

CodeWarrior™ Development Studio PowerPC™ ISA Communications Processors Edition Targeting Manual

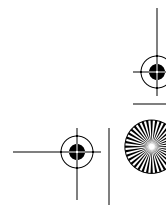


Revised: 28 March 2005

metrowerks



For More Information: www.freescale.com





Metrowerks, the Metrowerks logo, and CodeWarrior are trademarks or registered trademarks 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 © 2005 by Metrowerks, a Freescale Semiconductor company. 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 are 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 e-mail: sales@metrowerks.com
Technical Support	United States Voice: 800-377-5416 International Voice: +1-512-996-5300 e-mail: support@metrowerks.com



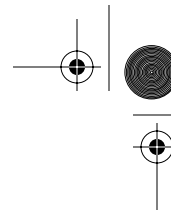


Table of Contents

1	Introduction	13
	Overview of This Manual	13
	Related Documentation	14
	CodeWarrior Information	14
	Embedded PowerPC Programming Information	15
	AltiVec Information	16
	CodeWarrior Development Tools	16
	CodeWarrior IDE	16
	CodeWarrior C/C++ Compiler	17
	CodeWarrior Assembler	17
	CodeWarrior Linker	17
	CodeWarrior Debugger	18
	Metrowerks Standard Libraries	18
	CodeWarrior Development Process	18
	Project Files	19
	Editing Code	19
	Compiling	19
	Linking	20
	Debugging	20
	Viewing Preprocessor Output	20
2	Working With Projects	21
	Types of Projects	21
	Using the EPPC New Project Wizard	22
	Using PowerPC EABI Templates	26
	Using the Makefile Importer Wizard	26
3	Working With Libraries and Support Code	29
	Metrowerks Standard Libraries	29
	Using the Metrowerks Standard Libraries	29
	Using Console I/O	30
	Allocating Memory and Heaps	31



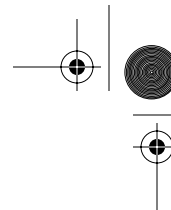
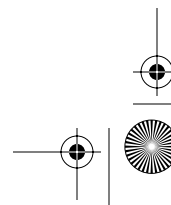


Table of Contents

Runtime Libraries	32
Library Naming Conventions	32
Required Libraries and Source Code Files	33
Board Initialization Code	34
4 Working With the C/C++ Compiler and Linker	35
Integer and Floating-Point Formats	36
Embedded PowerPC Integer Formats	36
Embedded PowerPC Floating-Point Formats	38
AltiVec Vector Data Formats	38
Data Addressing	40
Register Variables	41
Register Coloring Optimization	42
Pragma Directives	43
force_active	44
function_align	45
incompatible_return_small_structs	45
incompatible_sfpe_double_params	45
interrupt	46
pack	47
pooled_data	48
section	48
Linker Issues for Embedded PowerPC	53
Linker Generated Symbols	53
Deadstripping Unused Code and Data	54
Link Order	54
Linker Command Files	55
EXCLUDEFILES	56
EXTERNAL_SYMBOL	57
FORCEACTIVE	57
FORCEFILES	57
GROUP	57
INCLUDEDWARF	58
INTERNAL_SYMBOL	58
MEMORY	59



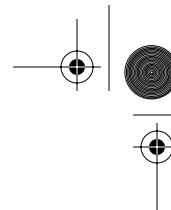
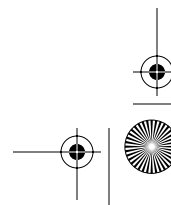


Table of Contents

SECTIONS	60
SHORTEN_NAMES_FOR_TOR_101	62
Using __attribute__((aligned(?)))	64
Variable Declaration Examples	64
Structure Definition Examples	64
Typedef Declaration Examples	65
Structure Member Examples	65
Small Data Area PIC/PID Support	66
Internal and External Segments and References	66
PIC/PID Linker Command File Directives	67
Linker-defined Symbols	67
Uses for SDA PIC/PID	68
Building an SDA PIC/PID Application	69

5 Working With the Inline Assembler 71

Working With Assembly Language	71
Assembler Syntax for Embedded PowerPC	72
Special Embedded PowerPC Instructions	74
Support for AltiVec Instructions	75
Creating Statement Labels	75
Using Comments	76
Using the Preprocessor in Embedded PowerPC Assembly	76
Using Local Variables and Arguments	76
Creating a Stack Frame in Embedded PowerPC Assembly	77
Specifying Operands in Embedded PowerPC Assembly	78
Assembler Directives	81
entry	81
fralloc	82
frfree	82
machine	83
nofralloc	84
opword	84
Intrinsic Functions	84
Low-Level Processor Synchronization	85
Absolute Value Functions	85



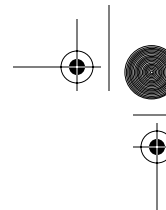
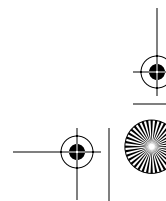


Table of Contents

Byte-Reversing Functions85
Setting the Floating-Point Environment85
Manipulating the Contents of a Variable or Register86
Data Cache Manipulation86
Math Functions87
Buffer Manipulation87
AltiVec Intrinsics Support88
6 Working With the Debugger	91
Using Remote Connections91
Accessing Remote Connections92
Understanding Remote Connections94
Editing Remote Connections95
Special Debugger Features108
Attaching to Processes108
Displaying Registers109
Register Details110
EPPC-Specific Debugger Features112
Using MetroTRK117
MetroTRK Overview117
Connecting to the MetroTRK Debug Monitor118
MetroTRK Memory Configuration120
Using MetroTRK for Debugging121
Debugging External ELF Files122
Additional Considerations125
Debugging Multiple ELF Files126
Debugging a Secondary ELF File with a Serial Connection126
Debugging a Secondary PIC/PID ELF File128
Debugging a Secondary External ELF File130
Accessing Translation Look-aside Buffers131
Initializing TLBs131
Accessing TLB Registers131
Using the Command-Line Debugger133



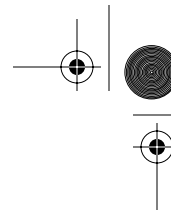
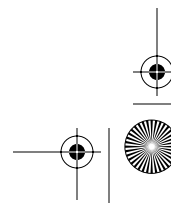


Table of Contents

7	Target Settings Reference	135
	Target Settings Overview	135
	Target Settings	137
	Target Name	137
	Linker	138
	Pre-Linker	138
	Post-Linker	138
	Output Directory	138
	Save Project Entries Using Relative Paths	138
	EPPC Target	138
	Project Type	139
	File Name	139
	Byte Ordering	140
	Disable CW Extensions	140
	DWARF	140
	ABI	140
	Tune Relocations	141
	Code Model	141
	Small Data	141
	Small Data2	142
	Heap Size	142
	Stack Size	142
	Optimize Partial Link	142
	Deadstrip Unused Symbols	143
	Require Resolved Symbols	143
	GNU Target	143
	EPPC Assembler	144
	Source Format	145
	GNU Compatible Syntax	145
	Generate Listing File	145
	Prefix File	145
	GNU Assembler	146
	EPPC Processor	146
	Struct Alignment	147



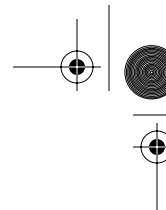
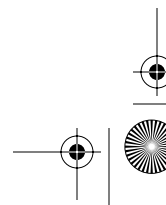


Table of Contents

Function Alignment	147
Processor	147
Floating Point.	148
Vector Support	148
Relax HW IEEE.	149
Use Fused Multi-Add/Sub	149
Generate FSEL Instruction	149
Assume Ordered Compares	149
Altivec Options	150
Altivec Structure Moves	150
Generate VRSAVE Instructions	150
Make Strings Read Only	150
Pool Data	150
Linker Merges FP Constants	151
Use Common Section	151
Use LMW & STMW	152
Inlined Assembler is Volatile	152
Instruction Scheduling	153
Peephole Optimization.	153
Profiler Information	153
e500 Options	153
EPPC Disassembler	153
Show Headers	154
Show Symbol Table	154
Show Code Modules	154
Show Data Modules	155
Show DWARF Info	155
Verbose Info	155
GNU Disassembler	155
Command Line Arguments	156
Show Assembly Output of Compiler When Disassembling Source	156
Display Content of Archive at Time of Disassembly	156
GNU Compiler	156
Command Line Arguments	157
Prefix File	157



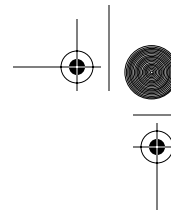
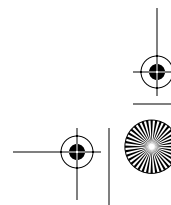


Table of Contents

Use Custom Debug Format	157
EPPC Linker	157
Link Mode	158
Generate DWARF Info	158
Generate Link Map	159
List Closure	159
List Unused Objects	159
List DWARF Objects	159
Suppress Warning Messages	160
Heap Address	160
Stack Address	160
Generate ROM Image	161
RAM Buffer Address	161
ROM Image Address	162
Segment Addresses	162
Use Linker Command File	162
Code Address	162
Data Address	163
Small Data	163
Small Data2	163
Generate S-Record File	164
Sort S-Record	164
Max Length	164
EOL Character	164
Entry Point	165
GNU Post Linker	165
GNU Linker	165
BatchRunner PreLinker	166
BatchRunner PostLinker	167
GNU Environment	167
GNU Tools	168
Use Custom Tool Commands	169
Tool Path	169
Commands	169
Analyzer Connection	170



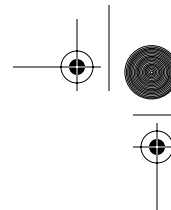
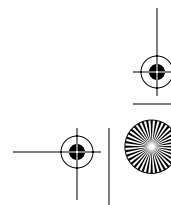


Table of Contents

Debugger PIC Settings	170
EPPC Debugger Settings	171
Target Processor	172
Target OS	172
Use Target Initialization File	172
Use Memory Configuration File	173
Program Download Options	173
Verify Memory Writes	173
PQ3 Trace Buffer	174
Enable Trace collection on Launch	174
Transaction Match Disable	174
Equal Context Enable	175
Not Equal Context Enable	175
Trace Only in TRACE Event	175
Interface Selection	175
Start Condition	176
Stop Condition	177
Source ID Enable	177
Target ID Enable	178
Address Match Enable	178
Source Folder Mapping	178
Build Folder	179
Current Folder	180
Add	180
Change	180
Remove	180
System Call Service Settings	180
Activate Support for System Services	181
Redirect stdout/stderr to	181
Use Shared Console Window	181
Trace Level	182
Redirect Trace to	182
Mount Root Folder to	182



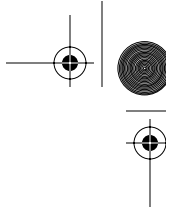
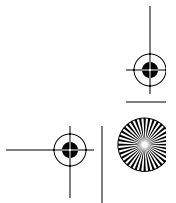


Table of Contents

8	Working With Hardware Tools	183
	Flash Programmer	183
	Hardware Diagnostics	185
	Logic Analyzer	186
	Logic Analyzer Menu	186
	Logic Analyzer Tutorial	187
	Trace Buffer	193
A	Debug Initialization Files	197
	Using Debug Initialization Files	197
	Debug Initialization File Commands	199
	alternatePC	200
	ANDmmr	200
	IncorMMR	201
	reset	201
	run	202
	setMMRBaseAddr	202
	sleep	203
	stop	203
	writemem.b	203
	writemem.w	204
	writemem.l	204
	writemem.r	205
	writemmr	205
	writereg	206
	writereg128	207
	writespr	207
	writeupma	208
	writeupmb	209
B	Memory Configuration Files	211
	Command Syntax	211
	Memory Configuration File Commands	211
	range	212



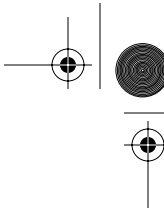
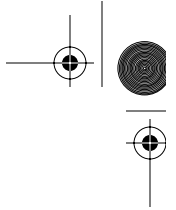


Table of Contents

reserved	212
reservedchar	213
translate	214
C Tested Jumper and Dipswitch Settings	215
Cogent CMA102 with CMA 278 Daughtercard	215
Axiom 555, 565	216
Motorola 555 ETAS	217
Motorola 5100 Ice Cube	218
Motorola Excimer 603e	219
Motorola MPC 8xx MBX	219
Motorola MPC 8xx FADS	220
Motorola MPC 82xx ADS	221
Motorola Maximer 7400	221
Motorola Sandpoint	222
Motorola MPC 8255/8260 ADS	222
Embedded Planet RPX Lite 8xx	223
Index	225



1

Introduction

This manual explains how to install and use the CodeWarrior™ Development Studio for Embedded PowerPC™ ISA Systems tools.

This chapter contains these sections:

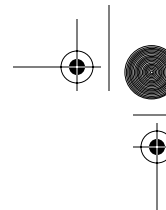
- Overview of This Manual — describes the contents of this manual
- Related Documentation — describes supplementary CodeWarrior documentation, third-party documentation, and references to helpful code examples and web sites
- CodeWarrior Development Tools — gives an overview of the CodeWarrior development tools, including descriptions of each of the major components of the CodeWarrior IDE
- CodeWarrior Development Process — describes the steps you take to write, debug, and create programs with the CodeWarrior IDE

Overview of This Manual

Table 1.1 describes the information contained in each chapter in this manual.

Table 1.1 Chapter Contents

Chapter	Description
Introduction	(this chapter)
Working With Projects	describes the different types of Embedded PowerPC™ projects you can create, provides an overview of CodeWarrior project stationery and wizards, and describes the Makefile Importer Wizard
Working With Libraries and Support Code	describes the ANSI-standard C and C++ libraries, runtime libraries, and other support code (such as startup code) included in this product
Working With the C/C++ Compiler and Linker	describes the CodeWarrior back-end compiler and linker for Embedded PowerPC™ software development
Working With the Inline Assembler	describes support for inline assembly in the CodeWarrior C and C++ compilers



Introduction

Related Documentation

Table 1.1 Chapter Contents

Chapter	Description
Working With the Debugger	describes supported remote connections in the CodeWarrior debugger, special debugger features, how to use MetroTRK to debug target hardware, how to debug ELF files, and how to use the command-line debugger
Target Settings Reference	a reference for target settings panels specific to developing software for Embedded PowerPC™ systems
Working With Hardware Tools	describes how to use the Flash Programmer, Hardware Diagnostics, and Logi Analyzer tools
Debug Initialization Files	describes how to initialize target boards before the debugger downloads code to them
Memory Configuration Files	describes how to define the accessible areas of memory for a specific board
Tested Jumper and Dipswitch Settings	describes various jumper and DIP switch settings you can use with supported hardware

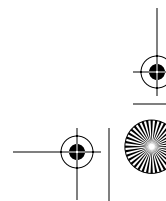
Related Documentation

This section provides information about documentation related to the CodeWarrior IDE and Embedded PowerPC development.

- CodeWarrior Information
- Embedded PowerPC Programming Information
- AltiVec Information

CodeWarrior Information

- For late-breaking information about new features, bug fixes, known problems, and incompatibilities, read the release notes in this folder:
`CWInstall\Release_Notes\`
- Example CodeWarrior projects are in this folder:
`CWInstall\CodeWarrior_Examples\PowerPC_EABI\`
- For general information about the CodeWarrior IDE and debugger, see the *CodeWarrior IDE User's Guide* in this folder:





Introduction Related Documentation

CWInstall\Help\PDF

- For information specific to the C/C++ front-end compiler, read the *C Compilers Reference* and *C++ Compiler's Reference* in this folder:

CWInstall\Help\PDF

- For information about the Metrowerks standard C/C++ libraries, read the *MSL C Reference* and the *MSL C++ Reference* in this folder:

CWInstall\Help\PDF

- For information about MetroTRK, including how to customize MetroTRK to work with a particular target board, read the *MetroTRK Reference* in this folder:

CWInstall\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) and available on the World Wide Web at this address:

<http://www.cloudcaptech.com/downloads.htm>

- *PowerPC Embedded Binary Interface, 32-Bit Implementation*., published by Motorola, Inc., and available on the World Wide Web at this address:

<http://e-www.motorola.com/brdata/PDFDB/docs/PPCEABI.pdf>

The PowerPC EABI specifies data structure alignment, calling conventions, and other information about how high-level languages can be implemented on a Embedded PowerPC processor. The code generated by this product conforms to the PowerPC EABI.

The PowerPC EABI also specifies the object and symbol file format. It specifies ELF (Executable and Linker Format) as the output file format and DWARF (Debug With Arbitrary Record Formats) as the symbol file format. For more information about those 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 and available at this address:





Introduction

CodeWarrior Development Tools

<http://www.nondot.org/sabre/os/files/Executables/dwarf-v1.1.0.pdf>

- *DWARF Debugging Information Format, Revision: Version 2.0.0*, Industry Review Draft, published by UNIX International, Programming Languages SIG, July 27, 1993.

AltiVec Information

To learn more about AltiVec™ technology, see:

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

<http://e-www.motorola.com/brdata/PDFDB/docs/ALTIVECPIM.pdf>

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

<http://e-www.motorola.com/collateral/ALTIVECPMCH.htm>

CodeWarrior Development Tools

Programming for Embedded PowerPC is much like programming for any other CodeWarrior target. If you have never used the CodeWarrior IDE before, the tools you will need to become familiar with are:

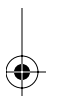
- CodeWarrior IDE
- CodeWarrior C/C++ Compiler
- CodeWarrior Assembler
- CodeWarrior Linker
- CodeWarrior Debugger
- Metrowerks Standard Libraries

If you are an experienced CodeWarrior user, you may need to become familiar with the Embedded PowerPC runtime environment.

CodeWarrior IDE

The CodeWarrior IDE lets you write your software. It controls the project manager, the source code editor, the class browser, the compilers and linkers, and the debugger.

Those who are more familiar with command-line development tools may find the CodeWarrior project new. The project manager organizes all files related to your project.





Introduction

CodeWarrior Development Tools

This allows you to see your project at a glance, and eases the organization of and navigation between your source code files.

The CodeWarrior IDE has an extensible architecture that uses plug-in compilers and linkers to target various operating systems and microprocessors. The CodeWarrior CD includes a C/C++ compiler for the Embedded PowerPC family of processors. Other CodeWarrior packages include C, C++, Pascal, and Java compilers for Mac OS, Win32, and other platforms.

For more information about the CodeWarrior IDE, you should read the *CodeWarrior IDE User's Guide*.

CodeWarrior C/C++ Compiler

The CodeWarrior compiler for Embedded PowerPC (EPPC) is an ANSI-compliant C/C++ compiler. This compiler is based on the same compiler architecture used in all of the CodeWarrior C/C++ compilers. You can generate Embedded PowerPC applications and libraries that conform to the PowerPC EABI by using the CodeWarrior compiler with the CodeWarrior linker for Embedded PowerPC.

For more information about the CodeWarrior C/C++ language implementation, you should read the *C Compilers Reference*.

CodeWarrior Assembler

The CodeWarrior assembler for EPPC is a standalone assembler that has an easy-to-use syntax. The CodeWarrior IDE supported assemblers for other platform targets use the same syntax.

For more information about the CodeWarrior assembler, see the *Assembler Guide*.

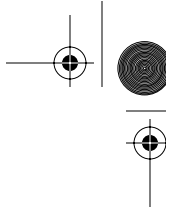
CodeWarrior Linker

The CodeWarrior linker for EPPC is an ELF linker. This linker lets you use absolute addressing to create code. It also lets you create multiple user-defined sections. In addition, you can generate an S-record output file for your application.

The EPPC linker also has a form of PIC/PID support. For more information, refer to this release notice:

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





Introduction

CodeWarrior Development Process

CodeWarrior 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 *CodeWarrior IDE User's 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 CCS.

Hardware protocols require additional hardware to communicate with the target board, such as a PowerTAP Pro, USB TAP, Ethernet TAP, or an MSI Wiggler.

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*.

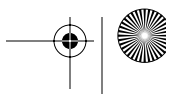
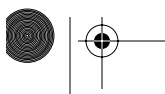
CodeWarrior Development Process

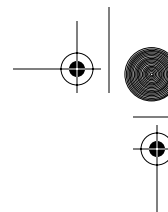
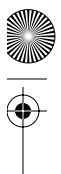
While working with the CodeWarrior IDE, you will proceed through the development stages familiar to all programmers: writing code, compiling and linking, and debugging. See the *CodeWarrior IDE User's Guide* for:

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

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

If you are unfamiliar with an integrated environment in general, or with the CodeWarrior IDE in particular, you may find the topics in this section helpful. Each topic explains how





Introduction

CodeWarrior Development Process

one component of the CodeWarrior tools relates to a traditional command-line environment.

Project Files

The CodeWarrior IDE *project* is analogous to a set of makefiles. Because you can have multiple build targets in the same project, the project is analogous to a collection of makefiles. For example, you can have one project that has both a debug version and a release version of your code. You can build one or the other, or both as you wish. In the CodeWarrior IDE, the different builds within a single project are called “build targets.”

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

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

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

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

Editing Code

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

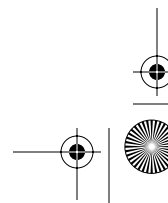
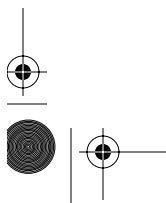
To edit a source code file, or any other editable file that is in a project, double-click the file name in the project window to open the file.

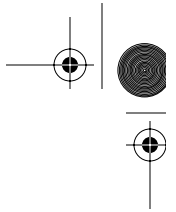
The editor window has excellent navigational features that allow you to switch between related files, locate any particular function, mark any location within a file, or go to a specific line of code.

Compiling

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

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





Introduction

CodeWarrior Development Process

Linking

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

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

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

Debugging

Select **Project>Debug** from the CodeWarrior menu bar to debug your project. This tells the compiler and linker to generate debugging information for all items in your project.

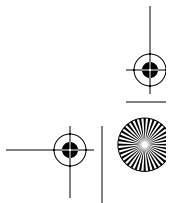
If you want to only generate debug information on a file-by-file basis, click in the debug column for that file. The debug column is located in the project window, to the right of the data column.

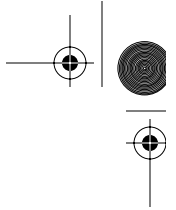
Viewing Preprocessor Output

To view preprocessor output, select the file in the project window and select **Project > Preprocess** from the CodeWarrior menu bar. 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. Rather than sending an entire source tree to technical support, you can preprocess the file causing problems and send the output along with the relevant project settings through e-mail.





2

Working With Projects

This chapter provides an overview of the steps required to create code that runs on Embedded PowerPC™ systems.

This chapter contains these sections:

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

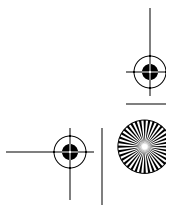
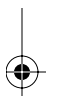
Types of Projects

The CodeWarrior compilers and linkers in this product generate 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.

To learn more about the ways you can create a new project read these sections:

- “Using the EPPC New Project Wizard” on page 22.
- “Using PowerPC EABI Templates” on page 26
- “Using the Makefile Importer Wizard” on page 26



Working With Projects

Using the EPPC New Project Wizard

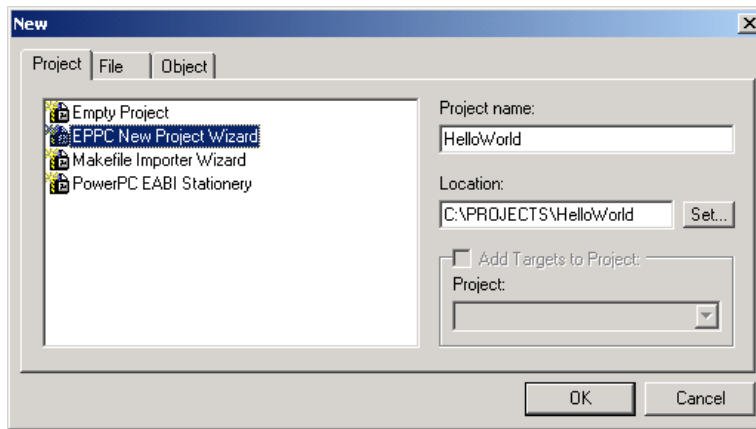
Using the EPPC New Project Wizard

This section explains how to use the EPPC New Project wizard to create a new project. After you create the project, you can modify project settings.

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

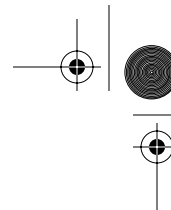
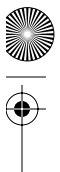
The New dialog box (Figure 2.1) appears.

Figure 2.1 New Dialog Box



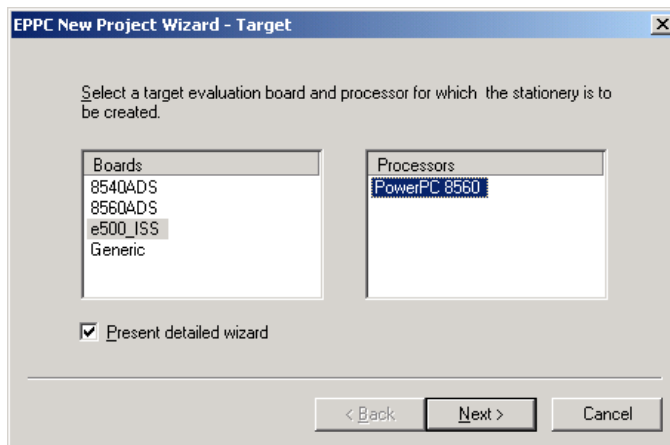
3. Select **EPPC New Project Wizard**.
4. In the **Project Name** text box, type HelloWorld.
5. In the **Location** text box, type the location where you want to save this project, or click **Set** to use a standard file dialog box to select a location.
6. Click **OK**.

The **EPPC New Project Wizard** dialog box appears (Figure 2.2).



Working With Projects
Using the EPPC New Project Wizard

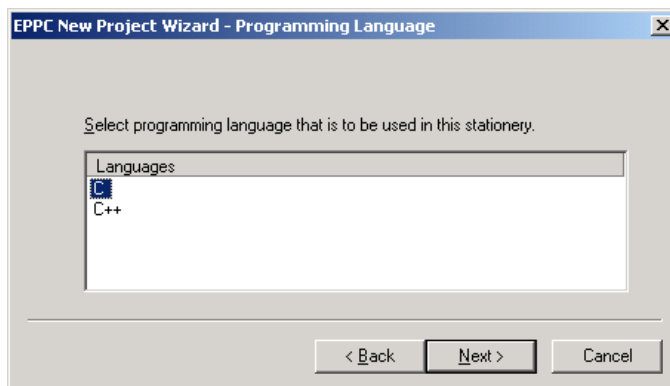
Figure 2.2 EPPC New Project Wizard — Target Page



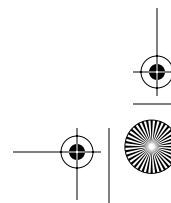
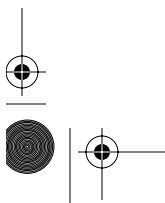
7. From the **Boards** list, select a target board.
8. From the **Processors** list, select a target processor.
9. Check the **Present detailed wizard** checkbox.
10. Click **Next**.

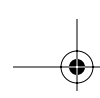
The **Programming Language** page (Figure 2.3) appears.

Figure 2.3 EPPC New Project Wizard — Programming Language Page



11. Select the programming language you want to use.
- For example, if you plan to use C source files in your project, select C.





Working With Projects

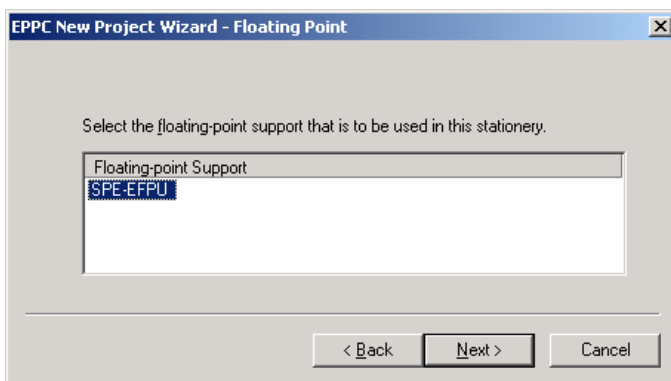
Using the EPPC New Project Wizard

NOTE Selecting a programming language primarily determines the libraries linked to the project and the setup of the main source file. If you select the C++ language, you may still add C source files to the project later.

12. Click the **Next** button.

The **Floating Point** page (Figure 2.4) appears.

Figure 2.4 EPPC New Project Wizard — Floating Point Page

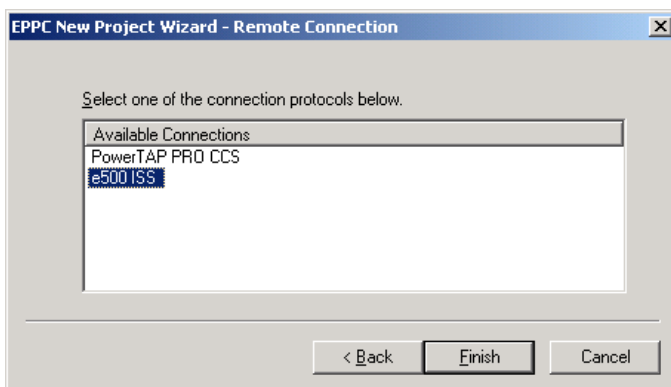


13. From the **Floating-point Support** list, select **SPE-EPPU**.

14. Click **Next**.

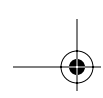
The **Remote Connection** page (Figure 2.5) appears.

Figure 2.5 EPPC New Project Wizard — Remote Connection Page



15. Select the remote connection for the debugger interface you want to use.





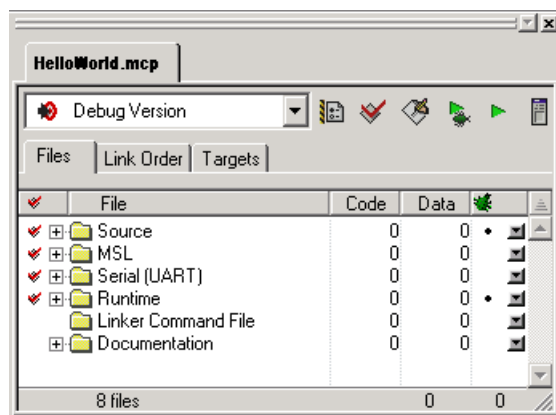
Working With Projects

Using the EPPC New Project Wizard

16. Click **Finish**.

The IDE creates the new project. The project window (Figure 2.6) appears.

Figure 2.6 Project Window



The new project is now ready for use. You can now add your own code to the project. The generated project includes two build targets:

- **Debug Version**

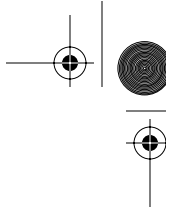
This build target includes only the user code and the standard and runtime libraries. This build target does not perform any hardware initialization or set up any exception vectors. You can continue using this build target until you need ISRs or to flash your code to the ROM.

- **ROM Version**

Use this build 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.

TIP To switch build targets, select **Project > Set Default Target > TargetName** from the CodeWarrior menu bar.





Working With Projects

Using PowerPC EABI Templates

Using PowerPC EABI Templates

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

The EABI template sources are here:

`CWInstall\Templates\PowerPC_EABI\Sources`

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

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

Using the Makefile Importer Wizard

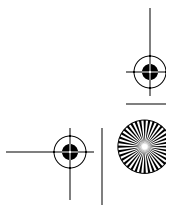
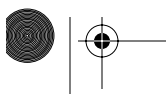
Use the Makefile Importer Wizard to convert most GNU makefiles into CodeWarrior projects. The Makefile Importer wizard lets you:

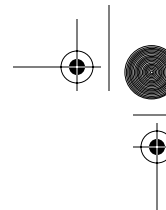
- parse the makefile to determine source files and build targets.
- create a project.
- add the source files and build targets determined during parsing.
- match makefile information, such as output name, output directory, and access paths, with the newly created build targets.
- select a project linker.

To convert a makefile to a CodeWarrior project, follow these steps:

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 with the **.mcp** extension.
 - d. Click **OK**.

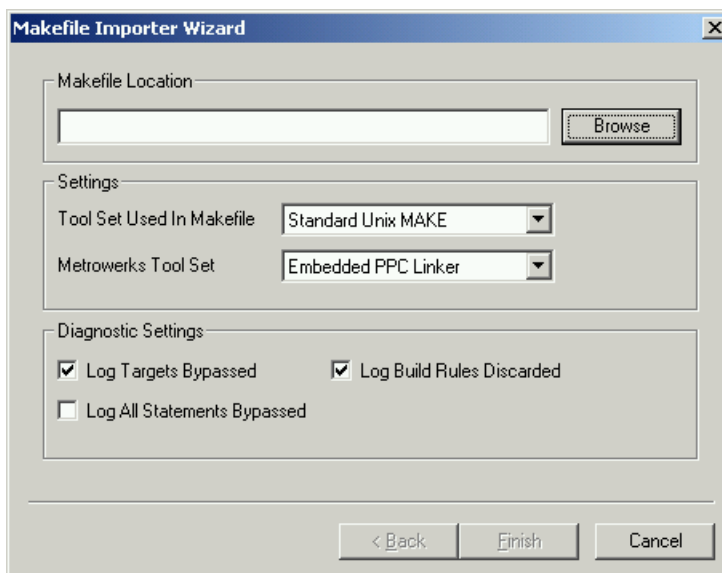
The **Makefile Importer Wizard** dialog box (Figure 2.7) appears.





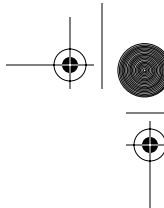
Working With Projects Using the Makefile Importer Wizard

Figure 2.7 Makefile Importer Dialog Box

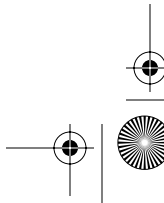


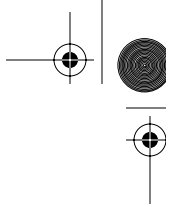
2. In the **Makefile Location** text box, type the path to the makefile. Alternatively, click **Browse** to navigate to the makefile.
3. Select the makefile conversion tool and the linker.
 - a. Use the **Tool Set Used In Makefile** list box to select the tool set that was used to create the make file being imported.
 - b. From the **Metrowerks Tool Set** list box, select **Embedded PPC Linker**.
4. Specify the desired diagnostic settings.
 - Select the **Log Targets Bypassed** checkbox to generate a log file containing information about makefile build targets that the conversion tool fails to convert to project build targets.
 - Select the **Log Build Rules Discarded** checkbox to generate a log file that contains information about makefile rules that the conversion tool discards during conversion.
 - Select the **Log All Statements Bypassed** checkbox 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. Generate the project.
Click **Finish**. The Makefile Importer wizard performs the conversion process and displays the log files you specified.





Working With Projects
Using the Makefile Importer Wizard





3

Working With Libraries and Support Code

CodeWarrior™ Development Studio for Embedded PowerPC ISA Systems 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).

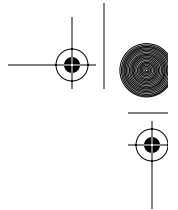
- Using the Metrowerks Standard Libraries
- Using Console I/O
- Allocating Memory and Heaps

Using the Metrowerks Standard Libraries

Metrowerks Standard Libraries (MSL) is a complete C and C++ library collection that you can use in your projects. All of the 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





Working With Libraries and Support Code

Metrowerks Standard Libraries

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.

Serial Driver Binary Files

Serial driver library binary files are in this subdirectory of the CodeWarrior installation directory (where *BoardName* is the name of the target board):

```
CWInstall\PowerPC_EABI_Tools\MetroTRK\Transport\ppc\BoardName\Bin
```

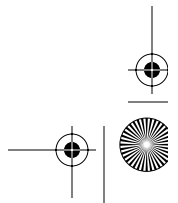
For each supported target board, there are two serial driver library versions: one for which `char` is unsigned by default, and one for which `char` is signed by default. Libraries with filenames that end with `.UC.a` are unsigned char libraries. Libraries with filenames that end with just `.a` are signed char libraries.

