTIVE and FORCEFILES give you more control over symbols that you don't want dead-stripped. The FORCEACTIVE directive has this form:

```
FORCEACTIVE { symbol1 symbol2 ... }
```

Use FORCEACTIVE with a list of symbols that you do not want to be dead-stripped.

FORCEFILES

Use FORCEFILES to list source files, archives, or archive members that you don't want dead-stripped. All objects in each of the files are included in the executable. The FORCEFILES directive has this form:

```
FORCEFILES { source.o object.o archive.a(member.o) ... }
```

If you only have a few symbols that you do not want deadstripped, use FORCEACTIVE.

GROUP

The GROUP directive lets you organize the linker command file. This directive has this form:

```
GROUP <address_modifiers> :{ <section_spec> ... }
```

Please see the topic SECTIONS for the description of the components.

Listing 7.5 shows that each group starts at a specified address. If no address_modifiers were present, it would start following the previous section or group. Although you normally do not have an address_modifier for an output_spec within a group, all sections in a group follow contiguously unless there is an address_modifier for that output_spec.

Listing 7.5 Example 1

```
SECTIONS {
 GROUP BIND(0x00010000) : {
  .text : {}
  .rodata : {*(.rodata) *(extab) *(extabindex)}
  }

 GROUP BIND(0x2000) : {
  .data : {}
  .bss : {}
  .sdata BIND(0x3500) : {}
  .sbss  : {}
  .sdata2 : {}
  .sbss2  : {}
  }

 GROUP BIND(0xffff8000) : {
  .PPC.EMB.sdata0  : {}
  .PPC.EMB.sbss0 : {}
  }
}
```

INCLUDEDWARF

The `INCLUDEDDWARF` directive allows you to debug source level code in the kernel while debugging your application. This directive has the form:
`INCLUDEDDWARF` { *executablename.extension* }

In the example `INCLUDEDDWARF` { kernel.elf }, `kernel.elf` is added to your project. The linker adds only the `.debug` and `.line` sections of `kernel.elf` to your application. This allows you to debug source level code in the kernel while debugging your application.

You are limited to one executable file when using this directive. If you need to process more than one executable, add this directive to another file.

INTERNAL_SYMBOL

Use the `INTERNAL_SYMBOL` and EXTERNAL_SYMBOL directives to force the addressing of global symbols. This directive is of the form: `XXXL_SYMBOL {sym1, sym2, symN}`, where symbols are the link time symbol names (mangled for C++).

For More Information: www.freescale.com

REGISTER

Use the `REGISTER` directive to assign one of the EPPC processor's non-volatile registers to a user-defined small data section.

This directive is of this form `REGISTER(nn [ , limit])` where:

- `nn` is one of the predefined small data base registers, a non-volative EPPC register, or -1
  - `0, 2, 13`

    These registers are for the predefined small data sections:
    `0 - .EMB.PPC.sdata0/.EMB.PPC.sbss0`
    `2 - .sdata2/sbss2`
    `13 - .sdata/sbss`

    You do not have to define these sections using `REGISTER` because they are predefined.
  - `14 - 31`

    Match any value in this range with the register reserved by your global register variable declaration.
  - `-1`

    This "register" value instructs the linker to treat relocations that refer to objects in your small data section as non-small data area relocations. These objects are converted to near absolute relocations, which means that the objects referenced must reside within the first 32 KB of memory. If they do not, the linker emits a "relocation out of range" error. To fix this problem, rewrite your code such that the offending objects use large data relocations.
- `limit` is the maximum size of the small data section to which register `nn` is bound.

  This value is the size of the initialized and uninitialized sections of the small data section combined. If `limit` is not specified, `0x00008000` is used.

NOTE Each small data section you create makes one less register available to the compiler; it is possible to starve the compiler of registers. As a result, create only the number of small data sections you need.

For More Information: www.freescale.com

Freescale Semiconductor, Inc.

MEMORY

A MEMORY directive is of the form MEMORY : { <memory_spec> ... }, where memory_spec is:

```
<symbolic name> : origin = num, length = num
```

origin may be abbreviated as org or o. length may be abbreviated as len or l. If you do not specify length, the memory_spec is allowed to be as big as necessary. In all cases, the linker warns you if sections overlap. The length is useful if you want to avoid overlapping an RTOS or exception vectors that might not be a part of your image.

You specify that a output_spec or a GROUP goes into a memory_spec with the ">" symbol.

Listing 7.6 shows the MEMORY directive added to the example code shown in Listing 7.5. The results of both examples are identical.

Listing 7.6 Example 2

```
MEMORY {

  text : origin = 0x00010000

  data : org = 0x00002000 len = 0x3000
  page0 : o = 0xffff8000, l = 0x8000
}

SECTIONS {

GROUP : {
  .text : {}
  .rodata : {*(.rodata) *(extab) *(extabindex)}

  } > text

GROUP : {
  .data : {}
  .bss : {}
  .sdata BIND(0x3500) : {}
  .sbss   : {}
  .sdata2 : {}
  .sbss2  : {}

  } > data
```


For More Information: www.freescale.com

```
GROUP   : {
  .PPC.EMB.sdata0   : {}
  .PPC.EMB.sbss0  : {}
  } > page0
}
```

SECTIONS

A `SECTIONS` directive has this form:

```
SECTIONS { <section_spec> ... }
```

where `section_spec` is

```
<output_spec> (<input_type>) <address_modifiers> :
{ <input_spec> ... }
```

`output_spec` is the section name for the output section.

`input_type` is one of `TEXT`, `DATA`, `BSS`, `CONST` and `MIXED`. `CODE` is also supported as a synonym of `TEXT`. One `input_type` is permitted and must be enclosed in `()`. If an `input_type` is present, only input sections of that type are added to the section. `MIXED` means that the section contains code and data `(RWX)`. The `input_type` restricts the access permission that are acceptable for the output section, but they also restrict whether initialized content or uninitialized content can go into the output section. Table 7.7 shows the types of input for `input_type`.

Table 7.7 Types of Input for input_type

Name	Access Permissions	Status
TEXT	RX	Initialized
DATA	RW	Initialized
BSS	RW	Uninitialized
CONST	R	Initialized
MIXED	RWX	Initialized

`address_modifiers` are for specifying the address of an output section.

The pseudo functions `ADDR()`, `SIZEOF()`, `NEXT()`, `BIND()`, and `ALIGN()` are supported.

For More Information: www.freescale.com

NOTE Other compiler vendors also support ways that you can specify the ROM Load address with the `address_modifiers`. With CodeWarrior IDE, this information is specified in the EPPC Linker settings panel. You may also simply specify an address with `BIND`.

`ADDR()` takes previously defined `output_spec` or `memory_spec` enclosed in () and returns its address.

`SIZEOF()` takes previously defined `output_spec` or `memory_spec` enclosed in () and returns its size.

`ALIGN()` takes a number and aligns the `output_spec` to that alignment.

`NEXT()` is similar to `ALIGN`. It returns the next unallocated memory address.

`BIND()` can take a numerical address or a combination of the above pseudo functions.

`input_spec` can be empty or a file name, a file name with a section name, the wildcard '*' with a section name singly or in combination.

When `input_spec` is empty, as in

```
.text : {}
```

all `.text` sections in all files in the project that aren't more specifically mentioned in another `input_spec` are added to that `output_spec`.

A file name by itself means that all sections go into the `output_spec`.

A file name with a section name means that the specified section goes into the `output_spec`.

A `"*"` with a section name means that the specified section in all files go into the `output_spec`.

In all cases, the `input_spec` is subject to `input_type`. For example,

```
.text (TEXT) : { red.c }
```

means that only sections of type TEXT in file `red.c` is added.

In all cases, if there is more that one `input_spec` that fits an input file, the more specific `input_spec` gets the file.

If an archive name is used instead of source file name, all referenced members of that archive are searched. You can further specify a member with `red.a(redsky.c)`. The linker doesn't support grep. If listing just the source file name is ambiguous, enter the full path.

Listing 7.7 shows how you might specify a `SECTIONS` directive without a `MEMORY` directive. The `.text` section starts at `0x00010000` and contains all sections named `.text` in all input files. The `.rodata` section starts just after the `.text` section, and is aligned on the largest alignment found in the input files. The input files are the read only sections (`.rodata`) found in all files. The `.data` section starting address is the sum of the starting address of `.rodata` and the size of `.rodata`. The resulting address is aligned on a `0x100` boundary. The address contains all sections of `.data` in all files. The `.bss` section follows the `.data` through `.sbss2` sections. The `.EMB.PPC.sdata0` starts at `0xffff8000` and the `.EMB.PPC.sbss0` follows it.

Listing 7.7 Example 3

```
SECTIONS {

.init : {}
  .text BIND(0x00010000) : {}
  .rodata : {}
  extab : {}
  extabindex : {}

  .data BIND(ADDR(.rodata) + SIZEOF(.rodata)) ALIGN(0x100) : {}
  .sdata : {}
  .sbss : {}
  .sdata2 : {}
  .sbss2 : {}
  .bss : {}

  .PPC.EMB.sdata0 BIND(0xffff8000) : {}
  .PPC.EMB.sbss0 : {}

}
```

NOTE `extab` and `extabindex` must be in separate sections.

SHORTEN_NAMES_FOR_TOR_101

The directive `SHORTEN_NAMES_FOR_TOR_101` instructs the linker to shorten long template names for the benefit of the WindRiver® Systems Target Server. To use this directive, simply add it to the linker command file on a line by itself.

```
SHORTEN_NAMES_FOR_TOR_101
```


For More Information: www.freescale.com

WindRiver Systems Tornado Version 1.0.1 (and earlier) does not support long template names as generated for the MSL C++ library. Therefore, the template names must be shortened if you want to use them with these versions of the WindRiver Systems Target Server.

Miscellaneous features

- Memory Gaps
- Symbols

Memory Gaps

You can create gaps in memory by performing alignment calculations such as

```
. = (. + 0x20) & ~0x20;
```

This kind of calculation can occur between `output_specs`, between `input_specs`, or even in `address_modifiers`. A "." refers to the current address. You may assign the . to a specific unallocated address or just do alignment as the example shows. The gap is filled with `0`, in the case of an alignment (but not with `ALIGN()`).

You can specify an alternate fill pattern with `= <short_value>`, as in

```
.text : { . = (. + 0x20) & ~0x20; *(.text) } = 0xAB > text
```

`short_value` is 2 bytes long. Note that the fill pattern comes before the `memory_spec`. You can add a fill to a GROUP or to an individual `output_spec` section. Fills cannot be added between `.bss` type sections. All calculations must end in a "`;`".

Symbols

You can create symbols that you can use in your program by assigning a symbol to some value in your linker command file.

```
.text : { _red_start = .; *(.text) _red_end = .;} > text
```

In the example above, the linker generates the symbols `_red_start` and `_red_end` as 32 bit values that you can access in your source files. `_red_start` is the address of the first byte of the `.text` section and `__red_end` is the byte that follows the last byte of the `.text` section.

You can use any of the pseudo functions in the `address_modifiers` in a calculation.

The CodeWarrior linker automatically generates symbols for the start address, the end address, and the start address for the section if it is to be burned into ROM. For a section `.red`, we create `_f_red`, `_e_red`, and `_f_red_rom`. In all cases, any "." in the name is replaced with a "_". Addresses begin with an "`_f`", addresses after the last byte in section begin with an "`_e`", and ROM addresses end in a "`_rom`". See the header file `__ppc_eabi_linker.h` for further details.

All user defined sections follow the preceding pattern. However, you can override one or more of the symbols that the linker generates by defining the symbol in the linker command file.

NOTE BSS sections do not have a ROM symbol.

Using __attribute__ ((aligned(?)))

