Freescale Semiconductor, Inc.

CodeWarrior™ Development Studio MPC5xx Edition Version 8.1

Targeting Manual

Revised 2004/03/17

metrowerks

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

How to Contact Metrowerks

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

Table of Contents

1 Introduction

This manual explains how to install and use the CodeWarrior™ Development Studio, MPC5xx Edition tools.

This chapter contains these sections:

- Read the Release Notes
- Related Documentation

Read the Release Notes

The release notes contain information about new features, bug fixes, and incompatibilities that do not appear in the documentation because of release deadlines. The release notes are in the `Release_Notes` directory on the CD-ROM.

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

- Look for the CodeWarrior tutorials in the *`InstallDir`*`\(CodeWarrior_Examples)` directory.
- For general information about the CodeWarrior IDE and debugger, see the *IDE User Guide* in the *`InstallDir`*`\Help\PDF` directory.
- For information specific to the C/C++ front-end compiler, see the *C Compilers Reference in the* *`InstallDir`*`\Help\PDF` directory.

- For information on Metrowerks' standard C/C++ libraries, see the *MSL C Reference* and the *MSL C++ Reference in the* `InstallDir\Help\PDF` directory.
- For general information about MetroTRK and instructions that explain how to customize MetroTRK to work with additional target boards, see *MetroTRK Reference in the* `InstallDir\Help\PDF` directory.

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 CodeWarrior for Embedded PowerPC 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 Format) 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:

  `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*, published by Motorola, Inc., and available at this address:

  `http://e-www.motorola.com/brdata/PDFDB/docs/ALTIVECPIM.pdf`
- *AltiVec Technology Programming Environments Manual*, published by Motorola, Inc., and available at this address:

  `http://e-www.motorola.com/collateral/ALTIVECPEMCH.htm`

2

Getting Started

This chapter provides installation instructions for the CodeWarrior™ Development Studio, MPC5xx Edition tools. In addition, the chapter includes an overview of the development process you follow when using the CodeWarrior IDE.

This chapter has these topics:

- System Requirements
- Supported Target Boards
- Installing your CodeWarrior™ Product
- CodeWarrior Development Tools
- CodeWarrior Development Process

System Requirements

The system requirements for the CodeWarrior™ Development Studio, MPC5xx Edition product are:

- Hardware:
  - PC with a 400 MHz Intel® Pentium® class processor, minimum
  - 128 MB RAM, minimum
  - 300 MB free hard disk space, minimum
  - CD-ROM drive
  - Serial port, parallel port, and Ethernet port.
- Software: Microsoft® Windows 2000/XP® or Windows NT® Workstation 4.0.

Supported Target Boards

Table 2.1 lists the supported target boards and their manufacturers.

Table 2.1 Supported Target Boards

Manufacturer	Boards
Axiom	Axiom 555, 565
Motorola	Motorola 555 ETAS
	Boards with Motorola MPC 56x chip
PHYTEC	phyCORE single-board computer subassembly for the MPC5xx family

Installing your CodeWarrior™ Product

To install your CodeWarrior product, follow these steps:

1. Put the installation CD in the CD drive.

   The CodeWarrior installation menu appears.

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

2. Click **Launch the installer**

   The Install Wizard starts and displays a welcome screen.

3. Click **Next**

   The Install Wizard displays a license agreement screen.

4. Select the **I accept the terms of the license agreement** option.
5. Click **Next**

   The installation procedure begins.

6. Follow the on-screen instructions, and accept the default for each option.
7. When prompted to check for updates, click **Yes**

   The **CodeWarrior Updater** window appears.

NOTE If the **CodeWarrior Updater** already has the correct Internet connection settings, proceed directly to step 11.

8. Click **Settings**

   The **Internet Properties** dialog box appears.

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

10. Click **OK**

    The **Internet Properties** dialog box closes.

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

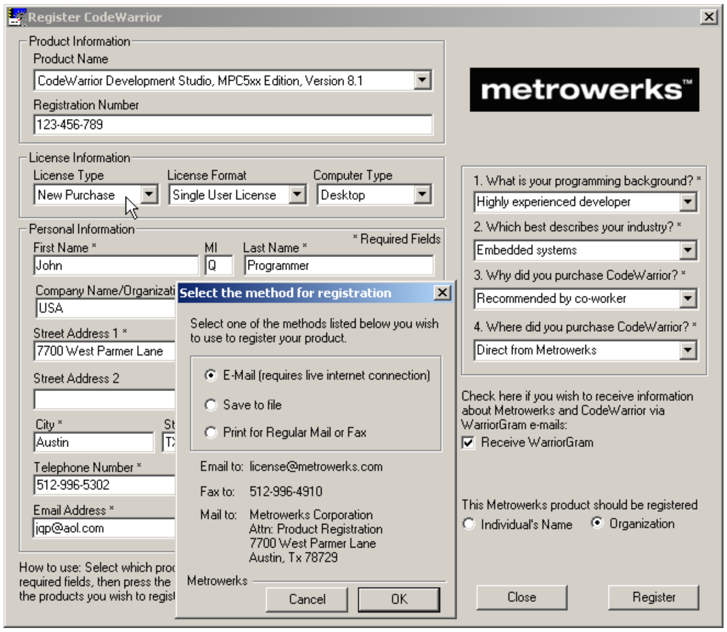
    The updater checks for newer versions of the CodeWarrior products installed on your PC.

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

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

    The **Register CodeWarrior** window appears. (See Figure 2.1.)

Figure 2.1 Register CodeWarrior and Registration Method Windows



Register CodeWarrior

Product Information
Product Name: CodeWarrior Development Studio, MPC5xx Edition, Version 8.1
Registration Number: 123-456-789

License Information
License Type: New Purchase
License Format: Single User License
Computer Type: Desktop

Personal Information
* Required Fields
First Name *: John
MI: Q
Last Name *: Programmer
Company Name/Organizati: USA
Street Address 1 *: 7700 West Parmer Lane
Street Address 2:
City *: Austin
St: TX
Telephone Number *: 512-996-5302
Email Address *: jqp@aol.com

How to use: Select which proc
required fields, then press the
the products you wish to regist

1. What is your programming background? *: Highly experienced developer
2. Which best describes your industry? *: Embedded systems
3. Why did you purchase CodeWarrior? *: Recommended by co-worker
4. Where did you purchase CodeWarrior? *: Direct from Metrowerks

Check here if you wish to receive information about Metrowerks and CodeWarrior via WarriorGram e-mails:
☑ Receive WarriorGram

This Metrowerks product should be registered
○ Individual's Name ● Organization

Select the method for registration

Select one of the methods listed below you wish to use to register your product.

● E-Mail (requires live internet connection)
○ Save to file
○ Print for Regular Mail or Fax

Email to: license@metrowerks.com
Fax to: 512-996-4910
Mail to: Metrowerks Corporation
Attn: Product Registration
7700 West Parmer Lane
Austin, Tx 78729

14. From the **License Type** list box, select New Purchase, Renewal, or Evaluation.

15. Fill in all other fields of registration window.

16. Click **Register**

    The Registration Method dialog box appears.

17. Select the **E-Mail** option.
18. Click **OK**

    A confirmation dialog box appears, the program sends your registration to Metrowerks, and Metrowerks e-mails the required license keys to you.

19. Click **OK**

    The confirmation dialog box closes.

20. Click **Close**

    The **Register CodeWarrior** window closes.

NOTE If you had problems registering, run `MWRegister.exe` after installation completes. This program is in the `License` directory of your CodeWarrior installation. E-mail registration questions to `license@metrowerks.com`.

21. Select **Yes, I want to restart my computer now** option.
22. Click **Finish**

    Your PC restarts, and installation completes.

23. Use Notepad to open `license.dat`. This file is in the CodeWarrior installation directory. The default installation directory is:

    `C:\Program Files\Metrowerks\CodeWarrior\`

24. On a new line at end of the file, paste or type the license keys received from Metrowerks.
25. Save and close `license.dat`

NOTE Do not move `license.dat`. If you do, your CodeWarrior product will not function.

26. Start the CodeWarrior IDE.

    The IDE uses the updated license file.

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
- C/C++ Compiler
- Standalone Assembler
- Linker
- Debugger
- Metrowerks Standard Libraries

If you are an experienced CodeWarrior user, this is the same IDE and debugger that you have been using all along. You will, however, 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. 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, refer to the *IDE User Guide.*

C/C++ Compiler

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

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

Standalone Assembler

The CodeWarrior EPPC assembler is a standalone assembler that supports an easy-to-use assembly language 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*.

Linker

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

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

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

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

Debugger

The CodeWarrior debugger controls the execution of your program and allows you to see what is happening internally as your program runs. You use the debugger to find problems in your program.

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

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

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

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

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 *IDE User Guide* for:

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

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

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

- Project Files
- Editing Code
- Compiling
- Linking

- Debugging
- Viewing Preprocessor Output

Project Files

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

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

The editor window has excellent navigational features that 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**.

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**.

Linking

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

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

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

Debugging

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

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

Viewing Preprocessor Output

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

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

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

3

Tutorial

This tutorial takes you step-by-step through the CodeWarrior programming environment. The tutorial does not teach you have to program; instead, it is designed to show you how to use the CodeWarrior Integrated Development Environment (IDE) to write and debug applications for the Embedded PowerPC platform.

The tutorial consists of these sections:

- Creating a Project
- Building and Debugging a Project

Creating a Project

You can create an Embedded PowerPC project using any of the techniques:

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

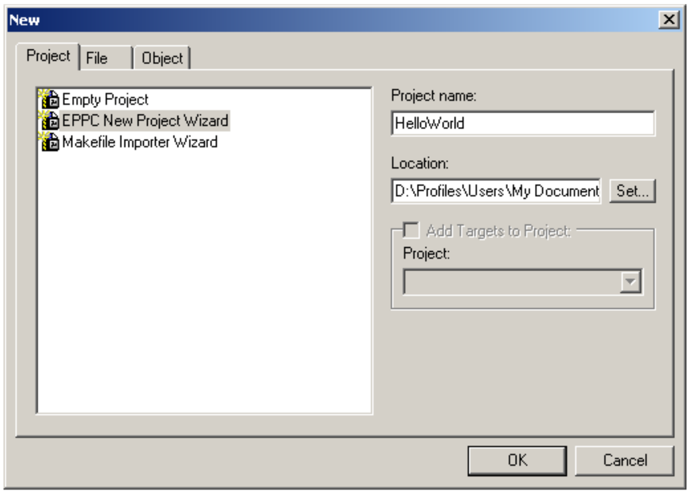
See the Creating a CodeWarrior™ Project chapter to learn how to create a project using the Makefile Importer Wizard, and PowerPC EABI templates.

This section explains how to create an Embedded Power PC project using the EPPC New Project Wizard. Any choices you make while creating a project with the EPPC New Project Wizard can be changed manually after the project has been created.

1. Create the project.

   a. Chose **File > New**. The **New** dialog box (Figure 3.1) appears.

Figure 3.1 New Dialog Box



Empty Project
EPPC New Project Wizard
Makefile Importer Wizard

Project name: HelloWorld

Location: D:\Profiles\Users\My Document

☐ Add Targets to Project:
Project:

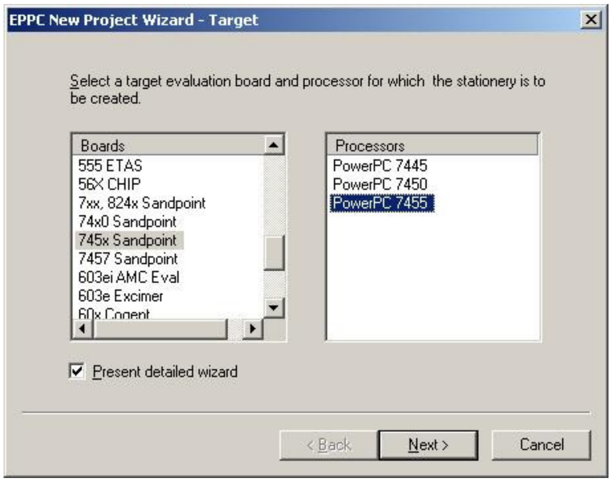
b. Select **EPPC New Project Wizard**.

c. In the **Project Name** text box, type the project name. For example, HelloWorld.

d. In the **Location** text box, type the location where you want to save this project or choose the default location.

e. Click **OK**. The **EPPC New Project Wizard — Target** dialog box (Figure 3.2) appears.

Figure 3.2 EPPC New Project Wizard — Target Dialog Box



Select a target evaluation board and processor for which the stationery is to be created.

Boards	Processors
555 ETAS	PowerPC 7445
56X CHIP	PowerPC 7450
7xx, 824x Sandpoint	PowerPC 7455
74x0 Sandpoint	
745x Sandpoint	
7457 Sandpoint	
603ei AMC Eval	
603e Excimer	
60x Cogent	

☑ Present detailed wizard

2. Select the target board and processor.

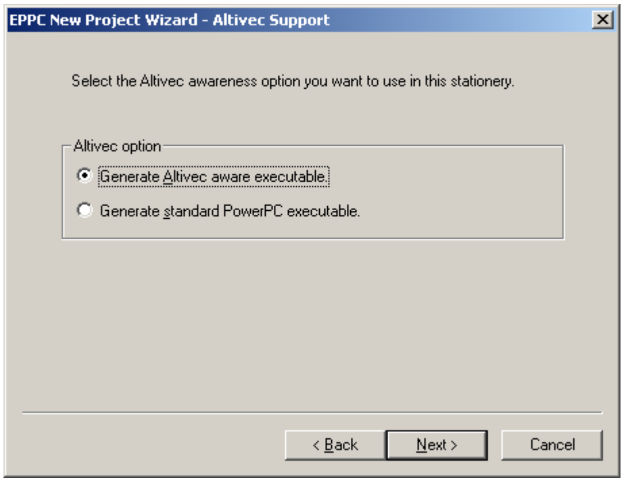
a. Select the evaluation board, such as the 745x Sandpoint board, from the **Boards** list.

b. Select the PowerPC processor, such as the PowerPC 7455 processor, from the **Processors** list.

c. Check the **Present detailed wizard** checkbox.

d. Click **Next**. The **EPPC New Project Wizard — Altivec Support** dialog box (Figure 3.3) appears.

Figure 3.3 EPPC New Project Wizard — Altivec Support Dialog Box



EPPC New Project Wizard - Altivec Support

Select the Altivec awareness option you want to use in this stationery.

Altivec option

● Generate Altivec aware executable.

○ Generate standard PowerPC executable.