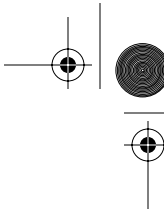
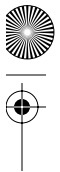
The serial driver library filename convention also uses these substrings:

- *Manufacturer* — the name of the target board manufacturer
- *BoardName* — the name of the particular target board

For example, `UART_Manufacturer_BoardName.UC.a` is an unsigned char library. `UART_Manufacturer_BoardName.a` is a signed char library.

Table 3.1 lists the serial driver library files for supported evaluation boards.





Working With Libraries and Support Code

Metrowerks Standard Libraries

Table 3.1 Serial I/O Libraries

Board	Library Filename
Motorola 5100 Ice Cube	UART1_MOT_5100_Icecube.a UART1_MOT_5100_Icecube.UC.a
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

Serial Driver Source Code

If a target board is not running at the default processor speed, you must modify the appropriate serial driver library source code so it works with the board. If you make such a change, you must add a baud-rate divisor table tailored to the target board's processor speed.

To modify a serial driver library, use the CodeWarrior projects included with your product. Serial driver library project files are in subfolders of the CodeWarrior installation directory:

CWInstall\PowerPC_EABI_Tools\MetroTRK\Processor\ppc\Board

The Board directory contains one subfolder for each board manufacturer. Within each manufacturer folder is a subfolder for each supported target board. To locate the serial driver library project for a given board, replace these elements of the full path:

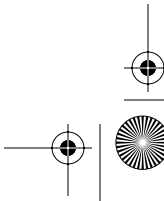
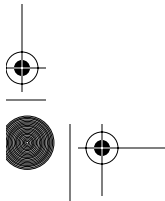
- *Manufacturer* — the name of the target board manufacturer
- *BoardName* — the name of the particular target board

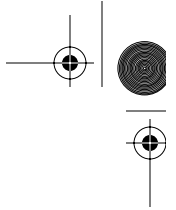
For a given target board, the full path to the serial driver library project is:

CWInstall\PowerPC_EABI_Tools\MetroTRK\Processor\ppc\Board\Manufacturer\BoardName

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`.





Working With Libraries and Support Code

Runtime Libraries

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.

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:

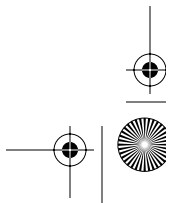
- Library Naming Conventions
- Required Libraries and Source Code Files

Library Naming Conventions

Substrings embedded in the name of a library indicate the type of support that library provides. Table 3.2 lists and defines the meaning of each substring.

Table 3.2 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 comm/host processor product.
E	The library is for e500/Zen targets. Does not apply to the comm/host processor product.
E.fast	The library is for e500/Zen targets. Does not apply to the comm/host processor product.
H	The library supports hardware floating-point operations.





Working With Libraries and Support Code

Runtime Libraries

Table 3.2 Runtime Library Naming Conventions (*continued*)

Substring	Meaning
HC	The library supports hardware floating-point operations and code compression. Does not apply to the comm/host processor product.
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 comm/host 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.

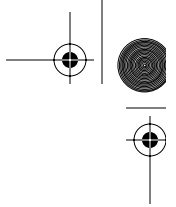
These libraries are in the directory:

`CWInstall\PowerPC_EABI_Support\Runtime\Lib\`

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)





Working With Libraries and Support Code

Board Initialization Code

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:

`CWInstall\PowerPC_EABI_Support\Runtime\Src`

The runtime library project files are in this directory:

`CWInstall\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.

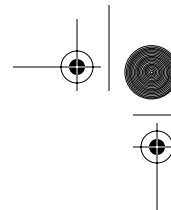
If your code is running standalone (without the debugger), you may need to include a board initialization file. These files have the extension `.asm` and are in this directory:

`CWInstall\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.





4

Working With the C/C++ Compiler and Linker

This chapter documents the back-end of the Metrowerks Embedded PowerPC C/C++ compiler. In addition, the chapter documents the Metrowerks Embedded PowerPC linker.

The *back-end* of the compiler is the module that generates code for the target processor. The *front-end* is the compiler module that parses and interprets C/C++ source code. The *C Compilers Reference* documents the compiler's front-end.

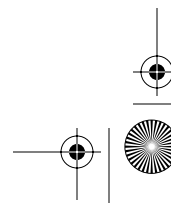
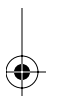
This chapter contains these sections:

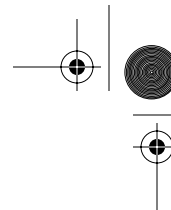
- Integer and Floating-Point Formats
- Data Addressing
- Register Variables
- Register Coloring Optimization
- Pragma Directives
- Linker Issues for Embedded PowerPC
- Using `__attribute__ ((aligned(?)))`
- Small Data Area PIC/PID Support

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 4.1 lists other useful compiler and linker documentation.

Table 4.1 Other Compiler and Linker Documentation

For this topic	Refer to
How the CodeWarrior IDE implements the C/C++ language	<i>C Compilers Reference</i>
Using C/C++ Language and C/C++ Warnings settings panels	<i>C Compilers Reference</i> , "Setting C/C++ Compiler Options" chapter
Controlling the size of C++ code	<i>C Compilers Reference</i> , "C++ and Embedded Systems" chapter





Working With the C/C++ Compiler and Linker

Integer and Floating-Point Formats

Table 4.1 Other Compiler and Linker Documentation (*continued*)

Using compiler pragmas	<i>C Compilers Reference</i> , “Pragmas and Symbols” chapter
Initiating a build, controlling which files are compiled, handling error reports	<i>CodeWarrior IDE User’s Guide</i> , “Compiling and Linking” chapter
Information about a particular error	<i>Error Reference</i> , which is available online
Embedded PowerPC assembler	<i>Assembler Guide</i>
PowerPC EABI calling conventions	<i>System V Application Binary Interface, Third Edition</i> , published by UNIX System Laboratories, 1994 (ISBN 0-13-100439-5) <i>System V Application Binary Interface, PowerPC Processor Supplement</i> , published by Sun Microsystems and IBM, 1995

Integer and Floating-Point Formats

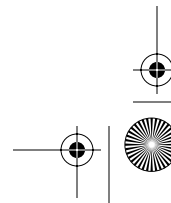
This section describes how the CodeWarrior C/C++ compilers implement integer and floating-point types for Embedded PowerPC processors. You also can read `limits.h` for more information on integer types, and `float.h` for more information on floating-point types. The `altivec.h` file provides more information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

The topics in this section are:

- Embedded PowerPC Integer Formats
- Embedded PowerPC Floating-Point Formats
- AltiVec Vector Data Formats

Embedded PowerPC Integer Formats

Table 4.2 shows the size and range of the Embedded PowerPC compiler’s integer types.



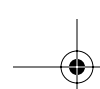


Working With the C/C++ Compiler and Linker Integer and Floating-Point Formats

Table 4.2 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 <i>C Compilers Reference</i>).	8 bits	-128 to 127
	Use Unsigned Chars is on	8 bits	0 to 255
signed char	n/a	8 bits	-128 to 127
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





Working With the C/C++ Compiler and Linker Integer and Floating-Point Formats

Embedded PowerPC Floating-Point Formats

Table 4.3 shows the sizes and ranges of the floating-point types of the embedded PowerPC compiler.

Table 4.3 PowerPC Floating Point 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

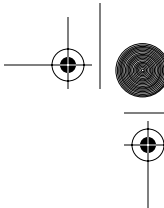
AltiVec Vector Data Formats

There are 11 new vector data types for use in writing AltiVec-specific code. (See Table 4.4). All the types are a constant size, 128 bits or 16 bytes. This is due to the AltiVec programming model, which is optimized for quantities of this size.

Table 4.4 AltiVec Vector Data Types

Vector Data Type	Bytes	Contents	Possible Values
<code>vector unsigned char</code>	16	16 unsigned char	0 to 255
<code>vector signed char</code>	16	16 signed char	-128 to 127
<code>vector bool char</code>	16	16 unsigned char	0 = false, 1 = true
<code>vector unsigned short [int]</code>	16	8 unsigned short	0 to 65535
<code>vector signed short [int]</code>	16	8 signed short	-32768 to 32767
<code>vector bool short [int]</code>	16	8 unsigned short	0 = false, 1 = true
<code>vector unsigned long [int]</code>	16	4 unsigned int	0 to $2^{32} - 1$
<code>vector signed long [int]</code>	16	4 signed int	-2^{31} to $2^{31}-1$





Working With the C/C++ Compiler and Linker
Integer and Floating-Point Formats

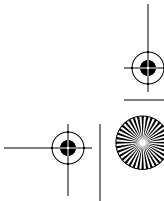
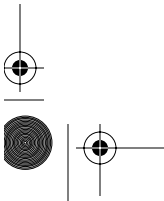
Table 4.4 AltiVec Vector Data Types (continued)

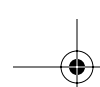
<code>vector bool long [int]</code>	16	4 unsigned int	0 = false, 1 = true
<code>vector float</code>	16	4 float	any IEEE-754 value
<code>vector pixel</code>	16	8 unsigned short	1/5/5/5 pixel

In the table, the `[int]` portion of the Vector Data Type is optional.

There are two additional keywords besides `pixel` and `vector`, `__pixel` and `__vector`. These keywords can be used in C or C++ code.

`bool` is not a reserved word in C unless it is used as an AltiVec vector data type.





Working With the C/C++ Compiler and Linker

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 Listing 4.1 becomes something like Listing 4.2.

Listing 4.1 Source Code

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

Listing 4.2 Generated Code

```
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)
```

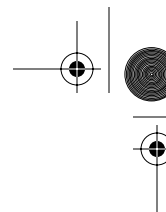
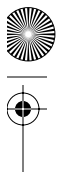
Each variable access takes two instructions and a total of four bytes to make a simple assignment. If we set the small data threshold in the **EPPC Target** panel to be at least the size of an **int**, we can fetch the variables with one instruction.

```
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.





Working With the C/C++ Compiler and Linker

Register Variables

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 code in Listing 4.3 has a zero small data threshold.

Listing 4.3 Zero Small Data Threshold

```
lis    r3,...bss.0@ha
addi   r3,r3,...bss.0@l
li     r0,1
stw    r0,0(r3)
li     r0,2
stw    r0,4(r3)
```

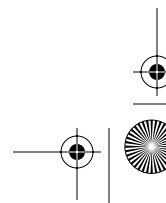
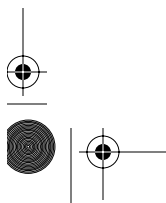
When pooled data is implemented, the first used variable of either the `.data`, `.bss` or `.rodata` section gets a two-instruction fetch of the first variable in that section. Subsequent fetches in that function use the register containing the already-loaded section address with a calculated offset.

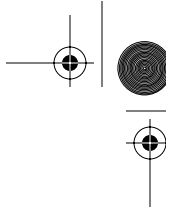
NOTE You can access small data in assembly files with the two-instruction fetch used with large data, because any data on your board can be accessed as if it were large data. The opposite is not true; large data can never be accessed with small data relocations (the linker issues an error if you try 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





Working With the C/C++ Compiler and Linker

Register Coloring Optimization

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

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 4.4, the compiler could place *i* and *j* in the same register:

Listing 4.4 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 4.4, *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





Working With the C/C++ Compiler and Linker

Pragma Directives

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 *C Compilers Reference*.

Pragma Directives

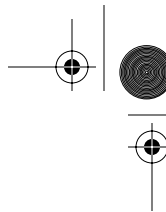
This section lists pragmas supported for PowerPC development and explains additional pragmas for Embedded PowerPC development.

Table 4.5 lists the pragmas supported by all versions of the PowerPC C/C++ compiler, including the Embedded PowerPC compiler. The *C Compilers Reference* documents these pragmas.

Table 4.5 General PowerPC C/C++ Compiler Pragmas

<code>align</code>	<code>align_array_members</code>
<code>ANSI_strict</code>	<code>ARM_conform</code>
<code>auto_inline</code>	<code>bool</code>
<code>check_header_flags</code>	<code>cplusplus</code>
<code>cpp_extensions</code>	<code>dont_inline</code>
<code>dont_reuse_strings</code>	<code>enumsalwaysints</code>
<code>exceptions</code>	<code>extended_errorcheck</code>
<code>fp_contract</code>	<code>global_optimizer</code>
<code>has8bytebitfields</code>	<code>ignore_oldstyle</code>
<code>longlong</code>	<code>longlong_enums</code>
<code>mark</code>	<code>no_register_save_helpers</code>
<code>once</code>	<code>only_std_keywords</code>
<code>optimize_for_size</code>	<code>optimizewithasm</code>
<code>peephole</code>	<code>pop</code>





Working With the C/C++ Compiler and Linker

Pragma Directives

Table 4.5 General PowerPC C/C++ Compiler Pragas (continued)

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
warn_unusedarg	warn_unusedvar
wchar_type	

Table 4.6 lists the pragmas supported by just the Embedded PowerPC (ELF/DWARF) C/C++ compiler. These pragmas are documented in this section.

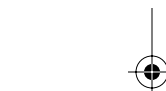
Table 4.6 Embedded PowerPC-Specific C/C++ Compiler Pragas

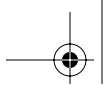
force_active	function_align
incompatible_return_small_structs	incompatible_sfpe_double_params
interrupt	pack
pooled_data	section

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.





Working With the C/C++ Compiler and Linker

Pragma Directives

NOTE `#pragma force_active` cannot be used with uninitialized variables because of language restrictions with 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 list box in the EPPC Processor settings panel.

incompatible_return_small_structs

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

This pragma makes CodeWarrior-built object files more compatible with those created using a GNU compiler.

As per PowerPC EABI settings, structures that are up to 8 bytes in size are to 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 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.

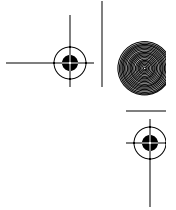
NOTE Different versions of GCC may fix these incompatibilities, so you should check your version if you will be mixing GCC and CodeWarrior objects.

incompatible_sfpe_double_params

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

This pragma makes CodeWarrior-built object files more compatible with those created with a GCC compiler.





Working With the C/C++ Compiler and Linker

Pragma Directives

The PowerPC EABI states that software floating-point double parameters always begin on an odd register. In other words, if you have the 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 if 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 create interrupt handlers in C and C++. For example:

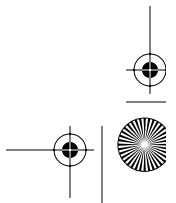
```
#pragma interrupt on
```

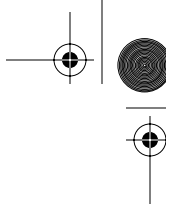
```
void MyHandler(void)
{
    my_real_handler();
}
```

```
#pragma interrupt off
```

The PowerPC architecture allows for 256 bytes at the interrupt vector. If the routine is larger, you can put an `interrupt_routine` at the site of the interrupt vector and the `interrupt_routine` can be any size. The compiler warns you if you are exceeding 256 bytes. You can pass the option `nowarn` to eliminate the warning.

Using the interrupt pragma saves all used volatile general purpose registers, as well as the CTR, XER, LR and condition fields. Then these registers and condition fields are restored before the RFI. You can optionally save certain special purpose registers (such as SRR0 and SRR1, DAR, DSISR), floating-point registers (`fprs`) AltiVec Registers (`vrs`), as well as re-enable interrupts while in the handler.





Working With the C/C++ Compiler and Linker

Pragma Directives

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 bitfields 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 bitfields.

For example, Listing 4.5 would be smaller if it were written as shown in Listing 4.6.

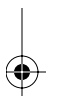
Listing 4.5 Before

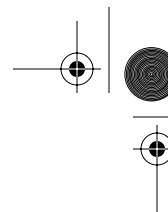
```
typedef struct red {
    unsigned a: 1;
    unsigned b: 1;
    unsigned c: 1;
} red;
```

Listing 4.6 After

```
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 bitfields. If you need portability, you are probably better off using shifts and masks instead of bitfields.





Working With the C/C++ Compiler and Linker

Pragma Directives

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.

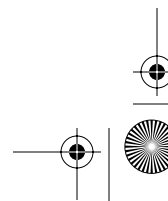
Parameters

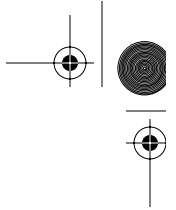
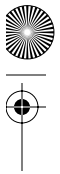
objecttype

This optional 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.





Working With the C/C++ Compiler and Linker

Pragma Directives

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`.

permission

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

- R — read only permission
- W — write permission
- X — execute permission

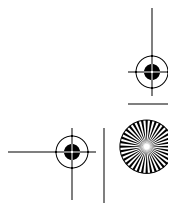
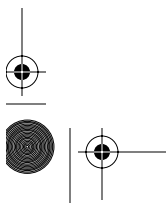
For information on access permission, see “Section access permissions” on page 51. Specify one or more of these permissions in any order, without quotes, and no spaces.

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 `xxxxxxx` 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.





Working With the C/C++ Compiler and Linker

Pragma Directives

Listing 4.7 shows an example where specific uninitialized variables are stored in the COMM section.

Listing 4.7 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.

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 can be used only 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.

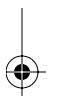
The optional `code_mode=codemode` parameter tells the compiler what kind of addressing mode to use for referring to executable routines of a section.

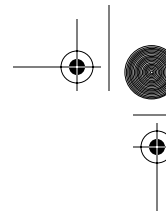
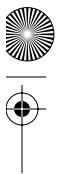
The permissible addressing modes for `codemode` are:

- `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.





Working With the C/C++ Compiler and Linker

Pragma Directives

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 of `#pragma section`” on page 52.

The compiler predefines the sections in Listing 4.8.

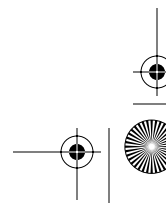
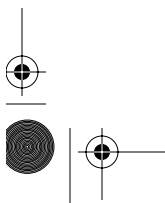
Listing 4.8 Predefined sections

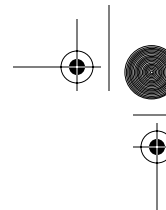
```
#pragma section code_type ".text" data_mode=far_abs code_mode=pc_rel

#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
```





Working With the C/C++ Compiler and Linker

Pragma Directives

```
#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
```

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 of #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.





Working With the C/C++ Compiler and Linker

Linker Issues for Embedded PowerPC

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.

Listing 4.9 shows examples.

Listing 4.9 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 4.7 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.

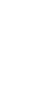
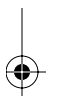
Linker Issues for Embedded PowerPC

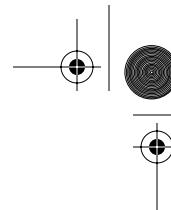
This section explains the background information on the Embedded PowerPC linker and how it works. The topics in this section are:

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

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





Working With the C/C++ Compiler and Linker

Linker Issues for Embedded PowerPC

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

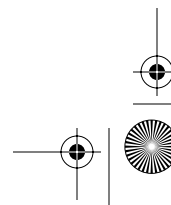
The Embedded PowerPC linker deadstrips unused code and data only from files compiled by the CodeWarrior C/C++ compiler. Assembler relocatable files and C/C++ object files built by other compilers are never deadstripped. Deadstripping is particularly useful for C++ programs. Libraries (archives) built 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.

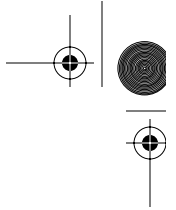
If the **Pool Data** checkbox is checked in the EPPC Processor panel, the pooled data is not stripped. However, all small data and code is still subject to deadstripping.

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 55 for information on how to prevent dead-stripping of unused symbols.

Link Order

The **Link Order** tab of the project window lets you specify the link order. For general information on setting the link order, see the *CodeWarrior IDE User's Guide*.





Working With the C/C++ Compiler and Linker

Linker Issues for Embedded PowerPC

Regardless of the link order you specify, the Embedded PowerPC linker always processes C/C++ files, assembler 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.

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.

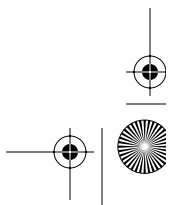
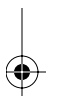
Setting up CodeWarrior IDE to accept LCF files

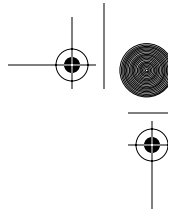
Projects created with the CodeWarrior IDE version 3 or earlier may not recognize the `.lcf` extension. Therefore, you may not be able to add a filename with the `.lcf` extension to the project. You need to create a file mapping to avoid this.

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.





Working With the C/C++ Compiler and Linker

Linker Issues for Embedded PowerPC

Linker Command File Directives

The CodeWarrior PPC EABI linker supports the directives listed below:

- EXCLUDEFILES
- EXTERNAL_SYMBOL
- FORCEACTIVE
- FORCEFILES
- GROUP
- INCLUDEDWARF
- INTERNAL_SYMBOL
- MEMORY
- 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

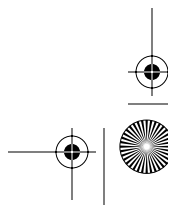
```
EXCLUDEFILES { executablename.extension }
```

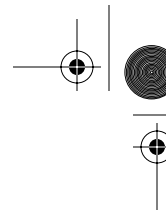
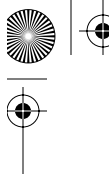
This directive is for partial link projects only. It makes your partial link file smaller. EXCLUDEFILES can be used independently of INCLUDEDWARF. Unlike INCLUDEDWARF, EXCLUDEFILES can take any number of executable files.

Example

```
EXCLUDEFILES { kernel.elf }
```

In this example, `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.





Working With the C/C++ Compiler and Linker

Linker Issues for Embedded PowerPC

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`.

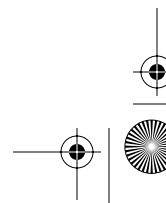
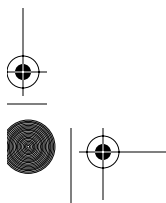
GROUP

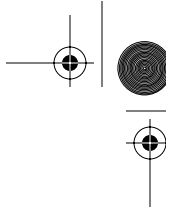
The `GROUP` directive lets you organize the linker command file. This directive has this form:

```
GROUP <address_modifiers> :{ <section_spec> ... }
```

Please see the topic `SECTIONS` for the description of the components.

Listing 4.10 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`.





Working With the C/C++ Compiler and Linker

Linker Issues for Embedded PowerPC

Listing 4.10 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 INCLUDEDWARF directive allows you to debug source level code in the kernel while debugging your application. This directive has the form

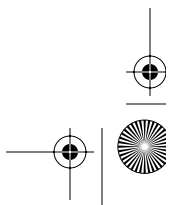
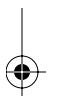
```
INCLUDEDWARF { executablename.extension }
```

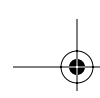
In the example INCLUDEDWARF { kernel.elf }, kernel.elf is added to your project. The linker adds only the .debug and .line sections of kernel.elf to your application. This lets you debug source level code in the kernel while debugging your application.

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++).





Working With the C/C++ Compiler and Linker

Linker Issues for Embedded PowerPC

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 4.11 shows the MEMORY directive added to the example code shown in Listing 4.10. The results of both examples are identical.

Listing 4.11 Example 2

```
MEMORY {

    text : origin = 0x00010000

    data : org = 0x00002000 len = 0x3000
    page0 : o = 0xffff8000, l = 0x8000
}

SECTIONS {

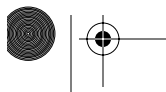
GROUP : {
    .text : {}
    .rodata : {*(.rodata) *(extab) *(extabindex)}

} > text

GROUP : {
    .data : {}
    .bss : {}
    .sdata BIND(0x3500) : {}
    .sbss : {}
    .sdata2 : {}
    .sbss2 : {}

} > data

GROUP : {
```



Working With the C/C++ Compiler and Linker

Linker Issues for Embedded PowerPC

```
.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.

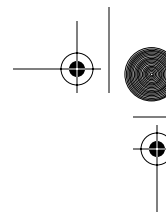
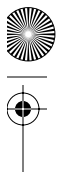
Table 4.7 shows the types of input for *input_type*.

Table 4.7 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.



Working With the C/C++ Compiler and Linker

Linker Issues for Embedded PowerPC

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,

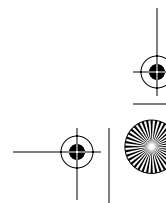
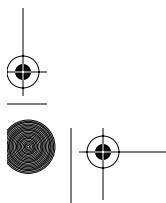
```
.text (TEXT) : { red.o }
```

means that only sections of type `TEXT` in file `red.o` is added.

In all cases, if there is more than 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 4.12 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





Working With the C/C++ Compiler and Linker

Linker Issues for Embedded PowerPC

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 4.12 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 : {}

}
```

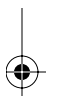
NOTE extab and extabindex must be in separate sections.

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`

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.





Working With the C/C++ Compiler and Linker

Linker Issues for Embedded PowerPC

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 zeroes, 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.

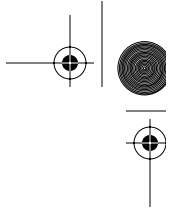
You can use any of the pseudo functions in the `address_modifiers` in a calculation.

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.





Working With the C/C++ Compiler and Linker

Using `__attribute__ ((aligned(?)))`

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
- Structure Definition Examples
- Typedef Declaration Examples
- Structure 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)));
```

The following variable declaration aligns V2 on a 2-byte boundary.

```
int V2[4] __attribute__ ((aligned (2)));
```

Structure Definition Examples

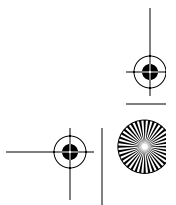
This section shows struct definitions that use `__attribute__ ((aligned(?)))`.

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;
```





Working With the C/C++ Compiler and Linker Using `__attribute__ ((aligned(?)))`

NOTE You must specify a minimum alignment of at least 4 bytes for structures; specifying a lower number for the alignment of a struct 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;
```

Structure 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`.





Working With the C/C++ Compiler and Linker

Small Data PIC/PID Support

Small Data Area PIC/PID Support

The basic requirement for position independent code and data in the small data area is, at runtime, maintaining the link time address relationships between the startup code (`.init`) and the `.sdata` and `.sdata2` segments. For example, if the link time addresses are:

```
.init = 0x00002000
.sdata2 = 0x00003000
.sdata = 0x00004000
```

but `.init` somehow is executed at 0x00002500, then those link time addresses must all increment by 0x00000500 for their runtime addresses.

Any segment that does not maintain the address relationship at runtime is considered external and must be addressed with absolute addresses. Segments that do maintain their link time address relationship at runtime are considered internal and must be addressed with PC-relative and SDA-relative addressing.

- Internal and External Segments and References
- PIC/PID Linker Command File Directives
- Linker-defined Symbols
- Uses for SDA PIC/PID
- Building an SDA PIC/PID Application

Internal and External Segments and References

The linker determines at link time whether code and data segments are external or internal. Internal segments reference their data as far or near offsets of the small data registers `r2` and `r13`. Their code references are normally PC-relative, but if far code references are required, they also use offsets of the small data registers.

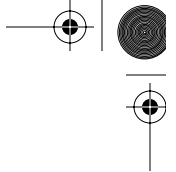
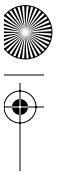
Internal segments can also reference code and data in other internal segments with the same addressing that they would use for their own code and data.

By default, the linker considers all segments in your application to be internal with the exception of segments that are at absolute addresses. Segments with names such as `.abs .xxxxxxx`, where `xxxxxxx` is a hex address, are considered external.

External segments reference their data with absolute addressing and code references within the segment may be either PC-relative or absolute. Any other segment must use absolute references to reference code or data in external segments. External segments must reference an internal segment with small data registers for code and data.

Related to external segments are external symbol references. These are symbols, usually linker-generated, that are determined not to be within any segment in your application.





Working With the C/C++ Compiler and Linker

Small Data Area PIC/PID Support

They are referenced with absolute addressing. All symbols in an external segment are considered to be external symbol references.

PIC/PID Linker Command File Directives

You may use linker command file directives to override PIC/PID related linker default settings.

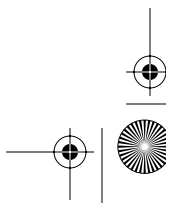
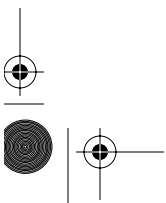
- “MEMORY” on page 59.
- “INTERNAL_SYMBOL” on page 58.
- “EXTERNAL_SYMBOL” on page 57.

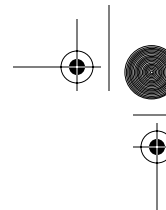
Linker-defined Symbols

The linker-generated start and end symbols that are automatically generated for loadable segments are internal if they are addresses into internal segments, and external if they are for external segments. All other linker defined symbols you create in a LCF are considered external unless you redefine them with `INTERNAL_SYMBOL`. The linker also defines some linker defined symbols for its own use (Table 4.8).

Table 4.8 Linker-defined Symbols

Symbol Name	Value	Description
<code>_stack_addr</code>	top of the stack -	External. Comes from pref panel settings.
<code>_stack_end</code>	bottom of the stack	External. Comes from pref panel settings.
<code>_heap_addr</code>	bottom of the heap	External. Comes from pref panel settings.
<code>_heap_end</code>	top of the heap	External. Comes from pref panel settings.
<code>_SDA_BASE_</code>	<code>.sdata + 0x00008000</code>	Internal per EABI requirement. May not be redefined.
<code>_SDA2_BASE_</code>	<code>.sdata2 + 0x00008000</code>	Internal per EABI requirement. May not be redefined.
<code>_ABS_SDA_BASE_</code>	<code>.sdata + 0x00008000</code>	External version of <code>_SDA_BASE_</code> that can be used as an absolute. May not be redefined.





Working With the C/C++ Compiler and Linker

Small Data Area PIC/PID Support

Table 4.8 Linker-defined Symbols

Symbol Name	Value	Description
<code>_ABS_SDA2_BASE</code> —	<code>.sdata2 + 0x00008000</code>	External version of <code>_SDA2_BASE</code> that can be used as an absolute. May not be redefined.
<code>_nbfunctions</code>	number of functions in program	Deprecated. External. This is a number, not an address. May not be redefined.
<code>SIZEOF_HEADERS</code>	size of the segment headers	External. This is a number, not an address. May not be redefined.

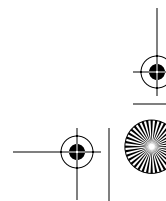
NOTE The symbols `_SDA_BASE` and `_SDA2_BASE` are not accessible until the small data registers are properly initialized before being accessible. The symbols `_ABS_SDA_BASE` and `_ABS_SDA2_BASE` allow you to access those pointers as absolute addresses, as it is difficult to initialize those pointers without accessing them as absolute addresses.

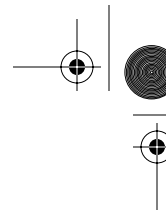
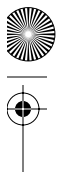
NOTE The stack and heap linker generated symbols are external. It may be more practical in a SDA PIC/PID application to make the heap and stack be contiguous with an internal segment and define them as internal.

Uses for SDA PIC/PID

The PIC/PID runtime can be used for different scenarios:

1. All code and data segments are internal. The simplest case would be for all segments to use the same `MEMORY` directive and to have all of the `.bss` type segments at the end. In such a simple case, the application could be converted to a binary file and linked into another application which could copy it to RAM and jump to its entry point.
2. All of the essential segments are internal and therefore moveable. But, there may be some external segments which are absolute. This situation is probably difficult to test but we can download the entire application to the chip and at least debug it at its link time addresses.
3. There are internal and external segments, but the application is linked as a ROM image (the application does not need to be flashed to ROM, however). It is possible to change the ROM Image Address to be an address into RAM and have the debugger download the image to the RAM address. Alternatively, we could have the ROM image converted to a binary file and linked into another application as in 1, above. The





Working With the C/C++ Compiler and Linker Small Data Area PIC/PID Support

structures used in `__init_data()`, `_rom_copy_info` and `__bss_init_info`, have been modified for SDA PIC/PID to have an extra field which tells the runtime where the segment is internal or external so that the internal segments are copied to position-relative addresses and the external segments copied to absolute addresses.

Building an SDA PIC/PID Application

To build a SDA PIC/PID application, select **SDA PIC/PID** in the **ABI** list box in the **EPPC Target** target preferences panel. The compiler defines a simple variable that we can use to guard PIC/PID source.

```
#if __option(sda_pic_pid) // is true if we have chosen SDA PIC/
PID ABI
```

At link-time, the linker generates a table used for the runtime files `__ppc_eabi_init.cpp` and `__ppc_eabi_init.c`.

If our application contains absolute addressing relocations, we will receive linker warnings telling us that those relocations may cause a problem. To resolve these warnings, either:

- change the **Code Model** listbox in the EPPC Target target preferences panel to be **SDA Based PIC/PID Addressing** for all of our sources and libraries
- check the **Tune Relocations** checkbox in the EPPC Target target preferences panel. This new option is only available for the EABI and SDA PIC/PID ABIs. For EABI, it changes 14-bit branch relocations to 24-bit branch relocations, but only if they can not reach the calling site from the original relocation.

For SDA PIC/PID, this option changes absolute-addressed references of data from code to use a small data register instead of `r0` and changes absolute code-to-code references to use the PC-relative relocations.

Linking Assembly Files

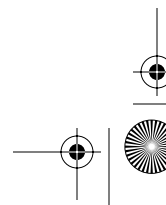
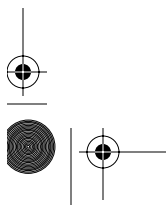
It is always possible to link in an assembly file that does not behave in a standard way. For example, taking the address of a variable with:

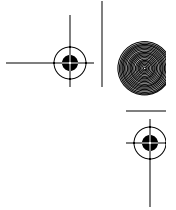
```
addis    rx,r0,object@h
ori      rx,rx,objec@l
```

generally can not be converted by the linker to SDA PIC/PID Addressing and the linker will warn us if it finds an occurrence.

The following will work with Absolute Addressing as well as allow the linker to convert the instructions to SDA PIC/PID Addressing:

```
addis    rx,r0,object@ha
addi     rx,rx,objec@l
```





Working With the C/C++ Compiler and Linker

Small Data Area PIC/PID Support

Another possible problem may arise if we put constant initialized pointers into a read-only section, thereby not letting the runtime convert the addresses.

Modifications to the Section Pragma

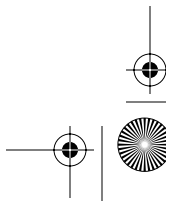
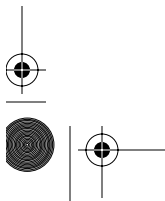
The pragma `#pragma section` has been modified to accept `far_sda_rel` for the `data_mode` and `code_mode` options, even if we are not using Code Model SDA Based PIC/PID Addressing. If we omit these options, the compiler uses the Code Model to determine the appropriate modes.

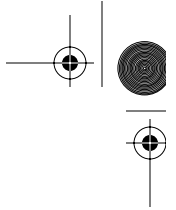
- Absolute Addressing


```
data_mode = far_abs
code_mode = pc_rel
```
- SDA Based PIC/PID Addressing


```
data_mode = far_sda_rel
code_mode = pc_rel
```

See “section” on page 48.





5

Working With the Inline Assembler

This chapter describes support for inline assembly language built into the CodeWarrior C/C++ compiler.

This chapter does not discuss the Embedded PowerPC stand-alone assembler. For information on the stand-alone assembler, see the *Assembler Guide*.

This chapter does not document the entire Embedded PowerPC assembly language instruction set. For documentation of Embedded PowerPC assembly language instruction set, see *PowerPC Microprocessor Family: The Programming Environment for 32-Bit Microprocessors*, published by Motorola, Inc.

You can find this manual and other useful information at this Web address:

http://e-www.motorola.com/webapp/sps/library/tools_lib.jsp

This chapter contains these sections:

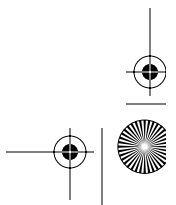
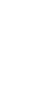
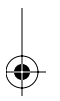
- Working With Assembly Language
- Assembler Directives
- Intrinsic Functions

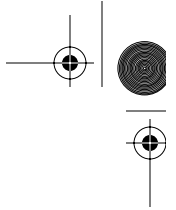
Working With Assembly Language

This section describes how to use the built-in support for assembly language programming included in the CodeWarrior compiler.

This section contains these topics:

- Assembler Syntax for Embedded PowerPC
- Special Embedded PowerPC Instructions
- Support for AltiVec Instructions
- Creating Statement Labels
- Using Comments
- Using the Preprocessor in Embedded PowerPC Assembly
- Using Local Variables and Arguments





Working With the Inline Assembler

Working With Assembly Language

- Creating a Stack Frame in Embedded PowerPC Assembly
- Specifying Operands in Embedded PowerPC Assembly

Assembler Syntax for Embedded PowerPC

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 in the **C/C++ Language** 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).

For instructions specific to the 5xx series of processors, see *MPC500 Family RCPU Reference Manual*, published by Motorola (serial number RCPURM/AD).

For instructions specific to the 8xx series of processors, see *MPC821 Data Book*, published by Motorola (serial number MPC821UM/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.

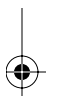
NOTE To enter a few lines of assembly language code within a single function, you can use the support for intrinsics included in the compiler. Intrinsics are an alternative to using `asm` statements within functions.

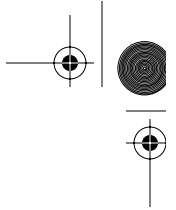
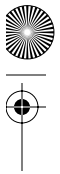
Function-level assembly code for PowerPC uses this syntax:

```
asm {function definition }
```

Assembly language functions must end with the `blr` instruction. For example:

```
asm long MyFunc(void)
{
    ... //assembly language instructions
    blr
}
```





Working With the Inline Assembler

Working With Assembly Language

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.

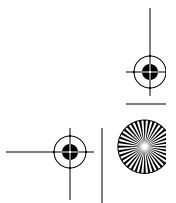
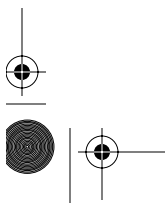
NOTE If you check the **Inlined Assembler is Volatile** checkbox in the EPPC Processor panel, functions that *contain* an `asm` block are only partially optimized, as the optimizer optimizes the function, but skips any `asm` blocks of code. If the **Inlined Assembler is Volatile** checkbox is clear, the optimizer treats `asm` blocks as compiler-generated instructions.

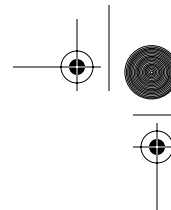
The built-in assembler uses all the standard PowerPC assembler instructions. It accepts some additional directives described in “Assembler Directives” on page 83. If you use the `machine` directive, you can also use instructions that are available only in certain versions of the PowerPC processors.

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:
`add r2,r3,r4`
- Every assembly function must end in an `blr` statement. For example:

```
asm void g(void)
{
    add r2,r3,r4
```





Working With the Inline Assembler

Working With Assembly Language

```
        blr
    }
```

Listing 5.1 shows an example of an assembly language function.

Listing 5.1 Creating an 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.

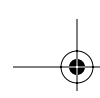
- overflow and record — `addo. r3,r4,r5`

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
```





Support for AltiVec Instructions

The full set of AltiVec assembly instructions is now supported in your inline assembly code. For more information, see *AltiVec Technology Programming Interface Manual* (available from Motorola, Inc.).