You can use `__attribute__ ((aligned(?)))` in several situations:

- Variable declarations
- Struct, union, or class definitions
- Typedef declarations
- Struct, union, or class members

NOTE Substitute any power of 2 up to 4096 for the question mark (?).

This section contains these topics:

- Variable Declaration Examples
- Struct Definition Examples
- Typedef Declaration Examples
- Struct Member Examples

Variable Declaration Examples

This section shows variable declarations that use `__attribute__ ((aligned(?)))`.

The following variable declaration aligns `V1` on a 16-byte boundary.

```
int V1[4] __attribute__ ((aligned (16)));
```


For More Information: www.freescale.com

The following variable declaration aligns V2 on a 2-byte boundary.

```
int V2[4] __attribute__ ((aligned (2)));
```

Struct Definition Examples

This section shows struct definitions that use `__attribute__ ((aligned(?)))`.

The following struct definition aligns all definitions of `struct` S1 on an 8-byte boundary.

```
struct S1 { short f[3]; }
    __attribute__ ((aligned (8)));
struct S1 s1;
```

The following struct definition aligns all definitions of `struct` S2 on a 4-byte boundary.

```
struct S2 { short f[3]; }
    __attribute__ ((aligned (1)));
struct S2 s2;
```

NOTE You must specify a minimum alignment of at least 4 bytes for structures. Specifying a lower number for the alignment of a structure causes alignment exceptions.

Typedef Declaration Examples

This section shows typedef declarations that use `__attribute__ ((aligned(?)))`.

The following typedef declaration aligns all definitions of T1 on an 8-byte boundary.

```
typedef int T1 __attribute__ ((aligned (8)));
T1 t1;
```

The following typedef declaration aligns all definitions of T2 on an 1-byte boundary.

```
typedef int T2 __attribute__ ((aligned (1)));
T2 t2;
```


For More Information: www.freescale.com

Struct Member Examples

This section shows struct member definitions that use `__attribute__ ((aligned(?)))`.

The following struct member definition aligns all definitions of `struct S3` on an 8-byte boundary, where `a` is at offset 0 and `b` is at offset 8.

```
struct S3 {
   char a;
   int b __attribute__ ((aligned (8)));
};
struct S3 s3;
```

The following struct member definition aligns all definitions of `struct S4` on a 4-byte boundary, where `a` is at offset 0 and `b` is at offset 4.

```
struct S4 {
   char a;
   int b __attribute__ ((aligned (2)));
};
struct S4 s4;
```

NOTE Specifying `__attribute__ ((aligned (2)))` does not affect the alignment of S4 because 2 is less than the natural alignment of int.

8

Inline Assembler

This chapter explains how to use the inline assembler built into the CodeWarrior™ C/C++ compiler for the mobileGT™ processor family.

The chapter does *not* discuss the standalone CodeWarrior EPPC assembler. For information about this tool, refer to the *Assembler Guide*.

Further, the chapter does not document all instructions in the EPPC instruction set. For complete coverage of the instruction set, see *PowerPC Microprocessor Family: The Programming Environment for 32-Bit Microprocessors,* published by Motorola.

Finally, refer to this web page for documentation of Motorola's entire semiconductor product line, including embedded versions of the PowerPC processor:

`http://e-www.motorola.com/webapp/sps/library/tools_lib.jsp`

The sections of this chapter are:

- Working With Assembly Language
- Assembler Directives
- Intrinsic Functions

Working With Assembly Language

This section explains how to use the built-in support for assembly language programming included in the CodeWarrior compiler.

This section contains these topics:

- Assembly Language Syntax
- Special Embedded PowerPC Instructions
- Creating Statement Labels
- Using Comments
- Using the Preprocessor in Embedded PowerPC Assembly
- Using Local Variables and Arguments

- Creating a Stack Frame
- Specifying Operands

Assembly Language Syntax

To specify that a block of code in your file should be interpreted as assembly language, use the `asm` keyword.

NOTE To ensure that the C/C++ compiler recognizes the `asm` keyword, you must clear the ANSI Keywords Only checkbox of the **C/C++ Language** target settings panel. This panel and its options are fully described in the *C Compilers Reference*.

As an alternative, the keyword `__asm` is always recognized even if the ANSI Keywords Only checkbox is checked.

The assembly instructions are the standard Embedded PowerPC instruction mnemonics. For information on Embedded PowerPC assembly language instructions, see *PowerPC Microprocessor Family: The Programming Environment for 32-Bit Microprocessors,* published by Motorola (serial number MPCFPE32B/AD).

There are two ways to use assembly language with the CodeWarrior compilers.

First, you can write code to specify that an entire function is in assembly language. This is called function-level assembly language. Alternatively, CodeWarrior compilers also support assembly statement blocks within a function. In other words, you can write code that is both in function-level assembly language and statement-level assembly language.

NOTE To enter a few lines of assembly language code within a single function, you can use the support for intrinsics included in the compiler. Intrinsics are an alternative to using `asm` statements within functions.

Function-level assembly code for PowerPC uses this syntax:

`asm {`*`function definition`*` }`

An assembly language function must end with `blr` instruction. For example:

```
asm long MyFunc(void)
{
  ... // assembly language instructions
```

```
  blr
}
```

Statement-level assembly language has this syntax:

```
asm { one or more instructions }
```

Blocks of assembly language statements are supported. For example:

```
long MyFunc (void)
{
  asm
  {
    ... // assembly language statements
  }
}
```

NOTE Assembly language functions are never optimized, regardless of compiler settings.

You can use an `asm` statement wherever a code statement is allowed.

NOTE If you check the Inlined Assembler is Volatile checkbox in the EPPC Processor panel, functions that *contain* an `asm` block are only partially optimized, as the optimizer optimizes the function, but skips any `asm` blocks of code. If the Inlined Assembler is Volatile checkbox is clear, the optimizer treats asm blocks as compiler-generated instructions.

The built-in assembler uses all the standard PowerPC assembler instructions. It accepts some additional directives described in “Assembler Directives” on page 174. If you use the `machine` directive, you can also use instructions that are available only in certain versions of the PowerPC processors.

Keep these tips in mind as you write assembly functions:

- All statements must follow this syntax:

  `[LocalLabel:] (instruction | directive) [operands]`

- Each instruction must end with a newline or a semicolon (`;`).
- Hex constants must be in C-style: `li r3, 0xABCDEF`
- Assembler directives, instructions, and registers are case-sensitive and must be in lowercase. For example,

Freescale Semiconductor, Inc.

```
add r2,r3,r4
```

- Every assembly function must end in an `blr` statement. For example,

```
asm void g(void)
{
  add r2,r3,r4
  blr
}
```

Listing 8.1 shows an example of an assembly language function.

Listing 8.1 Example Assembly Language Function

```
asm void mystrcpy(char *tostr, char *fromstr)

{
  addi  tostr,tostr,-1
  addi  fromstr,fromstr,-1
@1 lbzu  r5,1(fromstr)
  cmpwi r5,0
  stbu  r5,1(tostr)
  bne  @1
  blr
}
```

Special Embedded PowerPC Instructions

To set the branch prediction (`y`) bit for those branch instructions that can use it, use `+` or `-`. For example:

```
@1 bne+ @2
@2 bne- @1
```

Most integer instructions have four forms:

- normal form — `add r3,r4,r5`
- record form — `add.  r3,r4,r5`

  This form ends in a period. This form sets register `cr0` to whether the result is less, than, equal to, or greater than zero.
- overflow — `addo  r3,r4,r5`

  This form ends in the letter (`o`). This form sets the `SO` and `OV` bits in the `XER` if the result overflows.
- overflow and record — `addo. r3,r4,r5`

For More Information: www.freescale.com

This form ends in (`o.`). This form sets both registers.

Some instructions only have a record form (with a period). Always make sure to include the period. For example,

```
andi.  r3,r4,7
andis. r3,r4,7
stwcx. r3,r4,r5
```

Creating Statement Labels

The name of an inline assembly language statement label must follow these rules:

- A label name cannot be the same as the identifier of any local variables of the function in which the label name appears.
- A label name does not have to start in the first column of the function in which it appears; a label name can be preceded by white space.
- A label name can begin with an "at-sign" character (`@`) unless the label immediately follows a local variable declaration.

  For example:

  ```
  @red and red: are both valid label names.

  asm void func1(){
  int i;
    @x: li r0,1 //Invalid !!!
  }

  asm void func2(){
  int i;
    x:  li r0,1 //OK
    @y: add r3, r4, r5 //OK
  }
  ```

- A label name must end with a colon character (`:`) unless it begins with an at-sign character (`@`).

  For example, `red:` and `@red` are valid, but `red` is *not* valid.

- A label name *can* be the same as an assembly language statement mnemonic.

  For example, this statement is valid:

  ```
  add: add r3, r4, r5
  ```

This is an example of a complete inline assembly language function:

```
asm void red(void){
  x1:  add r3,r4,r5
```


For More Information: www.freescale.com

```
      @x2: add r6,r7,r8
    }
```

Using Comments

You cannot begin comments with a pound sign (#) because the preprocessor uses the pound sign. For example, this format is invalid:

```
add   r3,r4,r5 # Comment
```

Use C and C++ comments in this format:

```
add   r3,r4,r5 // Comment
add   r3,r4,r5 /* Comment */
```

Using the Preprocessor in Embedded PowerPC Assembly

You can use all preprocessor features, such as comments and macros, in the assembler. In multi-line macros, you must end each assembly statement with a semicolon (;) because the (\) operator removes newlines. For example:

```
#define remainder(x,y,z) \
divw z,x,y; \
mullw z,z,y; \
subf z,z,x

asm void newPointlessMath(void)
{
remainder(r3,r4,r5)
blr
}
```

Using Local Variables and Arguments

To refer to a memory location, you can use the name of a local variable or argument.

The rule for assigning arguments to registers or memory depends on whether the function has a stack frame.

If function has a stack frame, the inline assembler assigns:

- scalar arguments declared as `register` to `r14` — `r31`
- floating-point arguments declared as `register` to `fp14` — `fp31`

- other arguments to memory locations
- scalar locals declared as `register` to `r14` — `r31`
- floating-point locals declared as `register` to `fp14` — `fp31`
- other locals to memory locations

If a function has no stack frame, the inline assembler assigns arguments that are declared `register` and kept in registers. If you have variable or non-register arguments, the compiler will warn you that you should use `frfree`

NOTE Some opcodes require registers, and others require objects. For example, if you use `nofralloc` with function arguments, you may run into difficulties.

Creating a Stack Frame

You need to create a stack frame for a function if the function:

- calls other functions.
- declares non-register arguments or local variables.

To create a stack frame, use the `fralloc` directive at the beginning of your function and the `frfree` directive just before the `blr` statement. The directive `fralloc` automatically allocates (while `ffree` automatically de-allocates) memory for local variables, and saves and restores the register contents.

```
asm void red ()
{
  fralloc
  // Your code here
  frfree
  blr
}
```

The `fralloc` directive has an optional argument *number* that lets you specify the size in bytes of the parameter area of the stack frame. The stack frame is an area for storing parameters used by the assembly code. The compiler creates a 0 byte parameter area for you to pass variables into your assembly language functions.

In Embedded PowerPC, function arguments are passed using registers. If your assembly-language routine calls any function that requires more parameters than will fit into `r3` — `r10` and `fp1` — `fp8`, you need to pass that size to `fralloc`. In the case

of integer values, registers `r3` — `r10` are used. For floating-point values, registers `fp1` through `fp8` are used.

As an example, if you pass 12 long integer to your assembly function, this would consume 16 bytes of the parameter area. Registers `r3` through `r10` will hold eight integers, leaving four byte integers in the parameter area.

Specifying Operands

This section describes how to specify the operands for assembly language instructions.