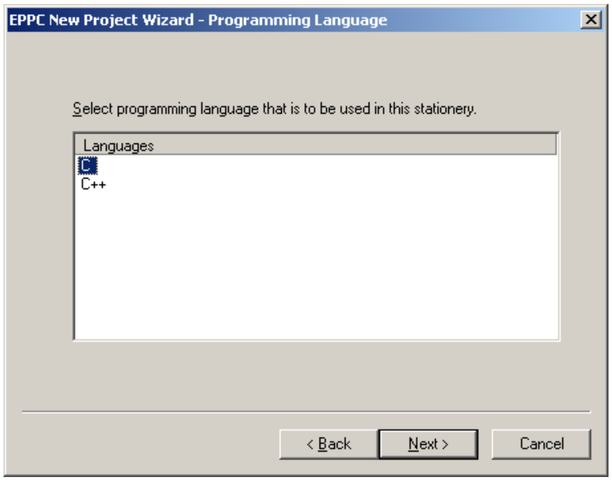
NOTE The **EPPC New Project Wizard — Altivec Support** dialog box appears only if the selected target system supports the Altivec awareness option.

3. Specify whether you want to generate an Altivec aware executable.

a. To generate an Altivec aware executable, select the **Generate Altivec aware executable** option button. If you do not want Altivec support, select the **Generate Standard PowerPC executable** option button.

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

Figure 3.4 EPPC New Project Wizard — Programming Language Dialog Box



EPPC New Project Wizard - Programming Language

Select programming language that is to be used in this stationery.

Languages
C
C++

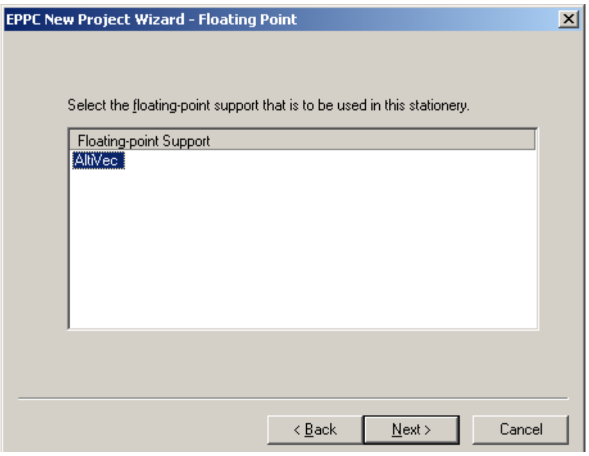
4. Select the programming language.

   a. Select either **C** or **C++** depending on the programming language you are using. For example, if you plan to use C source files in you project, select **C** from the **Languages** list.

NOTE Selecting a programming language primarily determines which libraries are linked to the project and how the main source file is set up. If you select the C++ language, you may still add C source files to the project later.

   b. Click **Next.** The **EPPC New Project Wizard — Floating Point** dialog box (Figure 3.5) appears.

Figure 3.5 EPPC New Project Wizard — Floating Point Dialog Box



EPPC New Project Wizard - Floating Point

Select the floating-point support that is to be used in this stationery.

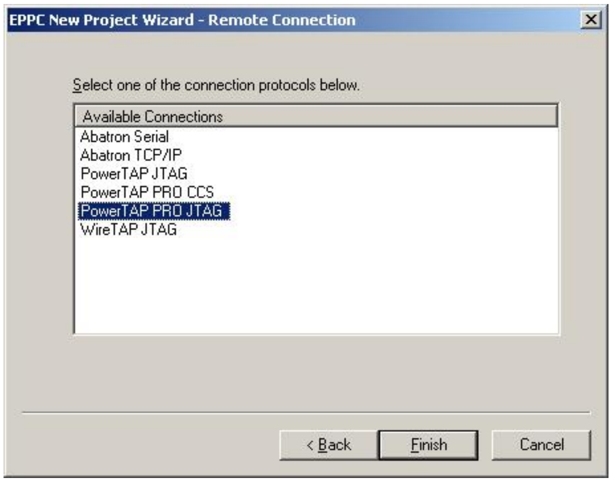
Floating-point Support
AltiVec

5. Select the floating point support.

   a. Select **AltiVec** in the **Floating-point Support** list.

   b. Click **Next**. The **EPPC New Project Wizard — Remote Connection** dialog box (Figure 3.6) appears.

Figure 3.6 EPPC New Project Wizard — Remote Connection Dialog Box



EPPC New Project Wizard - Remote Connection

Select one of the connection protocols below.

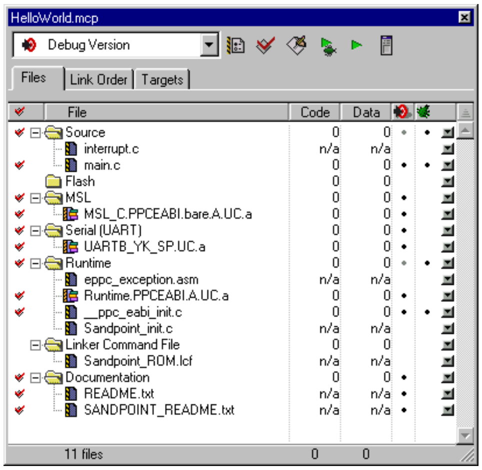
Available Connections
- Abatron Serial
- Abatron TCP/IP
- PowerTAP JTAG
- PowerTAP PRO CCS
- PowerTAP PRO JTAG
- WireTAP JTAG

6. Select the remote connection.

   a. Select the remote connection for the debugger interface you are using. For example, if you are using the PowerTAP PRO JTAG debugger interface, select **PowerTAP PRO JTAG** from the **Available Connections** list.

   b. Click **Finish** to create the new project; the project window (Figure 3.7) appears.

Figure 3.7 Project Window



HelloWorld.mcp

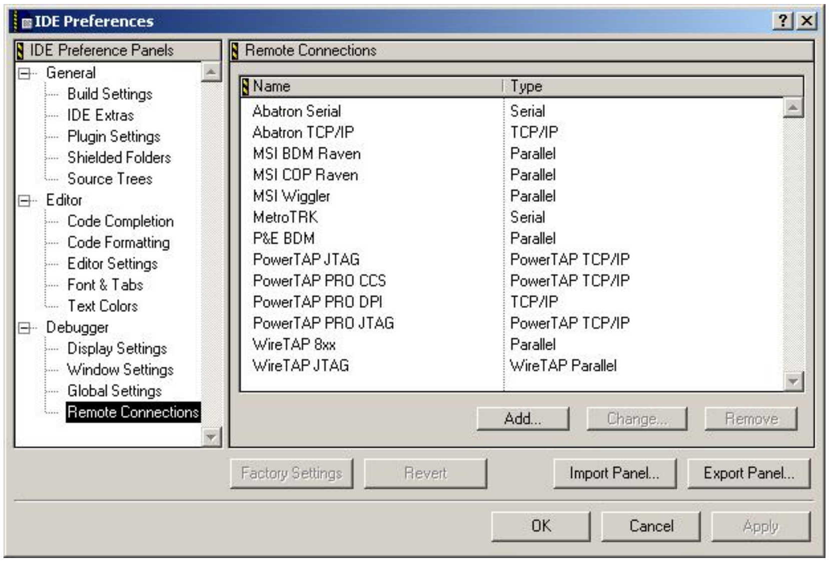
✓	File	Code	Data		
✓	Source	0	0	•	•
	interrupt.c	n/a	n/a		
✓	main.c	0	0	•	•
	Flash	0	0		
✓	MSL	0	0	•	
✓	MSL_C.PPCEABI.bare.A.UC.a	0	0	•	
✓	Serial (UART)	0	0	•	
✓	UARTB_YK_SP.UC.a	0	0	•	
✓	Runtime	0	0	•	•
	eppc_exception.asm	n/a	n/a		
✓	Runtime.PPCEABI.A.UC.a	0	0	•	
✓	__ppc_eabi_init.c	0	0	•	•
	Sandpoint_init.c	n/a	n/a		
	Linker Command File	0	0		
	Sandpoint_ROM.lcf	n/a	n/a		
✓	Documentation	0	0	•	
✓	README.txt	n/a	n/a	•	
✓	SANDPOINT_README.txt	n/a	n/a	•	
	11 files	0	0		

7. Specify the remote connection preferences.

   a. Select **Edit > Preferences**. The **IDE Preferences** window appears.

   b. Select the **Remote Connections** item in the **IDE Preference Panels** list. The **Remote Connections panel (Figure 3.8) appears.**

Figure 3.8 Remote Connections Panel

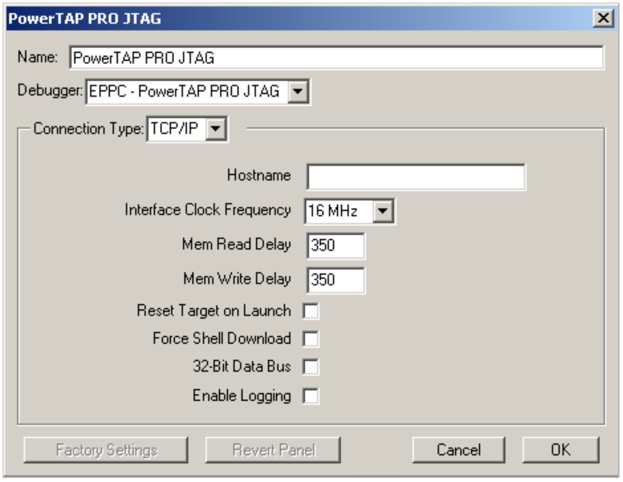


Name	Type
Abatron Serial	Serial
Abatron TCP/IP	TCP/IP
MSI BDM Raven	Parallel
MSI COP Raven	Parallel
MSI Wiggler	Parallel
MetroTRK	Serial
P&E BDM	Parallel
PowerTAP JTAG	PowerTAP TCP/IP
PowerTAP PRO CCS	PowerTAP TCP/IP
PowerTAP PRO DPI	TCP/IP
PowerTAP PRO JTAG	PowerTAP TCP/IP
WireTAP 8xx	Parallel
WireTAP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

   c. In the **Remote Connections** panel, click the remote connection name you selected for your project. For this tutorial, click **PowerTAP PRO JTAG**.

   d. Click **Change**. The **PowerTAP PRO JTAG** dialog box (**Figure 3.9**) appears.

Figure 3.9 PowerTAP PRO JTAG Dialog Box



Name: PowerTAP PRO JTAG

Debugger: EPPC - PowerTAP PRO JTAG

Connection Type: TCP/IP

Hostname

Interface Clock Frequency 16 MHz

Mem Read Delay 350

Mem Write Delay 350

Reset Target on Launch □

Force Shell Download □

32-Bit Data Bus □

Enable Logging □

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

NOTE For the purpose of this tutorial, the factory settings of the PowerTAP PRO JTAG connection are used. For details on the settings for this connection and other supported remote connections, see “Supported Remote Connections for Debugging”.

f. Click **OK**. The remote connection settings are saved; the **IDE Preferences** window appears.

g. Click **OK** in the **IDE Preferences** window. The IDE closes the window.

Building and Debugging a Project

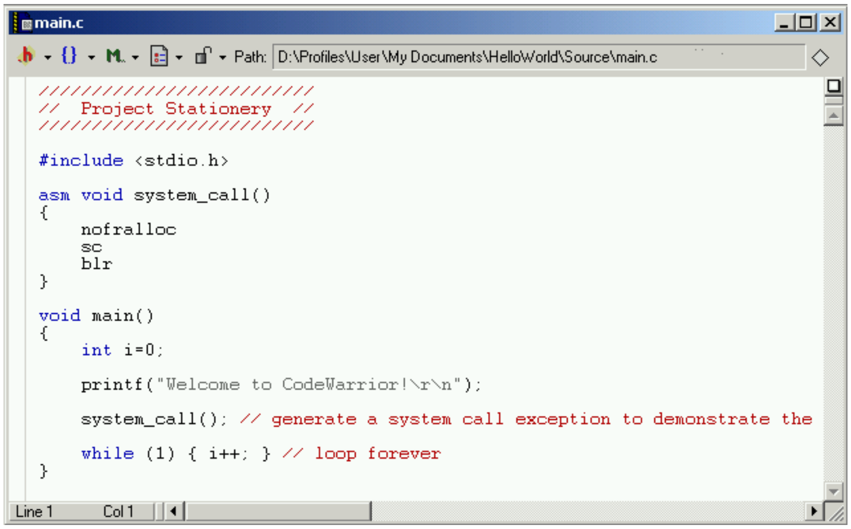
This section describes how to edit source code then compile and debug a project.

1. Edit the source code.

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

   b. Double-click the **main.c filename**. The editor window (Figure 3.10) appears. You can edit your code in this window.

Figure 3.10 main.c


main.c
Path: D:\Profiles\User\My Documents\HelloWorld\Source\main.c
Line 1
Col 1

```
//////////////////////////
//  Project Stationery  //
//////////////////////////

#include <stdio.h>

asm void system_call()
{
    nofralloc
    sc
    blr
}

void main()
{
    int i=0;

    printf("Welcome to CodeWarrior!\r\n");

    system_call(); // generate a system call exception to demonstrate the

    while (1) { i++; } // loop forever
}
```

2. Build the project.

   Select **Project > Make.** The IDE builds the project and stores the generated executable file in the project directory. The **Project** window shows the compiled code and data size in bytes after the project is built

   Note that the red checkmarks next to the filenames disappear. This indicates that the files no longer need to be built. Additionally, the IDE displays error messages in the **Errors & Warnings** window if it finds any errors during the make process.

NOTE Before starting the debug session, make sure that the debug monitor is connected to the target board and has a valid IP address. Also ensure that the serial cable appropriate for your target is connected between the COM A port of the target board and the serial port of the host computer.

3. Start the terminal emulation program.

   a. Select **Start > Programs > Accessories > Communications > HyperTerminal**. The **Connection Description** dialog box appears.

   b. Type the connection name in the **Name** text box

   c. Click **OK**. The **Connect To** dialog box appears.

   d. In the **Connect using** listbox, select the serial port to which the target board is connected.

   e. Click **OK**. The **COM1 Properties** dialog box appears.

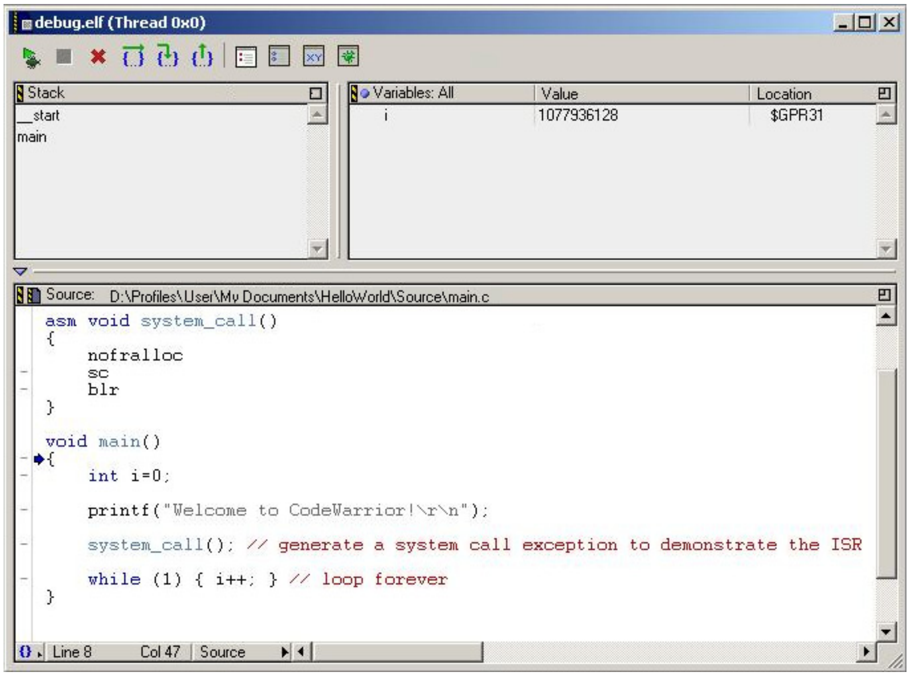
   f. Specify serial port settings as: **57600, 8, N, 1, N**

   g. Click **OK**. Terminal emulation starts; the HyperTerminal window appears.