NOTE You must select the AltiVec processor from the **Processor** list box in the **EPPC Processor** settings panel, or use the machine `altivec` directive or its equivalent.

You can also use intrinsics in your code.

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 foo(){
int i;
    @x: li r0,1 //Invalid !!!
}
asm void foo(){
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.

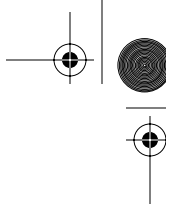
- A label name *can* be the same as an assembly language statement mnemonic.

For example, this statement is valid:

```
add: add r3, r4, r5
```

This is an example of a complete inline assembly language function:





Working With the Inline Assembler

Working With Assembly Language

```
asm void red(void)
{
    x1:  add r3,r4,r5
    @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
}
```

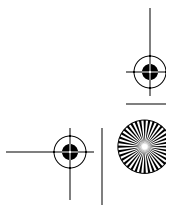
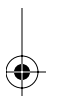
Using Local Variables and Arguments

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





Working With the Inline Assembler

Working With Assembly Language

- 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 in Embedded PowerPC Assembly

You need to create a stack frame for a function if the function:

- calls other functions.
- declares non-register arguments or local variables.

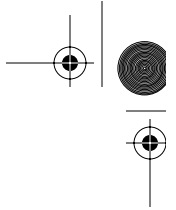
To create a stack frame, use the `fralloc` directive at the beginning of your function and the `frfree` directive just before the `blr` statement. The directive `fralloc` automatically allocates (while `ffree` automatically de-allocates) memory for local variables, and saves and restores the register 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 – fp8` are used.





Working With the Inline Assembler

Working With Assembly Language

As an example, if you pass 12 long integer to your assembly function, this would consume 16 bytes of the parameter area. Registers `r3` — `r10` will hold eight integers, leaving four byte integers in the parameter area.

Specifying Operands in Embedded PowerPC Assembly

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.

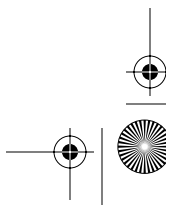
Listing 5.2 shows a block of code for specifying operands.

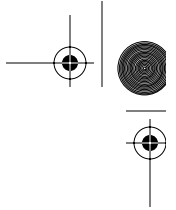
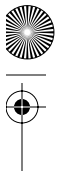
Listing 5.2 Using register variables and memory variables

```
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 5.2:

- 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





Working With the Inline Assembler Working With Assembly Language

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,

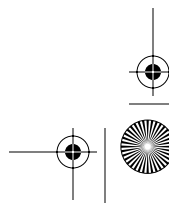
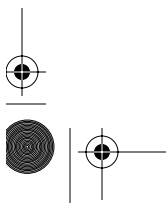
- `b @3` — correct syntax for branching to a local label
- `b red` — correct syntax for branching to external function `red`
- `bl @3` — correct syntax for calling a local label
- `bl red` — correct syntax for calling external function `red`
- `bne red` — incorrect syntax; short branch outside function `red`

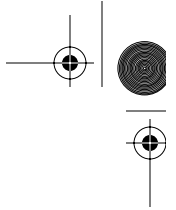
NOTE You cannot use local labels that have already been declared in other functions.

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)
```





Working With the Inline Assembler

Working With Assembly Language

```
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

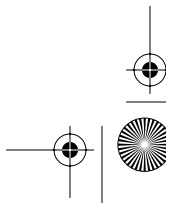
For an immediate operand, you can use an integer or enum constant, sizeof expression, and any constant expression using any of the C dyadic and monadic arithmetic operators.

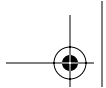
These expressions follow the same precedence and associativity rules as normal C expressions. The inline assembler carries out all arithmetic with 32-bit signed integers.

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.





Working With the Inline Assembler

Assembler Directives

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 because you can write:

```
lis x,var@ha
lwz v,var@l(x)
```

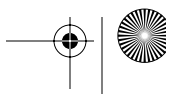
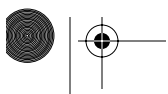
You cannot do this with `@h` because it requires that you use the `ori` instruction.

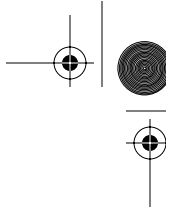
Assembler Directives

This section describes some special assembler directives that the Embedded PowerPC built-in assembler accepts.

entry

```
entry [ extern | static ] name
```





Working With the Inline Assembler

Assembler Directives

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 5.3 shows how to use the `entry` directive.

Listing 5.3 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]`

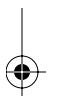
Embedded PowerPC assembler directive that creates a stack frame for a function and reserves registers for your local register variables. You need to create a stack frame for a function if the function:

- 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`





Working With the Inline Assembler

Assembler Directives

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 code is for. The value of `number` must be one of these:

Table 5.1 Machine Assembler Directive Values

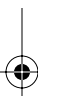
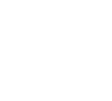
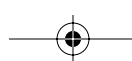
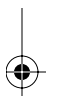
401	403	505	509
555	56x	601	602
603	604	740	750
801	821	823	850
852	860	885	7400
8240	8247	8248	8260
8271	8272	8280	PPC604e
PPC403GA	PPC403GB	PPC403GC	PPC403GCX
all	generic	altivec	

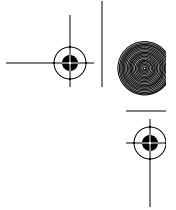
If you use `generic`, the CodeWarrior IDE supports the core instructions for the 603, 604, 740, and 750 processors. In addition, the CodeWarrior IDE supports all optional instructions.

If you use `all`, the CodeWarrior IDE supports all core and optional instructions for all Embedded PowerPC processors.

If you do not use the `machine` directive, the compiler uses the setting you selected from the **Processor** list box in the **EPPC Processor** settings panel. For example, `machine altivec`.

This enables the assembler AltiVec instructions. The pragma `#pragma altivec_codegen` on statement has the same effect.





Working With the Inline Assembler

Intrinsic Functions

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 of `nofralloc`, see the file `__start.c` in the directory:

`CWInstall\PowerPC_EABI_Support\Runtime\Src\`

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.

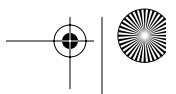
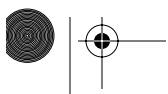
Intrinsic functions are a mechanism you can use to get assembly language into your source code.

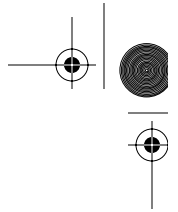
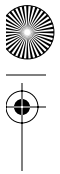
There is an intrinsic function for several common processor opcodes (instructions). Rather than using inline assembly syntax and specifying the opcode in an `asm` block, you call the intrinsic function that matches the opcode.

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 if the **Hardware** option button is cleared 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.

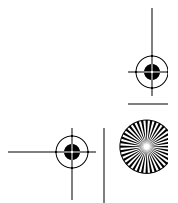
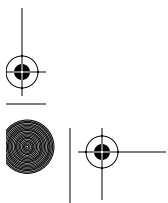
- `int __lhbrx(void *, int)` — Load halfword byte; reverse indexed
- `int __lwbrx(void *, int)` — Load word byte; reverse indexed
- `void __sthbrx(unsigned short, void *, int)` — Store halfword byte; reverse indexed
- `void __stwbrx(unsigned int, void *, int)` — Store word byte; reverse indexed

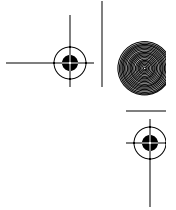
Setting the Floating-Point Environment

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 the **None** option button in the **EPPC Processor** settings panel.

```
float __setflm(float);
```





Working With the Inline Assembler

Intrinsic Functions

This example shows how to set and restore the FPSCR:

```
double old_fpscr;  
/* Clear all flag/exception/mode bits  
   and save original settings */  
oldfpscr = __setflm(0.0);  
/* Perform some floating-point operations */  
__setflm(old_fpscr); /* Restores the FPSCR */
```

Manipulating the Contents of 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);
```

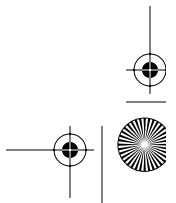
You can use inline assembly for a complete assembly language function, as well as individual assembly language statements.

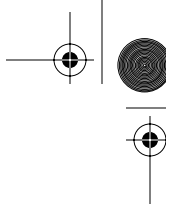
Data Cache Manipulation

The intrinsics shown in Table 5.2 map directly to PowerPC assembly instructions.

Table 5.2 Data Cache Intrinsics

Intrinsic Prototype	PowerPC Instruction
<code>void __dcbf(void *, int);</code>	dcbf
<code>void __dcbt(void *, int);</code>	dcbt





Working With the Inline Assembler
Intrinsic Functions

Table 5.2 Data Cache Intrinsics (*continued*)

<code>void __dcbst(void *, int);</code>	<code>dcbst</code>
<code>void __dcbtst(void *, int);</code>	<code>dcbtst</code>
<code>void __dcbz(void *, int);</code>	<code>dcbz</code>

Math Functions

The intrinsics shown in Table 5.3 map directly to PowerPC assembly instructions.

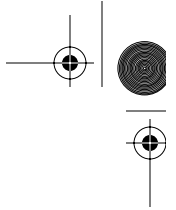
Table 5.3 Math Intrinsics

Intrinsic Prototype	PowerPC Instruction
<code>int __mulhw(int, int);</code>	<code>mulhw</code>
<code>uint __mulhwu(uint, uint);</code>	<code>mulhwu</code>
<code>double __fmadd(double, double, double);</code>	<code>fmadd</code>
<code>double __fmsub(double, double, double);</code>	<code>fmsub</code>
<code>double __fnmadd(double, double, double);</code>	<code>fnmadd</code>
<code>double __fnmsub(double, double, double);</code>	<code>fnmsub</code>
<code>float __fmadds(float, float, float);</code>	<code>fmadds</code>
<code>float __fmsubs(float, float, float);</code>	<code>fmsubs</code>
<code>float __fnmadds(float, float, float);</code>	<code>fnmadds</code>
<code>float __fnmsubs(float, float, float);</code>	<code>fnmsubs</code>
<code>double __mffs(void);</code>	<code>mffs</code>
<code>float __fabsf(float);</code>	<code>fabsf</code>
<code>float __fnabsf(float);</code>	<code>fnabsf</code>

Buffer Manipulation

Some intrinsics allow control over areas of memory, so you can manipulate memory blocks.





Working With the Inline Assembler

Intrinsic Functions

```
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.
```

AltiVec Intrinsics Support

You can use all the available AltiVec intrinsics in your code. You will find a list of these in the relevant Motorola documentation at this URL on the world-wide web:

<http://www.freescale.com/altivec/>

These are the generic and specific AltiVec intrinsics:

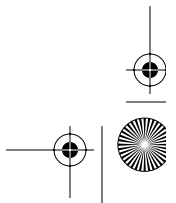
Table 5.4 Generic and AltiVec Intrinsics

vec_abs	vec_abss	vec_and
vec_adds	vec_ceil	vec_cmpb
vec_cmpgt	vec_cmple	vec_sums
vec_xor	vec_trunc	

These are the AltiVec predicates:

Table 5.5 AltiVec Predicates

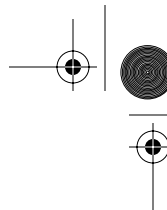
vec_all_eq	vec_all_ge
vec_all_le	vec_all_lt
vec_all_nga	vec_all_ngt
vec_all_numeric	vec_any_eq
vec_any_le	vec_any_lt
vec_any_nge	vec_any_ngt
vec_any_numerics	vec_any_out





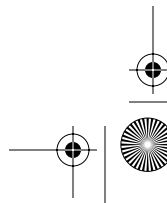
Working With the Inline Assembler
Intrinsic Functions





Working With the Inline Assembler

Intrinsic Functions





6

Working With the Debugger

This chapter explains how to use the CodeWarrior tools to debug Embedded PowerPC (EPPC) programs. The chapter covers those aspects of debugging that are specific to the Embedded PowerPC platform. See the *CodeWarrior IDE User's Guide* for debugger information that applies to all CodeWarrior products.

This chapter contains these sections:

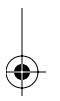
- Using Remote Connections
- Special Debugger Features
- Using MetroTRK
- Debugging External ELF Files
- Debugging Multiple ELF Files
- Accessing Translation Look-aside Buffers
- Using the Command-Line Debugger

Using Remote Connections

Remote connections are settings that describe how the CodeWarrior IDE should connect to and control program execution on target boards or systems, such as the debugger protocol, connection type, and connection parameters the IDE should use when it connects to the target system. This section shows you how to access remote connections in the CodeWarrior IDE, and describes the various debugger protocols and connection types the IDE supports.

NOTE We have included several types of remote connections in the default CodeWarrior installation. You can modify these default remote connections to suit your particular needs.

TIP When you import a Makefile into the CodeWarrior IDE to create a CodeWarrior project, the IDE asks you to specify the type of debugger interface (remote connection) you want to use. To debug the generated CodeWarrior project, you must properly configure the remote connection you selected when you created the project.





Working With the Debugger

Using Remote Connections

Accessing Remote Connections

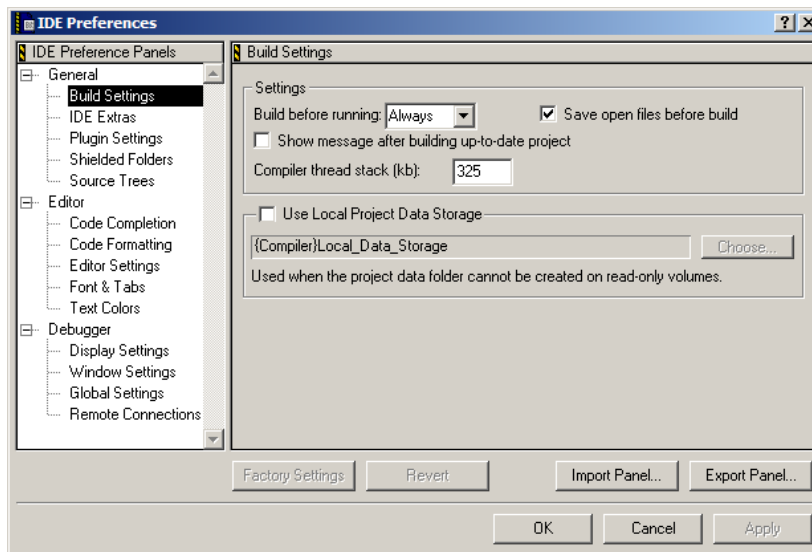
You access remote connections in the CodeWarrior **IDE Preferences** window. Remote connections listed in the preferences window are available for use in all CodeWarrior projects and build targets.

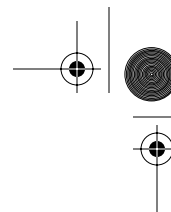
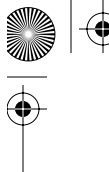
To access remote connections:

1. From the CodeWarrior menu bar, select **Edit > Preferences**.

The **IDE Preferences** window (Figure 6.1) appears.

Figure 6.1 IDE Preferences Window



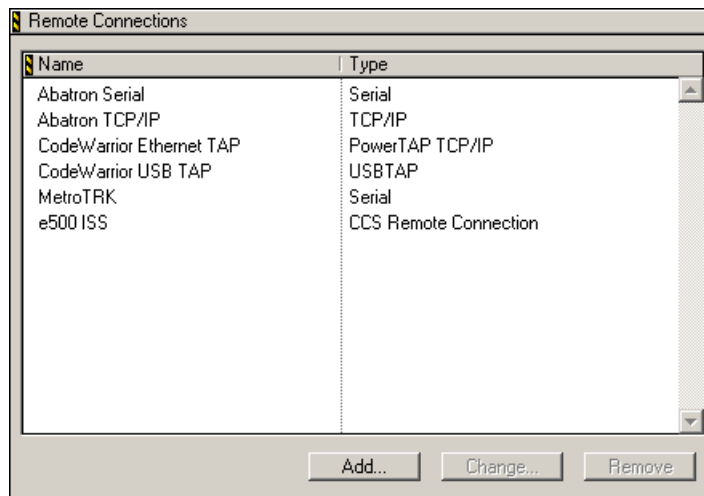


Working With the Debugger

Using Remote Connections

2. In the **IDE Preference Panels** list, select **Remote Connections**.
The **Remote Connections** preference panel (Figure 6.2) appears.

Figure 6.2 Remote Connections Preference Panel



NOTE The specific remote connections that appear in the Remote Connections list differ between CodeWarrior products and hosts.

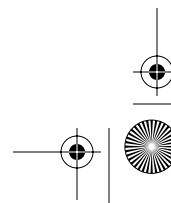
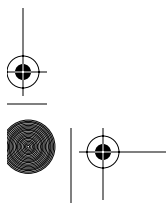
The **Remote Connections** preference panel lists all of the remote connections of which the CodeWarrior IDE is aware. You use this preference panel to add your own remote connections, remove remote connections, and configure existing remote connections to suit your needs.

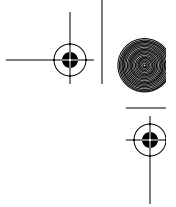
To add a new remote connection, click **Add**.

To configure an existing remote connection, select it and click **Change**.

To remove an existing remote connection, select it and click **Remove**.

TIP To specify a remote connection for a particular build target in a CodeWarrior project, you select the remote connection from the **Connection** list box in the **Remote Debugging** target settings panel. For an overview of the **Remote Debugging** settings panel, see the *CodeWarrior IDE User's Guide*.





Working With the Debugger Using Remote Connections

Understanding Remote Connections

Every remote connection specifies a debugger protocol and a connection type.

A *debugger protocol* is the protocol the IDE uses to debug the target system. This setting generally relates specifically to the particular device you use to physically connect to the target system.

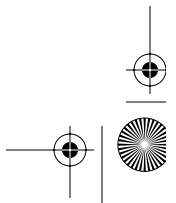
A *connection type* is the type of connection (such as Parallel, Serial, TCP/IP, and so on) the CodeWarrior IDE uses to communicate with and control the target system.

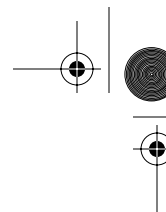
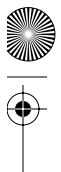
Table 6.1 describes each of the supported debugger protocols.

Table 6.1 Debugger Protocols

Debugger Protocol	Description
Logic Analyzer	Select to use the logic analyzer hardware tool with a target system.
EPPC - Abatron	Select to use serial or TCP/IP connections and an Abatron device with and debug a target system.
EPPC - MetroTRK	Select to use a serial connection with MetroTRK to debug a target system.
OSE Run Mode - EPPC	Select to use a TCP/IP connection with a target OSE system.
CCS EPPC Protocol Plugin	Select to use a CCS, Parallel, CodeWarrior Ethernet TAP, CodeWarrior USB TAP, or WireTAP Parallel connection with the target system.

Each of these protocols supports one or more types of connections (CCS, Parallel, TCP/IP, and so on). “Editing Remote Connections” describes each supported connection type and how to configure them.





Working With the Debugger Using Remote Connections

Editing Remote Connections

Based on the specified debugger protocol and connection type, the IDE makes different settings available to you. For example, if you specify a Serial connection type, the IDE presents settings for baud rate, stop bits, flow control, and so on. Table 6.2 describes the supported connection types for each debugger protocol.

Table 6.2 Supported Connection Types

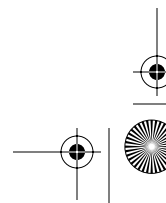
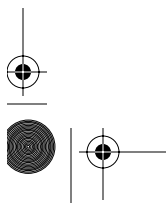
Debugger Protocol	Supported Connection Types
Logic Analyzer	Logic Analyzer Config Panel
EPPC - Abatron	Serial, TCP/IP
EPPC - MetroTRK	Serial
OSE Run Mode - EPPC	TCP/IP
CCS EPPC Protocol Plugin	CCS Remote Connection, PowerTAP TCP/IP, USBTAP

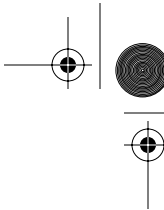
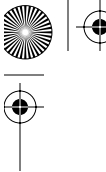
To configure a remote connection to correspond to your particular setup, you must edit the connection settings. You access the settings with the **Edit Connection** dialog box. You can view this dialog box in one of these ways:

- In the **Remote Connections** IDE preference panel, select a connection from the list, and click **Edit**. The **Edit Connection** dialog box appears.
- In the **Remote Connections** IDE preference panel, click **Add** to create a new remote connection. The **New Connection** dialog box appears.
- In the **Remote Debugging** target settings panel, select a connection from the **Connection** list box, then click the **Edit Connection** button. The **Edit Connection** dialog box appears.

This section describes the settings for each connection type:

- Logic Analyzer Config Panel
- Serial
- TCP/IP
- CCS Remote Connection
- PowerTAP TCP/IP
- USBTAP





Working With the Debugger
Using Remote Connections

Logic Analyzer Config Panel

Use this connection type to configure how the IDE connects the logic analyzer to the target system. This connection type is available only when the **Logic Analyzer** debugger protocol is selected.

Figure 6.3 shows the settings that are available to you when you select **Logic Analyzer Config Panel** from the **Connection Type** list box in the **Edit Connection** dialog box.

Figure 6.3 Logic Analyzer Config Panel Connection Settings

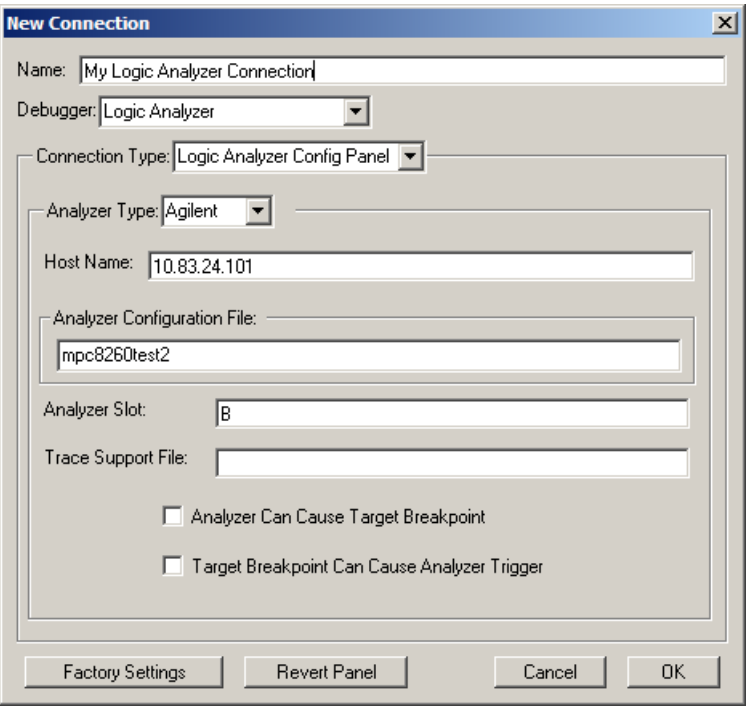
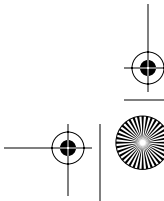
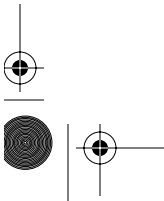
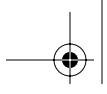


Table 6.3 describes the options in this dialog box.

Table 6.3 Logic Analyzer Config Panel Options

Option	Description
Name	Enter the name you want to use to refer to this remote connection within the CodeWarrior IDE.
Debugger	Select Logic Analyzer .

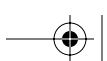




Working With the Debugger
Using Remote Connections

Table 6.3 Logic Analyzer Config Panel Options (*continued*)

Option	Description
Connection Type	Select Logic Analyzer Config Panel .
Analyzer Type	Select the type of logic analyzer you want to use.
Host Name	Enter the Internet Protocol (IP) address or fully-qualified host name of the logic analyzer device.
Analyzer Configuration File	Enter the name of the configuration file for the logic analyzer.
Analyzer Slot	Enter the name of the slot that identifies the location of the analyzer.
Trace Support File	Enter the name of the file that the logic analyzer requires to support trace data collection.
Analyzer Can Cause Target Breakpoint	Check to allow the logic analyzer to halt hardware execution. Clear to prevent the logic analyzer from halting hardware execution.
Target Breakpoint Can Cause Analyzer Trigger	Check to allow a hardware breakpoint to trigger the logic analyzer. Clear to prevent a hardware breakpoint from triggering the logic analyzer.





Working With the Debugger
Using Remote Connections

Serial

Use this connection type to configure how the IDE uses the serial interface of the host computer to connect with the target system. This connection type is available when the **EPPC - Abatron**, or **EPPC - MetroTRK** debugger protocol is selected.

Figure 6.4 shows the settings that are available to you when you select **Serial** from the **Connection Type** list box in the **Edit Connection** dialog box.

Figure 6.4 Serial Connection Settings

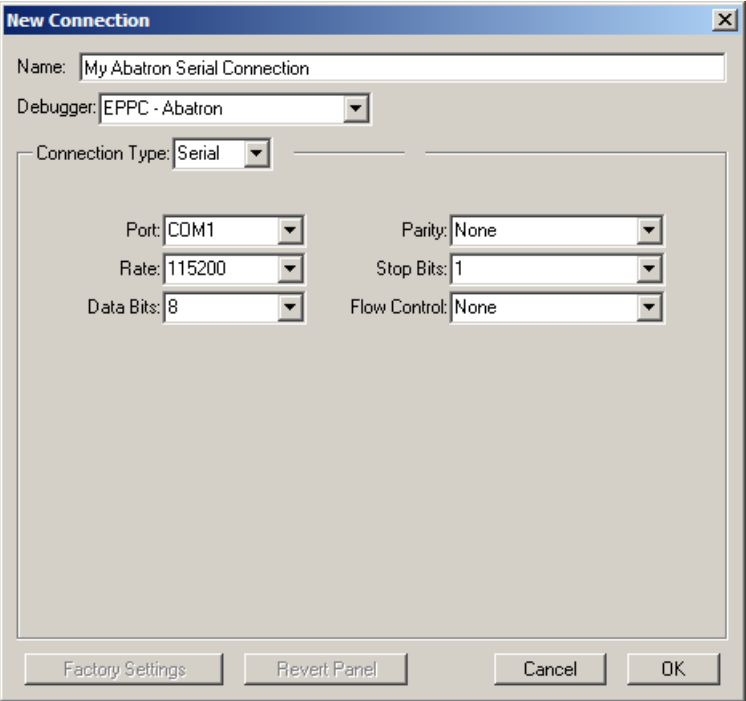
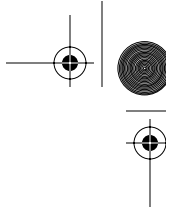


Table 6.4 describes the options in this dialog box.

Table 6.4 Serial Options

Option	Description
Name	Enter the name you want to use to refer to this remote connection within the CodeWarrior IDE.
Debugger	Select EPPC - Abatron , or EPPC - MetroTRK .



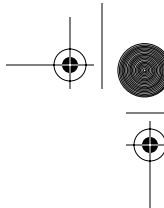


Working With the Debugger

Using Remote Connections

Table 6.4 Serial Options (continued)

Option	Description
Connection Type	Select Serial .
Port	Select the name of the serial port to which the target system is connected on the host computer.
Rate	For Abatron device connections, select the communication rate that the device supports. For MetroTRK connections, select the MetroTRK communication rate on the target system.
Data Bits	Select the number of data bits the IDE should use when it communicates with the target system.
Parity	Select the parity the IDE should use when it communicates with the target system.
Stop Bits	Select the stop bits the IDE should use when it communicates with the target system.
Flow Control	Select the flow control the IDE should use when it communicates with the target system.



Working With the Debugger
Using Remote Connections

TCP/IP

Use this connection type to configure how the IDE uses the TCP/IP protocol to connect with the target system. This connection type is available when the **EPPC - Abatron**, or **OSE Run Mode - EPPC** debugger protocol is selected.

Figure 6.5 shows the settings that are available to you when you select **TCP/IP** from the **Connection Type** list box in the **Edit Connection** dialog box.

Figure 6.5 TCP/IP Connection Settings

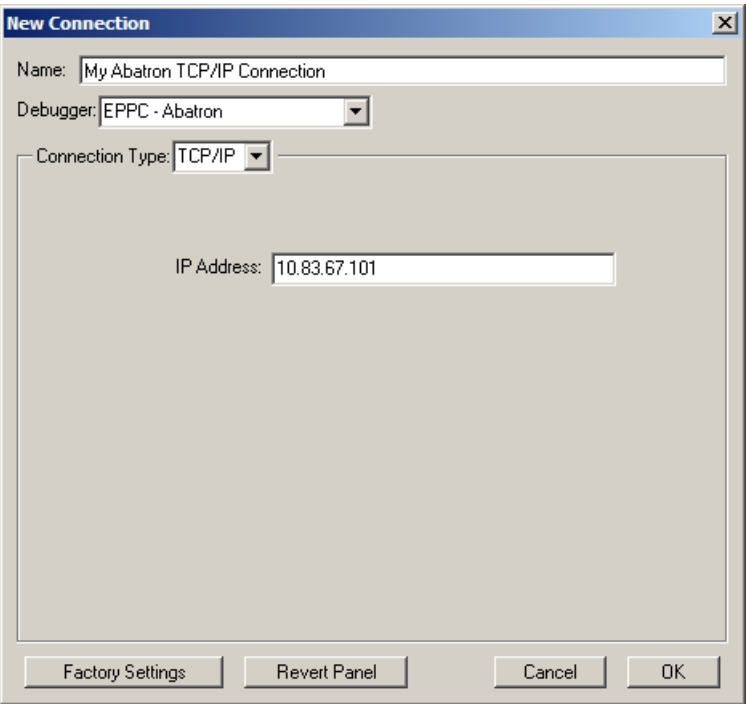
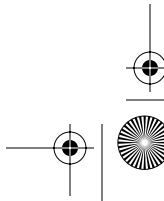
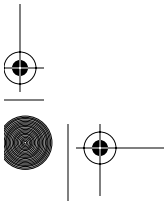
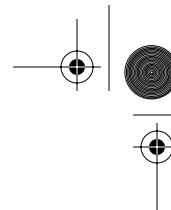
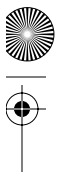


Table 6.5 describes the options in this dialog box.

Table 6.5 TCP/IP Options

Option	Description
Name	Enter the name you want to use to refer to this remote connection within the CodeWarrior IDE.
Debugger	Select EPPC - Abatron , or OSE Run Mode - EPPC .





Working With the Debugger Using Remote Connections

Table 6.5 TCP/IP Options (*continued*)

Option	Description
Connection Type	Select TCP/IP .
IP Address	Enter the Internet Protocol (IP) address assigned to the target system.

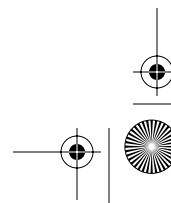
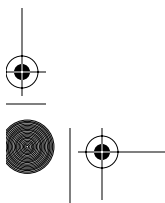
CCS Remote Connection

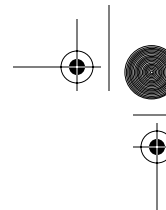
Use this connection type to configure how the IDE uses the Command Converter Server (CCS) protocol to connect with the target system. This connection type is available only when the **CCS EPPC Protocol Plugin** debugger protocol is selected.

Figure 6.6 shows the settings that are available to you when you select **CCS Remote Connection** from the **Connection Type** list box in the **Edit Connection** dialog box.

Figure 6.6 CCS Remote Connection Settings

The screenshot shows the 'e500 ISS' dialog box for editing a connection. The 'Name' field is 'e500 ISS'. The 'Debugger' is set to 'CCS EPPC Protocol Plugin'. The 'Connection Type' is 'CCS Remote Connection'. The 'Use Remote CCS' checkbox is unchecked. The 'Server IP Address' is 'hostname' and the 'Port #' is '40969'. The 'Specify CCS Executable' checkbox is checked, with the path '{Compiler}\ccs\bin\ccsim2.exe' and a 'Choose...' button. The 'Multi-Core Debugging' checkbox is unchecked, with a 'JTAG Configuration File' field and a 'Choose...' button. The 'CCS Timeout' is set to '10' seconds. At the bottom are buttons for 'Factory Settings', 'Revert Panel', 'Cancel', and 'OK'.





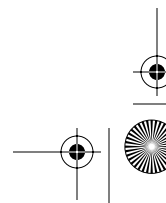
Working With the Debugger

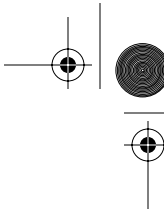
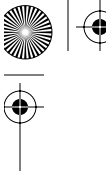
Using Remote Connections

Table 6.6 describes the options in this dialog box.