Using Register Variables and Memory Variables

When you use variable names as operands, the syntax you use depends on whether the variable is declared with or without the register keyword. For example, some instructions, such as `add`, require register operands. You can use a register variable wherever a register operand is used. The inline assembler allows a shortcut through use of locals and arguments that are not declared register in certain instructions.

Listing 8.2 shows a block of code for specifying operands.

Listing 8.2 Using Register Variables and Memory Variables

```
asm void red(register int *a)

{
   int b;
   fralloc
   lwz r4,a
   lwz r4,0(a)
   lwz r4,b
   lwz r4, b(SP)
   frfree
   blr
}
```

In Listing 8.2:

- The code at line number five is incorrect because the operand of the operand of register variable is not fully expressed.
- The code at line number six is correct because the operand is fully expressed.

- The code at line number seven is correct; the inline assembler allows use of locals and arguments that are not declared as register.
- The code at line number eight is correct because `b` is a memory variable.

Using Registers

For a register operand, you must use one of the register names of the appropriate kind for the instruction. The register names are case-sensitive. You also can use a symbolic name for an argument or local variable that was assigned to a register.

The general registers are `SP`, `r0` to `r31`, and `gpr0` to `gpr31`. The floating-point registers are `fp0` to `fp31` and `f0` to `f31`. The condition registers are `cr0` to `cr7`.

Using Labels

For a label operand, you can use the name of a label. For long branches (such as `b` and `bl` instructions) you can also use function names. For `bla` and `la` instructions, use absolute addresses.

For other branches, you must use the name of a label. For example,

- `b @3` — correct syntax for branching to a local label
- `b red` — correct syntax for branching to external function `red`
- `bl @3` — correct syntax for calling a local label
- `bl red` — correct syntax for calling external function `red`
- `bne red` — incorrect syntax; short branch outside function `red`

NOTE You cannot use local labels that have already been declared in other functions.

Using Variable Names as Memory Locations

Whenever an instruction, such as a load instruction, a store instruction, or `la`, requires a memory location, you can use a local or global variable name. You can modify local variable names with struct member references, class member references, array subscripts, or constant displacements. For example, all the local variable references in the following block of code are valid.

```
asm void red(void){
  long myVar;
  long myArray[1];
```

```
  Rect myRectArray[3];
  fralloc
  lwz r3,myVar(SP)
  la  r3,myVar(SP)
  lwz r3,myRect.top
  lwz r3,myArray[2](SP)
  lwz r3,myRectArray[2].top
  lbz r3,myRectArray[2].top+1(SP)
  frfree
  blr
}
```

You can also use a register variable that is a pointer to a struct or class to access a member of the struct in this manner:

```
void red(void){
  Rect q;
  register Rect *p = &q;
  asm {
  lwz r3,p->top;
  }
}
```

You can use the `@hiword` and `@loword` directives to access the high and low four bytes of 8 byte long longs and software floating-point doubles.

```
long long gTheLongLong = 5;
asm void Red(void);
asm void Red(void)
{
  fralloc
  lwz r5, gTheLongLong@hiword
  lwz r6, gTheLongLong@loword
  frfree
blr
}
```

Using Immediate Operands

For an immediate operand, you can use an integer or enum constant, `sizeof` expression, and any constant expression using any of the C dyadic and monadic arithmetic operators.

These expressions follow the same precedence and associativity rules as normal C expressions. The inline assembler carries out all arithmetic with 32-bit signed integers.

An immediate operand can also be a reference to a member of a struct or class type. You can use any struct or class name from a `typedef` statement, followed by any number of member references. This evaluates to the offset of the member from the start of the struct. For example:

```
lwz   r4,Rect.top(r3)
addi  r6,r6,Rect.left
```

As a side note, `la rD,d(rA)` is the same as `addi rD,rA,d`.

You also can use the top or bottom half-word of an immediate word value as an immediate operand by using one of the @ modifiers.

```
long gTheLong;
asm void red(void)
{
fralloc
lis r6, gTheLong@ha
addi r6, r6, gTheLong@h
lis r7, gTheLong@h
ori r7, br7, gTheLong@l
frfree
blr
}
```

The access patterns are:

```
lis x,var@ha
la  x,var@l(x)
```

or

```
lis x,var@h
ori x,x,var@l
```

In this example, `la` is the simplified form of `addi` to load an address. The instruction `las` is similar to `la` but shifted. Refer to the Motorola PowerPC manuals for more information.

Using `@ha` is preferred since you can write:

```
lis x,var@ha
lwz v,var@l(x)
```

You cannot do this with `@h` because it requires that you use the `ori` instruction.

For More Information: www.freescale.com

Assembler Directives

This section describes some special assembler directives that the Embedded PowerPC built-in assembler accepts. These directives are:

- entry
- fralloc
- frfree
- machine
- nofralloc
- opword

entry

`entry [ extern | static ]` *name*

Embedded PowerPC assembler directive that defines an entry point into the current function. Use the `extern` qualifier to declare a global entry point; use the `static` qualifier to declare a local entry point. If you leave out the qualifier, `extern` is assumed.

Listing 8.3 shows how to use the `entry` directive.

Listing 8.3 Using the entry directive

```
void __save_fpr_15(void);
void __save_fpr_16(void);
asm void __save_fpr_14(void)
{
    stfd   fp14,-144(SP)
    entry  __save_fpr_15
    stfd   fp15,-136(SP)
    entry  __save_fpr_16
    stfd  fp16,-128(SP)
    // ...
}
```

fralloc

`fralloc [` *number* `]`

Embedded PowerPC assembler directive that creates a stack frame for a function and reserves registers for your local register variables. You need to create a stack frame for a function if the function:

- calls other functions.
- uses more arguments than will fit in the designated parameters `(r3` — `r10, fp1` — `fp8).`
- declares local registers.
- declares non-registered parameters.

The `fralloc` directive has an optional argument *number* that lets you specify the size in bytes of the parameter area of the stack frame. The compiler creates a 0-byte parameter area. If your assembly-language routine calls any function that requires more parameters than will fit in `r3` — `r10` and `fp1` — `fp8`, you must specify a larger amount.

frfree

`frfree`

Embedded PowerPC assembler directive that frees the stack frame and restores the registers that `fralloc` reserved.

NOTE The `frfree` directive does not generate a `blr` instruction. You must include one explicitly.

machine

`machine` *number*

Embedded PowerPC assembler directive that specifies which CPU the assembly language code is for. The value of *number* must be one of those listed in Table 8.1.

Table 8.1 CPU Identifiers

5200	823	all	generic

Freescale Semiconductor, Inc.

If you use `generic`, the compiler supports the core instructions for the 603, 604, 740, and 750 processors. In addition, the compiler supports all optional instructions.

If you use `all`, the compiler supports all core and optional instructions for all Embedded PowerPC processors.

If you do not use the `machine` directive, the compiler uses the settings you selected from the Processor listbox of the **EPPC Processor** settings panel.

nofralloc

You can use the `nofralloc` directive so that an inline assembly function does not build a stack frame. When you use `nofralloc`, if you have local variables, parameters or make function calls, you are responsible for creating and deleting your own stack frame.

For an example code that shows how to use the `nofralloc` directive, see the file `__start.c`. This file is in this directory:

installDir`\PowerPC_EABI_Support\Runtime\Src`

where *installDir* is a placeholder for the path in which you installed your CodeWarrior product.

opword

The inline assembler supports the `opword` directive. For example, the line "`opword 0x7C0802A6`" is equivalent to "`mflr r0`". No error checking is done on the value of the opword; the instruction is simply copied into the executable file.

Intrinsic Functions

This section explains support for intrinsic functions in the CodeWarrior compilers. Support for intrinsic functions is not part of the ANSI C or C++ standards. They are an extension provided by the CodeWarrior compilers.

Intrinsic functions are a mechanism you can use to get assembly language into your source code.

There is an intrinsic function for several common processor opcodes (instructions). Rather than using inline assembly syntax and specifying the opcode in an `asm` block, you call the intrinsic function that matches the opcode.

For More Information: www.freescale.com

When the compiler encounters the intrinsic function call in your source code, it does not actually make a function call. The compiler substitutes the assembly instruction that matches your function call. As a result, no function call occurs in the final object code. The final code is the assembly language instructions that correspond to the intrinsic functions.

NOTE You can use intrinsic functions or the `asm` keyword to add a few lines of assembly code within a function. If you want to write an entire function in assembly, you can use the inline assembler.

For information on Embedded PowerPC assembly language instructions, see *PowerPC Microprocessor Family: The Programming Environment for 32-Bit Microprocessors,* published by Motorola.

Low-Level Processor Synchronization

These functions perform low-level processor synchronization.

- `void __eieio(void)` — Enforce in-order execution of I/O
- `void __sync(void)` — Synchronize
- `void __isync(void)` — Instruction synchronize

For more information on these functions, see the instructions `eieio`, `sync`, and `isync` in *PowerPC Microprocessor Family: The Programming Environments* by Motorola.

Absolute Value Functions

These functions generate inline instructions that take the absolute value of a number.

- `int __abs(int)` — Absolute value of an integer
- `float __fabs(float)` — Absolute value of a float
- `float __fnabs(float)` — Negative absolute value of a float
- `long __labs(long)` — Absolute value of a long int

`__fabs(float)` and `__fnabs(float)` are not available unless the Hardware floating-point option is selected in the **EPPC Processor** settings panel.

Byte-Reversing Functions

These functions generate inline instructions that can dramatically speed up certain code sequences, especially byte-reversal operations.

- `int __lhbrx(void *, int)` — Load halfword byte; reverse indexed
- `int __lwbrx(void *, int)` — Load word byte; reverse indexed
- `void __sthbrx(unsigned short, void *, int)` — Store halfword byte; reverse indexed
- `void __stwbrx(unsigned int, void *, int)` — Store word byte; reverse indexed

Setting the Floating-Point Environment

This function lets you change the Floating Point Status and Control Register (FPSCR). It sets the FPSCR to its argument and returns the original value of the FPSCR.

This function is not available if you select None floating-point option in the **EPPC Processor** settings panel.

```
float __setflm(float);
```

This example shows how to set and restore the FPSCR:

```
double old_fpscr;

oldfpscr = __setflm(0.0); /* Clears all flag/exception/mode
bits and save the original settings */

/* Peform some floating-point operations */

__setflm(old_fpscr); /* Restores the FPSCR */
```

Manipulating a Variable or Register

These functions rotate the contents of a variable to the left:

- `int __rlwinm(int, int, int, int)` — Rotate left word (immediate), then AND with mask
- `int __rlwnm(int, int, int, int)` — Rotate left word, then AND with mask
- `int __rlwimi(int, int, int, int, int)` — Rotate Left word (immediate), then mask insert

The first argument to `__rlwimi` is overwritten. However, if the first parameter is a local variable allocated to a register, it is both an input and output parameter. For this reason, this intrinsic should always be written to put the result in the same variable as the first parameter as shown here:

```
ra = __rlwimi( ra, rs, sh, mb, me );
```

You can count the leading zeros in a register using this intrinsic:

```
int __cntlzw(int);
```

You can use inline assembly for a complete assembly language function, as well as individual assembly language statements.

Data Cache Manipulation

The intrinsics shown in Table 8.2 map directly to PowerPC assembly instructions

Table 8.2 Data Cache Intrinsics

Intrinsic Function Prototype	PowerPC Instruction
`void __dcbf(void *, int);`	`dcbf`
`void __dcbt(void *, int);`	`dcbt`
`void __dcbst(void *, int);`	`dcbst`
`void __dcbtst(void *, int);`	`dcbtst`
`void __dcbz(void *, int);`	`dcbz`

Math Functions

The intrinsics shown in Table 8.3 map directly to PowerPC assembly instructions.