4. Debug the project.

   a. Select **Project > Debug.** The IDE launches the debugger; the debugger window (Figure 3.11) appears with the program counter at the `main` function.

Figure 3.11 Debugger Window



Stack:

```
__start
main
```

Variables: All	Value	Location
i	1077936128	$GPR31

Source: D:\Profiles\User\My Documents\HelloWorld\Source\main.c

```
asm void system_call()
{
    nofralloc
    sc
    blr
}

void main()
{
    int i=0;

    printf("Welcome to CodeWarrior!\r\n");

    system_call(); // generate a system call exception to demonstrate the ISR

    while (1) { i++; } // loop forever
}
```

b. Click in the gutter to the left of the `printf("Welcome to CodeWarrior!\r\n")` source line to set a breakpoint.

c. Click the **Run** icon. The program executes up to the breakpoint you set.

d. Click the **Step Into** icon to step into the code of the `printf` function. This steps you into the C Runtime Library `printf` code. You can now step through the C Runtime Library `printf` code by clicking the **Step Over** icon.

e. Click the **Step Out** icon to return to `main`.

5. End the debug session.

Click the **Kill** icon in the debugger window toolbar to end the debug session and close the debugger window.

4

Creating a CodeWarrior™ Project

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

This chapter includes these topics:

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

Types of Projects

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

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

You can create an Embedded PowerPC project by using the:

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

"Creating a Project" explains how to create a project by using the EPPC new project wizard.

Using PowerPC EABI Templates

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

The EABI template sources are here:

`*InstallDir*\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

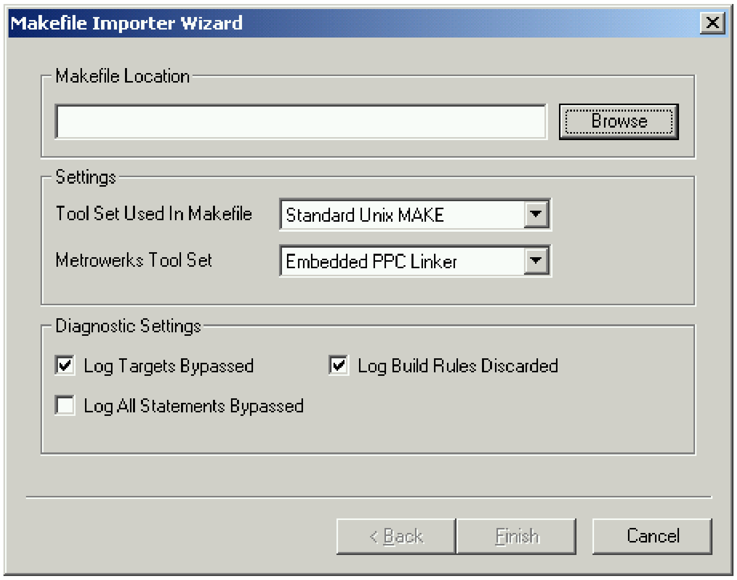
Use the Makefile Importer wizard to convert most GNU makefiles into CodeWarrior projects. The Makefile Importer wizard lets you:

- 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 makefiles to a CodeWarrior project:

1. Specify the project settings.
   a. Select **File > New**. The **New** dialog box appears.
   b. Select **Makefile Importer Wizard**.
   c. In the **Project Name** text box, type the project name with the `.mcp` extension.
   d. Click **OK**. The **Makefile Importer Wizard** dialog box (Figure 4.1) appears.

Figure 4.1 Makefile Importer Dialog Box



Makefile Location

Settings

Tool Set Used In Makefile: Standard Unix MAKE

Metrowerks Tool Set: Embedded PPC Linker

Diagnostic Settings

☑ Log Targets Bypassed ☑ Log Build Rules Discarded

☐ Log All Statements Bypassed

2. Specify the path of the makefile.

   Type the path of the makefile in the **Makefile Location** text box. 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** listbox to select the tool set that was used to generate the make files. For example, select **Standard UNIX Make** for GNU make files.

   b. Select **Embedded PPC Linker** from the **Metrowerks Tool Set** listbox.

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.

Project Targets

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

The available targets are:

- Debug Version

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

- ROM Version

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

- Auto Flash

  This target burns your program image to ROM. This target includes a small amount of code that programs the flash. The linker creates a RAM buffer that includes the image to flash followed by the flash code.

  If the CodeWarrior software successfully flashed the image, the program stops on the label `copy_successful`. If flashing the image was not successful, the program stops on the label `copy_failed`.

NOTE The Auto Flash target is not available for all boards.

5

Target Settings

Target settings define the behavior of the various development tools that a build target of a CodeWarrior project uses.

This chapter defines the Embedded PowerPC-specific target settings from which you can select. See the *CodeWarrior IDE User Guide* for information about settings available in all CodeWarrior projects.

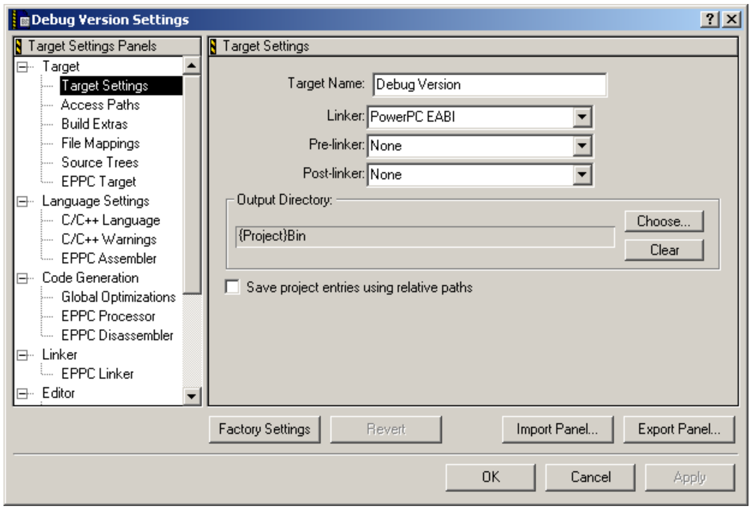
The sections of this chapter are:

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

Target Settings Overview

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

Figure 5.1 Target Settings Window



Debug Version Settings

Target Settings

Target Name: Debug Version

Linker: PowerPC EABI

Pre-linker: None

Post-linker: None

Output Directory:

{Project}Bin

□ Save project entries using relative paths

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

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

Embedded PowerPC Settings Panels

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

NOTE Certain target settings explained in the sections that follow may not be applicable to the product for which you have purchased the license.

The target settings panels covered in this section are:

- Target Settings
- EPPC Target
- EPPC Assembler
- Global Optimizations
- EPPC Processor
- EPPC Disassembler
- EPPC Linker
- Debugger PIC Settings
- EPPC Debugger Settings
- EPPC Exceptions
- System Call Service Settings

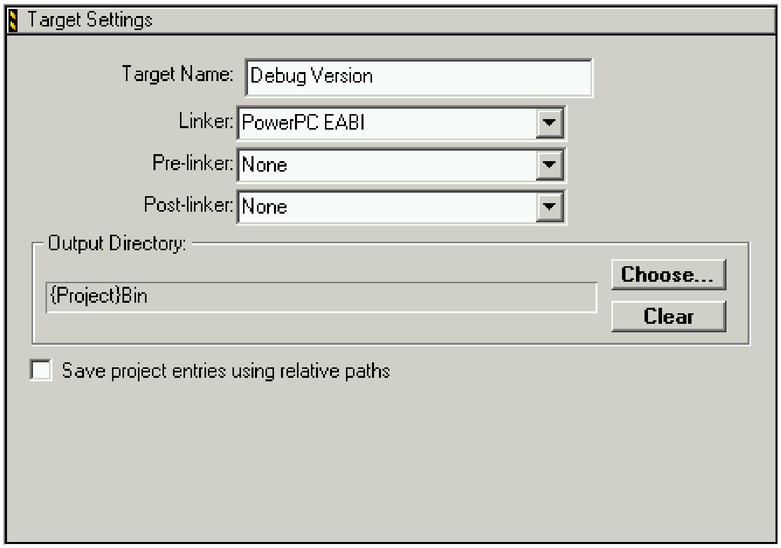
NOTE The **Global Optimizations** and **EPPC Processor** settings panels each provide code optimization options. Use both panels to select the best combination of optimization settings for your application.

Target Settings

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

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

Figure 5.2 Target Settings Panel



Target Settings

Target Name: Debug Version

Linker: PowerPC EABI

Pre-linker: None

Post-linker: None

Output Directory:

{Project}Bin

□ Save project entries using relative paths

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

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

Target Name

Type the name of the current project build target in the **Target Name** text box. If you display the **Targets** view of the **Project** window, the name of each build target in your project is displayed.

NOTE **Target Name** is not the name of your final output file; rather, it is the name of your project's current build target. You define the name of your final output file in the EPPC Target target settings panel.

Linker

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

- PowerPC EABI

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

- PCLint Linker

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

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

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

Pre-linker

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

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

CodeWarrior Development Studio, MPC5xx Edition includes just one pre-linker, the BatchRunner PreLinker.

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

Post-linker

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

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

CodeWarrior Development Studio, MPC5xx Edition includes just one post-linker, the BatchRunner PostLinker.

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

Output Directory

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

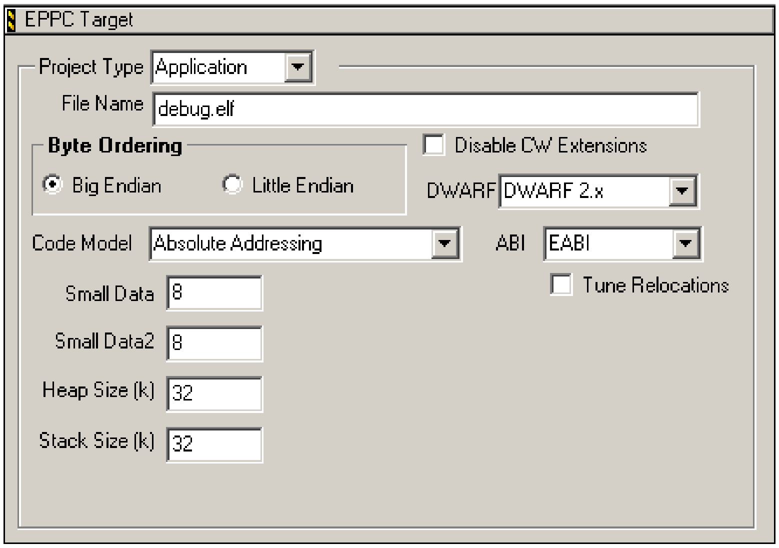
To add two or more files with the same name to a project, check this checkbox. The IDE includes information about the path used to access the file as well as the file name when it stores information about the file. When searching for a file, the IDE combines access path settings with the path settings 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 5.3) to specify the name and configuration of your final output file.

Figure 5.3 EPPC Target Settings Panel



EPPC Target

Project Type: Application

File Name: debug.elf

Byte Ordering: ● Big Endian ○ Little Endian

□ Disable CW Extensions

DWARF: DWARF 2.x

Code Model: Absolute Addressing

ABI: EABI

□ Tune Relocations

Small Data: 8

Small Data2: 8

Heap Size (k): 32

Stack Size (k): 32

Project Type

Select the kind of project you are creating from the **Project Type** listbox. 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 (k)**,** Stack Size (k), 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).

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** listbox 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** listbox.

ABI

Use the **ABI** listbox 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** listbox.

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, see this release notice:

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

Code Model

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

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.

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 (k)

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 (k)

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** listbox. 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 either `.text`, `.data` or `.bss`. 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`.

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

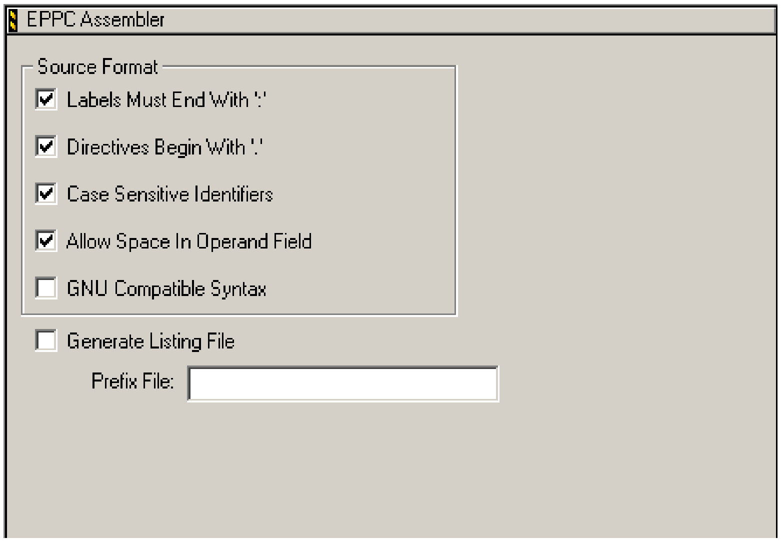
The **Require Resolved Symbols** checkbox is available only if you select the **Optimize Partial Link** checkbox. Check the **Require Resolved Symbols** checkbox to instruct the linker to resolve all symbols in your partial link. If any symbols are not present in one of the source files or libraries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

EPPC Assembler

Use the **EPPC Assembler** settings panel (Figure 5.4) to determine the format used for the assembly source files and the code generated by the EPPC assembler.

Figure 5.4 EPPC Assembler Settings Panel



EPPC Assembler

Source Format

☑ Labels Must End With ':'

☑ Directives Begin With '.'

☑ Case Sensitive Identifiers

☑ Allow Space In Operand Field

☐ GNU Compatible Syntax

☐ Generate Listing File

Prefix File:

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 listbox.

Source Format

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

GNU Compatible Syntax

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

GNU-compatibility allows:

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

- Using GNU compatible precedence rules for operators
- Implementing GNU compatible numeric local labels from 0 to 9
- Treating numeric constants beginning with 0 as octal
- Using semicolons as statement separators
- Using a single unbalanced quote for character constants. For example, .byte 'a.

Generate Listing File

A listing file contains 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.