Table 6.6 CCS Remote Connection Options

Option	Description
Name	Enter the name you want to use to refer to this remote connection within the CodeWarrior IDE.
Debugger	Select CCS EPPC Protocol Plugin .
Connection Type	Select CCS Remote Connection .
Use Remote CCS	Check to debug code on a target system when the system already has CCS running and connected.
Server IP Address	Enter the Internet Protocol (IP) address assigned to the target system.
Port #	Enter the port number on the target system to which the IDE should connect for CCS operations. The default port number for CCS hardware connections is 41475. Enter 41476 for the CCS Simulator.
Specify CCS Executable	Check to use another CCS executable file rather than the default CCS executable file: <code>CWInstall\ccs\bin\ccs.exe</code>
Multi-Core Debugging	Check to debug code on a target system with multiple cores where you need to specify the JTAG chain for debugging. Click Choose to specify the JTAG initialization file. A JTAG initialization file contains the names and order of the boards / cores you want to debug.
CCS Timeout	Enter the duration (in seconds) after which the CCS should attempt to reconnect to the target system if a connection attempt fails.





PowerTAP TCP/IP

Use this connection type to configure how the IDE uses a CodeWarrior Ethernet TAP device to connect with the target system. This connection type is available only when the **CCS EPPC Protocol Plugin** debugger protocol is selected.

Figure 6.7 shows the settings that are available to you when you select **PowerTAP TCP/IP** from the **Connection Type** list box in the **Edit Connection** dialog box.

Figure 6.7 PowerTAP TCP/IP Connection Settings

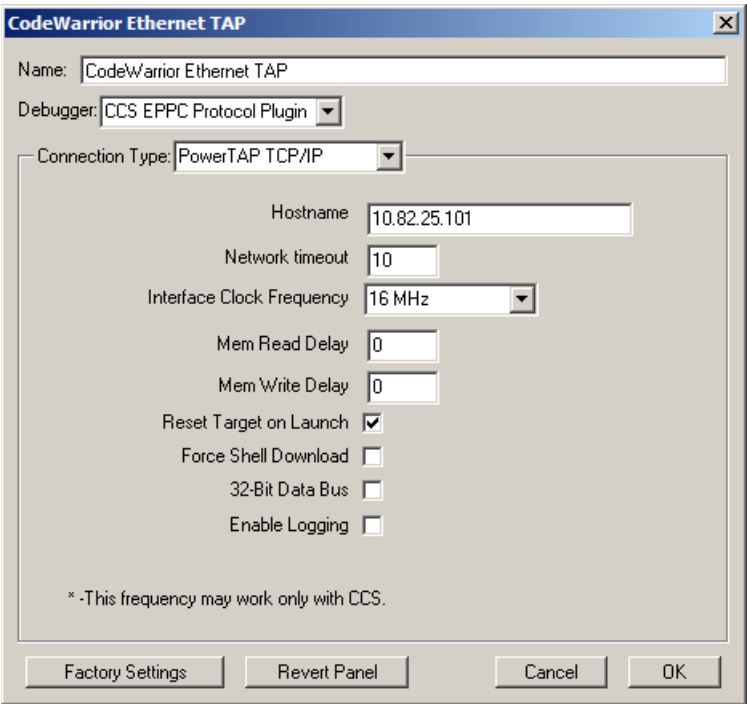
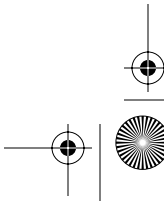
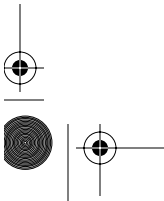


Table 6.7 describes the options in this dialog box.

Table 6.7 PowerTAP TCP/IP Options

Option	Description
Name	Enter the name you want to use to refer to this remote connection within the CodeWarrior IDE.
Debugger	Select CCS EPPC Protocol Plugin .

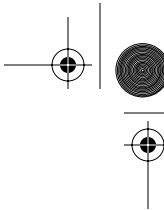
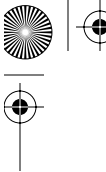


Working With the Debugger

Using Remote Connections

Table 6.7 PowerTAP TCP/IP Options (*continued*)

Option	Description
Connection Type	Select PowerTAP TCP/IP .
Hostname	Enter the host name or IP address that you assigned to the Ethernet TAP device during the emulator setup.
Network Timeout	Enter the maximum number of seconds the debugger should wait for a response from the Ethernet TAP device. By default, the debugger waits up to 10 seconds for responses.
Interface Clock Frequency	Select the clock frequency for the Ethernet TAP device. We recommended you set this to 16 MHz .
Mem Read Delay	Enter the number of additional processor cycles (in the range: 0 through 65024) the debugger should insert as a delay for completion of memory read operations. By default, the debugger delays for 350 cycles.
Mem Write Delay	Enter the number of additional processor cycles (in the range: 0 through 65024) the debugger should insert as a delay for completion of memory write operations. By default, the debugger delays for 350 cycles.
Reset Target on Launch	Check to have the debugger send a reset signal to the target system when you start debugging. Clear to prevent the debugger from resetting the target device when you start debugging.
Force Shell Download	Check to have the debugger start the Ethernet TAP shell when you start debugging. Clear to prevent the debugger from starting the Ethernet TAP shell when you start debugging.
32-Bit Data Bus	Check to use the 32-bit data bus.
Enable Logging	Check to have the IDE display a log of all debugger transactions during the debug session. If this checkbox is checked, a protocol logging window appears when you connect the debugger to the target system. Note: If you set the AMCTAP_LOG_FILE environment variable, the IDE directs log messages to the specified file.



USBTAP

Use this connection type to configure how the IDE uses CodeWarrior USB TAP device to connect with the target system. This connection type is available only when the **CCS EPPC Protocol Plugin** debugger protocol is selected.

Figure 6.7 shows the settings that are available to you when you select **USBTAP** from the **Connection Type** list box in the **Edit Connection** dialog box.

Figure 6.8 USBTAP Connection Settings

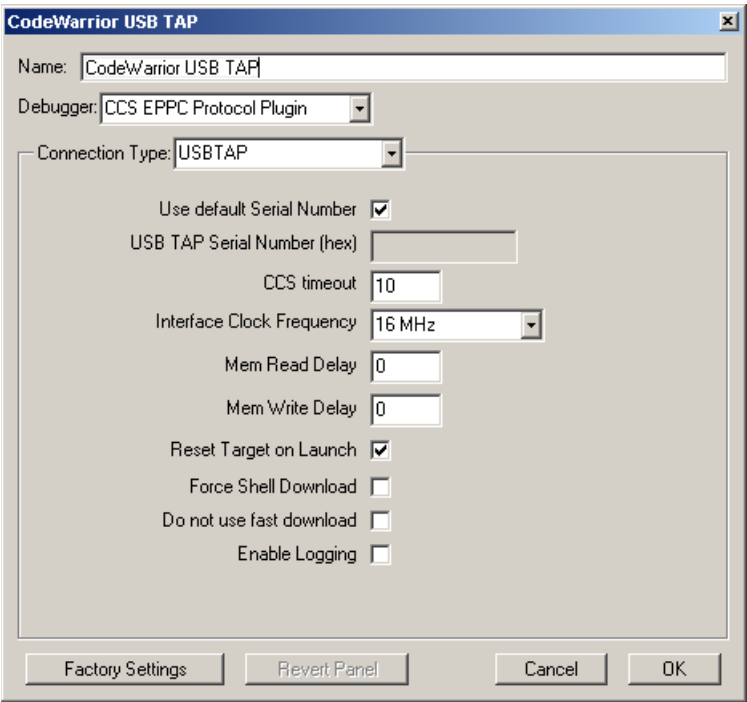
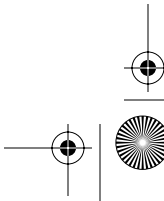
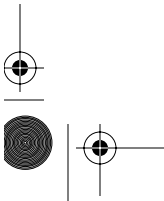
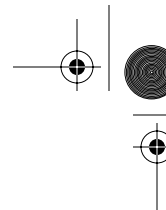


Table 6.8 describes the options in this dialog box.

Table 6.8 UBTAP Options

Option	Description
Name	Enter the name you want to use to refer to this remote connection within the CodeWarrior IDE.
Debugger	Select CCS EPPC Protocol Plugin .





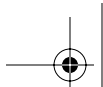
Working With the Debugger

Using Remote Connections

Table 6.8 UBTAP Options

Option	Description
Connection Type	Select USBTAP .
Use default serial number	Check if you only have one USB TAP device connected to the host computer. Clear if you have more than one USB TAP device connected to the host computer. When this checkbox is checked, the USB TAP Serial Number text box is available.
USB TAP Serial Number	If you have more than one USB TAP connected to the host computer, enter the serial number of the USB TAP you want to use for debugging. Note: The USB TAP serial number is located on a label on the bottom of the device.
CCS Timeout	Enter the maximum number of seconds the debugger should wait for a response from CCS. By default, the debugger waits up to 10 seconds for responses.
Interface Clock Frequency	Select the clock frequency for the Ethernet TAP device. We recommended you set this to 4 MHz .
Mem Read Delay	Enter the number of additional processor cycles (in the range: 0 through 65024) the debugger should insert as a delay for completion of memory read operations. By default, the debugger delays for 350 cycles.
Mem Write Delay	Enter the number of additional processor cycles (in the range: 0 through 65024) the debugger should insert as a delay for completion of memory write operations. By default, the debugger does not delay.
Reset Target on Launch	Check to have the debugger send a reset signal to the target system when you start debugging. Clear to prevent the debugger from resetting the target device when you start debugging.
Force Shell Download	Check to have the debugger start the Ethernet TAP shell when you start debugging. Clear to prevent the debugger from starting the Ethernet TAP shell when you start debugging.



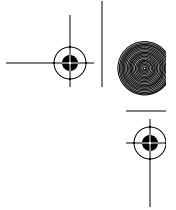


Working With the Debugger
Using Remote Connections

Table 6.8 UBTAPOptions

Option	Description
Do not use fast download	<p>Check to have the debugger use a standard (slow) procedure to write to memory on the target system.</p> <p>Clear to have the debugger use an optimized (fast) download procedure to write to memory on the target system.</p>
Enable Logging	<p>Check to have the IDE display a log of all debugger transactions during the debug session. If this checkbox is checked, a protocol logging window appears when you connect the debugger to the target system.</p> <p>Note: If you set the AMCTAP_LOG_FILE environment variable, the IDE directs log messages to the specified file.</p>





Working With the Debugger

Special Debugger Features

Special Debugger Features

The sections listed below explain how to use the debugger's Embedded PowerPC-specific features.

- Attaching to Processes
- Displaying Registers
- Register Details
- EPPC-Specific Debugger Features


Attaching to Processes

You can use the Attach function of the CodeWarrior debugger to attach the debugger to running processes on a target PQ3 board. The debugger can control execution of any process to which you attach it.

If the target board is running an operating system, or is running multiple processes, you can use the CodeWarrior **System Browser** window to view and attach to processes running on the board. To view this window:

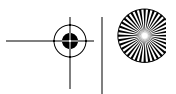
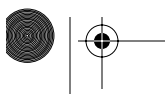
1. Open a CodeWarrior PowerQUICC III project.
2. Ensure that a linker is selected in the Target Settings panel in the **Target Settings** window (see "Target Settings" on page 137 for more information).
3. Ensure that a valid remote connection is selected in the **Remote Debugging** target settings panel.
4. Build the CodeWarrior project to generate a valid executable file.
5. Select **View > System > Connection** from the CodeWarrior menu bar (where **Connection** is the name of the selected remote connection).

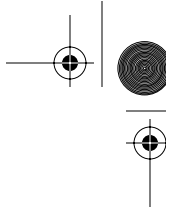
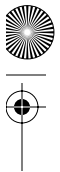
The **System Browser** window appears, displaying a list of the processes running on the target board.

6. In the System Browser window, select the process to which you want to attach, then click the Attach To Process button ().

NOTE For more details about the System Browser window, refer to the *CodeWarrior IDE User's Guide*.

If the target board is not running an operating system, and is only running a single process, you can use the **Debug > Attach To Process** CodeWarrior menu to attach directly to the running executable process on the board.





Working With the Debugger

Special Debugger Features

NOTE If you do not have a CodeWarrior project open when you select **Debug > Attach To Process**, the IDE asks you to specify which debugger and remote connection you want to use.

The Attach function differs from the Connect function in these ways:

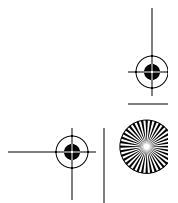
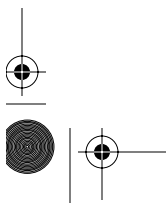
- The Connect function runs the hardware initialization file specified in the EPPC Debugger Settings panel to set up the board before connecting to it.
- The Attach function assumes that code is already running on the board, and therefore does not run a hardware initialization file. The state of the running program is undisturbed.
- The Connect function does not load any symbolic information for the current build target's generated executable. You therefore do not have access to source-level debugging and variable display.
- When you attach to a process, however, the debugger loads symbolic information for the current build target's generated executable. The result is that you have the same source-level debugging facilities you would have if you were to started a normal debug session (the ability to view source code and variables, and so on).

NOTE The debugger assumes that the current build target's generated executable matches the code currently running on the target.

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**.

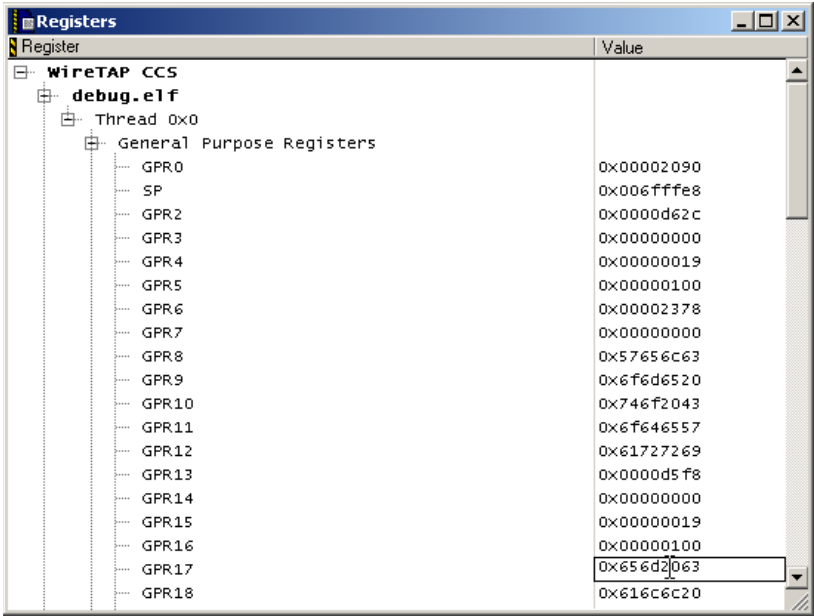
The **Registers** window displays categories of registers in a tree format. To display the contents of a particular category of registers, expand the tree element of the register category of interest. Figure 6.9 shows the **Registers** window with the General Purpose Registers tree element expanded.



Working With the Debugger

Special Debugger Features

Figure 6.9 Registers Window



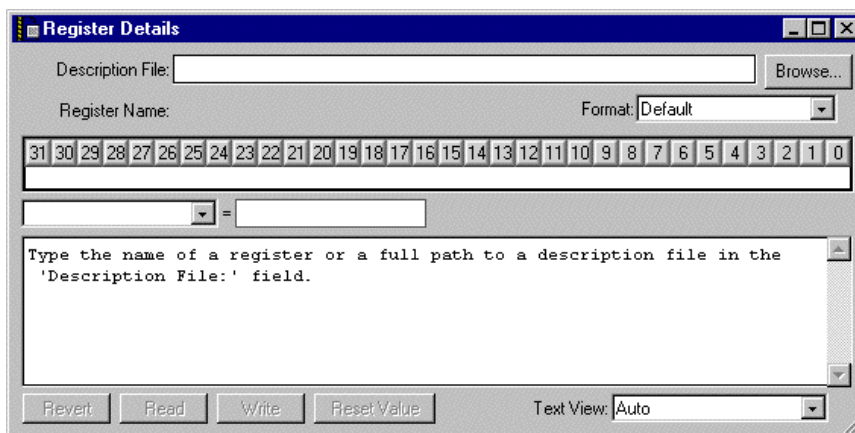
Register Details

You can use the **Register Details** dialog box to view different PowerPC registers by specifying the name of the register description file. Selecting **View > Register Details** displays the **Register Details** dialog box (Figure 6.10).



Working With the Debugger Special Debugger Features

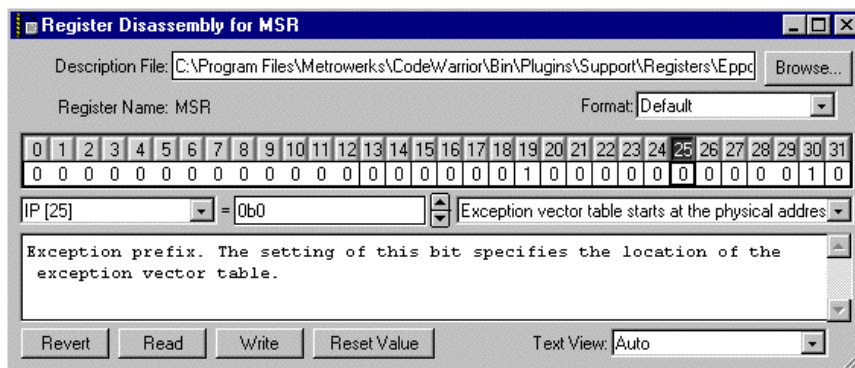
Figure 6.10 Register Details Dialog Box



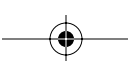
After the CodeWarrior software displays the **Register Details** dialog box, type the name of the register description file in the **Description File** text box to display the applicable register and its values. (Alternatively, you can use the **Browse** button to find the register description file.)

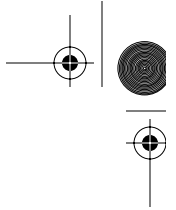
Figure 6.11 shows the **Register Details** dialog box displaying the **MSR** register.

Figure 6.11 Register Details Dialog Box Showing the MSR Register



You can change the format in which the CodeWarrior software displays registers using the **Format** list box. 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** list box.





Working With the Debugger

Special Debugger Features

NOTE For more information, see *CodeWarrior IDE User's Guide*.

EPPC-Specific Debugger Features

The debugger included with CodeWarrior Development Studio for Embedded PowerPC ISA Systems 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

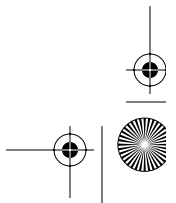
Set Stack Depth

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.

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.

NOTE The Change IMMR command is enabled only if you first select the 826x processor in the EPPC Processor target settings panel.





Working With the Debugger Special Debugger Features

Change MBAR

Unused by the communications/host variants of the Embedded PowerPC processor.

For communications/host EPPC processors, 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.
- 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.





Working With the Debugger

Special Debugger Features

Fill Memory

Select 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.

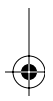
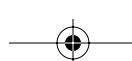
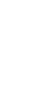
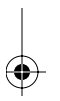
If you enable address translations, the debugger uses the address translation commands in your memory configuration file to perform virtual-to-physical address translations. Refer to Address Translation Commands for a definition of the syntax and effect of an address translation command.

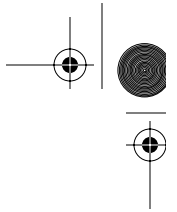
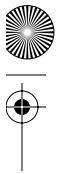
To perform MMU debugging, follow these steps:

1. Add required address translation commands to your memory configuration file.

NOTE To create the required address translation commands, you must know how your application maps memory.

2. In the EPPC Debugger Settings target settings panel, check the Use Memory Configuration File checkbox, and specify the memory configuration file described above in the related text box.
3. Select **Project > Debug**
The debugger downloads your executable to the target device. The executable enables the MMU of the target device.
4. Select **Debug > EPPC > Enable Address Translations**
The debugger performs address translations using the address translation commands it finds in the your memory configuration file.





Working With the Debugger

Special Debugger Features

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)
- 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

Automatically Enabling Address Translation

By default, address translations are disabled. However, if you must download an executable to a virtual address, you must enable address translation *before* the download. To enable address translations before a download, add this statement to your memory configuration file:

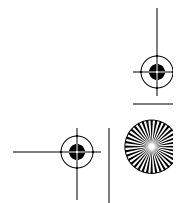
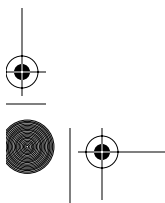
```
AutoEnableTranslations true
```

NOTE Typically, if using virtual addressing, you link your executable with virtual addresses and initialize the MMU of your target device from a target initialization file or boot-loader.

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.





Working With the Debugger

Special Debugger Features

- 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.

- 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.

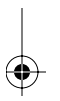
NOTE The Breakpoint Type command is available only if both the selected processor and your debug hardware support it.

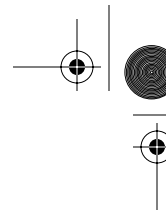
Setting Hardware Breakpoints

To set a hardware breakpoint:

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

Table 6.9 lists the number of breakpoints that can be set for various PowerPC processors. All the processors listed in the table support software breakpoints.





Working With the Debugger Using MetroTRK

Table 6.9 Hardware Breakpoints Supported by PowerPC Processors

CPU	Number of Hardware Breakpoints
5100, 603e, 603ei, 740, 7400, 7410, 745, 7450, 7445, 7455, 750, 755, 8240, 8245, 8250, 8255, 8260, 826x	1
5200, 8270, 8280, 8247, 8248, 8271, 8272	2
555, 565, 8xx	4

Using MetroTRK

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

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

MetroTRK Overview

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

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

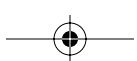
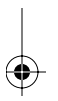
The CodeWarrior software installs the source code for MetroTRK, as well as ROM images and project files for several pre-configured builds of MetroTRK.

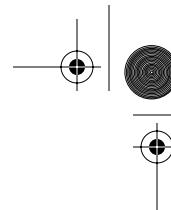
The board-specific directories that contain MetroTRK source code are here:

`CWInstall\PowerPC_EABI_Tools\MetroTRK\Processor\ppc\Board\`

If you are using a board other than the supported boards, 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.





Working With the Debugger Using MetroTRK

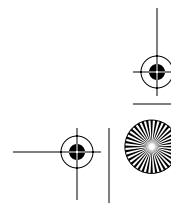
Connecting to the MetroTRK Debug Monitor

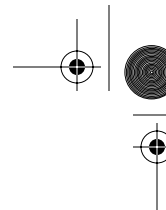
This section presents high-level steps for connecting to a debug monitor on the target board by using a serial port.

The type of serial cable connection that you can use depends on your target board. Table 6.10 lists the type of serial cable connection required for various embedded PowerPC target boards.

Table 6.10 Serial Cable Connection Type for Target Boards

Embedded PowerPC Board	Serial Cable Connection Type
AMC 603ei	Straight serial
Analogue & Micro Rattler MPC8250VR	Use the equipment and cable supplied with the board
Axiom 555, 565	Straight serial
Cogent CMA102 with CMA 278 daughter card	Use the equipment and cable supplied with the board
Embedded Planet RPX Lite 8xx	Use the equipment and cable supplied with the board
Embedded Planet SBC 852 M EP852-1.1	Use the equipment and cable supplied with the board
Embedded Planet SBC 866 H EP862-1.2	Use the equipment and cable supplied with the board
Motorola 555 ETAS	Null modem
Motorola 5100 Ice Cube	Null modem
Motorola 5200 Lite	Null Modem
Motorola Excimer 603e	Null modem
Motorola MPC 8xx MBX	Null modem
Motorola MPC 8xx FADS	Straight serial
Motorola Maximer 7400	Null modem
Motorola Sandpoint	Null modem
Motorola MPC 8255/8260 ADS	Straight serial





Working With the Debugger

Using MetroTRK

Table 6.10 Serial Cable Connection Type for Target Boards (*continued*)

Motorola DUET 885 ADS	Straight serial
Motorola MPC 8272 ADS (PQ27e)	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.

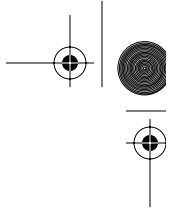
Depending on the board you are using, you can use a MetroTRK project provided by this product to place MetroTRK in flash memory. All the boards in Table 6.10 have self-flashable MetroTRK project targets, except these boards:

- Cogent CMA102 with CMA 278 Daughter card
- Motorola Maximer 7400
- Motorola 5100 Ice Cube
- Motorola 5200 Lite
- Motorola 8260 ADS

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.

NOTE See the project read me file regarding MetroTRK options.



Working With the Debugger Using MetroTRK

MetroTRK Memory Configuration

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

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.

NOTE For the MPC 555 ETAS board, the exception vectors remain in ROM.

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.

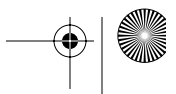
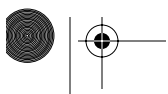
NOTE Do not change the location of the exception vectors because the processor expects the exception vectors to reside at the set location.

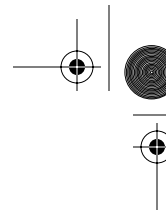
Data and Code Sections

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

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 the 603e and 7xx processors, the main exception handler must reside in cacheable memory if the instruction cache is enabled. On some boards the ROM is





Working With the Debugger Using MetroTRK

not cacheable; consequently, the main exception handler must reside in RAM if the instruction cache is enabled

RAM does contain a MetroTRK data section. For example, on the Motorola 8xx FADS board, the default address where MetroTRK data section starts is 0x3F8000 and ends at the address 0x3FA000.

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.

For example, on the Motorola 8xx ADS and Motorola 8xx MBX boards, the MetroTRK stack resides between the addresses 0x3F6000 and 0x3F8000.

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

MetroTRK Memory Map

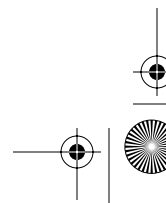
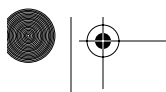
For more information on the MetroTRK memory map, see the board specific information provided with the MetroTRK source code.

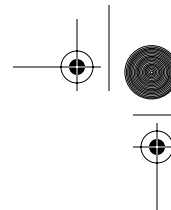
Using MetroTRK for Debugging

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

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.





Working With the Debugger

Debugging External ELF Files

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

Debugging External ELF Files

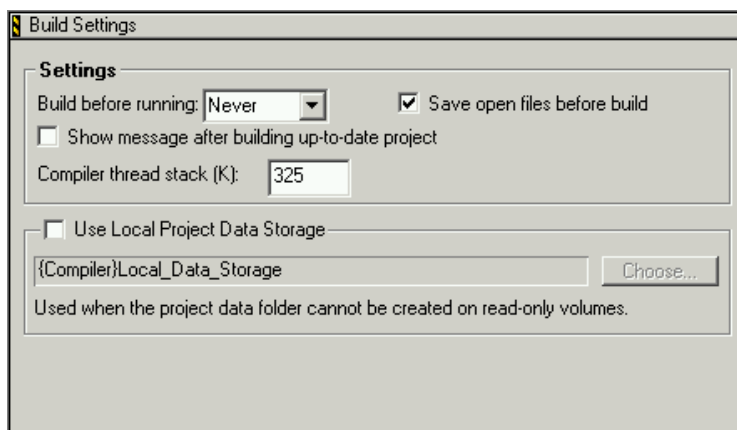
You can use the CodeWarrior debugger to debug an ELF file that you previously created and compiled in a different environment than the CodeWarrior IDE. Before you open the ELF file for debugging, you must examine some IDE preferences and change them if needed. In addition, you must customize the default 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.

Preparing to Debug an ELF File

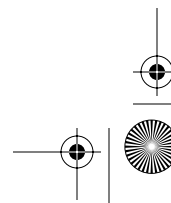
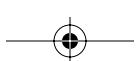
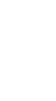
Before you debug an ELF file, you need to change certain IDE preferences and modify them if needed.

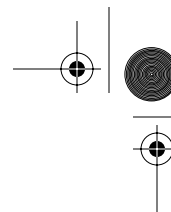
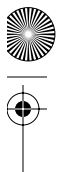
1. Select **Edit > Preferences**.
The **IDE Preferences** window appears.
2. In the **IDE Preference Panels** list, select **Build Settings**.
The **Build Settings** panel (Figure 6.12) appears.

Figure 6.12 Build Settings Panel



3. Make sure that the **Build before running** list box specifies **Never**.





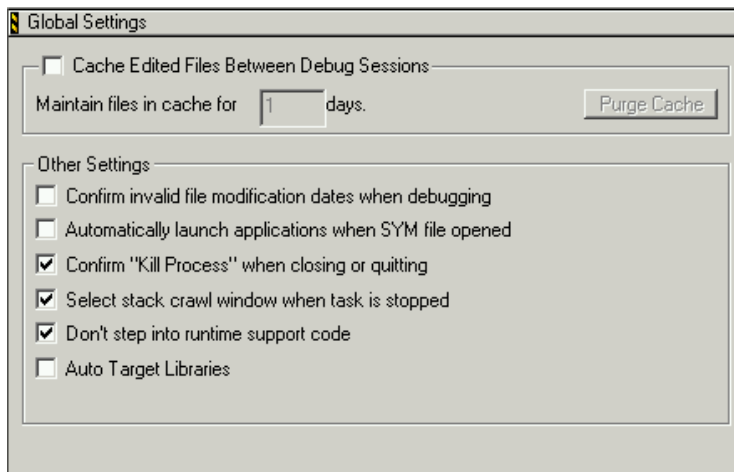
Working With the Debugger

Debugging External ELF Files

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 6.13) appears.

Figure 6.13 Global Settings Panel



5. Make sure that the **Cache Edited Files Between Debug Sessions** checkbox is clear.
6. Close the **IDE Preferences** window.

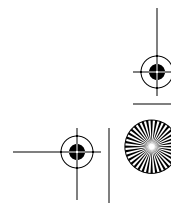
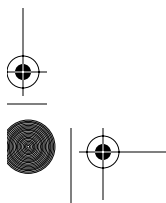
You successfully examined the relevant **IDE Preference** settings and changed them, if needed.

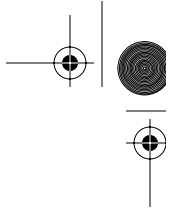
Customizing the Default XML Project File