Table 8.3 Math Intrinsics

Intrinsic Function Prototype	PowerPC Instruction
`int __mulhw(int, int);`	`mulhw`
`uint __mulhwu(uint, uint);`	`mulhwu`
`double __fmadd(double, double, double);`	`fmadd`
`double __fmsub(double, double, double);`	`fmsub`
`double __fnmadd(double, double, double);`	`fnmadd`

Freescale Semiconductor, Inc.

Table 8.3 Math Intrinsics

`double __fnmsub(double, double, double);`	`fnmsub`
`float __fmadds(float, float, float);`	`fmadds`
`float __fmsubs(float, float, float);`	`fmsubs`
`float __fnmadds(float, float, float);`	`fnmadds`
`float __fnmsubs(float, float, float);`	`fnmsubs`
`double __mffs(void);`	`mffs`
`float __fabsf(float);`	`fabsf`
`float __fnabsf(float);`	`fnabsf`

Buffer Manipulation

Some intrinsics allow control over areas of memory, so you can manipulate memory blocks.

```
void *__alloca(ulong);
__alloca implements alloca() in the compiler.
char *__strcpy(char *, const char *);
```

`__strcpy()` detects copies of constant size and calls `__memcpy()`. This intrinsic requires that a `__strcpy` function be implemented because if the string is not a constant it will call `__strcpy` to do the copy.

```
void *__memcpy(void *, const void *, size_t);
```

`__memcpy()` provides access to the block move in the code generator to do the block move inline.

For More Information: www.freescale.com

9 Support Libraries and Code

CodeWarrior™ Development Studio, mobileGT™ Processor Edition includes many libraries and support files that you can use in your projects. For example, the product includes ANSI-standard C and C++ libraries, runtime libraries, and other support code (such as startup code). This chapter explains how to use these materials.

With respect to the Metrowerks Standard Libraries (MSL) for C and C++, this chapter is an extension to the *MSL C Reference* and the *MSL C++ Reference.* Consult these documents for additional information about the standard libraries and the functions they implement.

The sections of this chapter are:

- Metrowerks Standard Libraries
- Runtime Libraries
- Board Initialization Code

Metrowerks Standard Libraries

These section explain how to use the Embedded PowerPC version of the Metrowerks Standard Libraries (MSL).

- Using the Metrowerks Standard Libraries
- Using Console I/O
- Allocating Memory and Heaps

Using the Metrowerks Standard Libraries

Your CodeWarrior product includes the Metrowerks Standard Libraries (MSL). MSL is a complete C and C++ library collection that you can use in your projects. All of the

Freescale Semiconductor, Inc.

source files required to build MSL are included with your product, along with project files for different MSL configurations.

To use MSL, you must use a version of the runtime libraries. You should not have to modify any of the source files included with MSL. If you have to make changes because of your board's memory configuration, you should make the changes to the runtime libraries.

MSL for Embedded PowerPC supports console I/O through the serial port on each supported evaluation board. The standard C library I/O is supported including `stdio`, `stderr`, and `stdin`. In addition, this version of MSL supports all functions that do not require disk I/O. Further, the memory management functions `malloc()` and `free()` are supported.

You may be able to use a third party standard C library with CodeWarrior product. To tell, compare the file `stdarg.h` of the third party library and the CodeWarrior library. The CodeWarrior EPPC C/C++ compiler can generate correct variable-length argument functions by using the header file included with the MSL. You may find that other implementations are also compatible. You may also need to modify the runtime library to support a different standard C library. In any event, you must include `__va_arg.c`.

You cannot use a third party standard C++ library with your CodeWarrior product.

Finally, if you are using an embedded operating system, you may need to customize MSL to work properly with this operating system.

Using Console I/O

For the console I/O functions of the MSL C and C++ libraries to work, you must include a special serial I/O library in your project. In addition, your hardware must be initialized properly so it works with this library.

Including UART libraries

For the standard C and C++ libraries to handle console I/O, you must include a special serial driver library in your project. The particular library to use depends on your target board.

For each supported target board, there are two version of the serial driver library, one for which `char` is unsigned by default and one for which `char` is signed by default. A library's name indicates which type it is. The naming convention is:

- UART_*Brd_mfr_Brd_name*.UC.a — unsigned char library
- UART_*Brd_mfr_Brd_name*.a — signed char library

For More Information: www.freescale.com

where *Brd_mfr* is a placeholder for the name of the board's manufacturer and *Brd_name* is a placeholder for the name of the board.

Table 9.1 lists the serial I/O library files for the supported evaluation boards. All files listed are in this directory:

```
installDir\PowerPC_EABI_Tools\
MetroTRK\Transport\ppc\board_name\Bin
```

where *board_name* is a placeholder for the name of a particular board.

Table 9.1 Serial I/O Libraries

Board	Library Filename
Motorola Lite5200, rev. I	UART1_MOT_Lite5200.a UART1_MOT_Lite5200.UC.a
Motorola Lite5200, rev. G	UART1_MOT_Lite5200.a UART1_MOT_Lite5200.UC.a
Motorola MPC 823 FADS	UART1_MOT_8XX_ADS.a UART1_MOT_8XX_ADS.UC.a

If your board is not running at the default processor speed, you must modify the appropriate serial driver library so it works with your board. If you make such a change, you must add a baud-rate divisor table tailored to your processor's speed.

To modify a serial driver library, use the CodeWarrior projects included with your product. These project files are in this directory:

```
installDir\PowerPC_EABI_Tools\
MetroTRK\Processor\ppc\Board\board_mfr_name\board_name
```

where *board_mfr_name* is a placeholder for the name of a manufacturer of a supported evaluation board and *board_name* is a placeholder for the name of a particular board.

Allocating Memory and Heaps

The heap you define using the Heap Address text box of the EPPC Linker panel is the default heap. The default heap needs no initialization. The code responsible for memory management is only linked into your code if you call `malloc` or `new`.

You may find that you do not have enough contiguous space available for your needs. In this case, you can initialize multiple memory pools to form a large heap.

For More Information: www.freescale.com

You create each memory pool by calling `init_alloc()`. You can find an example of this call in `__ppc_eabi_init.c` and `__ppc_eabi_init.cpp`. You do not need to initialize the memory pool for the default heap.

Runtime Libraries

Your CodeWarrior product includes many runtime libraries and support code files. These files are in this directory:

installDir\PowerPC_EABI_Support\Runtime

where *installDir* is a placeholder for the path in which you installed your product.

For your projects to build and run, you must include the correct runtime library and and startup code. These sections explain how to pick the correct files:

- Library Naming Conventions
- Required Libraries and Source Code Files

Library Naming Conventions

Substrings embedded in the name of a library indicate the type of support that library provides. Table 9.2 lists and defines the meaning of each substring.

Table 9.2 Runtime Library Naming Conventions

Substring	Meaning
`Runtime`	The library is a C language library.
`Run_EC++`	The library is an embedded C++ library.
`A`	The library provides AltiVec™ support. Does not apply to the mobileGT processor product.
`E`	The library is for e500/Zen targets. Does not apply to the mobileGT processor product.
`E.fast`	The library is for e500/Zen targets. Does not apply to the mobileGT processor product.
`H`	The library supports hardware floating-point operations.
`HC`	The library supports hardware floating-point operations and code compression. Does not apply to the mobileGT processor product.

Table 9.2 Runtime Library Naming Conventions

Substring	Meaning
S	The library provides software emulation of floating-point operations.
N	The library provides no floating-point support.
NC	The library provides no floating-point support, but supports code compression. Does not apply to the mobileGT processor product.
LE	The library is for a processor running in little-endian mode.
UC	The `char` parameters of library functions are `unsigned char`. The linker issues a warning if the `char` "signed-ness" selected in the C/C++ Language target settings panel conflicts with the library in a project.

Required Libraries and Source Code Files

In any C or C++ project, you must include one of these runtime libraries:

- `Runtime.PPCEABI.N.a` or `Run_EC++.PPCEABI.N.a`

  These libraries provide no floating-point support.

- `Runtime.PPCEABI.H.a` or `Run_EC++.PPCEABI.H.a`

  These libraries support hardware floating-point operations.

- `Runtime.PPCEABI.S.a` or `Run_EC++.PPCEABI.S.a`

  These libraries provide software emulation of floating-point operations.

These libraries are in the directory:

`installDir``\PowerPC_EABI_Support\Runtime\Lib\`

where *`installDir`* is a placeholder for the path in which you installed your CodeWarrior product.

In addition, you must include one of the startup files listed below in any C or C++ project. These files contain hooks from the runtime that you can customize if necessary. One kind of customizing is special board initialization. See the actual source file for other kinds of customizations possible.

- `__ppc_eabi_init.c`

  For C language projects.

- `__ppc_eabi_init.cpp`

  For C++ projects.

For More Information: www.freescale.com

Your CodeWarrior product includes the source and project files for the runtime libraries, so you can modify these libraries if necessary.

All runtime library source files are in this directory:

`installDir\PowerPC_EABI_Support\Runtime\Src`

The runtime library project files are in this directory:

`installDir\PowerPC_EABI_Support\Runtime\Project`

The project names are `Runtime.PPCEABI.mcp and Run_EC++.PPCEABI.mcp`. Each project has a different build target for each configuration of the runtime library.

For more information about customizing the runtime library, read the comments in the source files as well as any release notes for the runtime library.

NOTE The C and C++ runtime libraries do not initialize hardware. It is assumed that you load and run the programs with the Metrowerks debugger. When your program is ready to run as a standalone application, you must add the required hardware initialization code.

Board Initialization Code

Your CodeWarrior product includes several basic assembly language hardware initialization routines that you may want to use in your program. When you are debugging, it is not necessary to include this code because the debugger or debug kernel already performs the required board initialization.

If your code is running standalone (without the debugger), you may need to include a board initialization file. These files have the extension `.asm` and are in this directory:

`installDir\PowerPC_EABI_Support\Runtime\Src`

These files are included in source form, so you can modify them to work with other boards or hardware configurations.

Each board initialization file includes a function named `usr_init()`. This is the function you call to run the hardware initialization code. In the normal case, this would be put into the `__init_hardware()` function in either the `ppc_eabi_init.c` or `ppc_eabi_init.cpp` file. In fact, the default `__init_hardware()` function has a call into `usr_init()`, but it is commented out. Remove the comment tokens to have your program perform the hardware initializations.

10

Hardware Tools

This chapter explains how to use the CodeWarrior IDE's hardware tools. Use these tools for board bring-up, testing, and analysis.

The sections are:

- Flash Programmer
- Hardware Diagnostics
- Logic Analyzer

Flash Programmer

The CodeWarrior flash programmer can program the flash memory of a target board with code from any CodeWarrior IDE project or from any individual executable files. The CodeWarrior flash programmer provides features such as:

- Program
- Erase
- BlankCheck
- Verify
- Checksum

NOTE Certain flash programming features (such as view/modify, memory/register, and save memory content to a file) are provided by the CodeWarrior debugger. As a result, the CodeWarrior flash programmer does not include these features.

The CodeWarrior flash programmer uses the CodeWarrior Debugger Protocol API to communicate with the target boards. The CodeWarrior flash programmer runs as a CodeWarrior plug-in.

The CodeWarrior flash programmer lets you use the same IDE to program the flash of any of the embedded target boards.

Freescale Semiconductor, Inc.