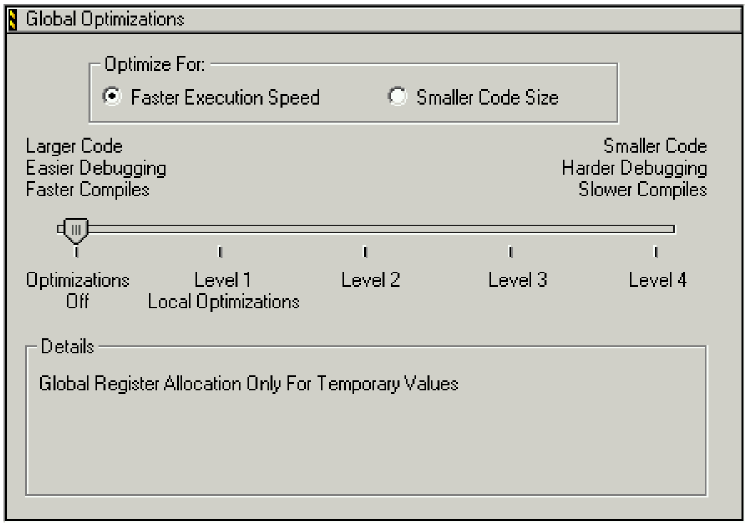
Prefix File

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.

Global Optimizations

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

Figure 5.5 Global Optimizations Settings Panel



Optimize For: ● Faster Execution Speed ○ Smaller Code Size

Details: Global Register Allocation Only For Temporary Values

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

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

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

The optimization levels are:

- **Optimizations Off** (level 0) consists of global register allocation (register coloring) only for temporary values.

NOTE To avoid ambiguity when debugging, set optimization level to **Optimizations Off**, which causes the compiler to use register coloring only for compiler-generated (temporary) variables.

- **Level 1** — dead code elimination and global register allocation
- **Level 2** — optimizations in **Level 1** plus common subexpression elimination and copy propagation; **Level 2** is best for most code
- **Level 3** — optimizations in **Level 2**, plus moving invariant expressions out of loops (also called Code Motion), strength reduction of induction variables, copy propagation, and loop transformation

  Level 3 is best for code with many loops
- **Level 4** — optimizations in **Level 3**, including performing some of them a second time for even greater code efficiency; **Level 4** can provide the best optimization for your code, but it takes more time to compile than with the other settings

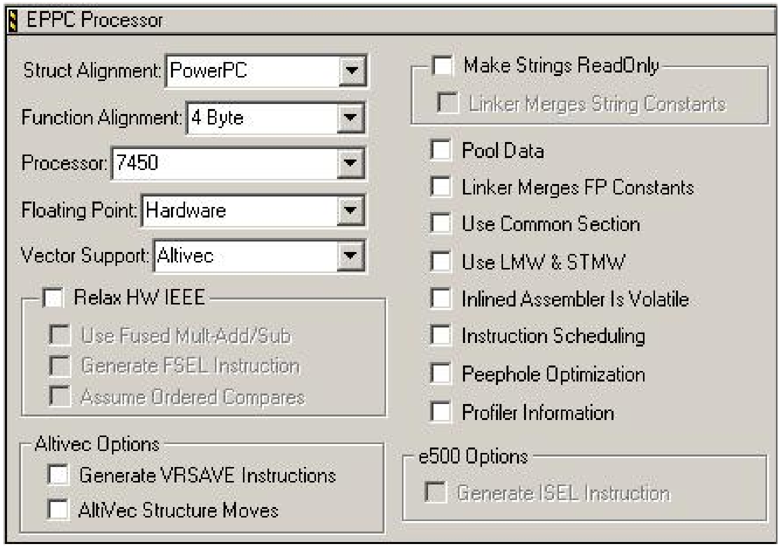
This option corresponds to `#pragma global_optimizer` and `#pragma optimization_level`.

NOTE Use compiler optimizations only after debugging your software. Using a debugger on an optimized program may affect the source code view that the debugger shows.

EPPC Processor

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

Figure 5.6 EPPC Processor Settings Panel



EPPC Processor

Struct Alignment: PowerPC
Function Alignment: 4 Byte
Processor: 7450
Floating Point: Hardware
Vector Support: Altivec

□ Relax HW IEEE
□ Use Fused Mult-Add/Sub
□ Generate FSEL Instruction
□ Assume Ordered Compares

Altivec Options
□ Generate VRSAVE Instructions
□ AltiVec Structure Moves

□ Make Strings ReadOnly
□ Linker Merges String Constants
□ Pool Data
□ Linker Merges FP Constants
□ Use Common Section
□ Use LMW & STMW
□ Inlined Assembler Is Volatile
□ Instruction Scheduling
□ Peephole Optimization
□ Profiler Information

e500 Options
□ Generate ISEL Instruction

Struct Alignment

The **Struct Alignment** listbox has the default selection **PowerPC**. To conform with the PowerPC EABI and inter-operate 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".

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** listbox to select alignments from 4 (the default) to 128 bytes. These selections corresponds to `#pragma function_align`. For more information, see "function_align".

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

Processor

Use the **Processor** listbox to specify the 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.
- 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** listbox 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** listbox
- include the corresponding runtime library in your project

  For example, if you select the **None** option, you also need to 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 button if you are targeting hardware without 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 **Vector Support** listbox. Currently, only the Altivec and SPE vector units are supported.

If you select the **Altivec** option from the **Vector Support** listbox, additional options appear in the Altivec Options area.

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** listbox. Check the 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 = (condition ? y : z)`, where `x` and `y` are floating point values.

FSEL is not accurate for denormalized numbers, and may have issues related to unorder 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)
```

into

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

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 VRSAVE Instructions

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

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

Check the **Generate VRSAVE Instructions** checkbox only when developing for a real-time operating system that supports AltiVec. Checking the **Generate VRSAVE 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.

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.

Use LMW & STMW

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

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

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

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

- `#pragma no_register_save_helpers on|off|reset`

  If this pragma is on, the compiler always inlines instructions.
- `#pragma use_lmw_stmw on|off|reset`

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

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

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

Inlined Assembler is Volatile

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

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

Instruction Scheduling

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

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

Peephole Optimization

Check the **Peephole Optimization** checkbox to have the compiler perform *peephole* optimizations. Peephole optimizations are small local optimizations that can reduce several instructions 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`.

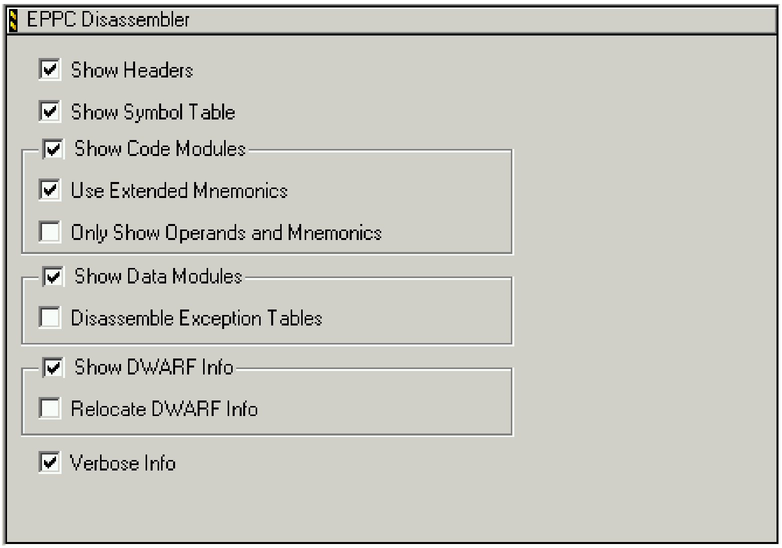
e500 Options

The **e500 Options** area is only available for the e500 family of processors, which are not supported in this product.

EPPC Disassembler

Use the **EPPC Disassembler** settings panel (Figure 5.7) to control the information displayed when you choose **Project > Disassemble** in the IDE.

Figure 5.7 EPPC Disassembler Settings Panel



EPPC Disassembler

☑ Show Headers

☑ Show Symbol Table

☑ Show Code Modules

- ☑ Use Extended Mnemonics
- □ Only Show Operands and Mnemonics

☑ Show Data Modules

- □ Disassemble Exception Tables

☑ Show DWARF Info

- □ Relocate DWARF Info

☑ Verbose Info

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

Show Headers

Check the **Show Headers** checkbox to have the 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.

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.

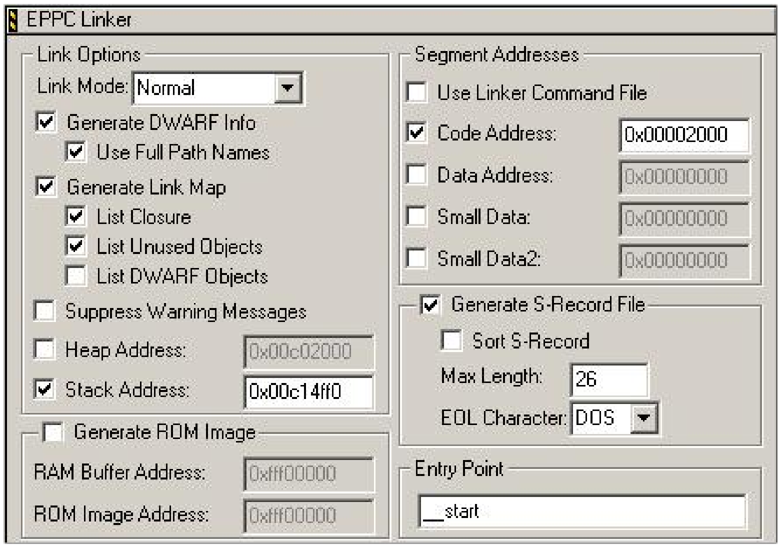
Verbose Info

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.

EPPC Linker

Use the **EPPC Linker** settings panel (Figure 5.8) to perform settings related to linking your object code into final form: executable file, library, or other type of code.

Figure 5.8 EPPC Linker Panel



EPPC Linker

Link Options

Link Mode: Normal

☑ Generate DWARF Info

☑ Use Full Path Names

☑ Generate Link Map

☑ List Closure

☑ List Unused Objects

☐ List DWARF Objects

☐ Suppress Warning Messages

☐ Heap Address: 0x00c02000

☑ Stack Address: 0x00c14ff0

☐ Generate ROM Image

RAM Buffer Address: 0xfff00000

ROM Image Address: 0xfff00000

Segment Addresses

☐ Use Linker Command File

☑ Code Address: 0x00002000

☐ Data Address: 0x00000000

☐ Small Data: 0x00000000

☐ Small Data2: 0x00000000

☑ Generate S-Record File

☐ Sort S-Record

Max Length: 26

EOL Character: DOS

Entry Point

__start

Link Mode

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

Use the **Link Mode** listbox to select the link mode. The link mode options are:

- Use Less RAM — In this link mode, the linker writes the output file directly to disk without using a buffer.
- Normal — In this link mode, the linker writes to a 512-byte buffer and then writes the buffer to disk. For most projects, this link mode is the best choice.
- Use More RAM **—** In this link mode, the linker writes each segment to its own buffer. When all segments have been written to their buffer, the buffers are flushed to the disk. This link mode is best suited to 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 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 Ma**p 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” 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 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 (k) is found in the EPPC Target panel). The possible addresses depend on your target hardware platform and how the memory is mapped. The heap must reside in RAM.

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

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

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

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.

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 (k) 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** checkbox if you wish to create a ROM image at link time by specifying the RAM Buffer Address and ROM Image Address.

NOTE A ROM image is defined as a file suitable for flashing to ROM.

RAM Buffer Address

The **RAM Buffer Address** text box specifies the address in RAM that is to be used as a buffer for the flash image programmer.

Enter an address value in this text box to load all code and data into consecutive addresses in ROM. Your application copies data, exception vectors, and possibly even code into their executing addresses.

The CodeWarrior flash programmer does not use a separate RAM buffer for flashing. If you are using the CodeWarrior flash programmer, make sure that the RAM Buffer Address equals the ROM Image Address.

NOTE Using the Flash to ROM target to flash your programs to ROM is substantially faster than using the flash programmer.

If you are not using the CodeWarrior flash programmer, some ROM flash programs, such as MPC8BUG for 821, expect to find your program in a RAM buffer in memory. This buffer address is different from where you want your program to execute. You enter the executing addresses for the different sections in the Code Address, Data Address, Small Data, and Small Data2 text boxes.

For example, MPC8BUG expects a RAM Buffer Address of `0x02800000`. MPC8BUG makes a copy of your program starting at address `0xFFE00000`. If `0xFFE00000` is where you want your `.text` section then you would put `0xFFE00000` as the Code Address. If you specify a different Code Address, you need to copy the code to that address from `0xFFE00000`. You also find linker-generated symbols for all ROM addresses and executing addresses for the sections to assist in copying them. For an explanation of linker generated symbols created for ROM addresses, see the file:

```
InstallDir\PowerPC_EABI_Support\Runtime\Inc\__ppc_eabi_linker.h
```

NOTE Not all flash programs require that you specify a buffer address. For example, MPC8BUG requires a buffer, but the CodeWarrior Flash Programmer does not. If you do not need a buffer, you *must* set the buffer address to be identical to the ROM Image Address.

ROM Image Address

The **ROM Image Address** text box specifies the address where you want your program to load in ROM.

Segment Address

Use the checkboxes in the **Segment Address** 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.

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

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

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

Small Data2

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

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

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

NOTE The CodeWarrior development tools create the three small data sections required by the PowerPC EABI specification. Further, the CodeWarrior tools let you define additional small data sections. See Additional Small Data Areas for instructions.

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

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

Entry Point

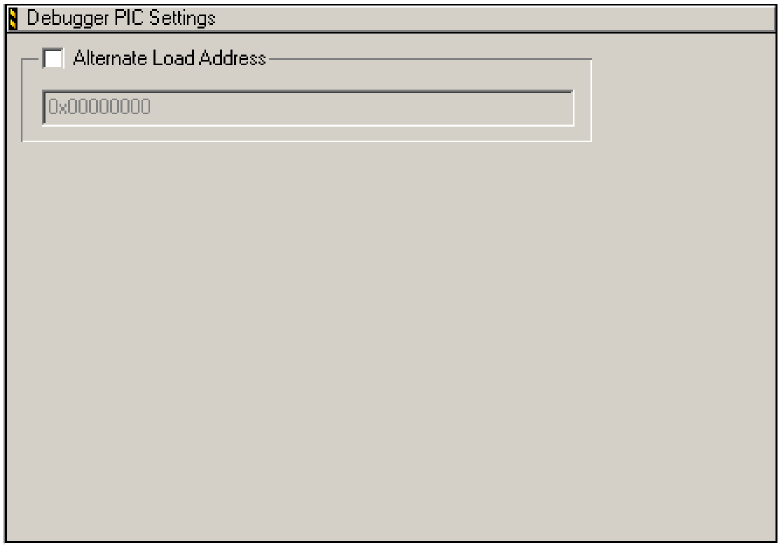
The **Entry Point** text box specifies the function that the linker uses first when the program launches. This is the starting point of the program.

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.

Debugger PIC Settings

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

Figure 5.9 Debugger PIC Settings Panel



Debugger PIC Settings

□ Alternate Load Address

0x00000000

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

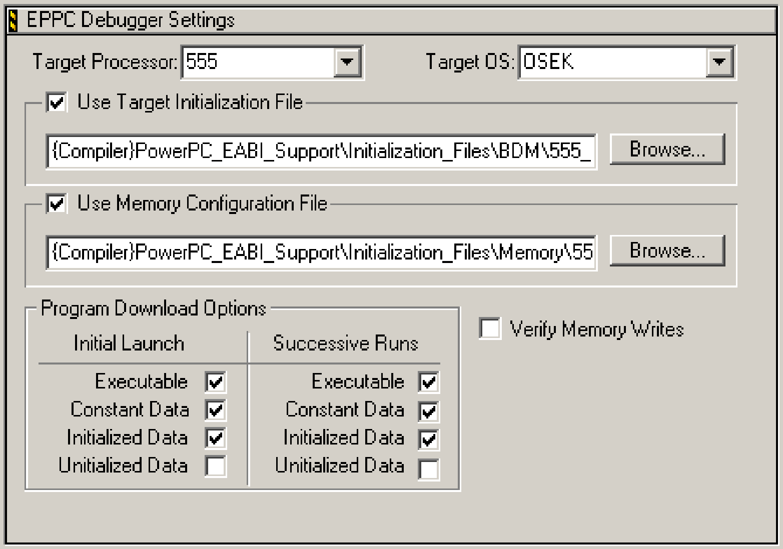
To specify the alternate load address, check the **Alternate Load Address** checkbox and enter the address in the associated text box. The debugger loads your ELF file on the target at the new address.

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

EPPC Debugger Settings

Use the **EPPC Debugger Settings** panel (Figure 5.10) to select the target processor.

Figure 5.10 EPPC Debugger Settings Panel



EPPC Debugger Settings

Target Processor: 555 Target OS: OSEK

☑ Use Target Initialization File

{Compiler}PowerPC_EABI_Support\Initialization_Files\BDM\555_

☑ Use Memory Configuration File

{Compiler}PowerPC_EABI_Support\Initialization_Files\Memory\55

Program Download Options

Initial Launch		Successive Runs	
Executable	☑	Executable	☑
Constant Data	☑	Constant Data	☑
Initialized Data	☑	Initialized Data	☑
Unitialized Data	☐	Unitialized Data	☐

☐ Verify Memory Writes

Target Processor

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

Target OS

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

- Bareboard

  Enables bareboard debugging.

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

- OSEK

  Enables OSEK Aware debugging.

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

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

Use Target Initialization File

Check this box 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` subdirectories of this path:

`*InstallDir*\PowerPC_EABI_Support\Initialization_Files`

Use Memory Configuration File

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

Samples of memory configuration file are here:

`*InstallDir*\PowerPC_EABI_Support\Initialization_Files\memory`

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

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

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

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.

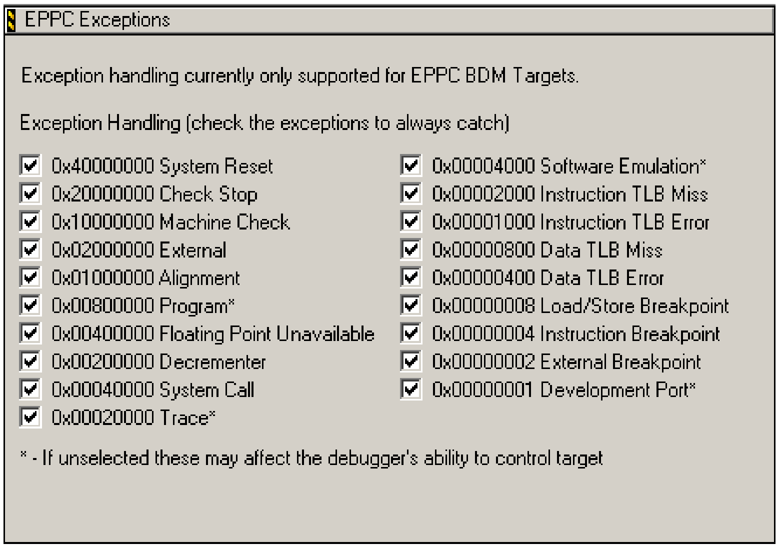
Verify Memory Writes

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.

EPPC Exceptions

The **EPPC Exceptions** settings panel (Figure 5.11) lists all the exceptions that the debugger is able to catch.

Figure 5.11 EPPC Exceptions Panel



EPPC Exceptions

Exception handling currently only supported for EPPC BDM Targets.

Exception Handling (check the exceptions to always catch)

☑ 0x40000000 System Reset
☑ 0x20000000 Check Stop
☑ 0x10000000 Machine Check
☑ 0x02000000 External
☑ 0x01000000 Alignment
☑ 0x00800000 Program*
☑ 0x00400000 Floating Point Unavailable
☑ 0x00200000 Decrementer
☑ 0x00040000 System Call
☑ 0x00020000 Trace*
☑ 0x00004000 Software Emulation*
☑ 0x00002000 Instruction TLB Miss
☑ 0x00001000 Instruction TLB Error
☑ 0x00000800 Data TLB Miss
☑ 0x00000400 Data TLB Error
☑ 0x00000008 Load/Store Breakpoint
☑ 0x00000004 Instruction Breakpoint
☑ 0x00000002 External Breakpoint
☑ 0x00000001 Development Port*

* - If unselected these may affect the debugger's ability to control target

NOTE The **EPPC Exceptions** panel is available for just the 5xx and 8xx series of processors.

Check the checkboxes of all the options in this panel if you want the debugger to catch all the exceptions. Leave the checkboxes cleared for those exceptions, which you prefer to handle. This panel is used to determine the value to which the Debug Enable Register (DER) sets the debugger. The DER controls which exceptions are caught or missed by the Background Debug Mode (BDM). Consult the user's guide of your processor for more information on the DER.

To ensure that the debugger performs properly, always select these exceptions:

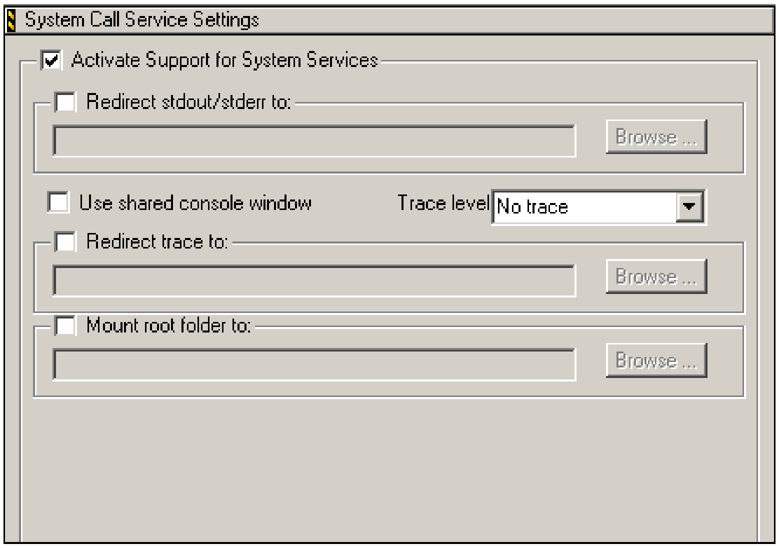
- 0x00800000 Program — for software breakpoints on some boards

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

System Call Service Settings

Use the **System Call Service Setting** panel (Figure 5.12) to activate support for system services and configure options for handling requests for system services.

Figure 5.12 System Call Service Setting Panel



System Call Service Settings

☑ Activate Support for System Services

□ Redirect stdout/stderr to:

□ Use shared console window

Trace level: No trace

□ Redirect trace to:

□ Mount root folder to:

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.

Trace Level

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

- No Trace — system calls are not traced
- Summary Trace — 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.

PC-lint Settings Panels

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

CodeWarrior Development Studio, MPC5xx Edition includes target settings panels and plug-ins that let you configure and use PC-lint from within the CodeWarrior IDE. However, the PC-lint software itself is *not* included with your CodeWarrior product. As a result, you must obtain and install a copy of PC-lint before you can use it with the CodeWarrior IDE. Among other places, PC-lint is available from its developers, Gimpel Software (`www.gimpel.com`).

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

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

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

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

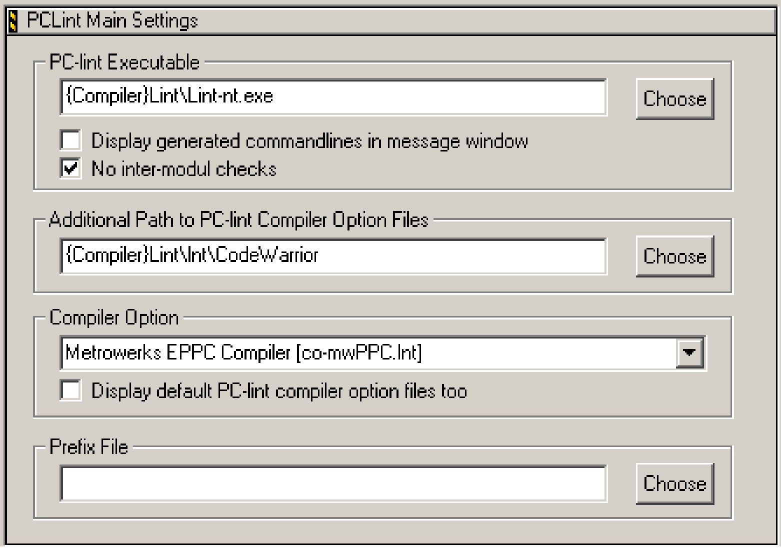
   The sections that follow explain how to use the PC-lint target settings panels.

PCLint Main Settings

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

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

Figure 5.13 PCLint Main Settings Target Settings Panel



{Compiler}Lint\Lint-nt.exe

□ Display generated commandlines in message window

☑ No inter-modul checks

{Compiler}Lint\lnt\CodeWarrior

Metrowerks EPPC Compiler [co-mwPPC.lnt]

□ Display default PC-lint compiler option files too

PC-lint Executable

Type the path to and name of the PC-lint executable file in this text box. Alternatively, click **Choose** to display a dialog box that lets you navigate to and select this file.

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

Display generated command lines in message window

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

No inter-modul checks

Check this box to instruct PC-lint to do *no* inter-module checking.

NOTE If you uncheck this box, PC-lint takes longer to process your build target's source files.

Additional Path to PC-lint Compiler Option Files

The IDE's default behavior is to use any PC-lint compiler option files (`*.lnt`) it finds in the directory `{Compiler}\Lint\lnt`.

To configure a build target to use a PC-lint compiler option file in addition to those in the default directory, enter the path to the directory that contains this file in the Additional Path to PC-lint Compiler Option Files text box. If the specified directory contains any files that end with the suffix `.lnt`, the Compiler Option listbox (see below) enables and displays these files.

The default CodeWarrior installation includes prewritten PC-lint compiler option files. They are in this directory:

```
{CodeWarrior}Lint\lnt\CodeWarrior
```

Each file in this directory is designed to work with a particular Metrowerks compiler. Many users enter this path in the Additional Path to PC-lint Compiler Option Files text box and then choose the file for the Metrowerks compiler they are using from the Compiler Option list.

You can leave this text box empty, if desired.

Compiler Option

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

This listbox displays all `.lnt` files in the directory specified in the Additional Path to PC-lint Compiler Option files text box. If this directory contains no `.lnt` files, the Compiler Option listbox is disabled.

Display default PC-lint compiler option files too

Check this box to include the default `.lnt` files (the files in `{Compiler}Lint\lnt`) in the Compiler Option listbox along with those in the directory specified in the Additional Path to PC-lint Compiler Option Files text box.

Prefix File

Type the name of a prefix file to pass to PC-lint. Alternatively, click **Choose** to display a dialog box that lets you navigate to and select this file.

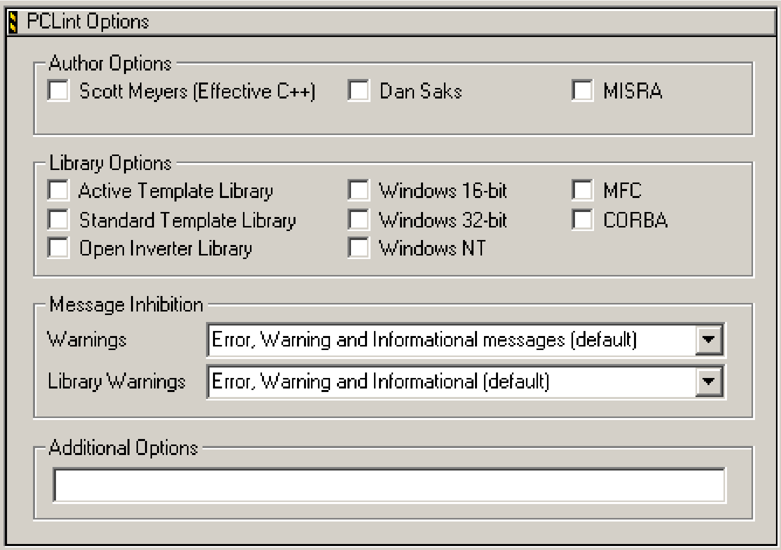
Typically, you use this feature to define macros to required values for a particular PC-lint run or to instruct PC-lint to check certain command-line commands. To do this, define this information in a prefix file.

You can leave this text box empty, if desired.

PCLint Options

Figure 5.14 shows the **PCLint Options** target settings panel. Use this panel to define the syntax rules PC-lint uses to validate your C/C++ source, to define the environment (libraries, operating system, remote procedure call standard, etc.) with which PC-lint must ensure your code conforms, and to pass command-line switches to PC-lint.

Figure 5.14 PC-lint Options Target Settings Panel



PCLint Options

Author Options

□ Scott Meyers (Effective C++) □ Dan Saks □ MISRA

Library Options

□ Active Template Library □ Windows 16-bit □ MFC
□ Standard Template Library □ Windows 32-bit □ CORBA
□ Open Inverter Library □ Windows NT

Message Inhibition

Warnings: Error, Warning and Informational messages (default)
Library Warnings: Error, Warning and Informational (default)

Additional Options

Author Options

This group of checkboxes lets you select the set of syntax rules that PC-lint uses as it checks your code. The options are:

- Scott Meyers (Effective C++)

  Check this box to instruct PC-lint to verify that your code adheres to the syntax rules documented in Effective C++.

- Dan Saks

  Check this box to instruct PC-lint to verify that your code adheres to the syntax rules recommended by Dan Saks.

- MISRA

  Check this box to instruct PC-lint to verify that your code adheres to the Motor Industry Software Reliability Association (MISRA) C language guidelines for safety-critical embedded software.

You can check none, some, or all boxes in this group.

Library Options

This group of checkboxes lets you define the environment with which PC-lint must ensure your code conforms. The options are:

- Active Template Library

  Check this box to instruct PC-lint to validate your Active X Template (ATL) library code.

- Standard Template Library

  Check this box to instruct PC-lint to validate your Standard Template Library (STL) code.

- Open Inverter Library

  Check this box to instruct PC-lint to validate your Open Inverter Library code.

- Windows 16-bit

  Check this box to instruct PC-lint to validate your 16-bit Windows API calls.

- Windows 16-bit

  Check this box to instruct PC-lint to validate your 32-bit Windows API calls.

- Windows NT

  Check this box to instruct PC-lint to validate your Windows NT API calls.

- MFC

  Check this box to instruct PC-lint to validate your Microsoft Foundation Classes (MFC) code.

- CORBA

  Check this box to instruct PC-lint to validate your Common Object Request Broker Architecture (CORBA) code.

Warnings

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

The default setting displays error, warning and information messages.

Library Warnings

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

The default setting displays error, warning and information messages.

Additional Options

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

6

Embedded PowerPC Debugging

This chapter explains how to use the CodeWarrior tools for debugging embedded PowerPC (EPPC) programs. The chapter covers those aspects of debugging that are specific to the Embedded PowerPC platform. See the *IDE User Guide* for more general information about the debugger.

This chapter has these topics:

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

Supported Remote Connections for Debugging

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

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

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

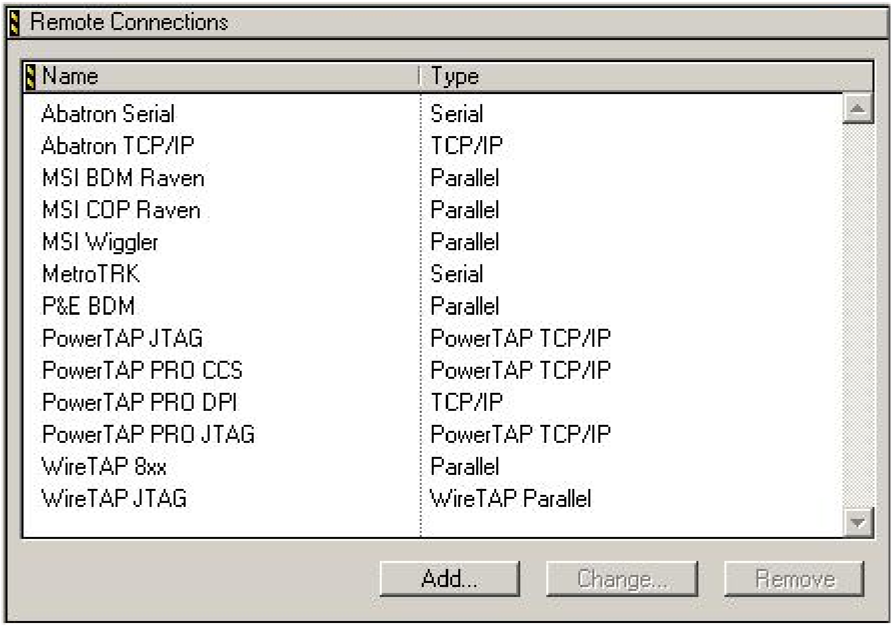
To specify the settings for a remote connection:

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

a. Select **Edit > Preferences**. The **IDE Preferences** window appears.

b. Select the **Remote Connections** item in the **IDE Preference Panels** list. The **Remote Connections panel (Figure 6.1) appears.**

Figure 6.1 Remote Connections Panel



Remote Connections

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

2. Select the remote connection name.

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

   b. Click **Change. A dialog box where you can specify the connection settings appears.**

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

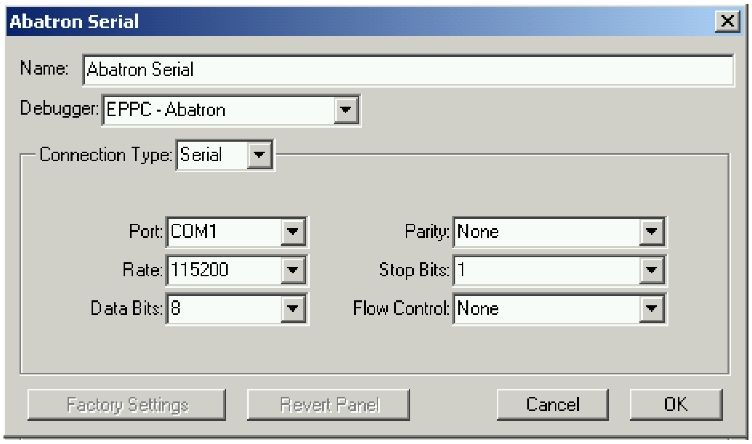
The other remote connection settings for each remote connection included in the default installation are described in these sections:

- Abatron Remote Connections
- MSI BDM Raven/MSI COP Raven/MSI Wiggler Remote Connection
- MetroTRK Remote Connection
- P&E BDM Remote Connection

Abatron Remote Connections

Figure 6.2 shows the dialog box where you specify the connection settings for the serial type Abatron remote connection.

Figure 6.2 Serial Type Abatron Remote Connection



Abatron Serial

Name: Abatron Serial

Debugger: EPPC - Abatron

Connection Type: Serial

Port: COM1

Rate: 115200

Data Bits: 8

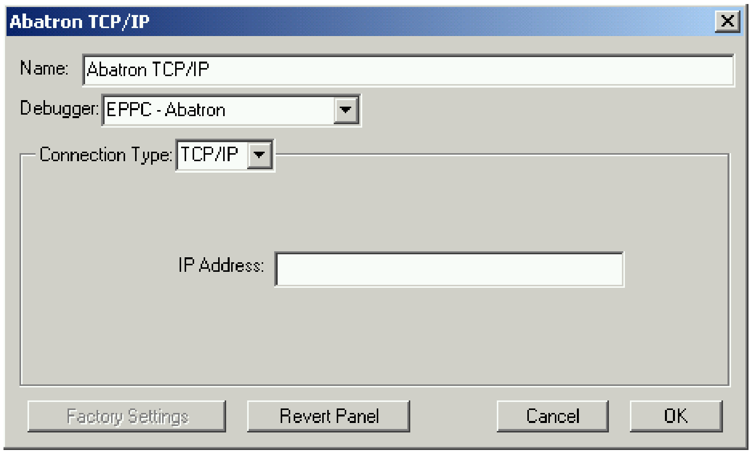
Parity: None

Stop Bits: 1

Flow Control: None

Figure 6.3 shows the dialog box where you specify the connection settings for the TCP/IP type Abatron remote connection.

Figure 6.3 TCP/IP Type Abatron Remote Connection



Abatron TCP/IP

Name: Abatron TCP/IP

Debugger: EPPC - Abatron

Connection Type: TCP/IP

IP Address:

IP Address

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

Port

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

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

Rate

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

Data Bits

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

Parity

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

Stop Bits

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

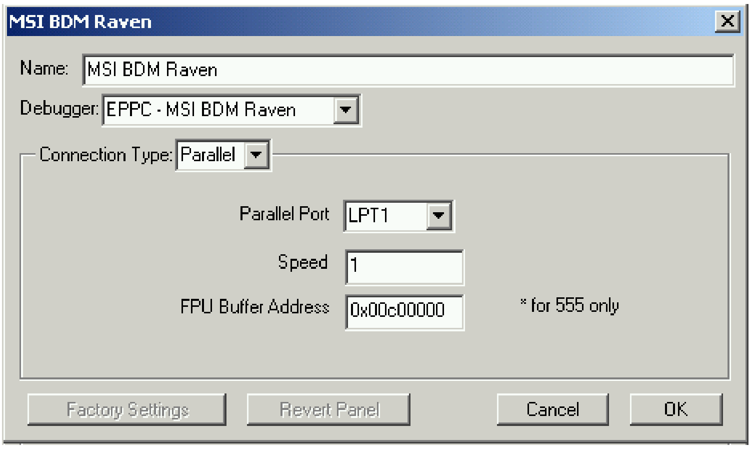
Flow Control

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

MSI BDM Raven/MSI COP Raven/MSI Wiggler Remote Connection

The remote connection settings for BDM Raven, COP Raven, and Wiggler are the same. Figure 6.4 shows the dialog box where you specify the connection settings for the MSI BDM Raven remote connection.

Figure 6.4 MSI BDM Raven/MSI COP Raven/MSI Wiggler Remote Connection



MSI BDM Raven

Name: MSI BDM Raven

Debugger: EPPC - MSI BDM Raven

Connection Type: Parallel

Parallel Port: LPT1

Speed: 1

FPU Buffer Address: 0x00c00000 * for 555 only

Parallel Port

Select the parallel port.

Speed

Use the default speed of 1.

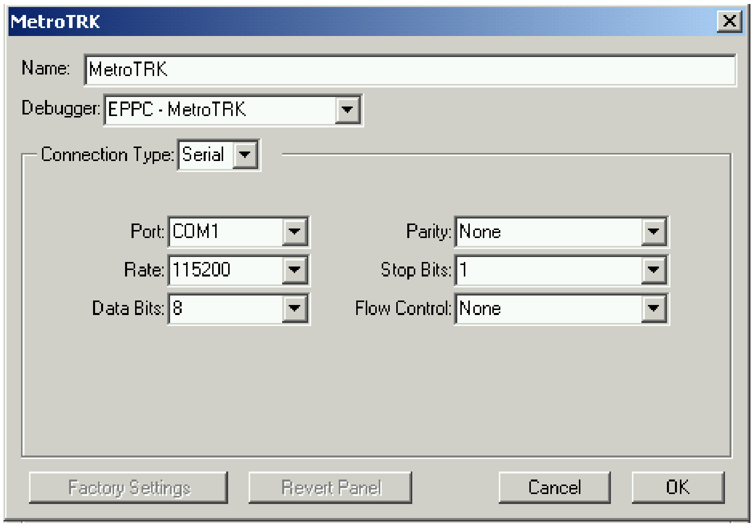
FPU Buffer Address

For the 555 processor only, you must specify a valid buffer address that the MSI device can use.

MetroTRK Remote Connection

Figure 6.5 shows the dialog box where you specify the connection settings for the MetroTRK remote connection.

Figure 6.5 MetroTRK Remote Connection



MetroTRK

Name: MetroTRK

Debugger: EPPC - MetroTRK

Connection Type: Serial

Port: COM1

Rate: 115200

Data Bits: 8

Parity: None

Stop Bits: 1

Flow Control: None

Port

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

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

Rate

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

Data Bits

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

Parity

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

Stop Bits

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

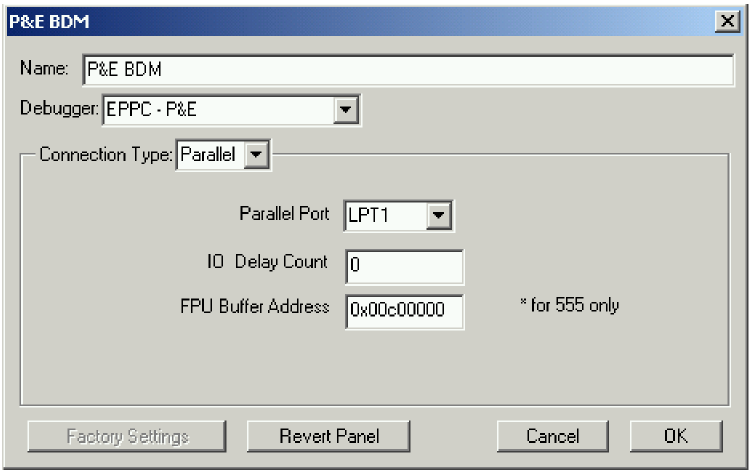
Flow Control

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

P&E BDM Remote Connection

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

Figure 6.6 P&E BDM Remote Connection



P&E BDM

Name: P&E BDM

Debugger: EPPC - P&E

Connection Type: Parallel

Parallel Port: LPT1

IO Delay Count: 0

FPU Buffer Address: 0x00c00000 * for 555 only

Parallel Port

Select the parallel port.

IO Delay Count

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

FPU Buffer Address

For the 555 processor only, you must specify a valid address that the P&E BDM device can use as a buffer.

Special Debugger Features

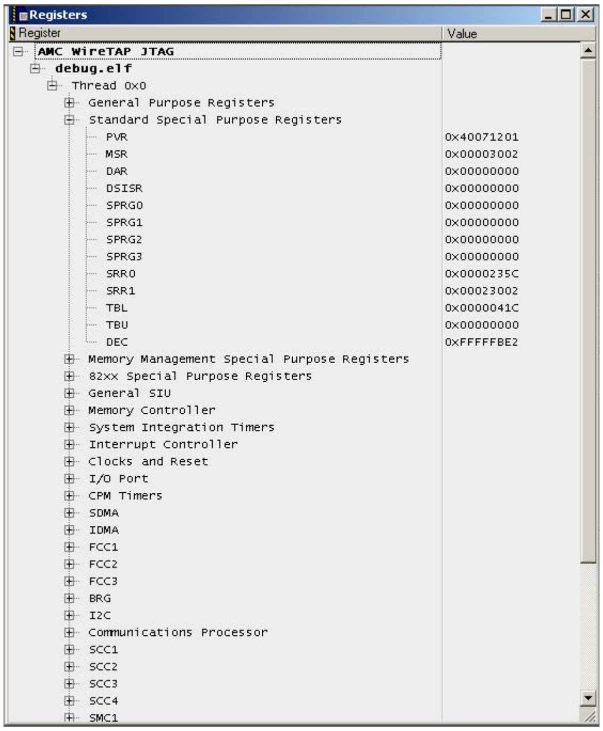
This section explains debugger features that are not found in the *IDE User Guide*. These features are unique to this platform target and enhance the debugger especially for Embedded PowerPC development.

- Displaying Registers
- EPPC Menu
- Register Details

Displaying Registers

Select **View > Registers** to display the **Registers** window (Figure 6.7).

Figure 6.7 Registers Window

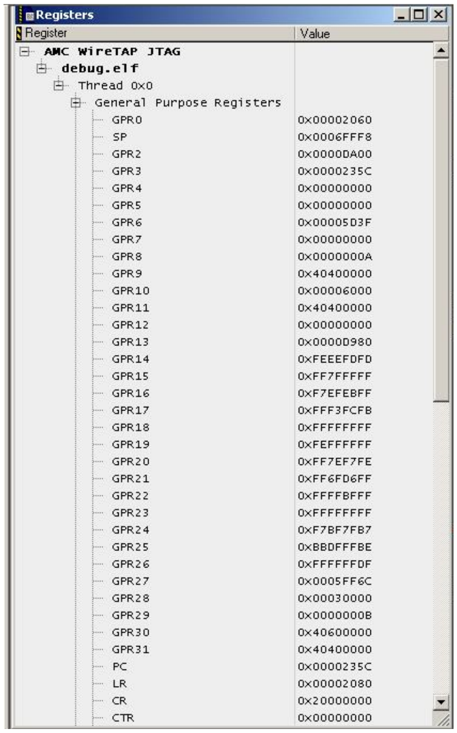