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.

```
CWInstall\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.





Working With the Debugger

Debugging External ELF Files

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.

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.

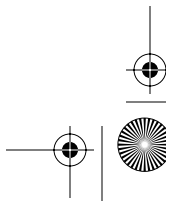
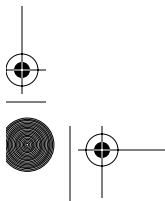
Debugging an ELF File

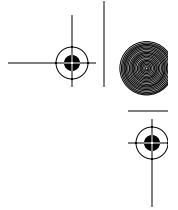
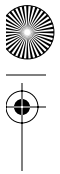
This section explains how to prepare for debugging an ELF file for the first time.

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





Working With the Debugger Debugging External ELF Files

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 *CodeWarrior IDE User's Guide*.

After debugging, the ELF file you imported is unlocked. If you choose to build your project in the CodeWarrior software (rather than using another compiler), you can select **Project > Make** to build the project, and the CodeWarrior software saves the new ELF file over the original one.

Additional Considerations

This section explains information that is useful when debugging ELF files.

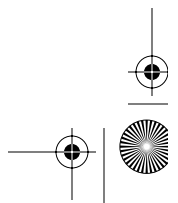
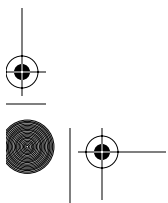
Deleting Old Access Paths From ELF-Created Projects

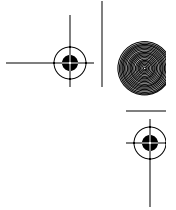
After you create a project to allow debugging an ELF file, you can delete old access paths that no longer apply to the ELF file by using these methods:

- Manually remove the access paths from the project in the **Access Paths** settings panel
- Delete the existing project for the ELF file and recreate it by dragging the ELF file icon to the IDE

Removing Files From ELF-Created Projects

After you create a project to allow debugging an ELF file, you may later delete one or more files from the ELF project. However, if you open the project again after rebuilding





Working With the Debugger

Debugging Multiple ELF Files

the ELF file, the CodeWarrior software does not automatically remove the deleted files from the corresponding project. For the project to include only the current files, you must manually delete the files that no longer apply to the ELF file from the project.

Recreating ELF-Created Projects

To recreate a project that you previously created from an ELF file:

1. Close the project if it is open.
2. Delete the project file. The project file has the file extension `.mcp` and resides in the same directory as the ELF file.
3. Drag the ELF file icon to the IDE. The CodeWarrior IDE opens a new project based on the ELF file.

Debugging Multiple ELF Files

This section describes how to use the CodeWarrior IDE to simultaneously debug multiple ELF files on bare board target systems. This is similar to debugging both an application and an associated shared library.

In order to debug multiple ELF files simultaneously with the CodeWarrior IDE, both ELF files must be available on the host computer, and must have DWARF 1.x, DWARF 2.x, or STABS symbolic information.

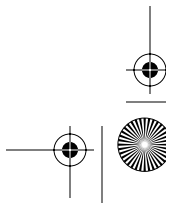
In this section, we show you how to debug multiple ELF files simultaneously under these scenarios:

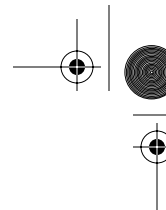
- Debugging a Secondary ELF File with a Serial Connection
- Debugging a Secondary PIC/PID ELF File
- Debugging a Secondary External ELF File

Debugging a Secondary ELF File with a Serial Connection

In this scenario, you have two CodeWarrior projects that generate ELF files. The first project builds the *main application*, an application that loads and launches a second application. The second project builds the *secondary application* that you send to the target system in S-Record format over a serial connection.

NOTE All source files for both projects must be available on the host system.





Working With the Debugger Debugging Multiple ELF Files

NOTE You can use two build targets in a single CodeWarrior project to generate both the main and the secondary applications, or you can use separate CodeWarrior projects to build each application. In this section, we use separate projects.

1. Connect a serial cable between the host computer and the target system.
2. Start the target system.
3. In the CodeWarrior IDE, open the main and secondary application projects.

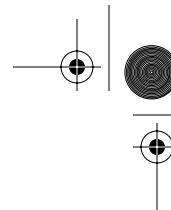
TIP You can easily create a CodeWarrior project from an existing ELF file by dragging the ELF file from Windows Explorer and dropping it on the CodeWarrior menu bar.

4. Configure both projects to connect to the target system using a CCS remote connection over the serial cable.

TIP Read “Using Remote Connections” on page 91 for detailed information about configuring remote connections.

5. In the main application project, check the **Generate S-Record** checkbox in the **EPPC Linker** target settings panel.
6. In the secondary application project, check the **Generate S-Record** checkbox in the **EPPC Linker** target settings panel.
7. Build the secondary application.
The IDE generates the secondary ELF file in the **Bin** subfolder of the project folder.
8. In the main application project, add the secondary ELF file to the **Other Executables** target settings panel.
9. Use a terminal application such as HyperTerminal to connect to the target system.
10. In the terminal, send the secondary ELF over the serial connection to the target system.
11. In the CodeWarrior IDE, open the **main.c** source code file in the main application project.
12. In the CodeWarrior editor, set a breakpoint in the **main.c** source code file where the loading of the S-Record (**.mot**) file is finished, before application launch.
13. Start a debug session of the main application project.
The debugger stops at the main application entry point.
14. In the terminal window, paste the content of the secondary S-Record (**.mot**) file the CodeWarrior IDE generated when you built the secondary project.
15. From the CodeWarrior menu bar, select **View > Symbolics** to view the **Symbolics** window.





Working With the Debugger

Debugging Multiple ELF Files

16. In the **Executables** pane of the Symbolics window, select **secondary.elf**.

17. In the **Files** pane of the Symbolics window, select **main.c**.

The **Symbolics** window displays the **main.c** source code file.

18. In the **Source** pane of the **Symbolics** window, click the breakpoint column (at the left side of the source code listing) to set a breakpoint somewhere in the **main.c** file.

19. From the CodeWarrior menu bar, select **Project > Run**.

The debugger executes the main application. The main application loads the secondary application and passes control it. The debugger halts secondary application execution when it reaches the breakpoint you set.

TIP You can also upload the secondary application to the target system by using the **Load/Save Memory** function of the IDE. For more information about this feature, read "Load/Save Memory" on page 113.

Debugging a Secondary PIC/PID ELF File

In this scenario, you create a simple CodeWarrior project to build an application that uses PIC/PID addressing.

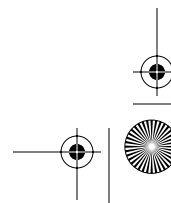
1. Connect a serial cable between the host computer and the target system.
2. Start the target system.
3. In the CodeWarrior IDE, open the main and secondary application projects.

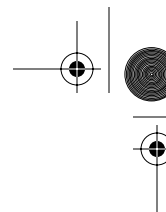
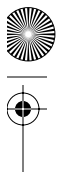
TIP You can easily create a CodeWarrior project from an existing ELF file by dragging the ELF file from Windows Explorer and dropping it on the CodeWarrior menu bar.

4. Configure both projects to connect to the target system using a CCS remote connection over the serial cable.

TIP Read "Using Remote Connections" on page 91 for detailed information about configuring remote connections.

5. In the main application project, check the **Generate S-Record** checkbox in the **EPPC Linker** target settings panel.
6. In the secondary application project, check the **Generate S-Record** checkbox in the **EPPC Linker** target settings panel.
7. In the secondary application project, set the **Code Address** text box in the **Segment Addresses** area of the **EPPC Linker** target settings panel to an appropriate value.



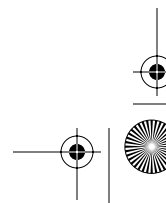
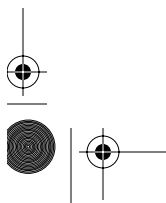


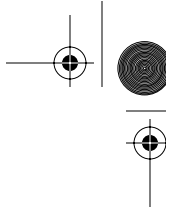
Working With the Debugger Debugging Multiple ELF Files

8. Build the secondary application.
The IDE generates the secondary ELF file in the `Bin` subfolder of the project folder.
9. In the main application project, add the secondary ELF file to the **Other Executables** target settings panel.
10. Use a terminal application such as HyperTerminal to connect to the target system.
11. Start a debug session of the main application project.
12. From the CodeWarrior menu bar, select **View > Symbolics** to view the **Symbolics** window.
13. In the **Executables** pane of the Symbolics window, select **secondary.elf**.
14. In the **Files** pane of the Symbolics window, select **main.c**.
The **Symbolics** window displays the **main.c** source code file.
15. In the **Source** pane of the **Symbolics** window, click the breakpoint column (at the left side of the source code listing) to set a breakpoint somewhere in the **main.c** file.
16. Start a CodeWarrior debug session for the main application project.
17. From the CodeWarrior menu bar, select **Debug > EPPC > Load/Save Memory** to display the **Load/Save Memory** window.
18. Use the **Load/Save Memory** window to load the secondary ELF at the address you set in the **Code Address** text box in the **EPPC Linker** target settings panel.

TIP For instructions showing how to use the **Load/Save Memory** dialog box, read “Load/Save Memory” on page 113.

19. From the CodeWarrior menu bar, select **View > Symbolics** to view the **Symbolics** window.
20. In the **Executables** pane of the Symbolics window, select **secondary.elf**.
21. In the **Files** pane of the Symbolics window, select **main.c**.
The **Symbolics** window displays the **main.c** source code file.
22. In the **Source** pane of the **Symbolics** window, drag the program counter indicator (the blue arrow) to a line of source code in the **main.c** file.
The thread window displays the source code of the **main.c** file. The program counter (blue arrow) appears at the line of source code you specified.
23. Change the view mode of thread window to **Mixed**. The **Address** line displays the address you set in the **Code Address** text box in the **Segment Addresses** area of the **EPPC Linker** target settings panel.





Working With the Debugger

Debugging Multiple ELF Files

Debugging a Secondary External ELF File

In this scenario, you have a main application CodeWarrior project, and a secondary application built without CodeWarrior tools.

NOTE The secondary ELF file you want to debug must have debug information in one of these formats: DWARF 1.x, DWARF 2.x, or Stabs.

NOTE All of the source files for the secondary ELF file must be available on the host system.

User will start the debug session of the main application.

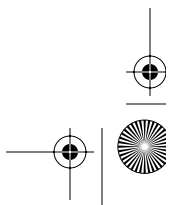
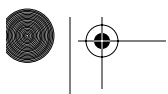
1. Using other tools, build the secondary ELF file outside of the CodeWarrior IDE.
2. Drag the secondary ELF file from Windows Explorer to the CodeWarrior menu bar.
The CodeWarrior IDE creates a CodeWarrior project for the secondary ELF file.
3. Open the main application CodeWarrior project.
4. In the main application project, add the secondary ELF file to the **Other Executables** target settings panel.
5. Start a CodeWarrior debug session for the main application project.
6. From the CodeWarrior menu bar, select **Debug > EPPC > Load/Save Memory** to display the **Load/Save Memory** window.
7. Use the **Load/Save Memory** window to load the secondary ELF at the address you set in the **Code Address** text box in the **EPPC Linker** target settings panel.

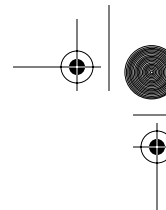
TIP For instructions showing how to use the **Load/Save Memory** dialog box, read "Load/Save Memory" on page 113.

8. Start a debug session of the main application project.
9. From the CodeWarrior menu bar, select **View > Symbolics** to view the **Symbolics** window.
10. In the **Executables** pane of the Symbolics window, select **secondary.elf**.
11. In the **Files** pane of the Symbolics window, select **main.c**.

The **Symbolics** window displays the **main.c** source code file.

The thread window displays the source code of the **main.c** file. The program counter (blue arrow) appears at the line of source code you specified.





Accessing Translation Look-aside Buffers

This section shows you how to use the CodeWarrior IDE to access PowerQUICC III Level 2 Memory Management Unit (MMU) translation look-aside buffers (TLBs). PowerQUICC III Level 2 MMUs have two TLBs:

- TLB0 — a 256-element, two-way set associative array supporting 4K page sizes
- TLB1 — a 16-element fully-associative array supporting nine page sizes

During debug sessions on supported PowerPC systems, the CodeWarrior **Registers** window displays **TLB0 Registers** and **TLB1 Registers** register groups.

Initializing TLBs

You can use `writereg128` commands in debug initialization files to set up TLBs at target system startup. For details, read “`writereg128`” on page 207.

Accessing TLB Registers

To view the **Registers** window:

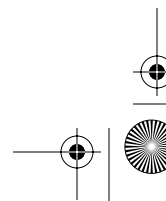
1. Start the CodeWarrior IDE.
2. Open (or create) a project that targets the PowerPC system you want to debug.
3. From the CodeWarrior menu bar, select **Project > Debug**.

The IDE starts a debug session, connects to the target system, and halts the system at the program entry point.

4. Select **View > Registers**.

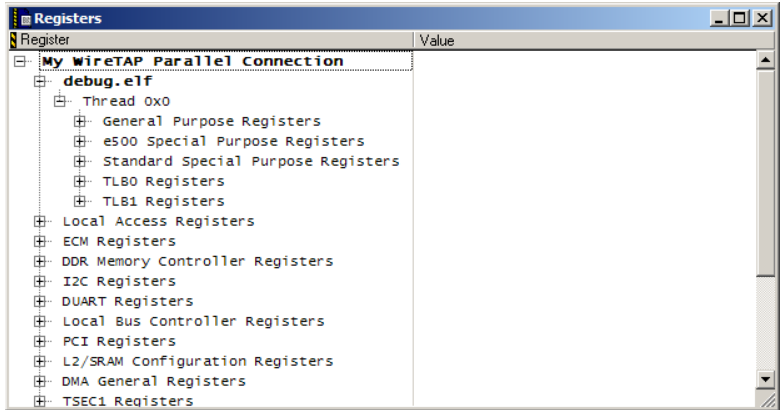
The **Registers** window (Figure 6.14 on page 132) appears.

The **Registers** window shows all of the supported registers for a target system. In this window, the IDE displays TLBs as **TLB0 Registers** and **TLB1 Registers** groups under the ELF file hierarchy, as shown in Figure 6.14 on page 132.



Working With the Debugger Accessing Translation Look-aside Buffers

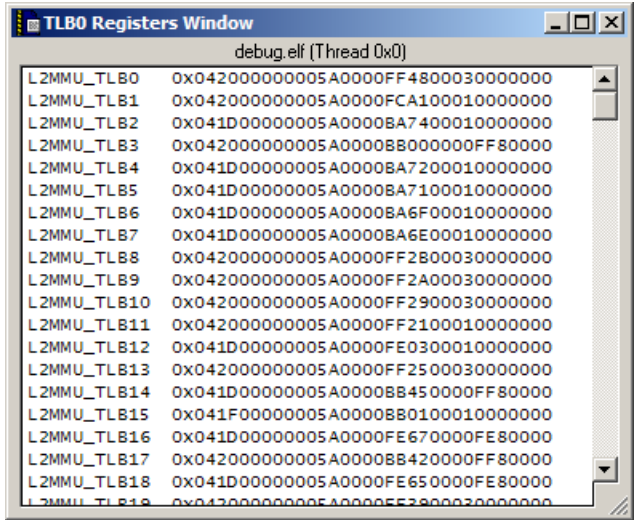
Figure 6.14 Registers Window — TLB Register Groups Displayed



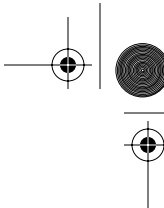
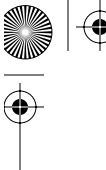
To view all of the elements of a TLB register group, double-click the group you want to view. A window appears that displays all of the elements of the selected TLB.

For example, if you double-click the TLB0 Registers group, the **TLB0 Registers** window (Figure 6.15) appears.

Figure 6.15 TLB0 Registers Window



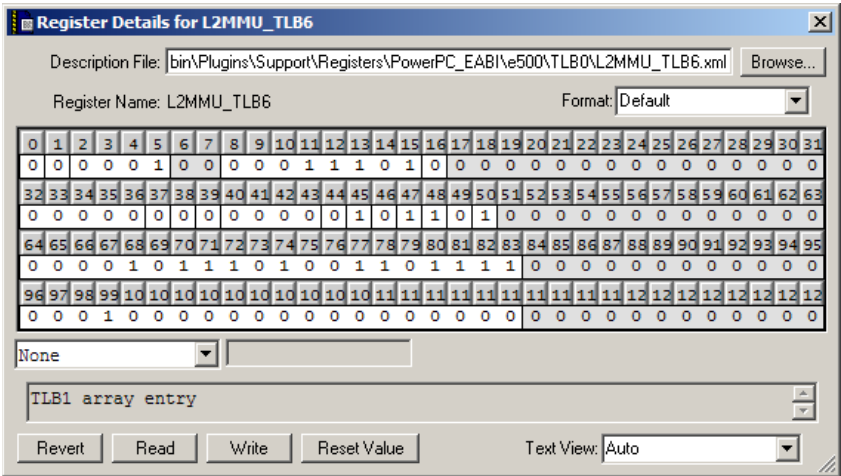
This window shows all of the TLB registers, and their contents. To modify TLB registers during a CodeWarrior debug session:



Working With the Debugger
Using the Command-Line Debugger

- 1. In the **Registers** window, select the TLB register you want to modify.
The IDE highlights your selection.
- 2. From the CodeWarrior menu bar, select **View > Register Details**.
The **Register Details** window () appears.

Figure 6.16 Register Details Window

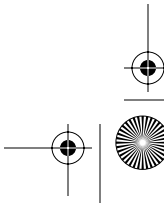
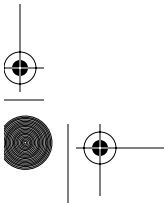


This window lets you view register contents in different formats, and change portions of the selected register.

Using the Command-Line Debugger

The CodeWarrior IDE supports a command-line interface to some of its features including the debugger. You can use the command-line interface together with various scripting engines, such as the Microsoft® Visual Basic® script engine, the Java™ script engine, TCL, Python, and Perl. You can also issue a command line that saves a log file of command-line activity.

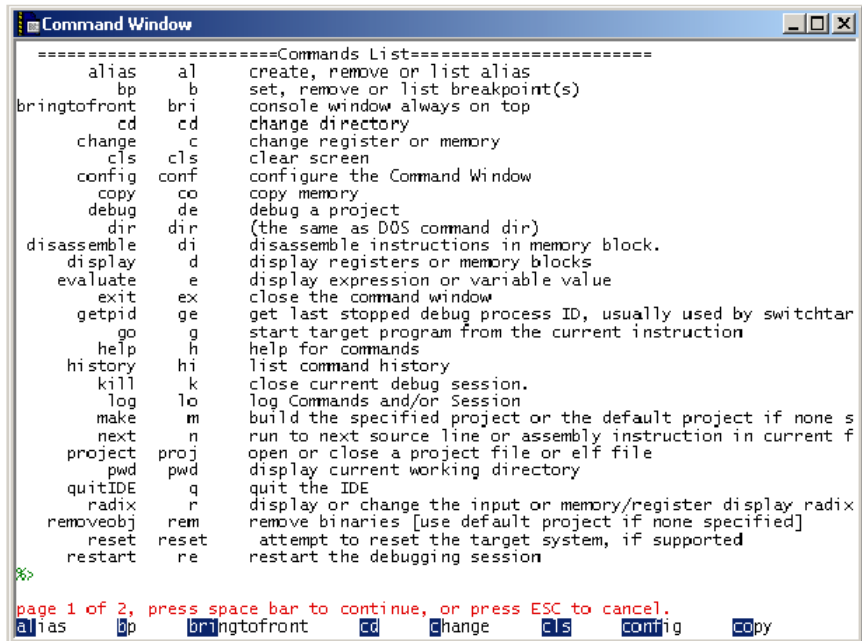
Use the **Command** window (Figure 6.17) to issue command lines to the IDE. For example, enter `debug` to start a debugging session. The **Command** window in the IDE shows the standard output and standard error streams of command-line activity. To open the **Command** window, select **View > Command Window**.



Working With the Debugger

Using the Command-Line Debugger

Figure 6.17 Command Window

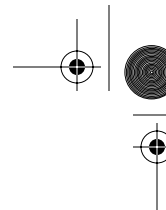


To issue a command line, bring forward the Command window, type the command line, and press Enter or Return. The IDE executes the command line that you entered.

If you work with hardware as part of your project, you can use the Command window to issue command lines to the IDE while the hardware is running.

NOTE Enter help to see a list of available commands and a brief explanation of each command.

Refer to the *IDE Automation Guide* for information about using the command-line debugger. This document has details on the definition, shortcut, syntax, and examples of the various command line options for debugging.



7

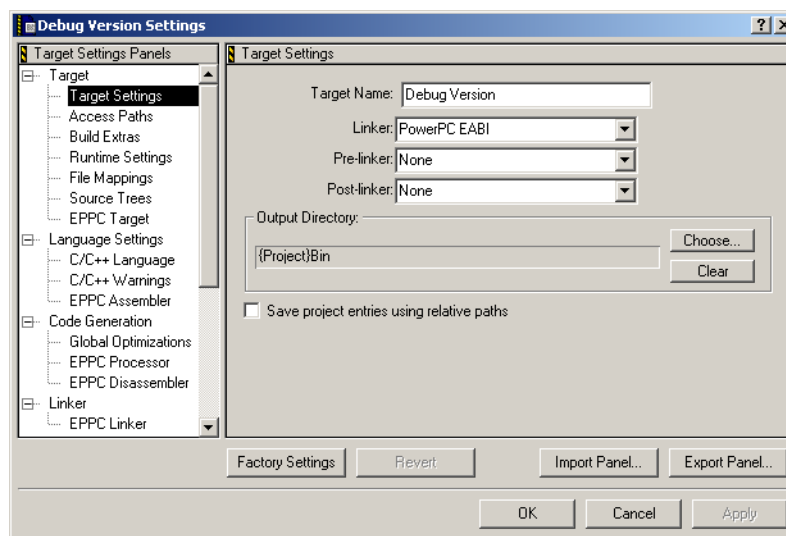
Target Settings Reference

This chapter describes the build target settings that affect code generation for embedded PowerPC systems development. *Target settings* allow you to control a variety of features such as compiler options, linker output, error and warning messages, and so forth.

Target Settings Overview

All CodeWarrior build target settings are located in the **Target Settings** window (shown in Figure 7.1).

Figure 7.1 Target Settings Window



To open the Target Settings window, select **Edit > TargetName Settings** (where *TargetName* is the current build target in the project) or display the **Targets** page of the project window and double-click the build target in the list.

NOTE All items in this window are described in the *CodeWarrior IDE User's Guide*.



Target Settings Reference

Target Settings Overview

NOTE When you use the EPPC New Project Wizard to create a new project, the wizard configures all settings in all panels to reasonable or default values.

This manual documents only those settings panels of specific interest to Mac OS developers. Refer to Table 7.1 to learn where to find documentation on the settings panels not documented in this manual.

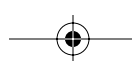
Table 7.1 Documentation About Other Settings Panels

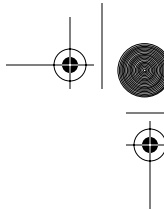
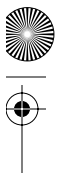
Settings Panel	Manual
Access Paths Build Extras Runtime Settings File Mappings Source Trees Global Optimizations Custom Keywords Other Executables Debugger Settings Remote Debugging	<i>CodeWarrior IDE User's Guide</i>
C/C++ Language C/C++ Preprocessor C/C++ Warnings	<i>C Compilers Reference</i>

Table 7.2 lists the settings panels in this chapter.

Table 7.2 EPPC Target Settings Panels

Target Settings	GNU Linker
EPPC Target	BatchRunner PreLinker
GNU Target	BatchRunner PostLinker
EPPC Assembler	GNU Environment
GNU Assembler	GNU Tools
EPPC Processor	Analyzer Connection
EPPC Disassembler	Debugger PIC Settings
GNU Disassembler	EPPC Debugger Settings
GNU Compiler	PQ3 Trace Buffer





Target Settings Reference
Target Settings

Table 7.2 EPPC Target Settings Panels (*continued*)

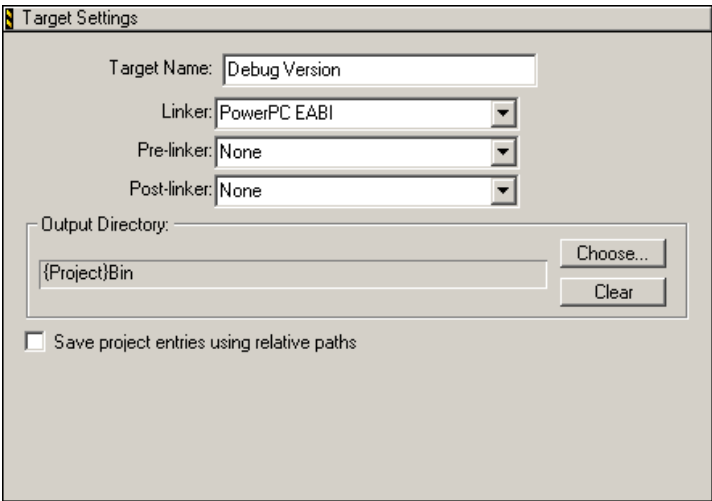
EPPC Linker	Source Folder Mapping
GNU Post Linker	System Call Service Settings

Target Settings

The **Target Settings** panel (shown in Figure 7.2) is the most important settings panel in the Target Settings window, because when you select a linker in this panel, you specify the target operating system and processor for the build target.

The IDE shows and hides other settings panels in this window based on the selected linker. Because the linker choice affects the visibility of other settings panels, you should always set the linker first.

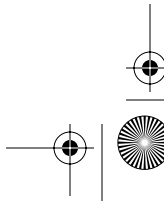
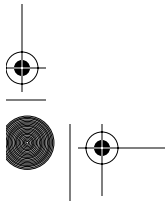
Figure 7.2 Target Settings Panel

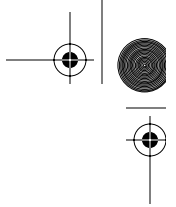


Target Name

The **Target Name** text box specifies the build target name. When you use the **Targets** view in the **Project** window, you see the name that you have specified.

The name you set here is not the name of your final output file. It is the name you assign to the build target for your personal use. Set the final output file name in the EPPC Target panel.





Target Settings Reference

EPPC Target

Linker

Choose a linker from the **Linker** list box. Selecting a linker determines will activate and deactivate the appropriate target settings panels. The available linkers are:

- **EPPC GNU Linker**
- **External Build Linker**
- **PowerPC EABI linker**

Pre-Linker

Some build targets have pre-linkers that perform work on object code before it is linked. The only pre-linker available in this product is the **BatchRunner PreLinker**.

Post-Linker

Some build targets have post-linkers that perform additional work (such as object code format conversion) on the final executable file. The only pre-linker available in this product is the **BatchRunner PostLinker**.

Output Directory

This directory contains your final linked output file. The project directory is the default location. Click **Choose** to specify another directory.

Save Project Entries Using Relative Paths

To add two or more files with the same name to a project, check this checkbox. The IDE includes information about the path used to access the file as well as the file name when it stores information about the file. When searching for a file, the IDE combines access path settings with the path settings it includes for each project entry.

If this checkbox is cleared, each project file must have a unique name. The IDE only records information about the file name of each project entry. When searching for a file, the IDE only uses access paths.

EPPC Target

Use the **EPPC Target** settings panel (Figure 7.3) to specify the name and configuration of your final output file.



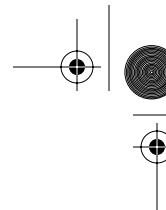
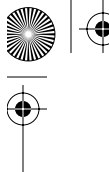
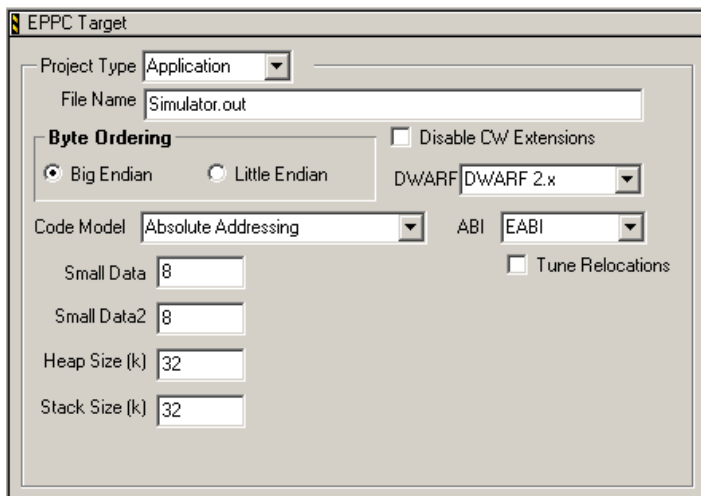


Figure 7.3 EPPC Target Settings Panel



Project Type

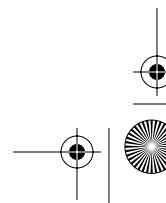
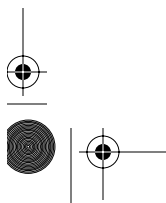
Select the kind of project you are creating from the **Project Type** list box. The options available are:

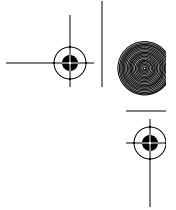
- **Application**
- **Library**
- **Partial Link**

The option you choose also controls the visibility of other items in this panel. If you choose **Library** or **Partial Link**, the **Heap Size**, **Stack Size**, and **Tune Relocations** items disappear from this panel 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

The **File Name** text box specifies the name of the executable or library you create. By convention, application names should end with the extension **.elf**, and library names should end with the extension **.a**. If the output 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-Records and Map files in the EPPC Linker panel).





Target Settings Reference

EPPC Target

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.

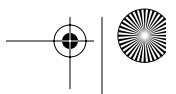
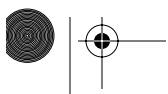
The **Disable CW Extensions** checkbox in the **Project** settings panel is checked when creating C libraries for use with third-party linkers. However, all third-party linkers do not require this checkbox to be checked; you may need to try both settings. When building a CW linked application, clear the **Disable CW Extensions** checkbox to avoid generating a larger application. Assembly files do not need this option; and C++ libraries are not portable to other linkers.

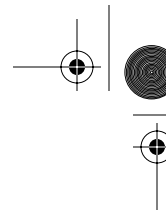
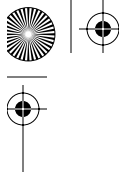
DWARF

Use the **DWARF** list box to select the version of the Debug With Arbitrary Record Format (DWARF) debugging information format. The linker ignores debugging information that is not in the format that you select from the **DWARF** list box.

ABI

Use the **ABI** list box to select the Application Binary Interface (ABI) used for function calls and structure layout.





Tune Relocations

The tune relocations functionality pertains to object relocation and is only available for the EABI and SDA PIC/PID ABIs.

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

Checking the **Tune Relocations** checkbox has these effects:

- For EABI, the 14-bit branch relocations are changed to 24-bit branch relocations only if they cannot reach the calling site from the original relocation
- For SDA PIC/PID, 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, refer to this release notice:

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

Code Model

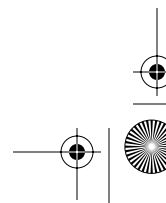
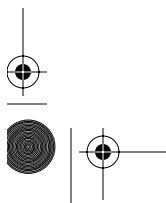
The **Code Model** list box lets you determine the addressing mode for the generated executable. Table 7.3 describes the items in this menu.

Table 7.3 Code Model Menu Items

Menu Item	Description
Absolute Addressing	Select to have the IDE generate non-relocatable binary files.
SDA Based PIC/PID Addressing	Select to have the IDE generate relocatable binary files that use position-independent-code (PIC) / position-independent-data (PID) addressing. You can load resulting binary files at any valid address.

Small Data

The **Small Data** text box specifies the threshold size (in bytes) for an item considered by the linker to be small data. The linker stores small data items in the **Small Data** address space. The compiler can generate faster code to access this data.





Target Settings Reference

EPPC Target

Small Data2

The **Small Data2** text box specifies the threshold size (in bytes) for an item considered by the linker to be small data. The linker stores read-only small data items in the **Small Data2** address space. The compiler can generate faster code to access this data.

Heap Size

The **Heap Size** text box specifies the amount of RAM allocated for the heap. The value that you enter is in kilobytes. The heap is used if your program calls `malloc` or `new`. This text box is not applicable when building a library project; heaps are associated only with applications.

Stack Size

The **Stack Size** text box specifies the amount of RAM allocated for the stack. The value you enter is in kilobytes. This text box is not applicable when building a library project; stacks are associated only with applications.

NOTE You can allocate stack and heap space based on the amount of memory that you have on your target hardware. If you allocate more memory for the heap and/or stack than you have available RAM, your program will not run correctly.

Optimize Partial Link

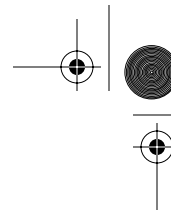
The **Optimize Partial Link** checkbox appears only if you select **Partial Link** from the **Project Type** list box. Check this checkbox to directly download the output of your partial link. This instructs the linker to:

- Allow the project to use a linker command file. This is important so that all of the diverse sections can be merged into the `.text`, `.data`, or `.bss` segments. If you do not let a linker command file merge them for you, the chances are good that the debugger is not be able to show you source code properly.
- Allow optional deadstripping. This is recommended.

NOTE The project must have at least one entry point for the linker to know to deadstrip.

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





Target Settings Reference

GNU Target

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

- Change common symbols to `.bss` symbols. This allows you to examine the variable in the debugger.
- Allow a special type of partial link that has no unresolved symbols. This is the same as the Diab linker's `-r2` command line argument.

When this checkbox is cleared, the output file remains as if you passed the `-r` argument on the command line.

Deadstrip Unused Symbols

The **Deadstrip Unused Symbols** checkbox is available only if you select the **Optimize Partial Link** checkbox. Check the **Deadstrip Unused Symbols** checkbox to have the linker deadstrip any symbols that are not used. This option makes your program smaller by stripping the symbols not referenced by the main entry point or extra entry points in the `force_active` linker command file directive.

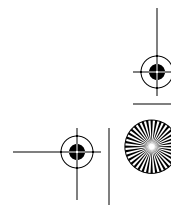
Require Resolved Symbols

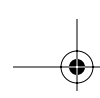
The **Require Resolved Symbols** checkbox is available only if you select the **Optimize Partial Link** checkbox. Check the **Require Resolved Symbols** checkbox to instruct the linker to resolve all symbols in your partial link. If any symbols are not present in one of the source files or libraries in your project, an error message is displayed.

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

GNU Target

The GNU Target settings panel (Figure 7.4) lets you select the project type and the name of the output file.

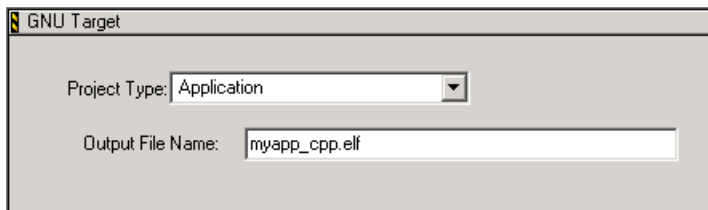




Target Settings Reference

EPPC Assembler

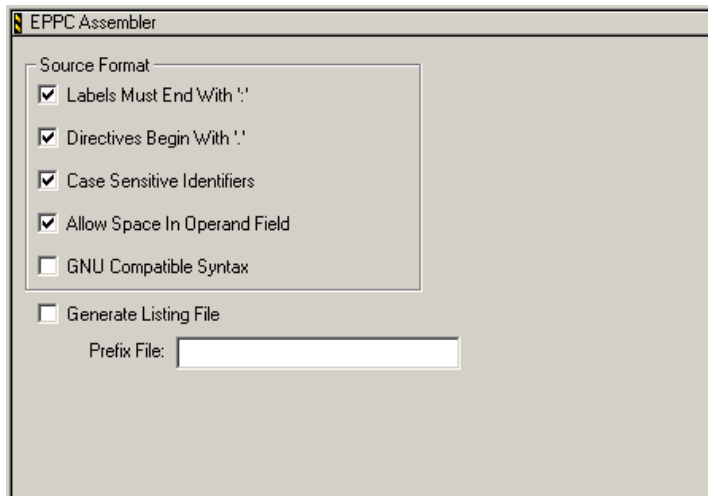
Figure 7.4 GNU Target



EPPC Assembler

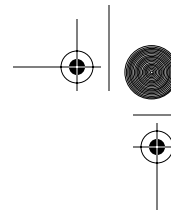
Use the **EPPC Assembler** settings panel (Figure 7.5) to determine the format used for the assembly source files and the code generated by the EPPC assembler.

Figure 7.5 EPPC Assembler Settings Panel



NOTE If you used a previous version of this panel, you may have noticed that the Processor region has disappeared. The processor settings for the assembler are now specified in the EPPC Processor settings panel, using the Processor list box.





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 mode allows:

- Redefining all equates regardless of whether they were defined using `.equ` or `.set`
- Ignoring the `.type` directive
- Treating undefined symbols as imported
- Using GNU compatible arithmetic operators. The symbols `<` and `>` mean left-shift and right-shift instead of less than and greater than. In addition, the symbol `!` means bitwise-not, not 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`

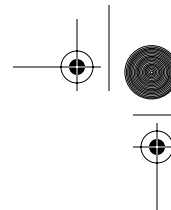
Generate Listing File

A listing file contains file source 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 assembling the source files in the project.

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.



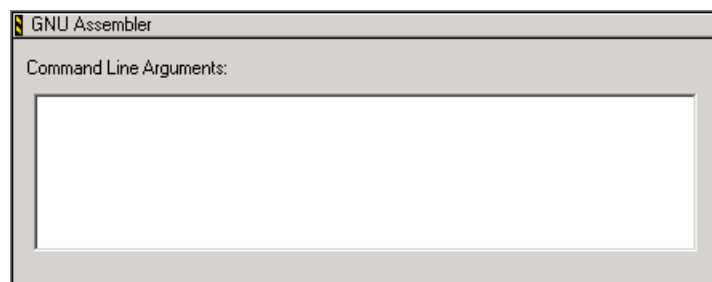
Target Settings Reference

GNU Assembler

GNU Assembler

The GNU Assembler panel (Figure 7.6) lets you enter command line arguments to be passed to the assembler. Type your desired command line arguments into the **Command Line Arguments** text field.

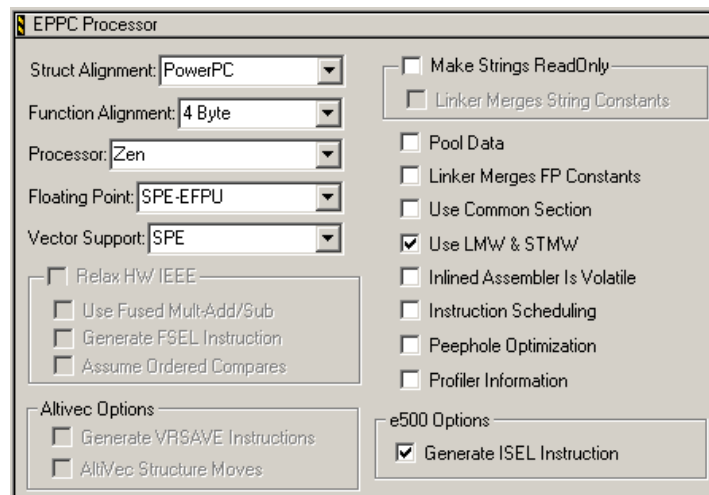
Figure 7.6 GNU Assembler

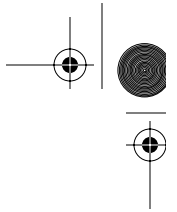
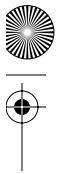


EPPC Processor

Use the **EPPC Processor** settings panel (Figure 7.7) to perform processor-dependent code-generation settings.

Figure 7.7 EPPC Processor Settings Panel





Target Settings Reference

EPPC Processor

Struct Alignment

The **Struct Alignment** list box has the default selection **PowerPC**. To conform with the PowerPC EABI and interoperate with third-party object code, this setting should remain **PowerPC**. Other settings may lead to reduced performance or alignment violation exceptions. For more information, refer to the explanation of pragma “pack” on page 47.

NOTE If you choose another setting for **Struct Alignment**, 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** list box to select alignments from 4 (the default) to 128 bytes. These selections corresponds to `#pragma function_align`. For more information, see “function_align” on page 45.

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

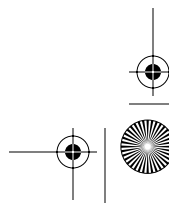
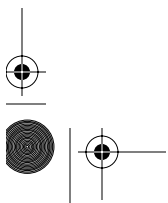
Use the **Processor** list box to specify the targeted processor. Choose **Generic** if the processor you are working with is not listed, or if you want to generate code that runs on any PowerPC processor. Choosing **Generic** allows the use of all optional instructions and the core instructions for the 603, 604, 740, and 750 processors.

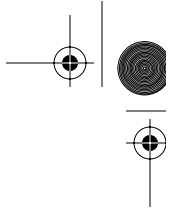
Selecting a particular target processor has these results:

- **Instruction scheduling** — If the **Instruction Scheduling** checkbox (also in the EPPC Processor panel) is selected, the processor selection helps determine 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 PowerPC processor being targeted:

```
#define __PPCnumber__ 1
```

For the PowerPC 821 processor, for instance, the symbol would be `__PPC821__`. If you select **Generic**, the macro `__PPCGENERIC__` is defined to 1.





Target Settings Reference

EPPC Processor

- Floating-point support — The **None** (no floating-point), **Software** and **Hardware** option buttons 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** option buttons. If the **Hardware** option button is cleared, the **Use FMADD & FMSUB** checkbox is not available.

Floating Point

Use the **Floating Point** list box to determine 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:

- choose an option from the **Floating Point** list box
- include the corresponding runtime library in your project

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

The description of each option follows:

- None** — Prevents floating-point operations.
- Software** — Emulates floating-point operations in software.

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

- Hardware** — Performs hardware floating-point operations.

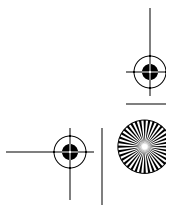
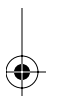
NOTE Do not select the **Hardware** option if your target device does not have hardware floating-point support.

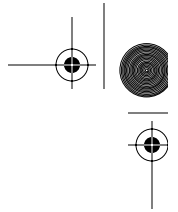
- SPE-EFPU** — Performs single float operations through e500-EFPU hardware instructions support. Performs double float operation by utilizing the software emulation library.

Vector Support

Several processors support vector instructions. If you want to allow vector instructions for your processor, select a vector type that your processor supports from the **Vector Support** list box. Currently, only the AltiVec and SPE vector units are supported.

If you select the **AltiVec** option from the **Vector Support** list box, additional options appear in the AltiVec Options area.





Target Settings Reference EPPC Processor

There are currently no additional options for SPE vector support.

Relax HW IEEE

The **Relax HW IEEE** checkbox is available only if you select **Hardware** from the **Floating Point** list box. Check the **Relax HW IEEE** checkbox to have the compiler generate faster code by ignoring certain 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 checkbox to generate PowerPC Fused Multi-Add/Sub instructions, which result in smaller and faster floating-point code.

This may generate unexpected results because of the greater precision of the intermediate values. The generated results are slightly more accurate than those specified by IEEE because of an extra rounding bit between the multiply and the add/subtract.

Generate FSEL Instruction

Check this checkbox to generate the faster executing FSEL instruction. The FSEL option allows the compiler to optimize the pattern $x = (\text{condition} ? y : z)$, where x and y are floating-point values.

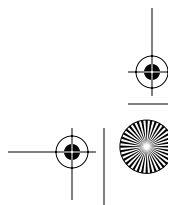
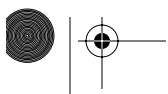
FSEL is not accurate for denormalized numbers and may have issues related to unordered compares.

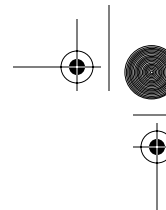
Assume Ordered Compares

Check this checkbox to allow the compiler to ignore issues with unordered numbers, such as NAN, while comparing floating-point values. In strict IEEE mode, any comparison against NAN, except not-equal-to, returns false.

This optimization ignores this provision, thus allowing the following conversion:

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





Target Settings Reference

EPPC Processor

Altivec Options

The **Altivec Options** area contains checkboxes for specifying additional options for Altivec vector support.

Altivec Structure Moves

Check the **Altivec Structure Move** checkbox if you want the CodeWarrior software to use Altivec instructions when the compiler copies a structure.

Generate VRSAGE Instructions

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

When a function call happens, the value of the VRSAGE 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 as well, in an area called the vector register save area, before changing the values in any of those registers.

Check the **Generate VRSAGE Instructions** checkbox only when developing for a real-time operating system that supports AltiVec. Checking the **Generate VRSAGE Instructions** checkbox tells the CodeWarrior software to generate instructions to save and restore these vector-register-related values.

Make Strings Read Only

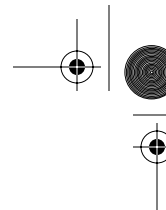
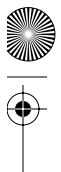
Check the **Make Strings Read Only** checkbox to store string constants in the read-only `.rodata` section. Leave this checkbox clear to store string constants in the ELF-file data section. The **Make Strings Read Only** checkbox corresponds to `#pragma readonly_strings`. The default setting is OFF.

If you check the **Make Strings Read Only** checkbox, the **Linker Merges String Constants** checkbox is 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 of `.data`, `.bss`, and `.rodata` so that the program can access it more quickly.





Target Settings Reference

EPPC Processor

This option only affects data that is actually defined in the current source file; it does not affect external declarations or any small data. The linker is normally aggressive in stripping unused data and functions from the C and C++ files in your project. However, the linker cannot strip any large data that has been pooled.

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 the floating-point constants in such a way so that the name contains the constant. This allows the linker to merge the floating-point constants automatically.

Use Common Section

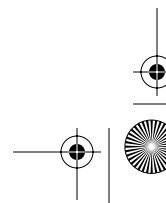
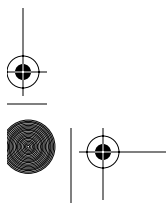
Check the **Use Common Section** checkbox to have the compiler place global uninitialized data in the common section. This section is similar to a FORTRAN Common Block. If the linker finds two or more variables with the same name and at least one of them is in a common section, those variables share 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.

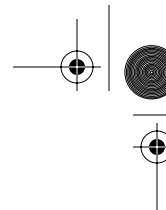
The `section` pragma provides fine control over which symbols the compiler includes in the common section.

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 those variables, otherwise you get unexpected results.

NOTE We recommend that you develop with **Use Common Section** checkbox cleared. When you have your program debugged, look at the data for especially large variables that are used in only one file. Change those variable names so that they are the same, and make sure that you initialize them before you use them. You can then turn the switch on.





Target Settings Reference

EPPC Processor

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** checkbox 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.

However, 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.

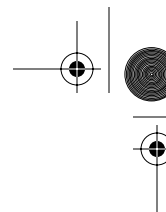
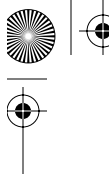
Consult the *Programming Environments Manual For 32-Bit Implementations of the PowerPC Architecture* for more information about **LMW** and **STMW** efficiency issues.

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.





Target Settings Reference

EPPC Disassembler

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** list box.)

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 into 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.

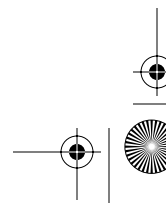
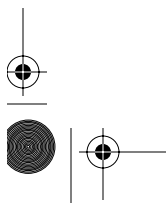
This checkbox corresponds to `#pragma profile`.

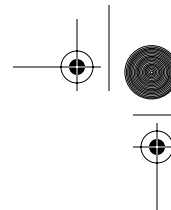
e500 Options

The settings in the **e500 Options** group apply to just the e500 family of processors. Because the CodeWarrior Development Studio for Embedded PowerPC ISA Systems product does not support this processor family, the options in the **e500 Options** group are always disabled.

EPPC Disassembler

Use the **EPPC Disassembler** settings panel (Figure 7.8) to control the information displayed when you choose **Project > Disassemble** in the IDE.

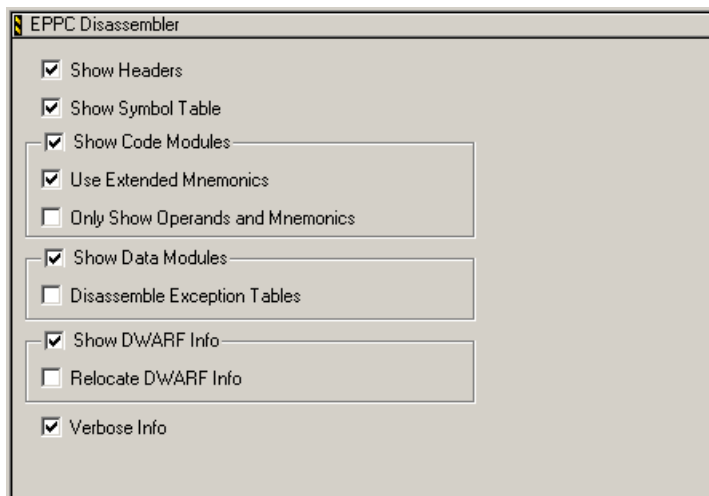




Target Settings Reference

EPPC Disassembler

Figure 7.8 EPPC Disassembler Settings Panel



See the “Compiling and Linking” chapter of the *CodeWarrior IDE User's Guide* for general information about the **Disassemble** command.

Show Headers

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

Show Symbol Table

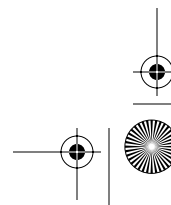
Check the **Show Symbol Table** checkbox to have the disassembler list the symbol table for the disassembled module.

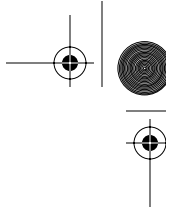
Show Code Modules

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.





Target Settings Reference

GNU Disassembler

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 provide 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.

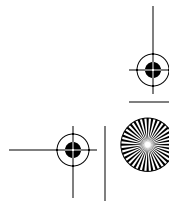
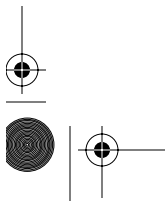
Checking this checkbox makes the **Relocate DWARF Info** checkbox available. The **Relocate DWARF Info** checkbox lets you relocate object and function addresses in the DWARF information.

Verbose Info

The **Verbose Info** checkbox tells 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 with an unstructured hex dump.

GNU Disassembler

The GNU Disassembler panel (Figure 7.9) lets you enter command line arguments to be passed to the disassembler.

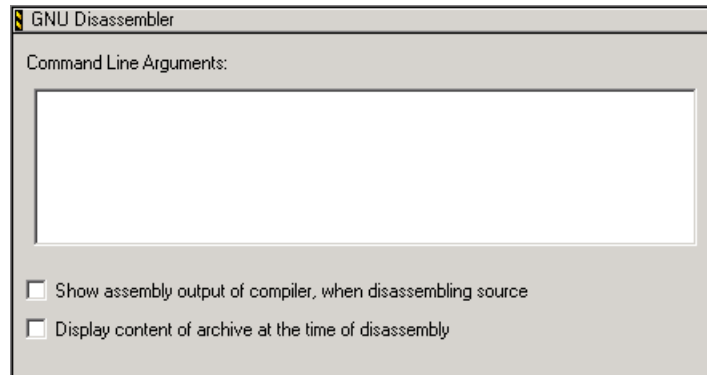




Target Settings Reference

GNU Compiler

Figure 7.9 GNU Disassembler



Command Line Arguments

Type your desired command line arguments into the **Command Line Arguments** text field.

Show Assembly Output of Compiler When Disassembling Source

Check this checkbox to have the compiler disassemble the source files. You can view the assembly output of the compiler as it disassembles the source.

If unchecked, the IDE uses the disassembler tool to disassemble the source files.

Display Content of Archive at Time of Disassembly

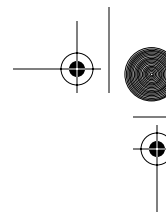
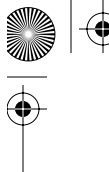
Check this checkbox to use the archiver tool to disassemble libraries. You can view the list of objects archived in the libraries.

If unchecked, the IDE uses the disassembler tool.

GNU Compiler

The GNU Compiler settings panel (Figure 7.10) lets you specify command line arguments, prefix files, and the format for generating debugger symbolics.





Target Settings Reference

EPPC Linker

Figure 7.10 GNU Compiler

GNU Compiler

Command Line Arguments:

-O0 -nostdinc

Prefix File:

☐ Use Custom Debug Format

Debug Option: -gdwarf-2 -g2

Command Line Arguments

Type your desired command line arguments into the **Command Line Arguments** text field.

Prefix File

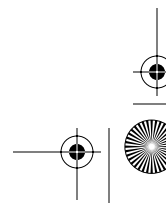
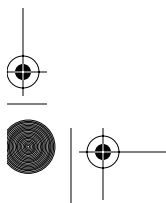
The Prefix File text box corresponds to the `-include` parameter. To include a prefix file before each source file in your project, enter the name of the prefix file in this text box.

Use Custom Debug Format

Check the Use Custom Debug Format checkbox to instruct the compiler to generate specific debugger symbolics. Uncheck this checkbox to generate debugger symbolics in the default format.

EPPC Linker

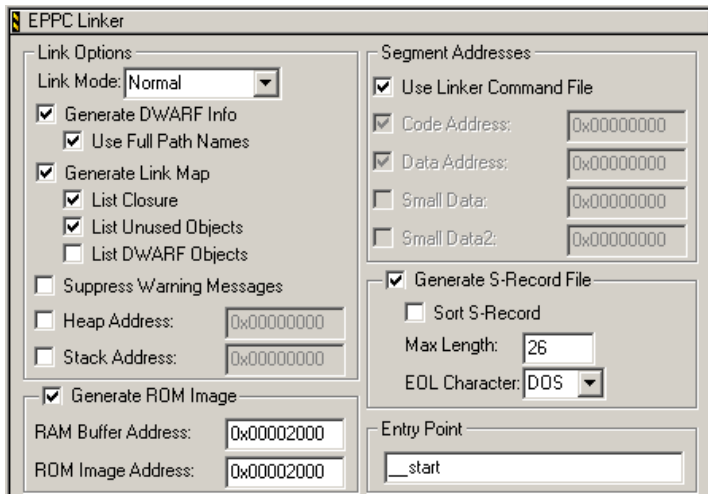
Use the **EPPC Linker** settings panel (Figure 7.11) to perform settings related to linking your object code into final form: executable file, library, or other type of code.



Target Settings Reference

EPPC Linker

Figure 7.11 EPPC Linker 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.

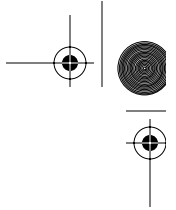
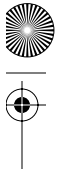
Use the **Link Mode** list box to select a link mode. The available link mode options are:

- **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 buffer, the buffers are flushed to the disk. This link mode is best suited for small projects.

Generate DWARF Info

Check the **Generate DWARF Info** checkbox to instruct the linker to generate debugging information. The debugger information is included within the linked ELF file. Checking this checkbox does not generate a separate file.

When you check the **Generate DWARF Info** checkbox, the **Use Full Path Names** checkbox becomes available. Use the **Use Full Path Names** checkbox to specify how the



Target Settings Reference

EPPC Linker

linker includes path information for source files. If the **Use Full Path Names** checkbox is checked, the linker includes path names within the linked ELF file (see the note that follows). If this checkbox is cleared, the linker uses only the file names.

NOTE To avoid problems while having the debugger locate your source code, clear the **Use Full Path Names** checkbox when building and debugging on different machines or platforms.

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

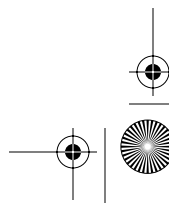
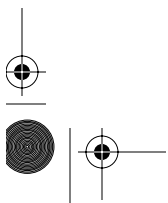
This checkbox is available only if you check the **Generate Link Map** checkbox. Check the **List Closure** checkbox to have all the functions called by the starting point of the program listed in the link map. See “Entry Point” on page 165 for details.

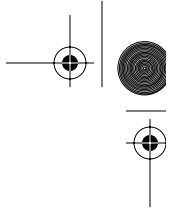
List Unused Objects

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 tell the linker to list all DWARF debugging





Target Settings Reference

EPPC Linker

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

The **Heap Address** text box specifies the location in memory where the program heap resides. 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 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 * 1024);
_heap_end = _stack_end;
_heap_addr = _heap_end - (Heap Size * 1024);
```

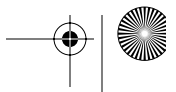
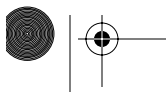
The MSL allocation routines do not require that you have a heap below the stack. You can set the heap address to any place in RAM that does not overlap with other sections. The MSL also allows you to have multiple memory pools, which can increase the total size of the heap.

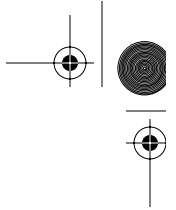
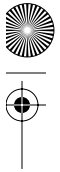
You can clear the **Heap Address** checkbox if your code does not make use of a heap. If you are using MSL, your program may implicitly use a heap.

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

The **Stack Address** text box specifies the location in memory where the program stack resides.





Target Settings Reference

EPPC Linker

If you wish 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 notation. The address you specify is the top of the stack and grows down the number of kilobytes you specify in the Stack Size text box in the EPPC Target panel. The address is aligned up to the nearest 8-byte boundary, if necessary. The possible addresses depend on your target hardware platform and how the 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`. This default address is suitable for the 8xx evaluation boards, but may not be suitable for boards with less RAM. For other boards, see the stationery projects for examples with suitable addresses.

NOTE Since the stack grows downward, it is common to place the stack as high 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

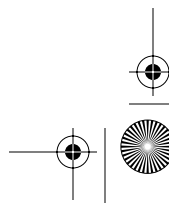
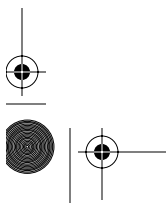
Check the Generate ROM Image box to instruct the linker to create a ROM image. A ROM image is a file that a flash programmer can write to flash ROM.

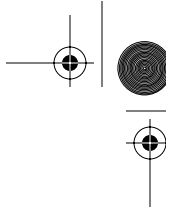
RAM Buffer Address

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`.





Target Settings Reference

EPPC Linker

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.

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

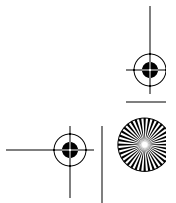
Check the **Use Linker Command File** checkbox to have the segment addresses specified in a linker command file. If the linker doesn't find the command file it expects, it issues an error message.

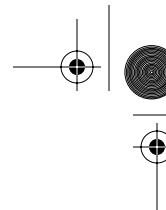
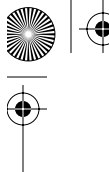
Leave this checkbox cleared if you want to specify the segment addresses directly in segment address text boxes: Code Address, Data Address, Small Data, and Small Data2.

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

The **Code Address** text box specifies the location in memory where the executable code resides.





Target Settings Reference

EPPC Linker

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 is suitable for the 8xx evaluation boards, but may not be suitable for boards with less RAM. For other boards, please 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

The **Data Address** text box specifies the location in memory where the global data of the program resides.

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 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** text box specifies the location in memory where the small data section resides.

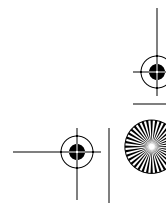
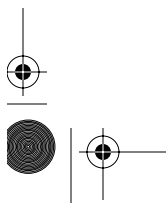
If you wish to specify a particular small data address, check the checkbox and type an address in the **Small Data** text box. You must specify the address in hexadecimal notation (using a format of 0x12345678). The possible addresses depend on your target hardware platform and how the memory is mapped. All types of data must reside in RAM.

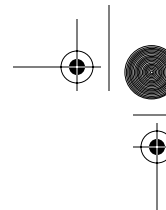
If you clear the checkbox, the linker calculates the small data address to begin immediately following the .data section.

Small Data2

The **Small Data2** text box specifies the location in memory where the small data2 section resides.

If you wish to specify a small data2 address, check the checkbox and type an address in the **Small Data2** text box. You must specify the address in hexadecimal notation. The





Target Settings Reference

EPPC Linker

possible addresses depend on your target hardware platform and how the memory is mapped. All types of data must reside in RAM.

If you clear the checkbox, the linker calculates the small data2 address to begin immediately following the `.sbss` section.

NOTE If the linker discovers that any of the sections, heap, or stack overlap, it issues a warning. You should fix the address problem and re-link your program.

Generate S-Record File

Check the **Generate S-Record File** checkbox to tell 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 the **Generate S-Record File** checkbox is checked. Check the **Sort S-Record** checkbox to have the generated S-Record files sorted in the ascending order of their addresses.

Max Length

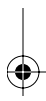
The **Max Length** text box specifies the maximum length of the S-record generated by the linker. This text box is available only if the **Generate S-Record File** checkbox is checked. The maximum value allowed for an S-Record length is 256 bytes.

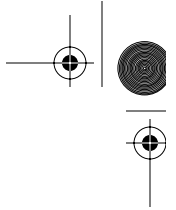
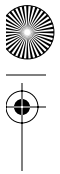
NOTE Most programs that load applications onto embedded systems have a maximum length allowed for the S-Records. The CodeWarrior debugger can handle S-Records of 256 bytes long. If you are using something other than the CodeWarrior debugger to load your embedded application, you need to find out what the maximum allowed length is.

EOL Character

Use the **EOL Character** list box to select the end-of-line character for the S-record file. This list box is available only if the **Generate S-Record File** checkbox is checked. The end of line characters are:

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





Target Settings Reference

GNU Post Linker

Entry Point

The **Entry Point** text box specifies the function that the linker uses first when the program launches. This is the starting point of the program.

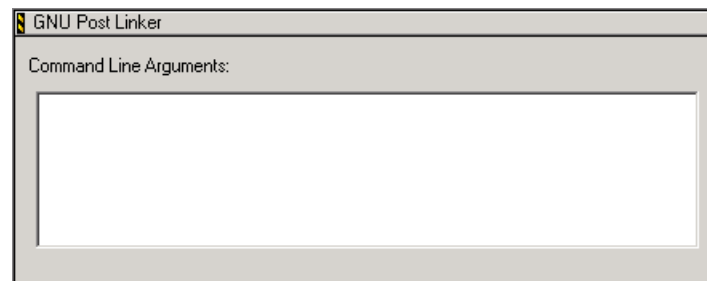
The default `__start` function is bootstrap or glue code that sets up the PowerPC EABI environment before your code executes. This function is in the `__start.c` file. The final task performed by `__start` is to call your `main()` function.

GNU Post Linker

The GNU Post Linker panel (Figure 7.12) lets you enter command line arguments to be passed to the post-linker specified in the GNU Tools panel. The default postlinker is `powerpc-eabispe-strip`.

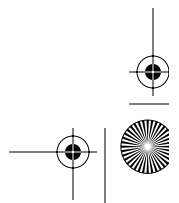
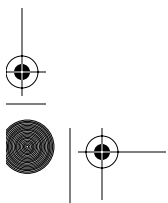
NOTE This settings panel is visible only when you select **EPPC GNU Post-linker** from the **Post-linker** menu in the **Target Settings** panel.

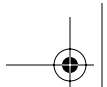
Figure 7.12 GNU Post Linker



GNU Linker

The GNU Linker panel (Figure 7.13) lets you enter command line arguments to be passed to the disassembler. The arguments that you specify are passed to the `gcc` command line for each file in your project as each file is linked.





Target Settings Reference

BatchRunner PreLinker

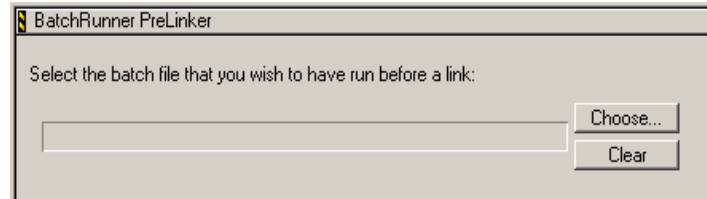
Figure 7.13 GNU Linker

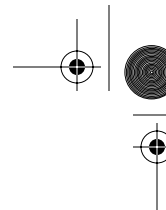


BatchRunner PreLinker

The **BatchRunner PreLinker** settings panel (Figure 7.14) lets you run a batch file before the IDE begins linking your project. Click the **Choose** button to select a batch file to run.

Figure 7.14 BatchRunner PreLinker

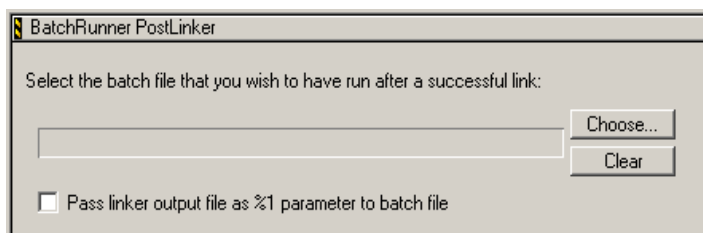




BatchRunner PostLinker

The **BatchRunner PostLinker** settings panel (Figure 7.15) lets you run a batch file after the IDE successfully builds your project.

Figure 7.15 BatchRunner Post-linker

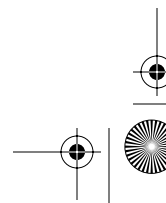
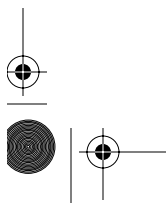


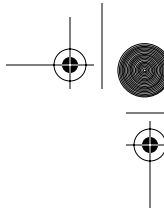
To specify the batch file to be run, click the **Choose** button.

To pass the name of the output file as a parameter to the batch file, check the **Pass Linker Output File as %1 Parameter to Batch File** checkbox.

GNU Environment

The GNU Environment settings panel (Figure 7.16) lets you specify the environment variables with which the IDE invokes the compiler, linker, assembler, and other build tool processes.

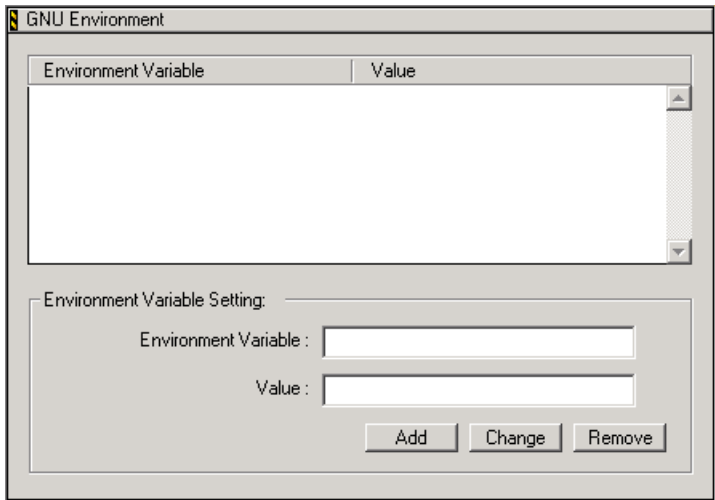




Target Settings Reference

GNU Tools

Figure 7.16 GNU Environment

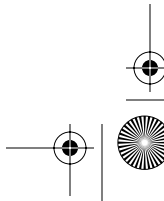


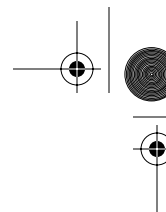
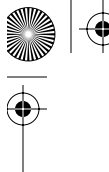
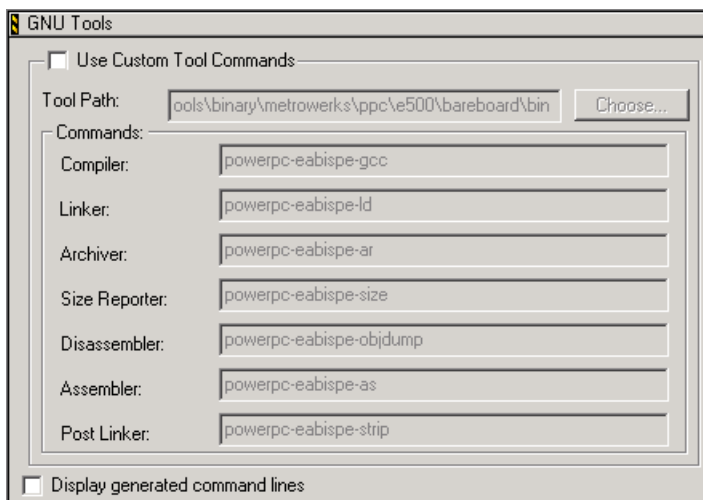
If you specify environment variables, the IDE uses the `cygwin1.dll` found in the `Cross_Tools` directory to invoke the GNU build tools.

If you already have Cygwin installed on your system, you can force the IDE to search for `cygwin1.dll` in the Windows system paths instead of the `Cross_Tools` directory. To do this, delete any environment variables defined here.

GNU Tools

The **GNU Tools** settings panel (Figure 7.17) lets you change the tool paths and executables that the IDE uses.




Figure 7.17 GNU Tools


Use Custom Tool Commands

Check the Use Custom Tool Commands checkbox to specify your own tool path or executables.

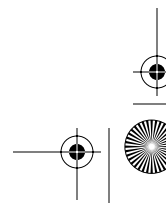
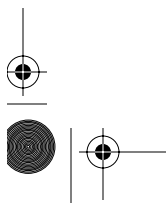
Tool Path

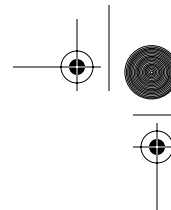
The Tool Path text box contains the path to the cross compiler tools on your system. Click the Choose button to specify a new location for these tools.

Commands

You may change the contents of the text boxes to specify other executables.

- Compiler
The compiler executable file
- Linker
The linker executable file
- Archiver
The executable file that builds static libraries
- Size Reporter





Target Settings Reference

Analyzer Connection

The executable file that reports the size of the code and data of your project files after they are compiled.

- Disassembler

The disassembler executable file.

- Assembler

The assembler executable file.

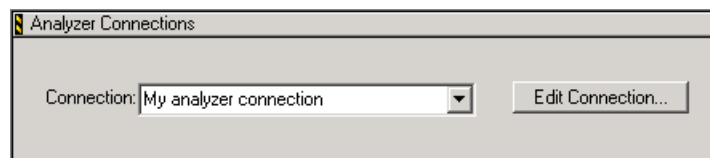
- Post Linker

The executable file for post linking operations. The default post linker executable, `powerpc-eabispe-strip`, strips debugger symbolics from the final output file.

Analyzer Connection

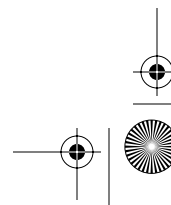
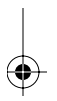
The Analyzer Connection settings panel (Figure 7.18) lets you select the remote connection to use for logic analyzer operations.

Figure 7.18 Analyzer Connection



Debugger PIC Settings

Use the **Debugger PIC Settings** panel (Figure 7.19) to specify an alternate address where you want your ELF image loaded on the target.

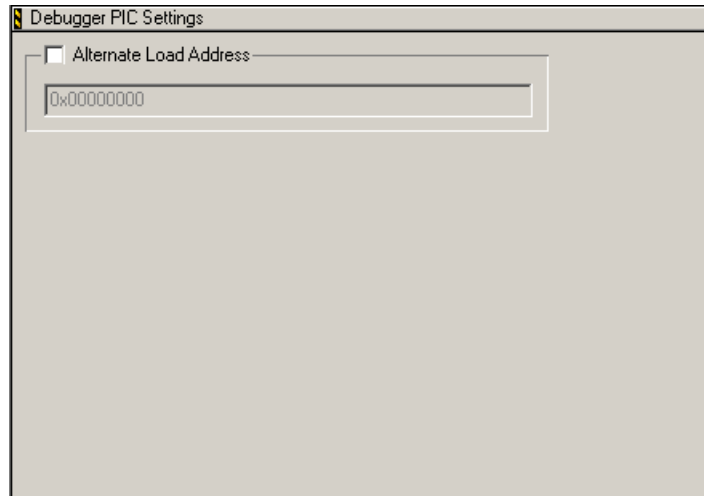




Target Settings Reference

EPPC Debugger Settings

Figure 7.19 Debugger PIC 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 7.20) to select the target processor.

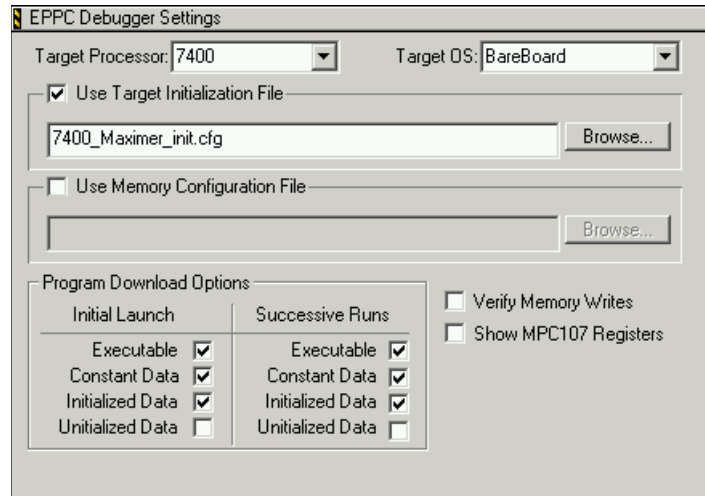




Target Settings Reference

EPPC Debugger Settings

Figure 7.20 EPPC Debugger Settings Panel



Target Processor

Use the **Target Processor** list box to specify the processor of your emulator or target board.

Target OS

Use the **Target OS** list box to select the operating system loaded on your board. If your board has no target operating system, select **Bareboard**.

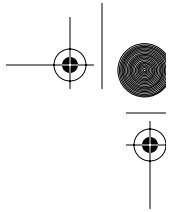
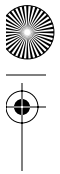
Use Target Initialization File

Check this checkbox if you want your project to use a target initialization file. Click **Browse** to locate and select the target initialization file. Prebuilt target initialization files are automatically selected for supported boards.

Sample target initialization files are in the BDM and Jtag directories. These directories are here:

`CWInstall\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 locate and select the memory configuration file.

Example memory configuration files are in this subdirectory of the CodeWarrior installation directory:

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 211.

Program Download Options

There are four section types listed in the **Program Download Options** section of this panel:

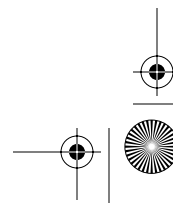
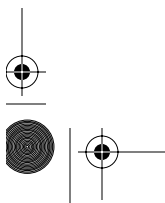
- Executable—the executable code and text sections of the program.
- Constant Data—the constant data sections of the program.
- Initialized Data—the initialized data sections of the program.
- Uninitialized Data—the uninitialized data sections of the program that are usually initialized by the runtime code included with CodeWarrior.

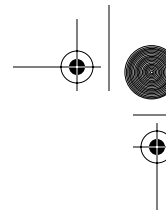
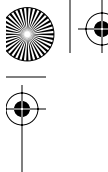
If one of these section types is selected, this means that it is to be downloaded when the program is debugged.

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.





Target Settings Reference

PQ3 Trace Buffer

PQ3 Trace Buffer

The PQ3 Trace Buffer panel (Figure 7.21) lets you configure the trace events you wish to capture during a PowerQUICC III debug session. The settings in this panel correspond to the appropriate bits in the trace configuration registers TBCR0 and TBCR1, the address register TBAR, the address mask register TBAMR, and the transaction mask register TBTMR.

NOTE For detailed information about using the trace buffer with the CodeWarrior IDE, refer to “Trace Buffer” on page 193.

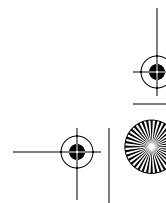
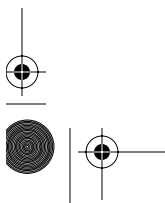
Figure 7.21 PQ3 Trace Buffer

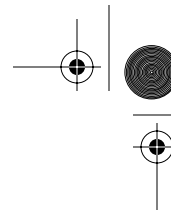
Enable Trace collection on Launch

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

Transaction Match Disable

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





Target Settings Reference

PQ3 Trace Buffer

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

Equal Context Enable

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

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

Not Equal Context Enable

Check this checkbox to record trace events only if the current context (the value of CCIDR register) is *not equal* to the programmed context (the value of PCIDR register).

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

Trace Only in TRACE Event

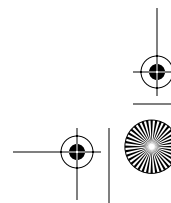
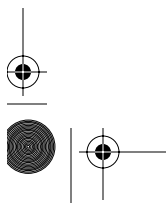
Check this checkbox to trace only cycles in which the monitor detects a trace event. Clear this checkbox to have the monitor trace all valid transactions.

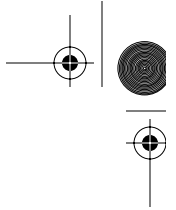
NOTE If the trace buffer is not properly configured to specify traceable events, the monitor traces every valid address.

Interface Selection

Select an item from this list box to specify the interface you want to trace. Selecting an interface activates tracing for all possible transaction types specific to that interface.