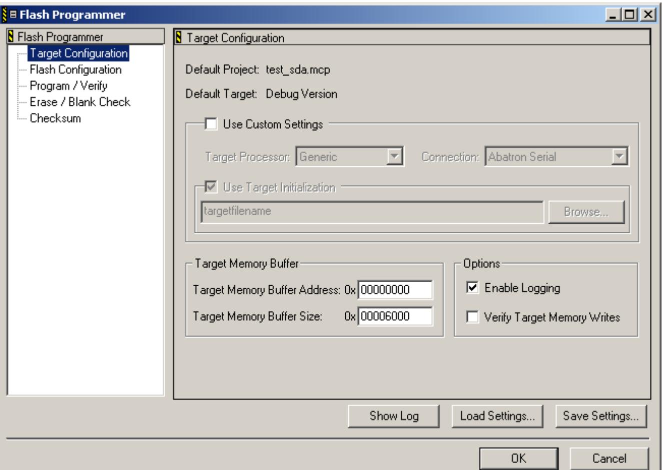
Table 10.1 lists the target boards that have flash modules that you can program using the CodeWarrior Flash Programmer.

Table 10.1 Supported Target Boards and Flash Modules

Target Board	Flash Module
Motorola Lite5200, rev.I	AM29LV065D
Motorola Lite5200, rev.G	AM29LV652D

The **Flash Programmer** window (Figure 10.1) lists global options for the flash programmer hardware tool. These preferences apply to every open project file.

Figure 10.1 Flash Programmer Window



To display the **Flash Programmer** window, select **Tools > Flash Programmer**.

The left pane of the **Flash Programmer** window shows a tree structure of panel names. Click a panel name to display that corresponding panel in the right pane of the **Flash Programmer** window.

Refer to the *IDE User Guide* for information on each panel in the **Flash Programmer** window.

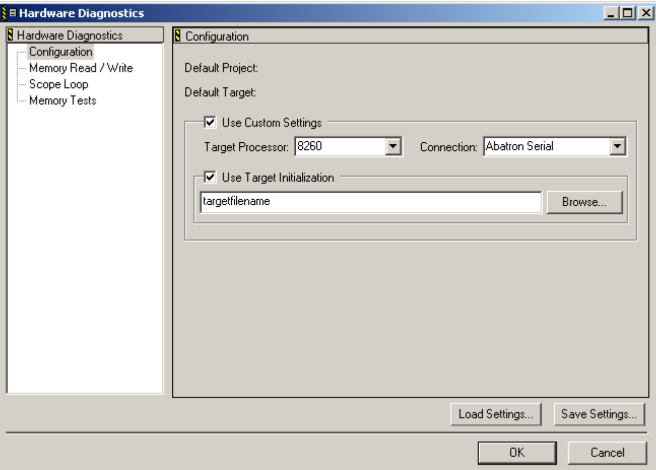
For More Information: www.freescale.com

Hardware Diagnostics

The **Hardware Diagnostics** window (Figure 10.2) lists global options for the hardware diagnostic tools. These preferences apply to every open project file.

Select **Tools > Hardware Diagnostics** to display the **Hardware Diagnostics** window.

Figure 10.2 Hardware Diagnostics window



The left pane of the **Hardware Diagnostics** window shows a tree structure of panel names. Click a panel name to display the corresponding panel in the right pane of the **Hardware Diagnostics** window.

Refer to the *IDE User Guide* for information on each panel in the **Hardware Diagnostics** window.

Logic Analyzer

This section explains how to use the logic analyzer feature of the CodeWarrior IDE. The logic analyzer collects trace data from the target and the debugger correlates the trace data with the currently running source code.

This section has these topics:

- Logic Analyzer Menu
- Logic Analyzer Tutorial

Freescale Semiconductor, Inc.

Logic Analyzer Menu

This topic explains how to use each command in the Logic Analyzer menu. The Logic Analyzer menu is a sub-menu of the Tools menu.

NOTE The Logic Analyzer menu is not available unless a logic analyzer connection has been established. See "Logic Analyzer Tutorial" on page 191 for details on establishing a logic analyzer connection.

The Logic Analyzer menu has these commands:

- Connect
- Arm
- Disarm
- Update Data
- Disconnect

Connect

Select **Tools > Logic Analyzer > Connect** to have the IDE:

- Open a connection to the analyzer
- Load the configuration file (if specified)

NOTE If your configuration file contains data besides the configuration information, the IDE may take a few minutes to load the configuration file.

- Retrieve all the label data for the columns in the Trace Window

To connect to the logic analyzer, the IDE uses the preferences you specify for the analyzer connection. See "Logic Analyzer Tutorial" on page 191 for details.

Arm

The Arm command is available only if the IDE is connected to the logic analyzer.

Select **Tools > Logic Analyzer > Arm** to instruct the logic analyzer to start collecting target cycles.

For More Information: www.freescale.com

Disarm

The Disarm command is not available if there is no connection between the IDE and the logic analyzer.

Select **Tools > Logic Analyzer > Disarm** to instruct the logic analyzer to stop collecting target cycles (*disarm*) if the analyzer is still running. You must disarm the logic analyzer before you update the trace data by using the Update Data command.

Update Data

The Update Data command is only available when the analyzer is disarmed.

Select **Tools > Logic Analyzer > Update Data** to retrieve the most recent trace data and display it in the Trace window. All previous data in the Trace window is replaced by the recent data.

Selecting Update Data does not update the label data for the columns. The label data is retrieved only when the IDE connects to the analyzer device. If the layout of the labels in the Listing window (in the Agilent analyzer) or the Group Name window (in the Tektronix analyzer) has changed, you must first disconnect and then re-connect to get the latest column headings and formats.

The Trace window displays up to 100,000 states or trace frames, beginning with the most recent frame.

Disconnect

Select **Tools > Logic Analyzer > Disconnect** to disconnect the system from the analyzer device. The IDE clears all the data in the Trace window.

Logic Analyzer Tutorial

The tutorial that follows explains how to use the logic analyzer functionality of the IDE to collect target cycles, retrieve trace data, and display trace data.

The tutorial refers to this hardware setup:

- Agilent logic analyzer
- Motorola MPC 8260 ADS board
- The Agilent 16700B modular frame with three 16717A boards

Freescale Semiconductor, Inc.

NOTE	The 16717A boards in slot A and C are configured as slaves to the master board in Slot B.

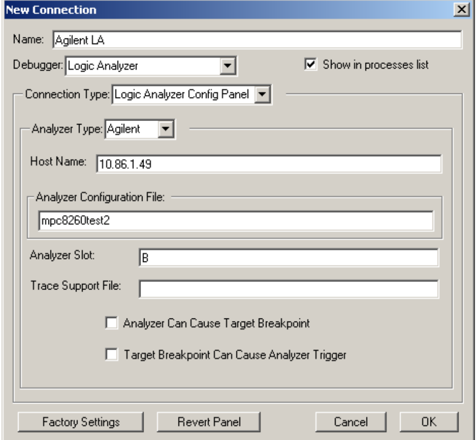
- A CodeWarrior IDE project configured to use a WireTAP JTAG remote connection to the MPC8260 ADS target.

To use the IDE's logic analyzer support, follow these steps:

1. Open your project.
   a. Start the CodeWarrior IDE.
   b. Select **File > Open**
      The **Open** dialog box appears.
   c. Navigate to the directory where you have stored your project.
   d. Select the project file name.
   e. Click **Open**
      The project window appears.
2. Create a logic analyzer connection.
   a. Click **Edit > IDE Preferences**.
      The **IDE Preferences** window appears.
   b. Select the Remote Connections item from the **IDE Preference Panels** list.
      The **Remote Connections** preference panel appears.
   c. In the **Remote Connections** preference panel, click **Add**
      The **New Connection** dialog box (Figure 10.3) appears.

For More Information: www.freescale.com

Figure 10.3 Logic Analyzer Connection Preferences



d. Type the connection name in the Name text box.

   For example, type `Agilent LA`.

e. Select the Logic Analyzer item from the Debugger listbox.

f. Select the Logic Analyzer Config Panel item from the Connection Type listbox.

g. Select the logic analyzer name from the Analyzer Type listbox.

   For example, select Agilent.

h. Type the IP address of the host machine in the Host Name text box.

i. In the Analyzer Configuration File text box, enter the name of the analyzer configuration file to be downloaded on the logic analyzer file system.

   For example, type `mpc8260test2`.

   To find out which configuration file to use for your target, refer to the analyzer trace support package documentation.

NOTE If you only enter the analyzer configuration file name in the Analyzer Configuration File text box, the system downloads the file at this location on the analyzer file system: `/logic/config`. If you want to download the configuration file somewhere else on the analyzer file system, enter the full path from root. For example, `/logic/config/myconfig/myconfigfile`. If you leave the

Analyzer Configuration File text box blank, you must load an analyzer configuration file through the Analyzer GUI. In this case, the logic analyzer connection will not load a configuration file.

j. In the Analyzer Slot text box, type the slot name that identifies the logic analyzer location.

k. In the Trace Support File text box, type the name of the file that the logic analyzer requires to support the collection of trace data.

l. Check the Analyzer Can Cause Target Breakpoint checkbox if you want to to let the logic analyzer cause a hardware breakpoint.

m. Check the Target Breakpoint Can Cause Analyzer Trigger checkbox if you want to let a hardware breakpoint trigger the logic analyzer.

n. Click **OK**

The system saves the connection settings.

o. In the **IDE Preferences** window, click **OK**

The **IDE Preferences** window closes.

3. Select the analyzer connection for your project.

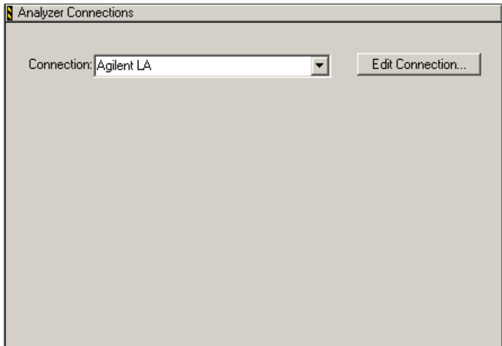
a. While your project window is active, select **Edit > Debug Version Settings**.

The **Target Settings** window appears.

b. Select the **Analyzer Connections** item from the **Target Setting Panels** list.

The **Analyzer Connections** settings panel (Figure 10.4) appears.

Figure 10.4 Analyzer Connections Panel



c. Select the analyzer connection name from the Connection list.

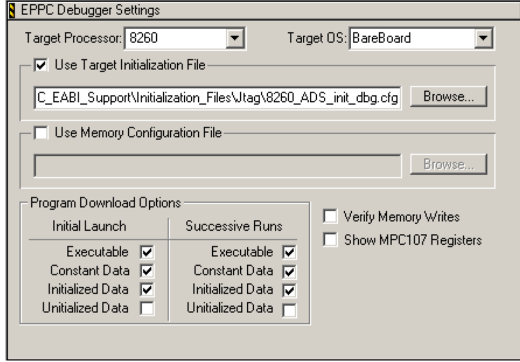
NOTE	Each build target supports just one connection to a logic analyzer. If your project must have more logic analyzer connections, create a separate build target for each additional connection.

4. Configure debugger settings of your project.

   a. Select EPPC Debugger Settings from the target settings panel list.

      The **EPPC Debugger Settings** panel (Figure 10.5) appears.

Figure 10.5 EPPC Debugger Panel



   b. Check the Use Target Initialization File checkbox.

   c. In the text box associated with the Use Target Initialization File checkbox, type the name of the target initialization file required by your target board. Alternatively, click **Browse** to display a dialog box you can use to select the required file.

      Table A.1 on page 200 lists the generic initialization file for each supported target board. These file are on this path:

      installDir`\PowerPC_EABI_Support\`
      `Initialization_Files\Jtag\`

   d. Click **OK**

      The system saves the target settings.

5. Connect the analyzer pods.

   Table 10.2 shows the Agilent analyzer pod connection scheme.

Freescale Semiconductor, Inc.

NOTE The pod connections are dependent on the trace support package installed on your analyzer. This package is available from the analyzer vendor. To know about the pod connection scheme for your target board, refer to the package documentation of the analyzer.