```
Registers
Register                                               Value
AMC WireTAP JTAG
  debug.elf
    Thread 0x0
      General Purpose Registers
      Standard Special Purpose Registers
        PVR                                            0x40071201
        MSR                                            0x00003002
        DAR                                            0x00000000
        DSISR                                          0x00000000
        SPRG0                                          0x00000000
        SPRG1                                          0x00000000
        SPRG2                                          0x00000000
        SPRG3                                          0x00000000
        SRR0                                           0x0000235C
        SRR1                                           0x00023002
        TBL                                            0x0000041C
        TBU                                            0x00000000
        DEC                                            0xFFFFFBE2
      Memory Management Special Purpose Registers
      82xx Special Purpose Registers
      General SIU
      Memory Controller
      System Integration Timers
      Interrupt Controller
      Clocks and Reset
      I/O Port
      CPM Timers
      SDMA
      IDMA
      FCC1
      FCC2
      FCC3
      BRG
      I2C
      Communications Processor
      SCC1
      SCC2
      SCC3
      SCC4
      SMC1
```

Expand a control tree to view a particular set of registers. For example, if you want to view general purpose registers, expand the **General Purpose Registers** control tree.

The general purpose registers are displayed, as shown in Figure 6.8.

Figure 6.8 General Purpose Registers



Register	Value
AMC WireTAP JTAG	
debug.elf	
Thread 0x0	
General Purpose Registers	
GPR0	0x00002060
SP	0x0006FFF8
GPR2	0x0000DA00
GPR3	0x0000235C
GPR4	0x00000000
GPR5	0x00000000
GPR6	0x00005D3F
GPR7	0x00000000
GPR8	0x0000000A
GPR9	0x40400000
GPR10	0x00006000
GPR11	0x40400000
GPR12	0x00000000
GPR13	0x0000D980
GPR14	0xFEEEFDFD
GPR15	0xFF7FFFFF
GPR16	0xF7EFEBFF
GPR17	0xFFF3FCFB
GPR18	0xFFFFFFFF
GPR19	0xFEFFFFFF
GPR20	0xFF7EF7FE
GPR21	0xFF6FD6FF
GPR22	0xFFFFBFFF
GPR23	0xFFFFFFFF
GPR24	0xF7BF7FB7
GPR25	0xBBDFFFBE
GPR26	0xFFFFFFDF
GPR27	0x0005FF6C
GPR28	0x00030000
GPR29	0x0000000B
GPR30	0x40600000
GPR31	0x40400000
PC	0x0000235C
LR	0x00002080
CR	0x20000000
CTR	0x00000000

EPPC Menu

When you use the debugger with CodeWarrior for Embedded PowerPC, the debugger provides the **EPPC** menu that is unique to this product. To see the menu, select **Debug > EPPC**.

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

Select the **Change IMMR** command to set the IMMR address when debugging for the 825x/826x processors.

NOTE The **Change IMMR** command is available only after you select 825x/826x as the target processor.

Soft Reset

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

NOTE The **Soft Reset** command is optional. It is only available if the debug option supports it.

Hard Reset

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

NOTE The **Hard Reset** menu option is optional. It is only available if the debug option supports it.

Load/Save Memory

The **Load/Save Memory** command lets you read data of user-specified size from a binary file and write it to a particular memory location on the target. For more information, see `Load_Save_Memory_Notes.txt`. This file is here:

`InstallDir``\Release Notes\Embedded_PowerPC\`
`CodeWarrior_Tools\Memory_Save_Load_Fill_Notes`

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. For more information, see `Fill_Memory_Notes.txt`. This file is here:

```
InstallDir\Release Notes\
Embedded_PowerPC\CodeWarrior_Tools\Memory_Save_Load_Fill_Notes
```

Save/Restore Registers

The **Save/Restore Registers** command lets you save groups of registers into a text file. You can specify which groups of registers need to be saved. This command also lets you restore the text file to which the groups of registers have been saved.

Watchpoint Type

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

- **Read**

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

  Program execution stops at the watchpoint when your program writes to memory at the watch address.
- **Read/Write**

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

NOTE The **Watchpoint Type** command is available if both the processor and debug connection 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** menu option is available if both the processor and debug connection support it.

Setting Hardware Breakpoints

To set a hardware breakpoint:

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

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

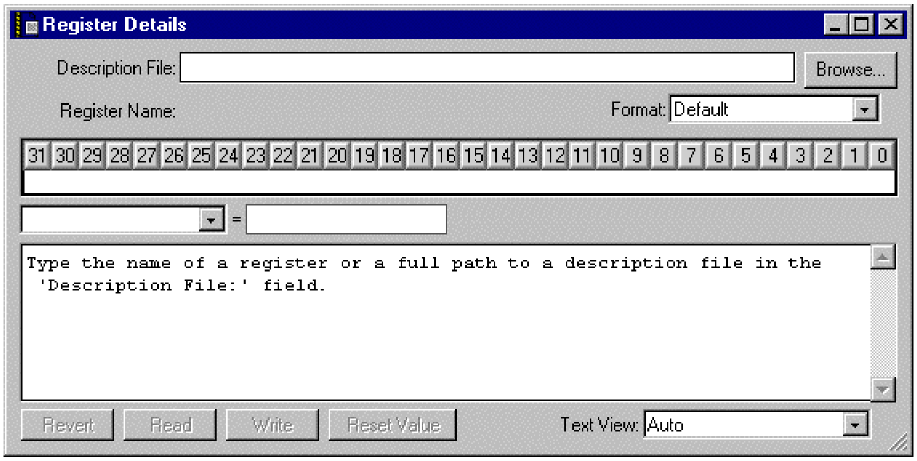
Table 6.1 Hardware Breakpoints Supported by PowerPC Processors

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	2
555, 565, 8xx	4

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 6.9).

Figure 6.9 Register Details Dialog Box



Description File:

Register Name:

Format: Default

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

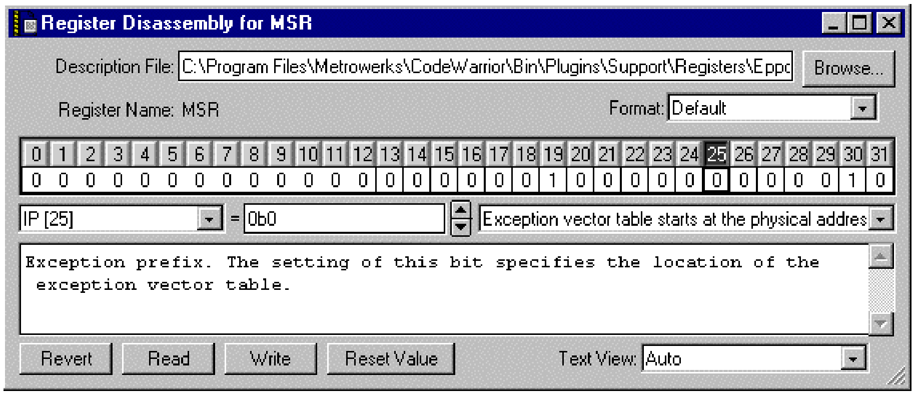
```
Type the name of a register or a full path to a description file in the
 'Description File:' field.
```

Text View: Auto

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 6.10 shows the **Register Details** dialog box displaying the **MSR** register.

Figure 6.10 Register Details Dialog Box Showing the MSR Register



Description File: C:\Program Files\Metrowerks\CodeWarrior\Bin\Plugins\Support\Registers\Eppc

Register Name: MSR

Format: Default

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0

IP [25] = 0b0 Exception vector table starts at the physical addres

```
Exception prefix. The setting of this bit specifies the location of the
 exception vector table.
```

Text View: Auto

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

NOTE For more information, see *IDE User Guide*.

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 the MetroTRK source code are here:

```
InstallDir\PowerPC_EABI_Tools\MetroTRK\Processor\ppc\Board
```

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

To modify a version of MetroTRK, find an existing MetroTRK project for your supported target board. You either can make a copy of the project (and its associated source files) or you can directly edit the originals. If you edit the originals, you always can revert back to the original version on your CodeWarrior CD.

Connecting to the MetroTRK Debug Monitor

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

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

Table 6.2 Serial Cable Connection Type for Target Boards

EPPC Board	Serial Cable Connection Type
Axiom 555, 565	Straight serial
Motorola 555 ETAS	Null modem
phyCORE single-board computer subassembly for MPC5xx family	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 6.2 have self-flashable MetroTRK project targets, except these boards:

   – Cogent CMA102 with CMA 278 Daughtercard

   – Motorola Maximer 7400

   – Motorola 5100 Ice Cube

   – 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.

MetroTRK Memory Configuration

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

This section contains these topics:

- Locations of MetroTRK RAM sections
- MetroTRK Memory Map

Locations of MetroTRK RAM sections

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

This section contains these topics:

- Exception Vectors
- Data and Code Sections
- The Stack

Exception Vectors

For a ROM-based MetroTRK, the MetroTRK initialization process copies the exception vectors from ROM to RAM.

NOTE For the MPC555 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 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.

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

Debugging ELF Files

You can use the CodeWarrior debugger to debug an ELF file that you previously created and compiled in a different environment than the CodeWarrior IDE. Before you open the ELF file for debugging, you must examine some IDE preferences and change them if needed. In addition, you must customize the default XML project file with appropriate target settings. The CodeWarrior IDE uses the XML file to create a project with the same target settings for any ELF file that you open to debug.

This section contains these topics:

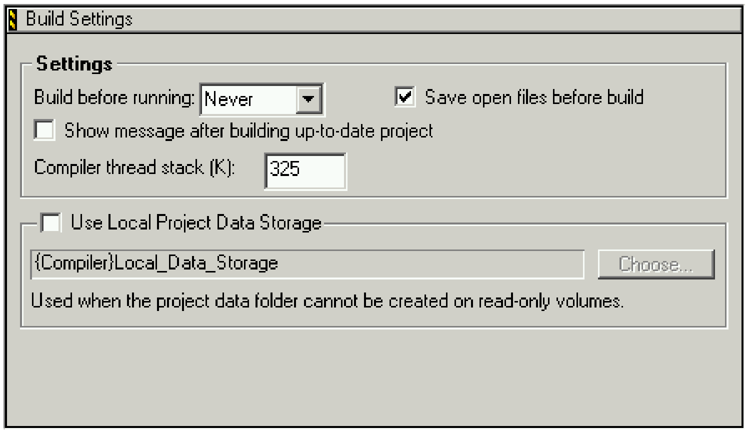
- Preparing to Debug an ELF File
- Customizing the Default XML Project File
- Debugging an ELF File
- ELF File Debugging: Additional Considerations

Preparing to Debug an ELF File

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

1. Select **Edit > Preferences**. The **IDE Preferences** window appears.
2. In the **IDE Preference Panels** list, click the **Build Settings** item. The **Build Settings** panel (Figure 6.11) appears.

Figure 6.11 Build Settings Panel



Build Settings

Settings

Build before running: Never

☑ Save open files before build

□ Show message after building up-to-date project

Compiler thread stack (K): 325

□ Use Local Project Data Storage

{Compiler}Local_Data_Storage

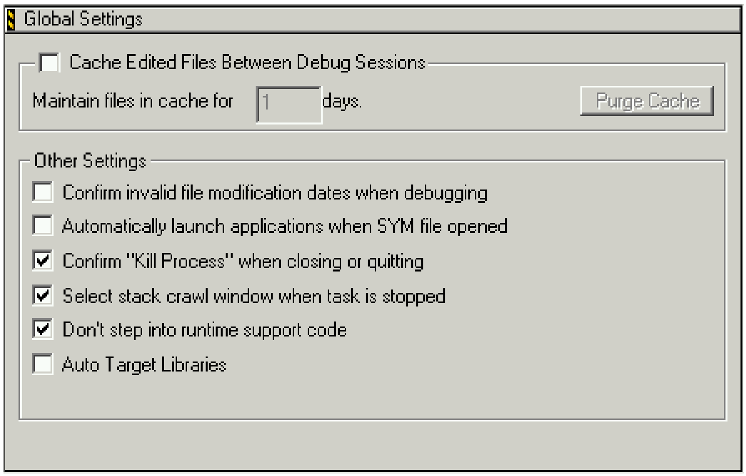
Used when the project data folder cannot be created on read-only volumes.

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

NOTE Selecting **Never** prevents the IDE from building the newly created project, which is useful if you prefer to use a different compiler.

4. In the **IDE Preference Panels** list, click the **Global Settings** item. The **Global Settings** panel (Figure 6.12) appears.

Figure 6.12 Global Settings Panel



Global Settings

☐ Cache Edited Files Between Debug Sessions

Maintain files in cache for 1 days.

Other Settings

☐ Confirm invalid file modification dates when debugging

☐ Automatically launch applications when SYM file opened

☑ Confirm "Kill Process" when closing or quitting

☑ Select stack crawl window when task is stopped

☑ Don't step into runtime support code

☐ Auto Target Libraries

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.