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





Target Settings Reference

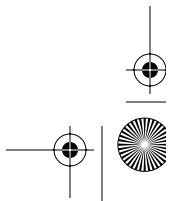
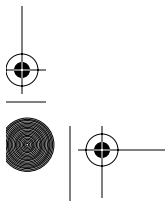
PQ3 Trace Buffer

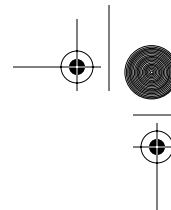
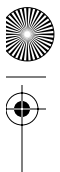
Start Condition

Select an item from this list box to specify the event that causes the monitor to start watching for traceable events. Table 7.4 describes the items in this menu.

Table 7.4 Start Conditions

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





Target Settings Reference
PQ3 Trace Buffer

Stop Condition

Select an item from this list box to specify the event that causes the monitor to stop watching for traceable events. Table 7.5 describes the items in this menu.

Table 7.5 Stop Conditions

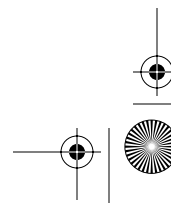
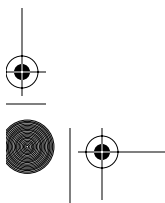
Item	Description
Buffer is full	Stop tracing once all 256 elements of the trace buffer are recorded.
Watchpoint event detected	Stop tracing once the monitor detects a watchpoint event.
Trace Buffer event detected	Stop tracing once the monitor detects a trace event.
Performance monitor overflow	Stop tracing when the performance monitor signals that an overflow occurred.
TRIG_IN transition	Stop tracing when the value of the TRIG_IN signal changes.
Context ID == programmed	Stop tracing when the current context ID is equal to the programmed context ID.
Context ID != programmed	Stop tracing when the current context ID is not equal to the programmed context ID.

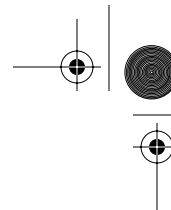
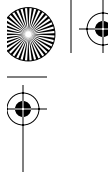
Source ID Enable

Check this checkbox and select a block or port from the list box to record only trace events whose transaction source ID matches the selected block or port. Table 7.6 lists the valid source IDs.

Table 7.6 Valid Source IDs

PCI	CPM
Boot Sequencer	DMA
Rapid IO	SAP
Local Processor Instruction Fetch	Ethernet X
Local Processor Data Fetch	





Target Settings Reference

Source Folder Mapping

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

Target ID Enable

Check this checkbox and select a block or port from the list box to record only trace events whose transaction target ID matches the selected block or port. Table 7.7 lists the valid target IDs.

Table 7.7 Valid Target IDs

PCI	Rapid IO
Local Bus	Local Space DDR
Config Space	DMA

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

Address Match Enable

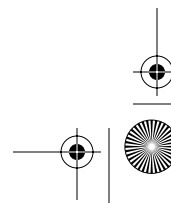
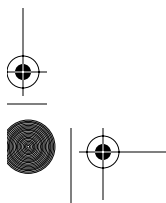
Check this checkbox to record only trace events whose trace address matches the transaction address. Enter an address in the **Trace Address** text box. Enter a mask in the **Trace Address Mask** text box. The monitor masks the trace address by excluding the address mask bits before comparison.

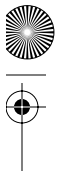
Source Folder Mapping

Use the **Source Folder Mapping** panel if you are debugging an executable file that was built in one place, but which is being debugged in 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 automatically creates entries in the **Source Folder Mapping** panel. The IDE creates these entries using the current folder information you provide during





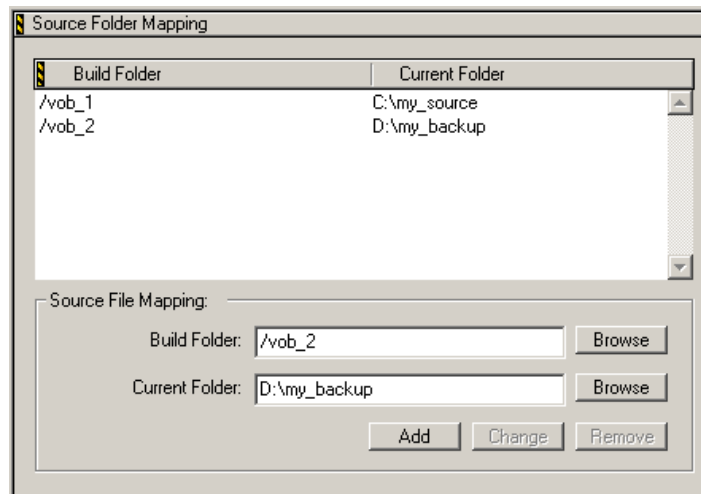
Target Settings Reference

Source Folder Mapping

the project creation process and the existing folder information in the ELF's debug information.

Figure 7.22 shows the **Source Folder Mapping** target settings panel.

Figure 7.22 Source Folder Mapping Panel



Build Folder

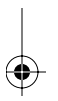
Use the **Build Folder** text box to enter the path that contained the executable's source files when this executable was originally built. Alternatively, click **Browse** to display a dialog box you can use to select the correct path.

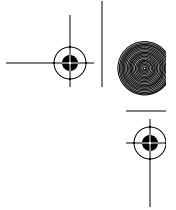
The supplied path can be the root of a source code tree. For example, if your source code files were in the directories

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

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

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.





Target Settings Reference

System Call Service Settings

Current Folder

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

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

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

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

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.

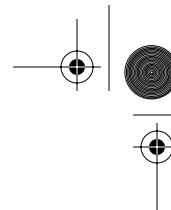
Remove

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 7.23) to activate support for system services and configure options for handling requests for system services.

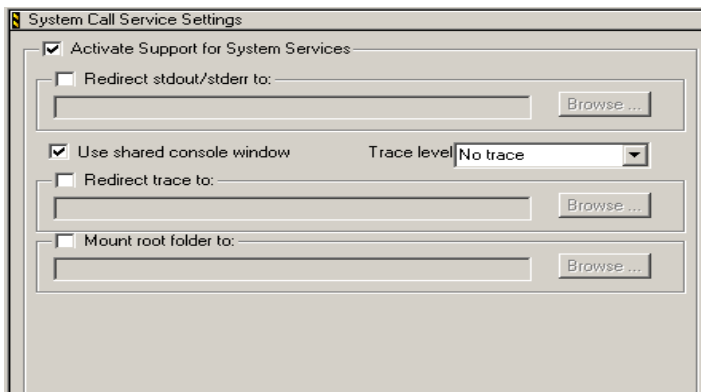




Target Settings Reference

System Call Service Settings

Figure 7.23 System Call Service 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 is useful when you do not have Board Support Package (BSP) for your target board.

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

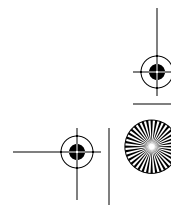
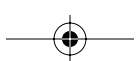
Check the **Activate Support for System Services** checkbox to enable support for system services. All the other options in the **System Call Service Setting** panel are available only if you check this checkbox.

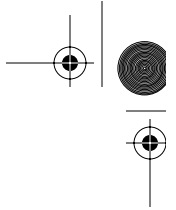
Redirect stdout/stderr to

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 specify the location of 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.





Target Settings Reference

System Call Service Settings

Trace Level

Use the **Trace level** list box to specify the system call trace level. The system call trace level options available are:

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

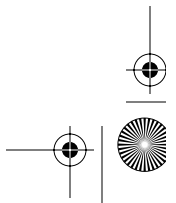
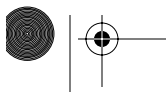
The place where the traced system service requests are displayed is determined by the **Redirect trace to** checkbox.

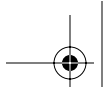
Redirect Trace to

The default 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 specify the location of the 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 specify the location of the root folder.





8

Working With Hardware Tools

This chapter explains how to use the CodeWarrior IDE hardware tools. Use these tools for board bring-up, test, and analysis.

This chapter contains these sections:

- Flash Programmer
- Hardware Diagnostics
- Logic Analyzer
- Trace Buffer

Flash Programmer

The CodeWarrior flash programmer can program the flash memory of the target board with code from any CodeWarrior IDE project or from any individual executable files. The CodeWarrior flash programmer provides features such as:

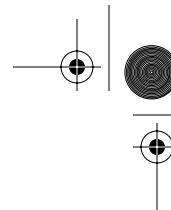
- Program
- Erase
- BlankCheck
- Verify
- Checksum

NOTE Certain flash programming features (such as view/modify, memory/register, and save memory contents to a file) are provided by the CodeWarrior debugger. Therefore, these features are not a part of the CodeWarrior flash programmer.

The CodeWarrior flash programmer lets you program the flash memory of any of the supported target boards from within the IDE. Further, CodeWarrior for EPPC includes a default flash configuration file for each supported target board. These files are in this directory:

`CWInstall\bin\Plugins\Support\Flash_Programmer\EPPC`





Working With Hardware Tools

Flash Programmer

To load a flash configuration file:

1. Select **Tools > Flash Programmer**

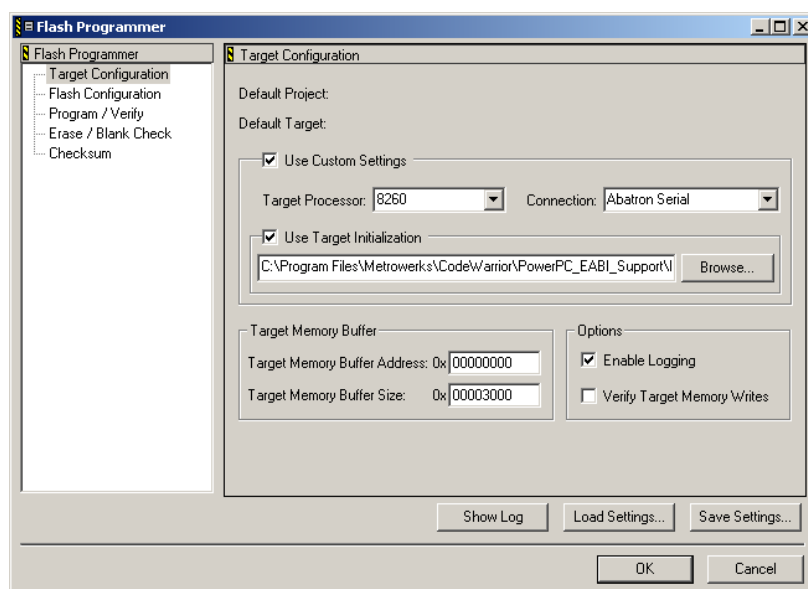
The **Flash Programmer** window appears.

NOTE The **Flash Programmer** window lets you define global setting for the flash programmer. These settings apply to each open project.

2. Select **Target Configuration** from the pane on the left side of the **Flash Programmer** window.

The Target Configuration preference panel appears on the right side of the **Flash Programmer** window. (See Figure 8.1).

Figure 8.1 Flash Programmer window



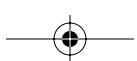
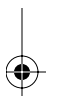
3. Click **Load Settings**

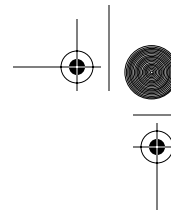
A standard open file dialog box appears.

4. Use the “open file” dialog box to select the flash programmer initialization file appropriate for your target board.

5. Click **Open**

The “open file” dialog box closes. The items in the Use Custom Settings group box are set using values from the selected initialization file.





Working With Hardware Tools

Hardware Diagnostics

6. Click **OK**

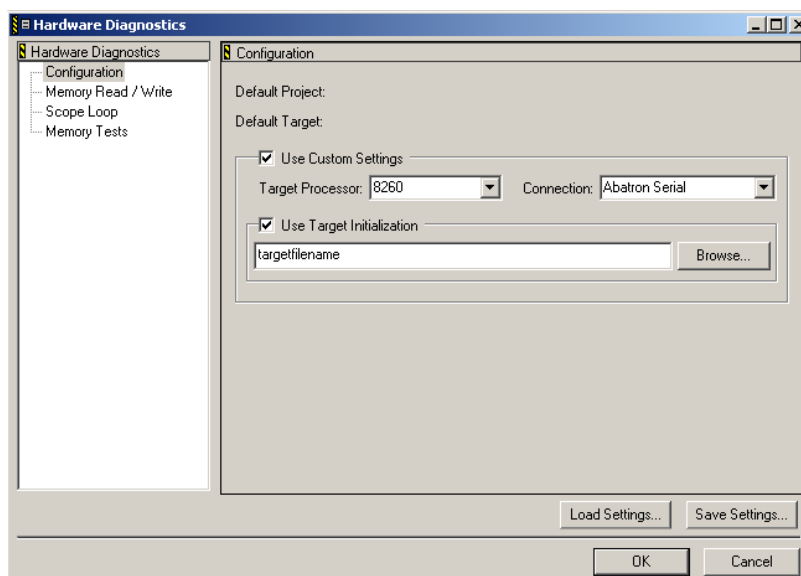
The **Flash Programmer** window saves your selections and closes.

NOTE See the *CodeWarrior IDE User's Guide* for documentation of the other preference panels available in the **Flash Programmer** window.

Hardware Diagnostics

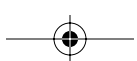
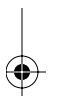
The **Hardware Diagnostics** window (Figure 8.2) lists global options for the hardware diagnostic tools. These preferences apply to every open project file. Select **Tools > Hardware Diagnostics** to display the **Hardware Diagnostics** window.

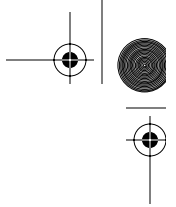
Figure 8.2 Hardware Diagnostics window



The left pane of the **Hardware Diagnostics** window shows a tree structure of panels. Click a panel name to display that panel in the right pane of the **Hardware Diagnostics** window.

Refer to the *CodeWarrior IDE User's Guide* for information on each panel in the **Hardware Diagnostics** window.





Working With Hardware Tools

Logic Analyzer

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

Logic Analyzer Menu

This topic explains the various commands in the **Logic Analyzer** menu. The **Logic Analyzer** menu is a sub-menu of the **Tools** menu. The **Logic Analyzer** menu is not available unless a logic analyzer connection has been established. See “Logic Analyzer Tutorial” on page 187 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)

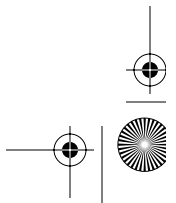
NOTE If your configuration file contains data besides the configuration information, the IDE may take a few minutes to load the configuration file.

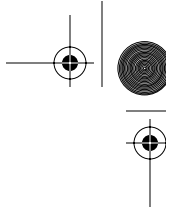
- Retrieve all the label data for the columns in the Trace Window

To connect to the logic analyzer, the IDE uses the preferences you specify for the analyzer connection. See “Logic Analyzer Tutorial” on page 187 for details.

Arm

The **Arm** command is available only if the IDE is connected to the logic analyzer.





Working With Hardware Tools Logic Analyzer

Select **Tools > Logic Analyzer > Arm** to instruct the logic analyzer to start collecting target cycles.

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

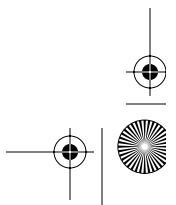
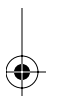
Select **Tools > Logic Analyzer > Disconnect** to disconnect the system from the analyzer device. The IDE clears all the data in the Trace window.

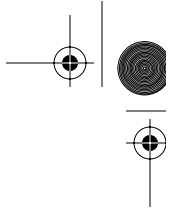
Logic Analyzer Tutorial

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





Working With Hardware Tools

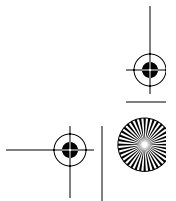
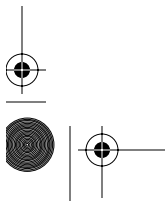
Logic Analyzer

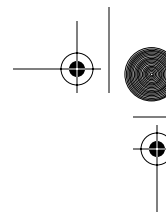
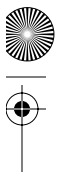
NOTE The 16717A boards in slot A and C are configured as slaves to the master board in Slot B.

- 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 8.3) appears.





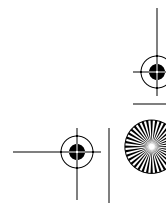
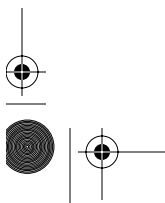
Working With Hardware Tools Logic Analyzer

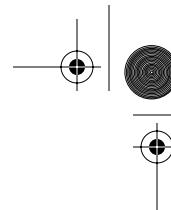
Figure 8.3 Logic Analyzer Connection Preferences

- 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,





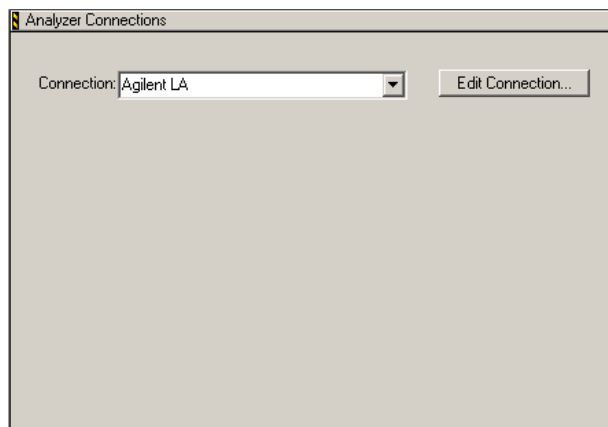
Working With Hardware Tools

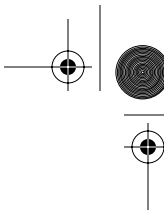
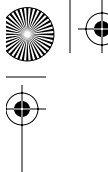
Logic Analyzer

`/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 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 closes the **IDE Preferences** window.
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 8.4) appears.

Figure 8.4 Analyzer Connections Panel





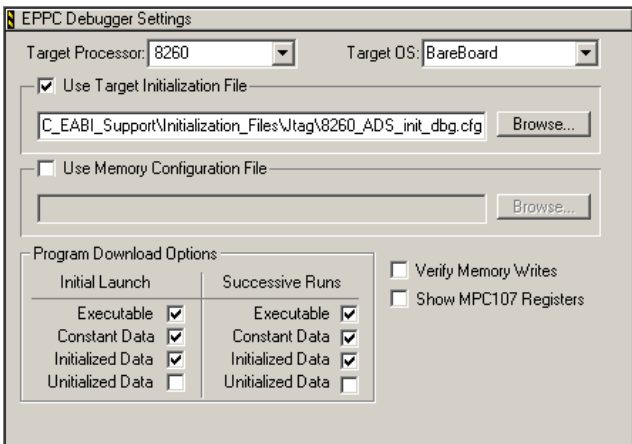
Working With Hardware Tools
Logic Analyzer

- c. Select the analyzer connection name from the **Connection** list.

NOTE Each build target supports only one connection to a logic analyzer. If you want your project to have more logic analyzer connections, create a build target for each additional connection.

- 4. Configure debugger settings of your project.
 - a. Select **EPPC Debugger Settings** from the **Target Setting Panels** list.The **EPPC Debugger Settings** panel (Figure 8.5) appears.

Figure 8.5 EPPC Debugger Panel



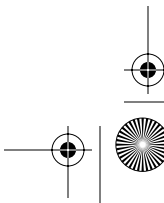
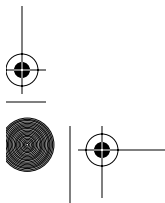
- b. Check the **Use Target Initialization File** checkbox.
- c. In the text box associated with the **Use Target Initialization File** checkbox, enter the name of the target initialization file required by your target board. Alternatively, click **Browse** to display a dialog box you can use to select the required file.

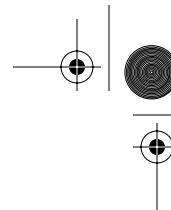
Table A.1 on page 197 lists the generic initialization file for each supported target board. These file are on this path:

CodeWarrior Install Dir\
PowerPC_EABI_Support\Initialization_Files

- d. Click **OK**
- The system saves the target settings.

- 5. Connect the analyzer pods.
- Table 8.1 shows the Agilent analyzer pod connection scheme.





Working With Hardware Tools

Logic Analyzer

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 8.1 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.

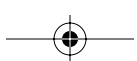
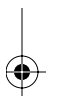
NOTE While the analyzer is armed the debugger periodically queries the analyzer for its Run status.

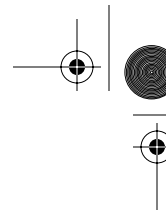
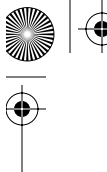
- d. Select **Tools > Logic Analyzer > Disarm**

The system instructs the logic analyzer to stop collecting target cycles.

- e. Select **Tools > Logic Analyzer > Update Data**

The system retrieves the trace data from the analyzer's buffer.





Working With Hardware Tools

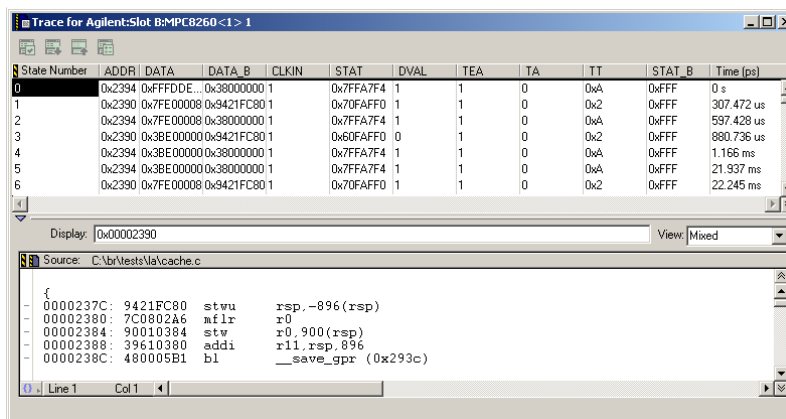
Trace Buffer

f. Select **Data > Trace View**

The trace window (Figure 8.6) appears.

The trace window displays the data collected.

Figure 8.6 Trace Window



g. Select **Tools > Logic Analyzer > Disconnect**

The system disconnects from the logic analyzer. The IDE erases the contents of the Trace window.

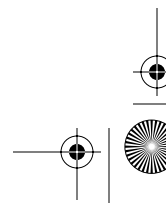
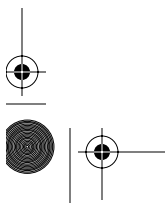
Trace Buffer

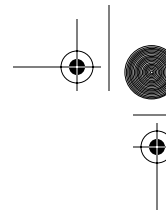
The trace buffer is a 256- x 64-bit buffer that holds information about the internal processing of transactions to the processing interfaces. This visibility into internal device operation is useful for debugging application software through inverse assembly and reconstruction of the fetch stream. On PowerQUICC III devices, trace buffer support is implemented in the hardware, thus the trace buffer does not affect application performance.

You can configure the trace buffer to trace the dispatch bus from any of these interfaces:

- e500 coherency module (ECM)
- Outbound host interface to the RapidIO controller
- Outbound host interface to the PCI controller
- Host interface to the DDR controller.

NOTE You can trace only one interface at a time.





Working With Hardware Tools

Trace Buffer

As transactions come into the ECM, the ECM arbitrates common resources and dispatches the transactions to target ports. You can capture information such as transaction types, source ID, and other attributes for any of the selected interfaces.

Trace events hold this information:

- Transaction type — the type of transaction (for example, write with local processor snoop, or read with unlock)
- Source of the transaction — the source block or port of the transaction (for example, the local processor for data fetches).
- Target of the transaction — the target block or port of the transaction (typically slave ports in a transaction, such as local memory)
- The size of the transaction, in bytes

NOTE The transaction target is meaningful only for monitoring the ECM dispatch bus.

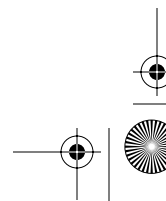
You can configure the trace buffer to record all transactions, or to record only:

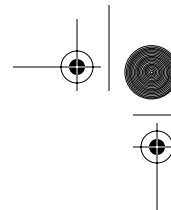
- transactions with a specified source ID
- transactions with a specified target ID
- transactions whose address matches a specified masked address
- transactions whose current context ID (the value of CCIDR register) matches or does not match the programmed context ID (the value of PCIDR register).

You can combine any of these conditions.

You use the **PQ3 Trace Buffer** target settings panel to configure the trace buffer for each build target in a CodeWarrior project.

For example, Figure 8.7 shows the settings panel configured to record only transactions dispatched by the ECM, with the source ID “Local Processor Data Fetch”, the target ID “Local Space DDR”, and with addresses in the range 0x00010000 through 0x0001FFFF.

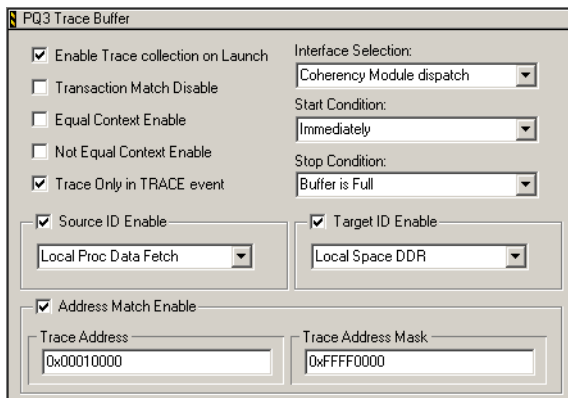




Working With Hardware Tools

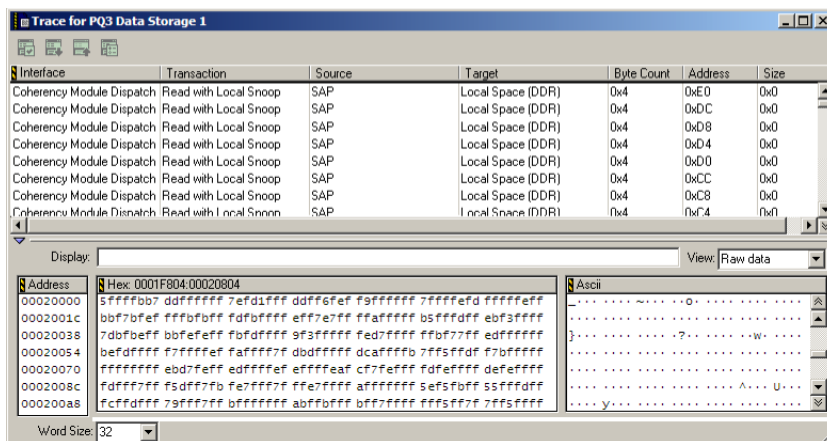
Trace Buffer

Figure 8.7 Example Trace Buffer Configuration



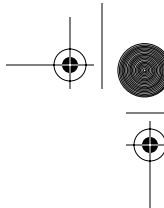
To view recorded trace events during a CodeWarrior debug session, select **Data > View Trace** from the CodeWarrior menubar. Figure 8.8 shows a sample trace event log generated by this configuration.

Figure 8.8 Trace Events Received

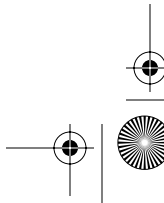


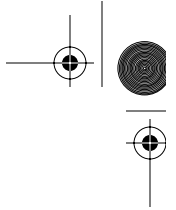
For a detailed description of the PQ3 Trace Buffer target settings panel, see “PQ3 Trace Buffer” on page 174.

For more details about the PowerQUICC III trace buffer, refer to the *PowerQUICC III Reference Manual*.



Working With Hardware Tools
Trace Buffer





A

Debug Initialization Files

You use debug initialization files to initialize the target board before the debugger downloads the code, ensuring that the target board's memory is initialized properly.

This appendix contains these sections:

- Using Debug Initialization Files
- Debug Initialization File Commands

Using Debug Initialization Files

A debug initialization file is a command file processed and executed each time the debugger is invoked. 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 need to be set for debugging purposes are set correctly. You specify whether or not to use an initialization file and which file to use in the EPPC Target settings panel.

NOTE You do not need to use an initialization file if you debug with MetroTRK.

We provide samples of initialization files for supported evaluation boards. The sample files can be found in these locations in the CodeWarrior installation directory:

PowerPC_EABI_Support\Initialization_Files\BDM\

PowerPC_EABI_Support\Initialization_Files\JTAG\

Table A.1 lists the sample initialization file for each supported target board.

Table A.1 Evaluation Board Initialization Files

Board	Configuration File Location
56x chip support	\BDM\56X_CHIP_init.cfg
AMC 603ei Eval	\Jtag\603ei_AMCEval_init.cfg
Analogue & Micro Rattler MPC8250VR	\JTAG\8250_AM_Rattler_init.cfg
AXIOM 555	\BDM\555_AXIOM_init.cfg



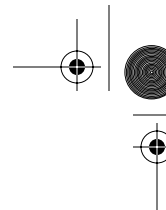
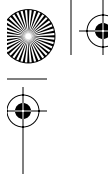
Debug Initialization Files

Using Debug Initialization Files

Table A.1 Evaluation Board Initialization Files (*continued*)

Board	Configuration File Location
AXIOM 555 with shadow flash	\BDM\555_AXIOM_vfSH_flash_init.cfg
AXIOM 565	\BDM\565_AXIOM_init.cfg
AXIOM 565 with shadow flash	\BDM\565_AXIOM_vfSH_flash_init.cfg
Embedded Planet EP8260 Rev. H	\Jtag\8260_EP_init.cfg
Embedded Planet EP8260 Rev. M	\Jtag\8260_EP_init_1.cfg
Embedded Planet RPX Lite rev.BW @50MHz	\BDM\8xx_RPX_BW_50M_init.cfg
Embedded Planet RPX Lite rev.DW @50MHz	\BDM\8xx_RPX_DW_50M_init.cfg
Embedded Planet SBC 857 M EP862-1.1	\BDM\862_EP_init.cfg
Embedded Planet SBC 852 M EP852-1.1	\BDM\852_EP_init.cfg
Embedded Planet SBC 866 H EP862-1.2	\BDM\862_EP_MPC866_init.cfg
Motorola 555 ETAS	\BDM\555_ETAS_init.cfg
Motorola 5100 IceCube EVB	\Jtag\5100_Icecube_init.cfg
Motorola 5200 Lite	\Jtag\5200_Icecube_init.cfg
Motorola 603 Excimer	\Jtag\603_Excimer_init.cfg
Motorola 7400 Maximer	\Jtag\7400_Maximer_init.cfg
Motorola DUET 885 ADS	\BDM\885ads_init.cfg
Motorola MPC 857 IAD	\BDM\857DSL_IAD_init.cfg
Motorola MPC 862 ADS	\BDM\862_ADS_init.cfg
Motorola MPC 862 ADS New Mode (EDO DRAM not used)	\BDM\862_ADS_NewMode_init.cfg
Motorola MPC 866 ADS	\BDM\866_ADS_init.cfg





Debug Initialization Files

Debug Initialization File Commands

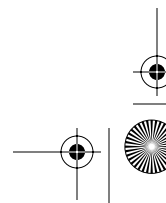
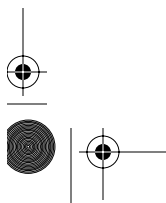
Table A.1 Evaluation Board Initialization Files (*continued*)

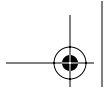
Board	Configuration File Location
Motorola MPC 8xx FADS	\BDM\8xx_FADS_init.cfg
Motorola MPC 8xx FADS eng rev	\BDM\8xx_FADS_Eng_init.cfg
Motorola MPC 8250 ADS-PCI	\Jtag\8250_ADS_init_dbg.cfg
Motorola MPC 8260 ADS	\Jtag\8260_ADS_init_dbg.cfg
Motorola MPC 8264 ADS-PCI	\Jtag\8264_ADS_init_dbg.cfg
Motorola MPC 8265 ADS-PCI	\Jtag\8265_ADS_init_dbg.cfg
Motorola MPC 8266 ADS-PCI	\Jtag\8266_ADS_init_dbg.cfg
Motorola MPC 8266 ADS-PCI-AI	\Jtag\8266AI_ADS_init_FLASH.cfg
Motorola MPC 8272 ADS (PQ27e)	\Jtag\PQ27e_init.cfg
Motorola MPC 82xx PQ2 FADS-ZU at 66 MHz	\Jtag\PQ2_FADS_ZU_Local_init_66MHz.cfg \Jtag\PQ2_FADS_ZU_PCI_init_66MHz.cfg
Motorola MPC 82xx PQ2 FADS-ZU at 100 MHz	\Jtag\PQ2_FADS_ZU_Local_init_100MHz.cfg \Jtag\PQ2_FADS_ZU_PCI_init_100MHz.cfg
Motorola Sandpoint 74xx	\Jtag\74xx_Sandpoint_init.cfg
Motorola Sandpoint 745x	\Jtag\7450_Sandpoint_init.cfg
Motorola Sandpoint 7457	\Jtag\7457_Sandpoint_init.cfg
Motorola Sandpoint 7xx, 824x	\Jtag\Sandpoint_init.cfg

Debug Initialization File Commands

In general, the syntax of debug initialization file commands follows these rules:

- white spaces and tabs are ignored
- character case is ignored
- unless otherwise notes, values may be specified in hexadecimal, octal, or decimal:





Debug Initialization Files

Debug Initialization File Commands

- hexadecimal values are preceded by 0x (for example, 0xDEADBEEF)
- octal values are preceded by 0 (for example, 01234567)
- decimal values start with a non-zero numeric character (for example, 1234)
- comments start with a semicolon (;) or pound sign (#), and continue to the end of the line

alternatePC

Sets the program counter (PC) register to the specified value. You can use this command when the program entry point differs from ELF image entry point.

Syntax

```
alternatePC address
```

Arguments

address

the address to which the command should set the program counter register

Example

To set the program counter register to the address 0xc28737a3:

```
alternatePC 0xc28737a3
```

ANDmmr

Performs an AND operation with the specified value and the specified MMR register, and writes the result back into the register.

Syntax

```
ANDmmr regName value
```

Arguments

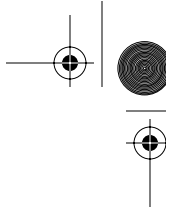
regName

the name of the register

value

the byte, word, or long value to write to the specified register





Debug Initialization Files

Debug Initialization File Commands

Example

To perform an AND operation with the value 0x00002000 on the ACFG register:

```
ANDmmr ACFG 0x00002000
```

IncorMMR

Performs an OR operation with the specified value and the specified MMR register, and writes the result back into the register.

Syntax

```
incorMMR regName value
```

Arguments

regName

the name of the register

value

the byte, word, or long value to write to the specified register

Example

To perform an OR operation with the value 0x00002000 on the ACFG register:

```
incorMMR ACFG 0x00002000
```

reset

Resets the target processor.

Syntax

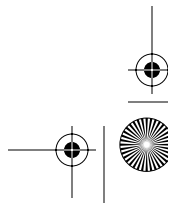
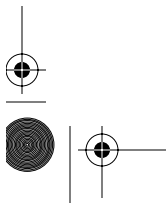
```
reset code
```

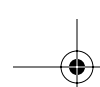
Arguments

code

a numeric value in the range 0 through 1 that determines what the debugger does after resetting the target

Specify one of the values in Table A.2.





Debug Initialization Files

Debug Initialization File Commands

Table A.2 Post Reset Actions

Value	Description
0	reset the target processor, then run
1	reset the target processor, then stop

run

Starts program execution at the current program counter (PC) address.

Syntax

run

setMMRBaseAddr

This command tells the debugger the base address of memory mapped registers, allowing you to send any `writemmr` commands from the debug initialization file, as well as read the memory mapped registers for the register views.

NOTE Starting with CodeWarrior Development Studio, PowerQUICC III Edition, version 1.1, this command is obsolete for PowerQUICC III target processors.

NOTE The debugger requires the base address of the memory mapped registers for 825x/826x hardware only. Therefore, this command must be in all debug initialization files for the 825x/826x processors.

NOTE For target board with 5100/5200 processors, use the `setMBARBaseAddr` command, which is functionally equivalent to this command.

Syntax

setMMRBaseAddr *baseAddress*





Debug Initialization Files

Debug Initialization File Commands

Arguments

`baseAddress`

the base address (in hexadecimal) of the memory mapped registers

Example

To set the base address for memory mapped registers to 0x0f00000:

```
setMMRBaseAddr 0x0f00000
```

sleep

Causes the processor to wait the specified number of milliseconds before continuing to the next command.