Table 10.2 Agilent Logic Analyzer Pods Connection Scheme

DS Connector	Signals	Analyzer Pod
P12	TS, AACK, etc.	A1/A2
P14	(A0-A31)	B1/B2
P15	SDCAS, SDRAS, etc.	B3/B4
P17	(D0-D31)	C3/C4
P18	(D32-C63)	C1/C2
No Connect		A3/A4

6. While your project window is active, select **Project > Debug**.

   The Debugger window appears.

7. Connect to the logic analyzer.

   a. Select **Tools > Logic Analyzer > Connect**.

      The IDE connects to the logic analyzer.

   b. Select **Tools > Logic Analyzer > Arm**

      The system instructs the logic analyzer to collect target cycles (*arm*). This is equivalent to invoking the Run command on the logic analyzer.

   c. In the debugger window, step through the code once.

      Stepping through code may generate trace frames in the analyzer. The analyzer's trigger mechanism affects if and when frames are collected.

NOTE While the analyzer is armed the debugger periodically queries the analyzer for its Run status.

   d. Select **Tools > Logic Analyzer > Disarm**

      The system instructs the logic analyzer to stop collecting target cycles.

For More Information: www.freescale.com

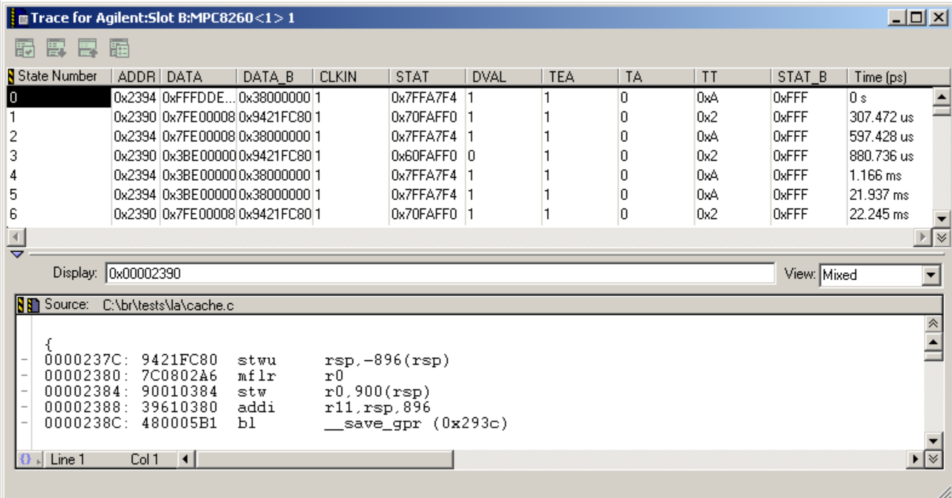
e. Select **Tools > Logic Analyzer > Update Data**

The system retrieves the trace data from the analyzer's buffer.

f. Select **Data > Trace View**

The trace window (Figure 10.6) appears. The trace window displays the data collected.

Figure 10.6 Trace Window



g. Select **Tools > Logic Analyzer > Disconnect**

The system disconnects from the logic analyzer. The IDE erases the contents of the Trace window.

A

Debug Initialization Files

A debug initialization file is used to initialize the target board before the debugger downloads your program's code. The primary purpose of an initialization file is to ensure that the target memory is initialized properly before it is accessed.

The sections are:

- Using Debug Initialization Files
- Debug Initialization File Commands

Using Debug Initialization Files

A debug initialization file is a command file that is executed each time you invoke the debugger. It is usually necessary to include an initialization file if debugging via BDM or JTAG to ensure that the target memory is initialized correctly and that any register values that must be set for debugging purposes are set correctly. You specify whether to use an initialization file and which file to use in the EPPC target settings panel.

NOTE You do not need an initialization file if debugging with MetroTRK.

Example initialization files are provided for the supported evaluation boards. These files are in this directory:

`installDir\PowerPC_EABI_Support\Initialization_Files\Jtag\`

where `installDir` is a placeholder for the path in which you installed your CodeWarrior product.

Table A.1 lists the generic initialization file for each supported target board.

Table A.1 Evaluation Board Initialization Files

Board	Configuration File Location
Motorola Lite5200, rev. I	`Lite5200_init.cfg`
Motorola Lite5200, rev. G	`Lite5200_init.cfg`
Motorola MPC 823 FADS	`8xx_FADS_init.cfg`

Debug Initialization File Commands

This section explains debug initialization file commands, and has these sections:

- Debug Initialization File Command Syntax
- Descriptions and Examples of Commands

Debug Initialization File Command Syntax

The following list shows the rules for the syntax of debug initialization file commands.

- Any white spaces and tabs are ignored.
- Character case is ignored in all commands.
- You can enter a number in hexadecimal, octal, or decimal:
  - Hexadecimal - preceded by 0x (0x00002222 0xA 0xCAfeBeaD)
  - Octal - preceded by 0 (0123 0456)
  - Decimal - starts with 1-9 (12 126 823643)
- Comments start with a “;” or “#”, and continue to the end of the line.

Descriptions and Examples of Commands

This section has the descriptions and examples of these commands:

- reset
- setMMRBaseAddr
- sleep
- writemem.b
- writemem.w
- writemem.l

- writemmr
- writereg
- writespr
- writeupma
- writeupmb

Each subsection explains these individual command lists:

- The command name
- A brief description of the command
- Command usage (prototype)
- Command examples
- Any important notes about the command

reset

The `reset` command is specific to debugging through the CCS protocol.

Description	This command determines a target reset depending on its parameter.
Usage	`reset <value>`, where `<value>` can be 0 or 1. Value 0 determines a reset to user and value 1 determines a reset to debug.
Example	`reset 0`

setMMRBaseAddr

The `setMMRBaseAddr` command works only with target boards that use the 825x/ 826x processors.

Description	The debugger requires the base address of the memory mapped registers on the 825x/826x since this register is memory mapped itself. This command must be in all debug initialization files for the 825x/826x processors. This command informs the debugger plug-in of the base address, which allows you to send any `writemmr` commands from the debug initialization file, as well as read the memory mapped registers for the register views.
Usage	`setMMRBaseAddr<value>`, where `<value>` is the base address for the memory mapped registers.
Example	`setMMRBaseAddr 0x0f00000`

For More Information: www.freescale.com

sleep

Description	Causes the processor to wait the specified number of milliseconds before continuing to the next command.
Usage	`sleep <value>`
Example	`sleep 10    # sleep for 10 milliseconds`

writemem.b

Description	Writes data to a memory location using a byte as the size of the write.
Usage	`writemem.b <address> <value>`, where: • `<address>` — the hex, octal, or decimal address in memory to modify • `<value>` — the hex, octal, or decimal value to write at the address
Example	`writemem.b 0x0001FF00  0xFF   # Write 1 byte to memory`

writemem.w

Description	Writes data to a memory location using a word as the size of the write.
Usage	`writemem.w <address> <value>`, where: • `<address>` — the hex, octal, or decimal address in memory to modify • `<value>` — the hex, octal, or decimal value to write at the address
Example	`writemem.w 0x0001FF00  0x1234   # Write 2 bytes to memory`

writemem.l

Description	Writes data to a memory location using a long as the size of the write.
Usage	`writemem.l <address> <value>`, where: • `<address>` — the hex, octal, or decimal address in memory to modify • `<value>` — the hex, octal, or decimal value to write at the address
Example	`writemem.l 0x00010000 0x00000000 # Write 4 bytes to memory`

For More Information: www.freescale.com

writemmr

Description	Writes a value to the specified MMR (Memory Mapped Register). All memory mapped register names for the supported processors should be accepted by this command. If any registers are found to not be supported, writemem commands can be used to accomplish the register modification.
Usage	`writemmr < register name> <value>`

Example	`writemmr SYPCR    0xffffffc3`
	`writemmr RMR      0x0001`
	`writemmr MPTPR    0x3200`

writereg

Description	Writes data to the specified register on the target. All register names that are part of the core processor are supported including GPRs and SPRs.
Usage	`writereg <registerName> <value>`
Example	`writereg MSR 0x00001002`

writespr

Description	Writes the value to the SPR with number regNumber, which is the same as writereg SPRxxxx but allows you to enter the SPR number in other bases (hex/octal/decimal).
Usage	`writespr <regNumber> <value>`, where: • `<regNumber>` — a hex/octal/decimal SPR number (0-1023) • `<value>` — a hex/octal/decimal value to write to SPR
Example	`writespr 638 0x02200000`

writeupma

Description	Maps the user-programmable machine (UPM) registers to define characteristics of the memory array.

Usage	`writeupma <offset> <ram_word>`, where: • `<offset>` — 0-3F, as defined in the UPM transaction type table in the Memory Controller section of the Motorola manual • `<ram_word>` — UPM RAM word for that offset
Example	`writeupma 0x08 0xffffcc24`

writeupmb

Description	Maps the user-programmable machine (UPM) registers to define characteristics of the memory array.
Usage	`writeupma <offset> <ram_word>`, where: • `<offset>` — 0-3F, as defined in the UPM transaction type table in the Memory Controller section of the Motorola manual • `<ram_word>` — UPM RAM word for that offset
Example	`writeupma 0x08 0xffffcc24`

For More Information: www.freescale.com

B

Memory Configuration Files

A memory configuration file contains commands that define the accessible areas of memory for your specific board.

The sections are:

- Command Syntax
- Memory Configuration File Commands

Command Syntax

The syntax rules for configuration file commands are:

- All syntax is case insensitive.
- Any white spaces and tabs are ignored.
- Comments can be standard C or C++ style comments.
- A number may be entered in hexadecimal, octal, or decimal.
  - Hexadecimal — preceded by 0x (0x00002222 0xA 0xCAfeBeaD)
  - Octal — preceded by 0 (0123 0456)
  - Decimal — starts with 1-9 (12 126 823643)

Memory Configuration File Commands

This section lists the command name, its usage, a brief explanation of the command, examples of how the command may appear in configuration files, and any important notes about the command.

Sample configuration files can be found at this location in CodeWarrior installation directory: `PowerPC_EABI_Support\Intialization_Files\Memory`

range

Description	Allows you to specify a memory range for reading and/or writing, and its attributes.
Usage	`range <loAddr> <hiAddr> <sizeCode> <access>`, where: • `<loAddr>` — start of memory range to be defined • `<hiAddr>` — ending address in the memory range to be defined • `<sizeCode>` — specifies the size, in bytes, to be used for memory accesses by the debug monitor or emulator. • `<access>` — can be `Read`, `Write`, or `ReadWrite`. This parameter allows you to make certain areas of your memory map read-only, write-only, or read/write only to the debugger.

Example	`range    0xFF000000 0xFF0000FF 4 Read`
	`range    0xFF000100 0xFF0001FF 2 Write`
	`range    0xFF000200 0xFFFFFFFF 1 ReadWrite`

reserved

Description	Allows you to specify a reserved range of memory. Any time the debugger tries to read from this location, the memory buffer is filled with the reservedchar. Any time the debugger tries to write to any of the locations in this range, no write will take place.
Usage	`reserved <loAddr> <hiAddr>`, where: • `<loAddr>` — start of memory range to be defined • `<hiAddr>` — ending address in memory range to be defined
Example	`reserved 0xFF000024 0xFF00002F`

reservedchar

Description	Allows you to specify a reserved character for the memory configuration file. This character is seen when you try to read from an invalid address. When an invalid read occurs, the debugger fills the memory buffer with this reserved character.
Usage	`reservedchar <char>`, where `<char>` can be any character (one byte).
Example	`reservedchar 0xBA`

C

Command-Line Tool Options

This appendix lists and defines each option you can pass to the command-line versions of the CodeWarrior software development tools.

The sections are:

- Compiler/Linker Options
- Disassembler Options

Compiler/Linker Options

Table C.1 lists the command-line options for the CodeWarrior EPPC C/C++ compiler and the CodeWarrior EPPC linker.

Table C.1 Compiler/Linker Command-line Options

<table>
<tr><th>Option</th><th colspan="2">Description</th></tr>
<tr><td><code>-big</code></td><td colspan="2">Generates code and links for a big-endian target; this option is the default.</td></tr>
<tr><td><code>-little</code></td><td colspan="2">Generates code and links for a little-endian target.</td></tr>
<tr><td rowspan="4"><code>-proc[essor]</code> keyword</td><td colspan="2">Specifies the processor for scheduling and inline assembler.</td></tr>
<tr><td>Parameter</td><td>Description</td></tr>
<tr><td><code>5200, 823</code></td><td>These are the processor numbers.</td></tr>
<tr><td><code>generic</code></td><td>This is the default option.</td></tr>
</table>

Freescale Semiconductor, Inc.

Table C.1 Compiler/Linker Command-line Options

<table>
<tr><th>Option</th><th colspan="2">Description</th></tr>
<tr><td rowspan="7"><code>-fp</code> keyword</td><td colspan="2">Specifies floating-point code generation options.</td></tr>
<tr><td>Parameter</td><td>Description</td></tr>
<tr><td><code>efpu | spfp</code></td><td>e500 SPE-EFPU hardware FP plus software double FP emulation.
This option is only applicable to e500 family of processors, which are not supported by this product.</td></tr>
<tr><td><code>none | off</code></td><td>Indicates not to use floating-point.</td></tr>
<tr><td><code>soft[ware]</code></td><td>Indicates software floating-point emulation; this option is the default.</td></tr>
<tr><td><code>hard[ware]</code></td><td>Hardware floating-point codegen.</td></tr>
<tr><td><code>fmadd</code></td><td>Same as the following items:
<code>-fp hard</code>
<code>-fp_contract</code></td></tr>
<tr><td><code>-sdata[threshold]</code> short</td><td colspan="2">Sets the maximum size in bytes for mutable data objects before being spilled from a small data section into a data section; the default is 8.</td></tr>
<tr><td><code>-sdata2[threshold]</code> short</td><td colspan="2">Sets the maximum size in bytes for constant data objects before being spilled from a constant section into a data section; the default is 8.</td></tr>
<tr><td rowspan="4"><code>-model</code> keyword</td><td colspan="2">Specifies the code model.</td></tr>
<tr><td>Parameter</td><td>Description</td></tr>
<tr><td><code>absolute</code></td><td>Specifies absolute code and data addressing; this is the default option.</td></tr>
<tr><td><code>sda_pic_pid</code></td><td>SDA PIC/PID</td></tr>
</table>

For More Information: www.freescale.com

Table C.1 Compiler/Linker Command-line Options

<table>
<tr><th>Option</th><th colspan="2">Description</th></tr>
<tr><td rowspan="7"><code>-abi</code> keyword</td><td colspan="2">Specifies the ABI to use.</td></tr>
<tr><td>Parameter</td><td>Description</td></tr>
<tr><td><code>eabi</code></td><td>Specifies EABI; this is the default option.</td></tr>
<tr><td><code>SysV</code></td><td>Specifies SysV ABI without gnu-isms</td></tr>
<tr><td><code>SuSE</code></td><td>Specifies SuSE Linux with gnu-ism</td></tr>
<tr><td><code>YellowDog</code></td><td>Specifies YellowDog Linux with gnu-isms</td></tr>
<tr><td><code>sda_pic_pid</code></td><td>Specifies SDA PIC/PID</td></tr>
<tr><td><code>-g[dwarf]</code></td><td colspan="2">Generates DWARF 1.x debugging information.</td></tr>
<tr><td><code>-gdwarf-2</code></td><td colspan="2">Generates DWARF 2.x debugging information.</td></tr>
<tr><td rowspan="5"><code>-align keyword[,...]</code></td><td colspan="2">Specifies structure and array alignment options.</td></tr>
<tr><td>Parameter</td><td>Description</td></tr>
<tr><td><code>power[pc]</code></td><td>Specifies PowerPC alignment; this option is the default.</td></tr>
<tr><td><code>mac68k</code></td><td>Specifies Macintosh 680x0 alignment.</td></tr>
<tr><td><code>mac68k4byte</code></td><td>Specifies Mac 680x0 4-byte alignment.</td></tr>
<tr><td></td><td><code>array[members]</code></td><td>Specifies to align members of arrays.</td></tr>
<tr><td><code>-common on|off</code></td><td colspan="2">Specifies whether to move all uninitialized data into a common section; the default is off.</td></tr>
<tr><td><code>-fp_contract | -maf on|off</code></td><td colspan="2">Specifies whether to generate fused multiply-add instructions; the default is off.</td></tr>
</table>

Freescale Semiconductor, Inc.

Table C.1 Compiler/Linker Command-line Options

<table>
<tr><th>Option</th><th colspan="2">Description</th></tr>
<tr><td rowspan="8"><code>-func_align</code> keyword</td><td colspan="2">Specifies function alignment.</td></tr>
<tr><td>Parameter</td><td>Description</td></tr>
<tr><td><code>4</code></td><td>Specifies four-byte alignment; this is the default.</td></tr>
<tr><td><code>8</code></td><td>Specifies eight-byte alignment.</td></tr>
<tr><td><code>16</code></td><td>Specifies 16-byte alignment.</td></tr>
<tr><td><code>32</code></td><td>Specifies 32-byte alignment.</td></tr>
<tr><td><code>64</code></td><td>Specifies 64-byte alignment.</td></tr>
<tr><td><code>128</code></td><td>Specifies 128-byte alignment.</td></tr>
<tr><td><code>-pool[data] on|off</code></td><td colspan="2">Specifies whether to pool like data objects; the default is on.</td></tr>
<tr><td><code>-profile on|off</code></td><td colspan="2">Specifies whether to generate calls at function entry and exit for use with a profiler.</td></tr>
<tr><td><code>-rostr | -readonlystrings</code></td><td colspan="2">Specifies to make string constants read-only.</td></tr>
<tr><td><code>-schedule on|off</code></td><td colspan="2">Specifies whether to schedule instructions; the default is off.</td></tr>
<tr><td><code>-use_lmw_stmw on|off</code></td><td colspan="2">Specifies whether to use multiple-word load/store instructions for structure copies; the default is on.</td></tr>
</table>

For More Information: www.freescale.com

Disassembler Options

Table C.2 lists the command-line options for the CodeWarrior EPPC disassembler.

Table C.2 Disassembler Command-line Options

<table>
<tr><th>Option</th><th colspan="2">Description</th></tr>
<tr><td rowspan="3"><code>-fmt | -format</code> keyword</td><td colspan="2">Specifies formatting options; this option exists for compatibility reasons.</td></tr>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td><code>[no]x</code></td><td>Specifies whether to show extended mnemonics; the default is to not show the extended mnemonics.</td></tr>
<tr><td rowspan="3"><code>-show</code> keyword<code>[,...]</code></td><td colspan="2">Specifies display options.</td></tr>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td><code>only | none</code></td><td>Examples:

<code>-show none</code>
<code>-show only,code,data</code></td></tr>
</table>

Freescale Semiconductor, Inc.

Table C.2 Disassembler Command-line Options

Option	Description	
	`all`	Specifies to show everything.
	`[no]binary`	Specifies whether to show binary information, such as addresses and opcodes, for object code; the default is to show the binary information.
	`[no]code` | `[no]text`	Specifies whether to show `.text` sections; the default is to show the `.text` sections.
	`[no]data`	Specifies whether to show data; the default is to show data.
	`[no]detail`	Specifies whether to show detailed dump information.
	`[no]extended`	Specifies whether to show extended mnemonics; the default is to show extended mnemonics.
	`[no]exceptions` | `[no]xtab[les]`	Specifies whether to show exception tables; these options also imply the following item: `-show data`
	`[no]headers`	Specifies whether to show object headers; the default is to show the object headers.
	`[no]debug` | `[no]dwarf`	Specifies whether to show DWARF information.
	`[no]tables`	Specifies whether to show string and symbol tables; the default is to show the string and symbol tables.
	`[no]xtables`	Specifies whether to show exception tables.

For More Information: www.freescale.com

Table C.2 Disassembler Command-line Options

Option	Description
`-[no]relocate`	For DWARF information, specifies whether to relocate addends in `.rela.text` and `.rela.debug`.
`-xtables on|off`	Specifies whether to show exception tables; the default is off. This option exists for compatibility reasons.

For More Information: www.freescale.com

Freescale Semiconductor, Inc.

For More Information: www.freescale.com

D

Using the Dhrystone Benchmark Software with the Lite5200

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 show you how to use the Dhrystone benchmark example program included with your CodeWarrior™ software. This example works with a Motorola Lite5200 evaluation board. You can use the example as the basis for your own Dhrystone benchmark programs.

The sections 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.

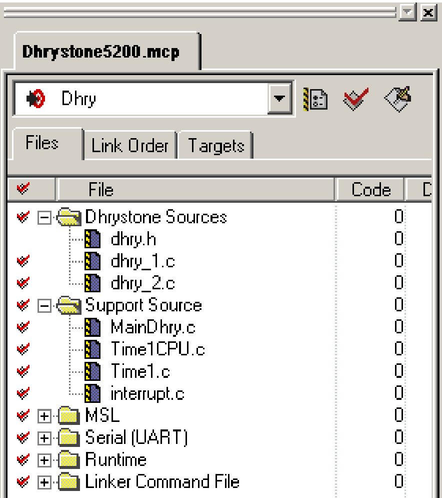
For More Information: www.freescale.com

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 Motorola Lite5200 evaluation board.

Running the Dhrystone Program

To run the Dhrystone example program on a Lite5200 board, 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\`

3. Connect your debug hardware to the Lite5200 and to your PC.

   For example, connect a WireTAP run-control tool to the JTAG port of the Lite5200 and to a parallel port of your PC.

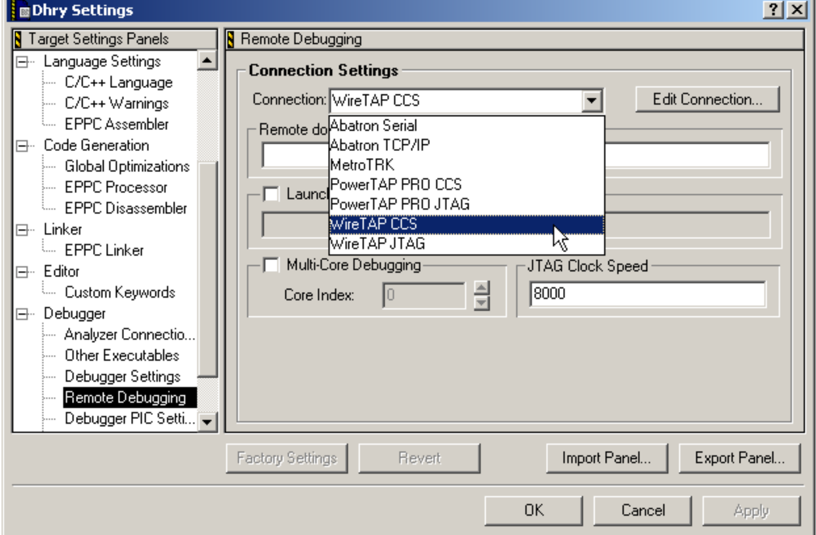
4. Press **Alt-F7**

   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 listbox, select the remote connection appropriate for your debug hardware.

7. Click **Edit Connection**

   A "remote connection" dialog box appears. Use this dialog box to configure your debug hardware.

   See "Supported Remote Connections" on page 87 for a definition of each option for each available remote connection.

Freescale Semiconductor, Inc.