Syntax

```
sleep milliseconds
```

Arguments

`milliseconds`

the number of milliseconds (in decimal) to pause the processor

Example

To pause execution for 10 milliseconds:

```
sleep 10
```

stop

Stops program execution and halts the target processor.

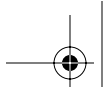
Syntax

```
stop
```

writemem.b

Writes a byte (8 bits) of data to the specified memory location.





Debug Initialization Files

Debug Initialization File Commands

Syntax

```
writemem.b address value
```

Arguments

address

the memory address to modify (in hexadecimal, octal, or decimal)

value

the 8-bit value (in hexadecimal, octal, or decimal) to write to the memory address

Example

To write the byte value 0xFF to memory location 0x0001FF00:

```
writemem.b 0x0001FF00 0xFF
```

writemem.w

This command writes a word (16 bytes) of data to the specified memory location.

Syntax

```
writemem.w address value
```

Arguments

address

the memory address to modify (in hexadecimal, octal, or decimal)

value

the 16-bit value (in hexadecimal, octal, or decimal) to write to the memory address

Example

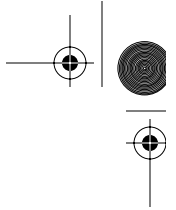
To write the word value 0x1234 to memory location 0x0001FF00:

```
writemem.w 0x0001FF00 0x1234
```

writemem.l

Writes a long integer (32 bytes) of data to the specified memory location.





Debug Initialization Files

Debug Initialization File Commands

Syntax

```
writemem.l address value
```

Arguments

address

the memory address to modify (in hexadecimal, octal, or decimal)

value

the 32-bit value (in hexadecimal, octal, or decimal) to write to the memory address

Example

To write the long integer value 0x12345678 to memory location 0x0001FF00:

```
writemem.w 0x0001FF00 0x12345678
```

writemem.r

Writes a value to the specified register.

Syntax

```
writemem.r regName value
```

Arguments

regName

the name of the register

value

the value (in hexadecimal, octal, or decimal) to write to the register

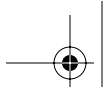
Example

To write the value 0xffffffffc3 to the SYPCR register:

```
writemem.r SYPCR 0xffffffffc3
```

writemmr

Writes a value to the specified MMR (Memory Mapped Register).



Debug Initialization Files

Debug Initialization File Commands

Syntax

```
writemmr regName value
```

Arguments

regName

the name of the register

NOTE In general, all memory mapped register names for the supported processor are accepted by this command. If any registers are not accepted by this command, however, you may instead use writemem commands to modify the register.

value

the value (in hexadecimal, octal, or decimal) to write to the register

Example

To write the value 0xffffffffc3 to the SYPCR register:

```
writemmr SYPCR 0xffffffffc3
```

To write the value 0x0001 to the RMR register:

```
writemmr RMR 0x0001
```

To write the value 0x3200 to the MPTPR register:

```
writemmr MPTPR 0x3200
```

writereg

Writes the specified data to a register.

Syntax

```
writereg regName value
```

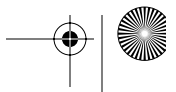
Parameters

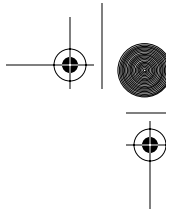
regName

the name of the register

value

the value (in hexadecimal, octal, or decimal) to write to the register





Debug Initialization Files

Debug Initialization File Commands

Example

To write the value 0x00001002 to the MSR register:

```
writereg MSR 0x00001002
```

writereg128

Writes specified values to a TLB register.

Syntax

```
writereg128 register value1 value2 value3 value4
```

Arguments

register

the name (or number) of the TLB register you want to modify

TIP Valid TLB0 register names range from L2MMU_TLB0 through L2MMU_TLB255. These registers are sequentially numbered 5000 through 5255.

TIP Valid TLB1 register names range from L2MMU_CAM0 through L2MMU_CAM15. These registers are sequentially numbered 5300 through 5315.

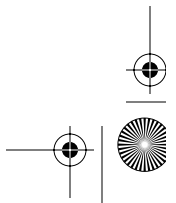
value1, value2, value3, value4

four 32-bit values (each in hexadecimal format), making up the 128-bit value that you want to write to the TLB register

writespr

Writes the specified value to the specified SPR register.

NOTE This command is similar in function, to a `writereg SPRxxx` command, except that you may specify the SPR number in hexadecimal, octal, or decimal.





Debug Initialization Files

Debug Initialization File Commands

Syntax

```
writespr regNumber value
```

Arguments

regNumber

the SPR register number (in hexadecimal, octal, or decimal)

value

the value (in hexadecimal, octal, or decimal) to write to the register

Example

To write the value 0x0220000 to SPR register 638:

```
writespr 638 0x02200000
```

writeupma

Maps the user-programmable machine (UPM) register, defining characteristics of the memory array.

Syntax

```
writeupma offset ramWord
```

Arguments

offset

a value in the rang of 0 through 3F representing the UPM transaction type (for details, refer to the “UPM Transaction Type” table in the “Memory Controller” section of the hardware manual)

ramWord

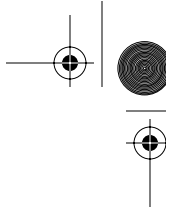
the UPM RAM word for the specified offset

Example

To map the 0x08 transaction type to the UPM registers with RAM word 0xffffcc24:

```
writeupma 0x08 0xffffcc24
```





Debug Initialization Files

Debug Initialization File Commands

writeupmb

Maps the user-programmable machine (UPM) register, defining characteristics of the memory array.

Syntax

```
writeupmb offset ramWord
```

Arguments

offset

a value in the range of 0 through 3F representing the UPM transaction type (for details, refer to the “UPM Transaction Type” table in the “Memory Controller” section of the hardware manual)

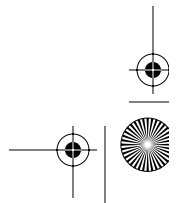
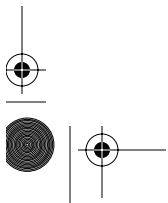
ramWord

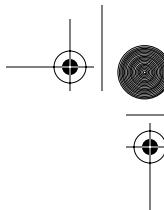
the UPM RAM word for the specified offset

Example

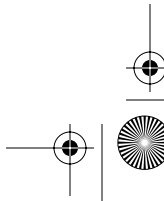
To map the 0x08 transaction type to the UPM registers with RAM word 0xffffcc24:

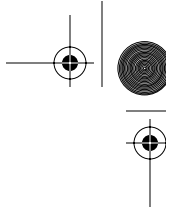
```
writeupmb 0x08 0xffffcc24
```





Debug Initialization Files
Debug Initialization File Commands





B

Memory Configuration Files

A memory configuration file contains commands that define the accessible areas of memory for your specific board.

This appendix consists of these topics:

- Command Syntax
- Memory Configuration File Commands

Command Syntax

In general, the syntax of memory configuration file commands follows these rules:

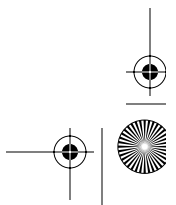
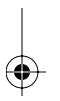
- white spaces and tabs are ignored
- character case is ignored
- unless otherwise notes, values may be specified in hexadecimal, octal, or decimal:
 - hexadecimal values are preceded by 0x (for example, 0xDEADBEEF)
 - octal values are preceded by 0 (for example, 01234567)
 - decimal values start with a non-zero numeric character (for example, 1234)
- comments start with standard C and C++ comment characters, and continue to the end of the line

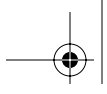
Memory Configuration File Commands

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.

We provide a sample configuration file in this folder within the CodeWarrior install directory:

```
CodeWarrior\PowerPC_EABI_Support\Config\
Memory\8560ADS_Example.mem
```





Memory Configuration Files

Memory Configuration File Commands

range

This command sets debugger access to a block of memory.

Syntax

`range loAddress hiAddress size access`

Arguments

loAddress

the starting address of the memory range

hiAddress

the ending address of the memory range

size

the size, in bytes, the debug monitor or emulator uses for memory accesses

access

controls what type of access the debugger has to the memory block — supply one of: Read, Write, or ReadWrite

Examples

To set memory locations 0xFF000000 through 0xFF0000FF to read-only with a size of 4 bytes:

```
range 0xFF000000 0xFF0000FF 4 Read
```

To set memory locations 0xFF0001000 through 0xFF0001FF to write-only with a size of 2 bytes:

```
range 0xFF000100 0xFF0001FF 2 Write
```

To set memory locations 0xFF0002000 through 0xFFFFFFFF to read and write with a size of 1 byte:

```
range 0xFF000200 0xFFFFFFFF 1 ReadWrite
```

reserved

This command allows you to specify a reserved range of memory. If the debugger attempts to read reserved memory, the resulting buffer is filled with the reserved character. If the debugger attempts to write to reserved memory, no write takes place.





Memory Configuration Files

Memory Configuration File Commands

NOTE Refer to “reservedchar” on page 213 for information showing how to set the reserved character.

Syntax

```
reserved loAddress hiAddress
```

Arguments

loAddress

the starting address of the memory range

hiAddress

the ending address of the memory range

Examples

To reserve memory starting at 0xFF000024 and ending at 0xFF00002F:

```
reserved 0xFF000024 0xFF00002F
```

reservedchar

This command sets the reserved character for the memory configuration file. When the debugger attempts to read a reserved or invalid memory location, it fills the buffer with this character.

Syntax

```
reservedchar rChar
```

Arguments

rChar

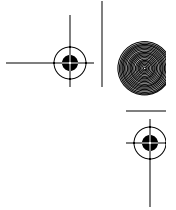
the one-byte character the debugger uses when it accesses reserved or invalid memory

Example

To set the reserved character to “∞”:

```
reservedchar 0xB0
```





Memory Configuration Files

Memory Configuration File Commands

translate

This command allows you to configure how the debugger performs virtual-to-physical memory address translations. You typically use address translations to debug programs that use a memory management unit (MMU) that performs block address translations.

Syntax

```
translate virtualAddress physicalAddress numBytes
```

Arguments

virtualAddress

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

physicalAddress

the address of the first byte of the physical address range to which the debugger translates virtual addresses

numBytes

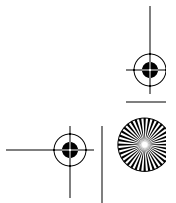
the size (in bytes) of the address range to translate

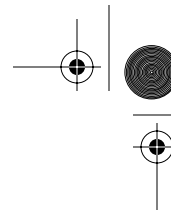
Example

The following `translate` command:

- defines a one-megabyte address range (0x100000 bytes is one megabyte)
- instructs the debugger to convert a virtual address in the range 0xC0000000 to 0xC0100000 to the corresponding physical address in the range 0x00000000 to 0x00100000

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





C

Tested Jumper and Dipswitch Settings

This appendix provides tested jumper and dipswitch settings for a number of supported target boards.

Cogent CMA102 with CMA 278 Daughtercard

Table C.1 lists the tested jumper settings for the Cogent CMA102 target board when used with a CMA 278 daughtercard. Table C.2 lists the tested dipswitch setting for the Cogent CMA278 daughtercard.

NOTE The CMA 278 daughtercard uses a 603/740 processor.

Table C.1 Cogent CMA102 Jumper Settings

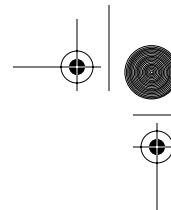
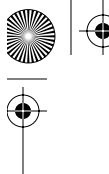
Jumper Locations	Settings
P6	All pins are OPEN.
W2	Use the factory default settings.

Table C.2 Cogent CMA278 Daughtercard Dipswitch Settings

Jumper Location	Settings
SW2	Set 6 and 8 to OFF. All others are ON.

For more information see these documents:

- *CMA102 Motherboard User's Manual* by Cogent Computer Systems, Inc.
- *CMA278 PPC60x/740 User's Manual* by Cogent Computer Systems, Inc.



Tested Jumper and Dipswitch Settings

Axiom 555, 565

Axiom 555, 565

The tested jumper and dipswitch settings for the Axiom 555 board are listed in Table C.3 and Table C.4, respectively.

NOTE Subswitches 5 and 6 are ON for programming internal Flash, otherwise they are OFF.

Table C.3 Axiom 555 Jumper Settings

Jumper Label	Setting
RAM-SEL	Jumper at position 2
FLSH-SEL	Jumper at position 1
M-SEL	Jumper at position 3

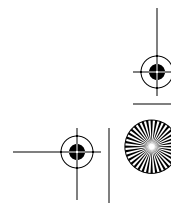
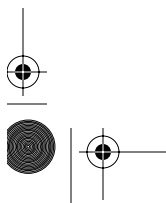
Table C.4 Axiom 555 Dipswitch Settings

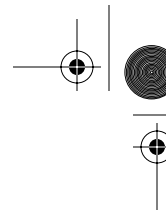
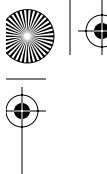
Dipswitch Location	Setting
MODE_SW1	Set all subswitches (1 through 8) to OFF
MODE_SW2	Set all subswitches (1 through 8) to OFF
Config Switch	Set 2, 4, 5, and 6 to ON. All others are OFF Subswitches 5 and 6 are ON for programming internal Flash, otherwise they are OFF.

The chip select and dipswitch settings for the Axiom 565 board are listed in Table C.5 and Table C.6, respectively.

Table C.5 Axiom 565 Chip Selects

Chip Select	Memory Bank
CS1	FSRAM (U4-U7)
CS0	Flash EPROM (U15-U16)
CS3	EPROM (U2-U3)





Tested Jumper and Dipswitch Settings

Motorola 555 ETAS

Table C.6 Axiom 565 Dipswitch Settings

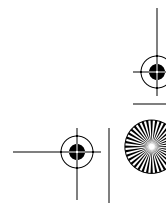
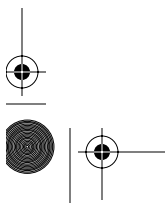
Dipswitch Location	Setting
MAP_SW	Set 2 and 7 to ON. All others are OFF
MODE_SW1	Set all subswitches (1 through 8) to OFF
MODE_SW2	Set all subswitches (1 through 8) to OFF
MODE_SW3	Set 5 to ON. All others OFF Set 5 to ON when booting from internal Flash, otherwise set to OFF
MODE_SW4	Set all subswitches (1 through 8) to OFF
COM_SW	Set 1 through 4 to ON. All others are OFF
CONFIG-SW	Set 2, 4, 5, and 6 to ON. All others are OFF Subswitches 5 and 6 are ON for programming internal Flash, otherwise they are OFF

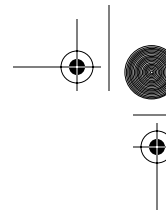
Motorola 555 ETAS

Table C.7 lists the tested dipswitch settings for the Motorola 555 ETAS target board.

Table C.7 Motorola 555 ETAS Dipswitch Settings

Dipswitch Location	Setting
Reset Configuration Word (32bit)	Set 2 and 20 to OFF. All others are ON.
SW100	Set 6, 7 to ON. All others are OFF.
SW101	Switch to A.
SW102	Switch to A to use BDM (Macraigor Wiggler) or MetroTRK.
SW200	Set 1 and 3 to OFF, 2 and 4 to ON.
MODCK 1	Set to OFF.





Tested Jumper and Dipswitch Settings

Motorola 5100 Ice Cube

Table C.7 Motorola 555 ETAS Dipswitch Settings (continued)

Dipswitch Location	Setting
MODCK 2	Set to ON.
MODCK 3	Set to OFF.

Motorola 5100 Ice Cube

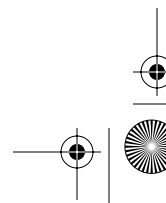
Table C.8 and Table C.9 list the initial jumper settings for the Motorola 5100 Ice Cube board. These jumper settings set the boot configuration of the board.

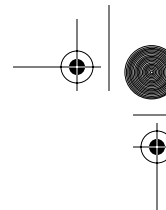
Table C.8 Motorola 5100 Ice Cube Jumper Settings (J75, J74, J73)

Jumper Label	Setting
XLB	High — Jumper between pin 1 of J75 and pin 1 of J74
SYS	Low — Jumper between pin 2 of J73 and pin 2 of J74
RES	Low — Jumper between pin 3 of J73 and pin 3 of J74
RES	Low — Jumper between pin 4 of J73 and pin 4 of J74
RES	Low — Jumper between pin 5 of J73 and pin 5 of J74
B H/L	High — Jumper between pin 6 of J75 and pin 6 of J74
WAIT	High — Jumper between pin 7 of J75 and pin 7 of J74
SWAP	Low — Jumper between pin 8 of J73 and pin 8 of J74
SIZE	Low — Jumper between pin 9 of J73 and pin 9 of J74
MUX	Low — Jumper between pin 10 of J73 and pin 10 of J74

Table C.9 Motorola 5100 Ice Cube Jumper Settings (J35, J36, J37)

Jumper Label	Setting
PPC0	Low — Jumper between pin 1 of J35 and pin 1 of J36
PPC1	High — Jumper between pin 2 of J37 and pin 2 of J36
PPC2	High — Jumper between pin 3 of J37 and pin 3 of J36





Tested Jumper and Dipswitch Settings
Motorola Excimer 603e

Table C.9 Motorola 5100 Ice Cube Jumper Settings (J35, J36, J37) (*continued*)

Jumper Label	Setting
PPC3	High — Jumper between pin 4 of J37 and pin 4 of J36
PPC4	Low — Jumper between pin 5 of J35 and pin 5 of J36

Motorola Excimer 603e

Table C.10 lists the tested jumper settings for the Motorola Excimer 603e EVB target board.

Table C.10 Motorola 555 ETAS Jumper Settings

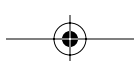
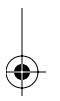
Jumper Location	Setting
J3	1-2 CLOSED; 3 OPEN
J4	1 OPEN; 2-3 CLOSED
J5	1 OPEN; 2-3 CLOSED

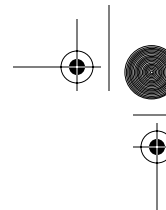
Motorola MPC 8xx MBX

Table C.11 lists the tested jumper settings for the Motorola MPC 8xx MBX target board.

Table C.11 Motorola MPC 8xx MBX Dipswitch and Jumper Settings

Jumper Location	Setting
Jumper 1	Depends on battery setup. See the reference manual for your target board.
Jumper 2	3-4 CLOSED (Normal Mode setting)
Jumper 3	1-2 CLOSED (Boot ROM Write Protect)
Jumper 4	2-3 CLOSED (Uses Flash for Boot)
Jumper 5	2-3 CLOSED (Select DEBUG Port Signal Pins)
Jumper 6	Depends on setup. See the reference manual for the board.
Jumper 7	Any setting is allowed.





Tested Jumper and Dipswitch Settings

Motorola MPC 8xx FADS

Table C.11 Motorola MPC 8xx MBX Dipswitch and Jumper Settings (*continued*)

Jumper Location	Setting
Jumper 8	Depends on DRAM setup. See the reference manual for the board.
Jumper 9	Depends on DRAM setup. See the reference manual for the board.
Jumper 10	Depends on DRAM setup. See the reference manual for the board.
Jumper 11	Depends on PCMCIA setup. See the reference manual for the board.

Motorola MPC 8xx FADS

Table C.12 lists the tested dipswitch and jumper settings for the Motorola MPC 8xx FADS target board (main board).

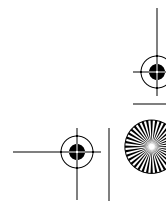
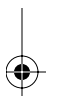
Table C.12 Motorola MPC 8xx FADS Dipswitch and Jumper Settings

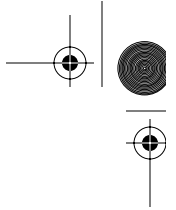
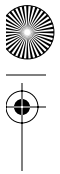
Dipswitch and Jumper Location	Setting (Main Board)
J1	1-2 CLOSED; 3 OPEN.
DS1	Set all 4 to ON.
DS1	Set all 4 to OFF.

Table C.13 lists the tested settings for the Motorola MPC 8xx FADS daughtercard.

Table C.13 Motorola MPC 8xx FADS Jumper Settings (Daughtercard)

Jumper Location	Setting (Daughtercard)
J1	Determines a level at which Power-On-Reset is generated. Any setting is allowed.
J2	1-2 CLOSED; 3 OPEN (3.3V setting)
J3	Set to 1-2 CLOSED; 3 OPEN (factory default).





Tested Jumper and Dipswitch Settings
Motorola MPC 82xx ADS

Motorola MPC 82xx ADS

Table C.14 lists the factory settings for the Motorola MPC 82xx ADS board.

Table C.14 Motorola MPC 82xx ADS Jumper Settings

Jumper Location	Settings
JP1	2-3 CLOSED; 1 OPEN (Fast download JTAG machine disabled)
JP3	2-3 CLOSED; 1 OPEN (BCSR is hard reset)
JP5	1-2 CLOSED; 3 OPEN (Range of 1.7V to 1.9V on VDDL)

Table C.15 lists the factory dipswitch settings for the Motorola MPC 82xx ADS board.

Table C.15 Motorola MPC 82xx ADS Dipswitch Settings

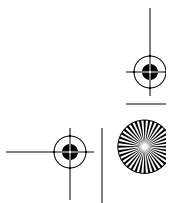
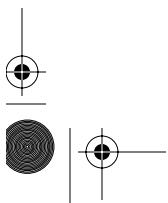
Jumper Location	Settings
SW2	Set all the subswitches (1 to 4) to the ON position.
SW3	Set subswitches 1 and 2 to the ON position. Set subswitches 3 and 4 to the OFF position.
SW4	Set subswitches 2, 4, and 8 to the OFF position. Set all remaining subswitches to the ON position.

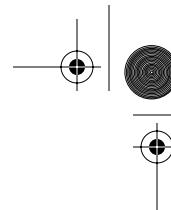
The factory settings for other components are as follows:

- Set the ATX chassis main AC power switch to 230-volts AC position.
- Set the VDDL supply voltage level selector (R26) in the middle range.

Motorola Maximer 7400

For the Motorola Maximer 7400 board, the tested settings are the factory default jumper settings.





Tested Jumper and Dipswitch Settings

Motorola Sandpoint

Motorola Sandpoint

NOTE Before applying the following changes, save your original switch settings.

The following Sandpoint switch settings are known to work with both MetroTRK and COP debug assist hardware.

Table C.16 lists the tested dipswitch and jumper settings for the Motorola Sandpoint.

Table C.16 Motorola Sandpoint MPC 8240 Jumper and Dipswitch Settings

Dipswitch / Jumper Location	Settings (Main Board)
VIO	Set for 5V. (Each pin on J30 is connected to the corresponding pin on J32; all pins on J31 are unconnected.)
J34	Closed.
J33	Open.
S3	The red top is positioned away from the power supply.
S4	The red top is positioned away from the power supply.
S5	The red top is positioned away from the power supply.
S6	The red top is positioned away from the power supply.

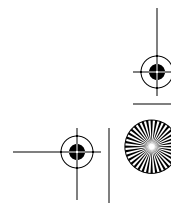
Table C.17 lists the tested settings for the Motorola Sandpoint x2 and x4 daughtercards.

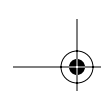
Table C.17 Motorola Sandpoint Settings (Daughtercard)

Daughtercard	Settings
X2	SW1=10101110 SW2=01000101
X4	SW2=10110000 SW3=01001001

Motorola MPC 8255/8260 ADS

Table C.18 lists the tested dipswitch and jumper settings for the Motorola MPC 8255/8260 ADS target board.





Tested Jumper and Dipswitch Settings
Embedded Planet RPX Lite 8xx

Table C.18 Motorola MPC 8255/8260 ADS Jumper and Dipswitch Settings

Dipswitch / Jumper Location	Settings
J1	1-2 CLOSED; 3 OPEN (5V+ setting)
P10	Set all to OFF
DS1	Set 1 and 3 to OFF Set 2 and 4 to ON
DS2	Set all 4 to ON
DS3	Set all 4 to ON

Embedded Planet RPX Lite 8xx

Table C.19 lists the tested dipswitch settings for the RPX Lite 8xx target board.

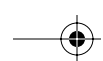
Table C.19 RPX Lite 8xx Dipswitch Settings

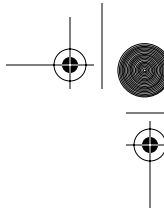
Dipswitch location	Settings
1	Set to on.
2	Set to on.
3	Set to on.
4	Set to on.

Table C.20 lists the tested jumper settings for the RPX Lite 8xx target board.

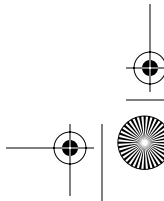
Table C.20 RPX Lite 8xx Jumper Settings

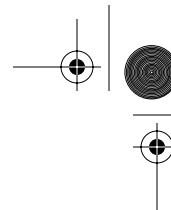
Jumper Location	Settings
JP1	Set to off (no jumper connected).
JP4	1 -2 = DEBUG connector valid (P6 - BDM port).





Tested Jumper and Dipswitch Settings
Embedded Planet RPX Lite 8xx





Index

Symbols

__abs() 85
 __attribute__ ((aligned(?))) 64–65
 __cntlzw() 86
 __eieio() 85
 __fabs() 85
 __fnabs() 85
 __isync() 85
 __labs() 85
 __lhrx() 85
 __lwbrx() 85
 __pixel 39
 __rlwimi() 86
 __rlwinm() 86
 __rlwnm() 86
 __setflm() 85
 __sthbrx() 85
 __stwbrx() 85
 __sync() 85
 __vector 39
 _ABS_SDA_BASE_ 67
 _ABS_SDA2_BASE_ 68
 _heap_addr 67
 _heap_end 67
 _nbfunctions 68
 _SDA_BASE_ 67
 _SDA2_BASE_ 67
 _stack_addr 67
 _stack_end 67

A

alternatePC 200
 AltiVec, vector data formats 38
 Andmmr 200
 asm blocks not supported 72
 asm keyword 72
 assembler
 stand-alone described 17
 See also inline assembler

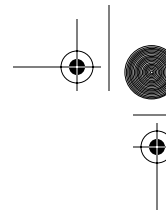
B

back-end compiler *See* compiler
 BatchRunner PostLinker panel 167
 BatchRunner PreLinker panel 166
 binary files 21
 board initialization code 34
 bool 39
 bool size 37
 build folder 179

C

char size 37
 CodeWarrior
 compare to command line 18
 compiler described 17
 components 16
 debugger described 18
 development process 18–20
 IDE described 16
 linker described 17
 stand-alone assembler described 17
 tools listed 16
 Cogent CMA102 target board jumper
 settings 215
 Cogent CMA 278 daughtercard dipswitch
 settings 215
 command syntax
 memory configuration files 211
 command window 133
 command-line and CodeWarrior compared 18
 command-line debugger 133
 commands
 memory configuration files syntax 211
 comments in inline assembler 76
 compiler
 back-end for PowerPC 35
 described 17
 other documentation 35
 support for inline assembly 71
 See also C Compilers Reference
 compiling 19





configuration files 197
configuration files, memory, command syntax
of 211
connection type 94
console I/O 30
converting, makefiles to CodeWarrior
project 26–27
current folder 180

D

Data Addressing 40
data cache window 116
DDR controller 193
deadstripping unused code 54
debug initialization files
commands
alternatePC 200
Andmmr 200
incorMMR 201
reset 201
run 202
setMMRBaseAddr 202
sleep 203
writemem.b 203
writemem.w 204
writemmr 205
writereg 206
writespr 207
writeupma 208
writeupmb 209
using 197
debugger features, special
displaying registers 109
EPPC menu 112–116
register details 110
Debugger PIC Settings panel 170
debugger protocol 94
debugger, described 18
debugging 20
ELF files 122–126
for PowerPC Embedded 91–122
supported remote connections 91–107
using command lines 133
using MetroTRK 121

See also Debugger User Guide

development tools 16
dipswitch settings
Cogent CMA 278 daughtercard 215
Embedded Planet RPX Lite 8xx target
board 223
Motorola 555 ETAS target board 217
Motorola MPC 8xx FADS target board 220
Motorola MPC 8xx MBX target board 219
Motorola Sandpoint 8240 target board 222
directives, assembler
entry 81
fralloc 82
frfree 82
machine 83
nofralloc 84
opword 84
disassembly, Register Details window 110
double size 38

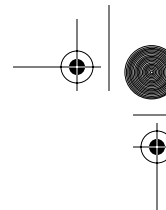
E

e500 coherency module 193
EABI templates, for PowerPC projects 26
editing code 19
See also IDE User Guide
__eieio() 85
ELF files, debugging 122–126
entry assembly statement 81
entry directive 81
EPPC Assembler settings panel 144–145
EPPC Debugger Settings panel 171–173
EPPC Disassembler settings panel 153–155
EPPC Linker settings panel 157–164
EPPC menu, options explained 112–116
EPPC Processor settings panel 146–153
EPPC Target panel 138–143
EXCLUDEFILES 56
external segments 66

F

float size 38
floating-point formats 38
floating-point support 148
force_active 44





FORCEACTIVE 57

FORCEFILES 57

FPSCR 85

fralloc assembly statement 77

fralloc directive 82

frfree assembly statement 77

frfree directive 82

function level assembly 72

function_align 45

G

GNU Assembler panel 146

GNU Compiler panel 156

GNU Disassembler panel 155

GNU Environment panel 167

GNU Linker panel 165

GNU Post Linker panel 165

GNU Target panel 143

GNU Tools panel 168

GROUP 57

H

Hardware tools

flash programmer 183–185

hardware diagnostics 185

logic analyzer 186–193

I

IDE described 16

INCLUDEDDWARF 58

incompatible_return_small_structs 45

incompatible_sfpe_double_params 45

incorMMR 201

inline assembler

asm blocks not supported 72

comments 76

directives 81–84

for PowerPC 71

function level support 72

instructions 71, 72

local variables 76

operands 78

preprocessor use 76

special PowerPC instructions 74

stack frame 77

statement labels 75

syntax 72

using for PowerPC 71

int size 37

integer formats 36, 38

integer formats for PowerPC 36

internal segments 66

interrupt 46

intrinsic functions

described 84

See also inline assembler 84

J

jumper settings

Cogent CMA102 target board 215

Embedded Planet RPX Lite 8xx target board 223

Motorola Excimer 603e target board 219

Motorola Maximer 7400 target board 221

Motorola MPC 8xx FADS daughtercard 220

Motorola MPC 8xx FADS target board 220

Motorola MPC 8xx MBX target board 219

Motorola Sandpoint 8240 target board 222

L

labels in inline assembly language 75

__labs() 85

libraries

console I/O 30

MSL for PowerPC Embedded 29

runtime 32

support for PowerPC Embedded 29–34

using MSL 29–30

link order 54

linker

.a files 55

.o files 55

and executable files 55

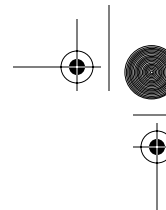
described 17

for PowerPC 53–55

multiply defined symbols 55

other documentation 35





linker generated symbols 53

linker-defined symbols 67

linking 20

See also IDE User Guide

local variables in inline assembler 76

L1 Data Cache window 116

long double size 38

long long size 37

long size 37

M

machine assembly statement 83

machine directive 83

Makefile Importer wizard, using 26–27

MEMORY 59

memory configuration, MetroTRK 120–121

MetroTRK

connecting 118–119

memory configuration 120–121

overview 117

using, for debugging 121

Metrowerks Standard Libraries *See* MSL

Motorola Excimer 603e target board jumper

settings 219

Motorola 555 ETAS target board dipswitch

settings 217

Motorola Maximer 7400 target board jumper

settings 221

Motorola MPC 8xx FADS daughtercard jumper

settings 220

Motorola MPC 8xx FADS target board dipswitch

settings 220

Motorola MPC 8xx FADS target board jumper

settings 220

Motorola MPC 8xx MBX target board dipswitch

settings 219

Motorola MPC 8xx MBX target board jumper

settings 219

MSL

using 29–30

and runtime libraries 29

described 18

for PowerPC Embedded 29

using console I/O 30

See also MSL C Reference, MSL C++

Reference

multiple symbols and linker 55

N

nofralloc directive 84

number formats

for PowerPC 36–39

integers 36

O

operands in inline assembler 78

optimizing

for PowerPC 42–43

inline assembly disables 73

register coloring 42

opword directive 84

P

pack 47

PCI controller 193

PIC/PID 66

building applications 69

linker-defined symbols 67

scenarios 68

uses 68

pixel 39

pooled_data 48

position-independent code. *See* PIC/PID

position-independent data. *See* PIC/PID

postlinker 165, 167

PowerPC EABI templates, using 26

PowerPC Embedded debugging *See* debugging

PQ3 Trace Buffer panel 174

pragma

for PowerPC 43

overload 55

prelinker 166

preprocessing 20

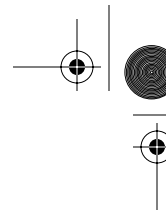
See also IDE User Guide

preprocessor, using in inline assembler 76

project

types of 21





project targets
 debug version 25
 ROM version 25
 project window
 Targets page 135

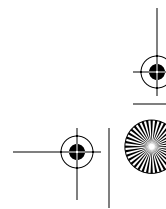
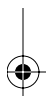
R

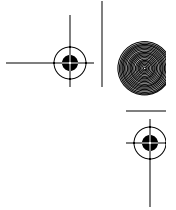
RapidIO controller 193
 Register 41
 register coloring optimization 42
 register details 110
 Register Details window 110
 register variables for PowerPC 41
 registers
 displaying 109
 floating point 109
 general 109
 Register Details window 110
 special purpose 109
 TLB 131
 variables 41
 vector registers 109
 registers, displaying 109
 reset 201
 __rlwimi() 86
 __rlwinm() 86
 __rlwnm() 86
 RPX Lite 8xx dipswitch settings 223
 RPX Lite 8xx jumper settings 223
 run 202
 runtime libraries
 and MSL 29
 customizing 34
 for PowerPC Embedded 32
 in projects 33
 initializing hardware 34

S

Sandpoint 8240 dipswitch settings 222
 Sandpoint 8240 jumper settings 222
 section 48
 section pragma 48
 SECTIONS 60
 segments, internal/external 66

__setfml() 85
 setMMRBaseAddr 202
 settings panels
 BatchRunner PostLinker 167
 BatchRunner PreLinker 166
 Debugger PIC Settings 170
 EPPC Assembler 144–145
 EPPC Debugger Settings 171–173
 EPPC Disassembler 153–155
 EPPC Linker 157–164
 EPPC Processor 146–153
 EPPC Target 138–143
 GNU Assembler 146
 GNU Compiler 156
 GNU Disassembler 155
 GNU Environment 167
 GNU Linker 165
 GNU Post Linker 165
 GNU Target 143
 GNU Tools 168
 PQ3 Trace Buffer 174
 Source Folder Mapping 178–180
 System Call Service Settings 180
 Target Settings 137
 short size 37
 SHORTEN_NAMES_FOR_TOR_101 62
 signed char size 37
 SIZEOF_HEADERS 68
 sleep 203
 small data 66
 Source Folder Mapping panel 178–180
 special debugger features 108–112
 stack frame in inline assembler 77
 Stack Size edit field 142
 stand-alone assembler 17
 See also Assembler Guide
 statement labels, in inline assembly language 75
 __sthbrx() 85
 __stwbrx() 85
 symbols
 linker generated 53
 multiple linker 55
 symbols, linker-defined 67
 __sync() 85





System Call Service Settings 180

T

target initialization files 197
Target Settings panel 137
TLB 131
TLB0 131
TLB1 131
trace buffer 174, 193–195
translation look-aside buffers 131

U

UART libraries
 and processor speed 31
unsigned char size 37
unsigned int size 37
unsigned long long size 37
unsigned long size 37
unsigned short size 37

V

variables
 register 41
vector 39
vector registers (AltiVec), displaying 109

W

writemem.b 203
writemem.w 204
writemmr 205
writereg 206
writespr 207
writeupma 208
writeupmb 209