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.

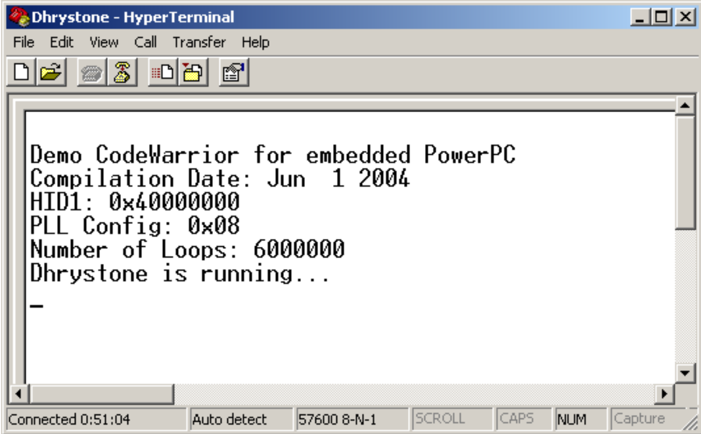
11. Start a terminal emulation program and configure it as follows:

    - Bits per second — `57600`
    - Data bits — `8`
    - Parity — `None`
    - Stop bits — `1`
    - Flow control — `None`

12. From the menu bar of the IDE, select **Project > Run**

    The IDE 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.)

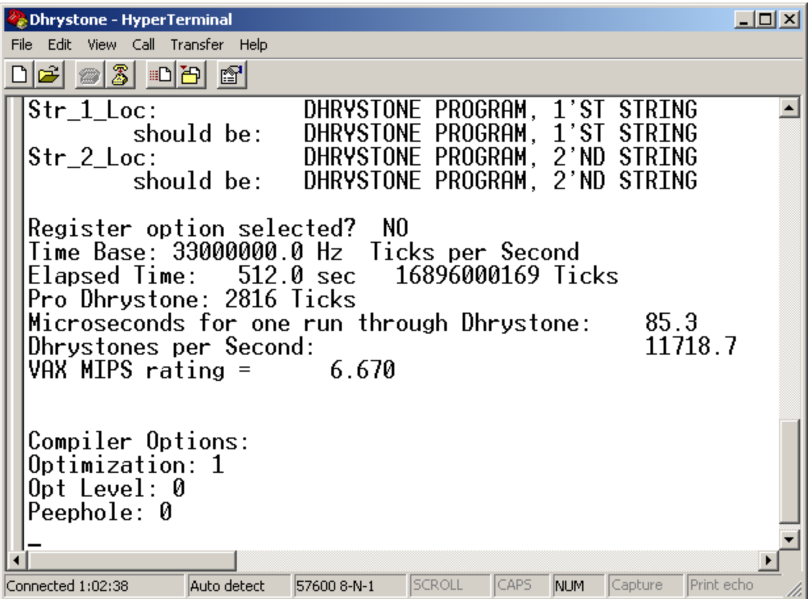
Figure D.3 Terminal Emulator Showing Test "Start" Information



13. Upon completion, the Dhrystone example program displays the results of its tests in the terminal emulator window. (See Figure D.4.)

For More Information: www.freescale.com

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.

For More Information: www.freescale.com

Index

Symbols

__abs() 177
__attribute__ ((aligned(?))) 159–161
__cntlzw() 179
__eieio() 177
__fabs() 177
__fnabs() 177
__isync() 177
__labs() 177
__lhbrx() 178
__lwbrx() 178
__rlwimi() 178
__rlwinm() 178
__rlwnm() 178
__setflm() 178
__sthbrx() 178
__stwbrx() 178
__sync() 177

A

Additional Options 86
Additional Path to PC-lint compiler options files 83
additional small data sections 143–147
  how to create 144–147
asm blocks not supported 164
asm keyword 164
assembler
  stand-alone described 20
  See also inline assembler
Author Options 85

B

back-end compiler *See* compiler
BatchRunner
  post-linker 43
  pre-linker 42
benchmark software, Dhrystone, using with Lite5200 215
binary files 35
board initialization code 186
bool size 126
build folder 77
build target, defined 19

C

char size 126
CodeWarrior
  compared to command-line 22
  compiler described 20
  components 17
  debugger described 21
  development process 22–24
  IDE described 17
  installing 14–15
  linker described 21
  project manager described 18
  registering 15–16
  stand-alone assembler described 20
  tools listed 17
command syntax
  memory configuration files 205
command-line and CodeWarrior compared 22
command-line tool options
  EPPC-specific compiler/linker options 207
  EPPC-specific disassembler options 211
commands
  memory configuration files syntax 205
comments in inline assembler 168
compiler
  back-end for PowerPC 125
  described 20
  other documentation 125
  support for inline assembly 163
  See also C Compilers Reference
Compiler Option 83
compiling 23
configuration files 200
configuration files, memory, command syntax of 205
console I/O 182
  UART libraries 182
converting, makefiles to CodeWarrior project 36–38
current folder 77

D

Data Addressing 127
data cache window 106

deadstripping unused code 148
debug initialization files
  command syntax 200
  commands
    setMMRBaseAddr 201
    sleep 202
    writemem.b 202
    writemem.l 202
    writemem.w 202
    writemmr 203
    writereg 203
    writespr 203
    writeupma 203
    writeupmb 204
  using 199
debugger features, special
  displaying registers 100
  EPPC menu 101–106
  register details 106
Debugger PIC Settings panel 73
debugger, described 21
debugging 24
  ELF files 120–124
  for PowerPC Embedded 87–112
  supported remote connections 87–99
  using MetroTRK 111
  See also Debugger User Guide
development tools 17
  project manager
    project, defined 18
    project, related terms 19
Dhrystone benchmark software
  example project
    building 215
    running 216
  using with Lite5200 215
directives, assembler
  entry 174
  fralloc 175
  frfree 175
  machine 175
  nofralloc 176
  opword 176
disassembly, Register Details window 106
Display default PC-lint compiler option files too 84
Display generated command lines in message window 82
`double` size 127

E

EABI templates, for PowerPC projects 36
editing code 23
  See also IDE User Guide
`__eieio()` 177
ELF files, debugging 120–124
`entry` assembly statement 174
`entry` directive 174
EPPC Assembler settings panel 50–51
EPPC Debugger Settings panel 74–76
EPPC Disassembler settings panel 63–64
EPPC Linker settings panel 65–72
EPPC menu, options explained 101–106
EPPC Processor settings panel 54–62
EPPC Target panel 44–49
EXCLUDEFILES 150

F

flash modules, supported 188
`float` size 127
floating-point formats 127
floating-point support 55
force_active 134
FORCEACTIVE 151
FORCEFILES 151
FPSCR 178
`fralloc` assembly statement 169
`fralloc` directive 175
`frfree` assembly statement 169
`frfree` directive 175
function level assembly 164
function_align 134

G

Global Optimizations settings panel 52–53
GROUP 151

H

Hardware tools
  flash programmer 187–188
  hardware diagnostics 189
  logic analyzer 189–197
host, defined 19

I

IDE described 17
INCLUDEDDWARF 152
incompatible_return_small_structs 134
incompatible_sfpe_double_params 135
inline assembler
  asm blocks not supported 164
  comments 168
  directives 174–176
  for PowerPC 163
  function level support 164
  instructions 163, 164
  local variables 168
  operands 170
  preprocessor use 168
  special PowerPC instructions 166
  stack frame 169
  statement labels 167
  syntax 164
  using for PowerPC 163
inline_max_auto_size 133
installing CodeWarrior software 14–15
int size 127
integer formats 126
integer formats for PowerPC 126
interrupt 135
intrinsic functions
  described 176
  See also inline assembler 176

L

labels in inline assembly language 167
__labs() 177
libraries
  console I/O 182
  MSL for PowerPC Embedded 181
  runtime 184
  support for PowerPC Embedded 181–186
  UART 182
  using MSL 181–182
Library Options 85
Library Warnings 86
link order 148
linker
  .a files 148
  .o files 148
  and executable files 149
  described 21
  for PowerPC 143–149
  multiply defined symbols 148
  other documentation 125
linker generated symbols 147
linking 24
  See also IDE User Guide
local variables in inline assembler 168
L1 Data Cache window 106
long double size 127
long long size 127
long size 127

M

machine assembly statement 175
machine directive 175
Makefile Importer wizard, using 36–38
MEMORY 154
memory configuration, MetroTRK 110–111
MetroTRK
  connecting 109
  memory configuration 110–111
  overview 108
  using with Lite5200 for debugging 112
  using, for debugging 111
Metrowerks Standard Libraries *See* MSL
MISRA 85
MSL
  and runtime libraries 182
  described 22
  for PowerPC Embedded 181
  using 181–182
  using console I/O 182
  See also MSL C Reference, MSL C++ Reference
multiple symbols and linker 148

N

No inter-modul 83
nofralloc directive 176
number formats
  floating-point 127
  for PowerPC 126–127
  integers 126

O

operands in inline assembler 170

opt_findoptimalunrollfactor 132
opt_full_unroll_limit 132
opt_unroll_count 133
opt_unroll_instr_count 133
opt_unrollpostloop 133
optimizing
  for PowerPC 130
  inline assembly disables 165
  register coloring 130
opword directive 176

P

pack 136
PC-lint Executable 82
platform target, defined 19
pooled_data 137
PowerPC EABI templates, using 36
PowerPC Embedded debugging *See* debugging
ppc_no_fp_blockmove 133
pragma
  for PowerPC 131
  overload 149
Prefix File 84
preprocessing 24
  See also IDE User Guide
preprocessor, using in inline assembler 168
project
  build target, defined 19
  build target, example 19–20
  defined 18
  host, defined 19
  platform target, defined 19
  related terms 19
  types of 35
project manager described 18
project targets
  debug version 38
  ROM version 38

R

REGISTER 153
Register 129
register coloring optimization 130
register details 106
Register Details window 106
register variables for PowerPC 129
registering CodeWarrior software 15–16
registers
  Register Details window 106
  variables 129
registers, displaying 100
requirements *See* system requirements
__rlwimi() 178
__rlwinm() 178
__rlwnm() 178
runtime libraries
  and MSL 182
  customizing 186
  for PowerPC Embedded 184
  in projects 185
  initializing hardware 186

S

section 137
section pragma 137
SECTIONS 155
__setflm() 178
setMMRBaseAddr 201
settings panels
  Debugger PIC Settings 73
  EPPC Assembler 50–51
  EPPC Debugger Settings 74–76
  EPPC Disassembler 63–64
  EPPC Linker 65–72
  EPPC Processor 54–62
  EPPC Target 44–49
  Global Optimizations 52–53
  Source Folder Mapping 76–78
  System Call Service Settings 79
  Target Settings 40
short size 127
SHORTEN_NAMES_FOR_TOR_101 157
signed char size 126
sleep 202
small data sections, additional 143–147
Source Folder Mapping panel 76–78
special debugger features 100–108
stack frame in inline assembler 169
Stack Size edit field 47
stand-alone assembler 20
  See also Assembler Guide
statement labels, in inline assembly language 167
__sthbrx() 178

For More Information: www.freescale.com

__stwbrx() 178
supported target boards 14
symbols
  linker generated 147
  multiple linker 148
__sync() 177
System Call Service Settings 79
system requirements 13

T

target boards supported 14
target initialization files 200
Target Settings panel 40

U

UART libraries
  and console I/O 182
  and processor speed 183
unsigned char size 127
unsigned int size 127
unsigned long long size 127
unsigned long size 127
unsigned short size 127

V

variables
  register 129

W

Warnings 86
writemem.b 202
writemem.l 202
writemem.w 202
writemmr 203
writereg 203
writespr 203
writeupma 203
writeupmb 204

For More Information: www.freescale.com