```
InstallDir\bin\Plugins\Support\
PowerPC_EABI\EPPC_Default_Project.XML
```

You must import the default XML project file, adjust the target settings of the new project, and export the changed project back to the original default XML project file. The CodeWarrior software then uses the changed XML file to create projects for any ELF files that you open to debug.

NOTE The CodeWarrior software overwrites the existing `EPPC_Default_Project.XML` file if you customize it again for a different target board or debugging setup. If you want to preserve the file that you originally customized for later use, rename it or save it in another directory.

To customize the default XML project file:

1. Import the default XML project file.
    a. Select **File > Import Project**.
    b. Navigate to this location in the CodeWarrior installation directory: `bin\Plugins\Support\PowerPC_EABI\`
    c. Select the `EPPC_Default_Project.XML` file name.
    d. Click **OK.** The CodeWarrior software displays a new project based on `EPPC_Default_Project.XML`.
2. Change the target settings of the new project.

    Select **Edit > Target Settings** to display the **Target Settings** window. In this window, you can change the target settings of the new project as per the requirements of your target board and debugging devices.
3. Export the new project with its changed target settings.

    Export the new project back to the original default XML project file (`EPPC_Default_Project.XML`) by selecting **File > Export Project** and saving the new XML file over the old one.

    The new `EPPC_Default_Project.XML` file reflects any target settings changes that you made. Any projects that the CodeWarrior software creates when you open an ELF file to debug use those target settings.

Debugging an ELF File

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

To debug an ELF file:

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

    The CodeWarrior software creates a new project using the previously customized default XML project file. The CodeWarrior software bases the name of the new project on the name of the ELF file. For example, an ELF file named `cw.ELF` results in a project named `cw.mcp`.

    The symbolics in the ELF file specify the files in the project and their paths. Therefore, the ELF file must include the full path to the files.

    The DWARF information in the ELF file does not contain full path names for assembly (.s) files. Therefore, the CodeWarrior software cannot find them when creating the project. However, when you debug the project, the CodeWarrior software finds and uses the assembly files if the files reside in a directory that is an access path in the project. If not, you can add the directory to the project, after which the CodeWarrior software finds the directory whenever you open the project. You can add access paths for any other missing files to the project as well.

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

   Select **Project > Debug**.

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

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

ELF File Debugging: Additional Considerations

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

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

Deleting old access paths from an ELF-created project

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

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

Removing files from an ELF-created project

After you create a project to allow debugging an ELF file, you may later delete one or more files from the ELF project. However, if you open the project again after rebuilding the ELF file, the CodeWarrior software does not automatically remove the deleted files from the corresponding project. For the project to include only the current files, you must manually delete the files that no longer apply to the ELF file from the project.

Recreating an ELF-created project

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

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

7

C/C++ Compiler and Linker

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

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

This chapter contains these topics:

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

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

Table 7.1 Other Compiler and Linker Documentation

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

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

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

Integer and Floating-Point Formats

This section describes how the CodeWarrior C/C++ compilers implement integer and floating-point types for Embedded PowerPC processors. You also can read `limits.h` for more information on integer types, and `float.h` for more information on floating-point types. The `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 7.2 shows the size and range of the integer types for the Embedded PowerPC compiler.

Table 7.2 PowerPC Integer Types

For this type	Option setting	Size	Range
bool	n/a	8 bits	true or false

Table 7.2 PowerPC Integer Types

For this type	Option setting	Size	Range
char	Use Unsigned Chars is off (see language preferences panel in the “C Compilers Guide.”)	8 bits	-128 to 127
	Use Unsigned Chars is on	8 bits	0 to 255
signed char	n/a	8 bits	-128 to 127
unsigned char	n/a	8 bits	0 to 255
short	n/a	16 bits	-32,768 to 32,767
unsigned short	n/a	16 bits	0 to 65,535
int	n/a	32 bits	-2,147,483,648 to 2,147,483,647
unsigned int	n/a	32 bits	0 to 4,294,967,295
long	n/a	32 bits	-2,147,483,648 to 2,147,483,647
unsigned long	n/a	32 bits	0 to 4,294,967,295
long long	n/a	64 bits	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long long	n/a	64 bits	0 to 18,446,744,073,709,551,615

Embedded PowerPC Floating-Point Formats

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

Table 7.3 PowerPC Floating-Point Data Types

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

AltiVec Vector Data Formats

There are 11 new vector data types for use in writing AltiVec-specific code, shown in Table 7.4. 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 7.4 AltiVec Vector Data Types

Vector Data Type	Size (bytes)	Contents	Possible Values
vector unsigned char	16	16 unsigned char	0 to 255
vector signed char	16	16 signed char	-128 to 127
vector bool char	16	16 unsigned char	0 = false, 1 = true
vector unsigned short [int]	16	8 unsigned short	0 to 65535
vector signed short [int]	16	8 signed short	-32768 to 32767
vector bool short [int]	16	8 unsigned short	0 = false, 1 = true
vector unsigned long [int]	16	4 unsigned int	0 to 2^{32} - 1
vector signed long [int]	16	4 signed int	-2^{31} to 2^{31}-1
vector bool long [int]	16	4 unsigned int	0 = false, 1 = true
vector float	16	4 float	any IEEE-754 value
vector pixel	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.

Data Addressing

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

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

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

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

becomes something similar to:

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

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

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

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

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

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

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

The following example has a zero small data threshold.

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

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

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

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

Register Variables

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

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

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

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

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

Register Coloring Optimization

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

Listing 7.1 Register coloring example

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

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

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

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

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

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

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

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

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

Pragmas

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

Table 7.5 lists the pragmas documented in the *C Compilers Reference* that are supported by all Metrowerks PowerPC C/C++ compilers.

Table 7.5 Pragmas Supported by All PPC Compilers—See *C Compilers Reference*

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

Table 7.5 Pragmas Supported by All PPC Compilers—See *C Compilers Reference*

warn_unusedarg	warn_unusedvar
wchar_type	

Table 7.6 lists the pragmas documented in this manual that are supported by all Metrowerks PowerPC C/C++ compilers.

Table 7.6 Pragmas Supported by All PPC Compilers—Documented in this Manual

opt_full_unroll_limit	opt_findoptimalunrollfactor
opt_unroll_count	opt_unrollpostloop
opt_unroll_instr_count	inline_max_auto_size
ppc_no_fp_blockmove	

opt_full_unroll_limit

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

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

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

opt_findoptimalunrollfactor

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

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

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

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

opt_unroll_count

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

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

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

opt_unrollpostloop

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

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

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

opt_unroll_instr_count

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

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

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

inline_max_auto_size

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

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

This pragma is ignored if auto-inlining is disabled.

ppc_no_fp_blockmove

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

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

Table 7.7 lists the pragmas supported by just Metrowerks C/C++ compilers for embedded PowerPC systems.

Table 7.7 Pragmas Supported by Just Embedded PowerPC Compilers

force_active	function_align
incompatible_return_small_structs	incompatible_sfpe_double_params
interrupt	pack
pooled_data	section

force_active

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

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

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

function_align

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

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

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

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

incompatible_return_small_structs

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

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

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

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

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

incompatible_sfpe_double_params

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

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

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

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

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

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

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

interrupt

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

This pragma allows you to 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 may put an `interupt_routine` at the site of the interrupt vector and the `interupt_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.

pack

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

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

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

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

For example,

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

would be smaller if rewritten as:

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

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

pooled_data

`#pragma pooled_data on | off | reset`

This pragma changes the state of pooled data.

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

section

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

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

- Parameters
- Section access permissions
- Predefined sections and default sections

- Forms for #pragma section
- Forcing individual objects into specific sections
- Using #pragma section with #pragma push and #pragma pop

Parameters

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

- `code_type` — executable object code
- `data_type` — non-constant data of a size greater than the size specified in the small data threshold option in the **PowerPC EABI Project** 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 **PowerPC EABI Project** settings panel
- `const_type` — constant data of a size greater than the size specified in the small const data threshold option in the **PowerPC EABI Project** 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 **PowerPC EABI Project** settings panel
- `all_types` — all code and data

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

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

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

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

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

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

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

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

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

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

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

Listing 7.2 Storing Uninitialized Data in the COMM Section

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

int red;
int sky;

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

You may not use any of the object types, data modes, or code modes as the names of sections. Also, you may not use pre-defined section names in 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 may only be used with the "`.sdata`", "`.sbss`", "`.sdata2`", "`.sbss2`", "`.EMB.PPC.sdata0`", and "`.EMB.PPC.sbss0`" sections.

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

Specify one of these addressing modes without quotes.

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

The permissible addressing modes for *codemode* are:

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

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

Specify one of these addressing modes without quotes.

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

Section access permissions

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

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

Predefined sections and default sections

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

The compiler predefines the sections in Listing 7.3.

Listing 7.3 Predefined sections

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

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

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

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

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

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

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

NOTE The .EMB.PPC.sdata0 and .EMB.PPC.sbss0 sections are predefined as an alternative to the sdata_type object type. The

`.init` section is also predefined, but it is not a default section. The `.init` section is used for startup code.

Forms for #pragma section

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

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

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

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

`#pragma section `*`objecttype`*

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

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

Forcing individual objects into specific sections

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

Listing 7.4 shows examples.

Listing 7.4 Using __declspec to Force Objects into Specific Sections

```
__declspec(section ".data") extern int myVar;

#pragma section "constants"

__declspec(section "constants") const int myConst = 0x12345678;
```

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

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

See Listing 7.2 for an example.

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

EPPC Linker Issues

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

The topics in this section are:

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

Additional Small Data Areas

The PowerPC EABI specification mandates that compliant build tools predefine three small data sections. The EPPC Linker target settings panel lets you specify the address at which the CodeWarrior linker puts two of these sections (if the default locations are unsatisfactory).

The CodeWarrior Development Studio, MPC5xx Edition lets you create small data sections in addition to those mandated by the PowerPC EABI specification. The CodeWarrior tools let you specify that the contents of a given user-defined section will be accessed by the small data base register selected from the available non-volatile registers. To do this, you use a combination of source code statements and linker command file directives.

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

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

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

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

   a. Add a statement that creates a global register variable.

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

      ```
      // _dummy does not have to be defined
      extern int _dummy asm("r14");
      ```

   b. Turn off the “unsafe global register variables” warning using this pragma:

      ```
      #pragma unsafe_global_reg_vars off
      ```

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

      For example:

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

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

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

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

7. Save the prefix file and close the editor window.

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

   For example, the statement:

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

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

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

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

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

   The **EPPC Linker** target settings panel appears.

10. In the Segment Addresses group box, check the Use Linker Command File checkbox.

    The other checkboxes and text boxes in the group become disabled.

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

    The **EPPC Target** settings panel appears.

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

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

15. Modify the linker command file such that it instructs the linker to use the global register declared above as the base register for your new small data section.

    To do this, follow these steps:

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

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

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

    b. Add the linker command file to each build target in which you want to use the new small data section.

16. Open the CodeWarrior project for the runtime library used by your project. The runtime library project is here:

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

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

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

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

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

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

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

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

    The IDE builds a new runtime library.

28. Close the runtime library project.
29. Return to your main project.
30. Press **F7**

    The IDE builds your project.

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

NOTE You can create more small data segments by following the procedure above. Remember, however, that for each small data section created, the compiler loses one non-volatile register to use for other purposes.

Linker Generated Symbols

You can find a complete list of the linker generated symbols in either the C include file `__ppc_eabi_linker.h` or the assembly include file `__ppc_eabi_linker.i`. The CodeWarrior linker automatically generates symbols for the start address, the end address (the first byte after the last byte of the section), and the start address for the section if it will be burned into ROM. With a few exceptions, all CodeWarrior linker-generated symbols are immediate 32 bit values.

If addresses are declared in your source file as `unsigned char _f_text[];` you can treat `_f_text` just as a C variable even though it is a 32-bit immediate value.

```
unsigned int textsize = _e_text - _f_text;
```

If you do need linker symbols that are not addresses, you can access them from C.

```
unsigned int size = (unsigned int)&_text_size;
```

The linker generates four symbols:

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

These four symbols are actually not 32-bit immediates but are variables with storage. You access them just as C variables. The startup code now automatically handles initializing all bss type sections and moves all necessary sections from ROM to RAM, even for user defined sections.

Deadstripping Unused Code and Data

The Embedded PowerPC linker deadstrips unused code and data only from files compiled by the CodeWarrior C/C++ compiler. Assembler relocatable files and C/C++ object files built by other compilers are never deadstripped. Deadstripping is particularly useful for C++ programs. Libraries (archives) built with the CodeWarrior C/C++ compiler only contribute the used objects to the linked program. If a library has assembly or other C/C++ compiler built files, only those files that have at least one referenced object contribute to the linked program. Completely unreferenced object files are always ignored.

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

There are, however, situations where there are symbols that you don't want dead-stripped even though they are never used. See “Linker Command Files” for information on how to prevent dead-stripping of unused symbols.

Link Order

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

Regardless of the link order you specify, the Embedded PowerPC linker always processes C/C++ files, 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, enter `TEXT` and in the **Extension** text box, enter `.lcf`
4. Click **None** from the **Compiler** listbox. Click the **Add** button to save your settings.

Now, when you add an `.lcf` file to your project, the compiler recognizes the file as a linker command file.

Linker Command File Directives

The CodeWarrior PPC EABI linker supports the directives listed below:

- EXCLUDEFILES
- EXTERNAL_SYMBOL
- FORCEACTIVE
- FORCEFILES
- GROUP
- INCLUDEDWARF
- INTERNAL_SYMBOL
- MEMORY
- REGISTER
- SECTIONS
- SHORTEN_NAMES_FOR_TOR_101

NOTE You can only use one `SECTIONS`, `MEMORY`, `FORCEACTIVE`, and `FORCEFILES` directive per linker command file.

NOTE If you want to mention a source file such as `main.c` in an `.lcf` file, type `main.o`. The `.lcf` only recognizes object and architecture extensions.

EXCLUDEFILES

The `EXCLUDEFILES` directive is for partial link projects only. It makes your partial link file smaller. The directive has this form.

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

In the example:

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

`kernel.elf` is added to your project. The linker does not add any section from `kernel.elf` to your project. However, it does delete any weak symbol from your partial link that also exists in `kernel.elf`. Weak symbols can come from templates or out-of-line inline functions.

EXCLUDEFILES can be used independently of INCLUDEDWARF. Unlike INCLUDEDWARF, EXCLUDEFILES can take any number of executable files.

EXTERNAL_SYMBOL

Use the EXTERNAL_SYMBOL and INTERNAL_SYMBOL directives to force the addressing of global symbols. This directive is of the form: XXXL_SYMBOL {sym1, sym2, symN}, where symbols are the link time symbol names (mangled for C++).

FORCEACTIVE

The directives FORCEACTIVE and FORCEFILES give you more control over symbols that you don't want dead-stripped. The FORCEACTIVE directive has this form:

```
FORCEACTIVE { symbol1 symbol2 ... }
```

Use FORCEACTIVE with a list of symbols that you do not want to be dead-stripped.

FORCEFILES

Use FORCEFILES to list source files, archives, or archive members that you don't want dead-stripped. All objects in each of the files are included in the executable. The FORCEFILES directive has this form:

```
FORCEFILES { source.o object.o archive.a(member.o) ... }
```

If you only have a few symbols that you do not want deadstripped, use FORCEACTIVE.

GROUP

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

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

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

Listing 7.5 shows that each group starts at a specified address. If no address_modifiers were present, it would start following the previous section or group. Although you normally do not have an address_modifier for an output_spec within a group, all sections in a group follow contiguously unless there is an address_modifier for that output_spec.

Listing 7.5 Example 1

```
SECTIONS {
 GROUP BIND(0x00010000) : {
  .text : {}
  .rodata : {*(.rodata) *(extab) *(extabindex)}
  }

 GROUP BIND(0x2000) : {
  .data : {}
  .bss : {}
  .sdata BIND(0x3500) : {}
  .sbss  : {}
  .sdata2 : {}
  .sbss2  : {}
  }

 GROUP BIND(0xffff8000) : {
  .PPC.EMB.sdata0  : {}
  .PPC.EMB.sbss0 : {}
  }
}
```

INCLUDEDWARF

The `INCLUDEDDWARF` directive allows you to debug source level code in the kernel while debugging your application. This directive has the form:
`INCLUDEDDWARF` { *executablename.extension* }

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

You are limited to one executable file when using this directive. If you need to process more than one executable, add this directive to another file.

INTERNAL_SYMBOL

Use the `INTERNAL_SYMBOL` and EXTERNAL_SYMBOL directives to force the addressing of global symbols. This directive is of the form: `XXXL_SYMBOL {sym1, sym2, symN}`, where symbols are the link time symbol names (mangled for C++).

REGISTER

Use the `REGISTER` directive to assign one of the EPPC processor's non-volatile registers to a user-defined small data section.

This directive is of this form `REGISTER(nn [ , limit])` where:

- `nn` is one of the predefined small data base registers, a non-volative EPPC register, or -1
  - `0, 2, 13`

    These registers are for the predefined small data sections:
    `0 - .EMB.PPC.sdata0/.EMB.PPC.sbss0`
    `2 - .sdata2/sbss2`
    `13 - .sdata/sbss`

    You do not have to define these sections using `REGISTER` because they are predefined.
  - `14 - 31`

    Match any value in this range with the register reserved by your global register variable declaration.
  - `-1`

    This "register" value instructs the linker to treat relocations that refer to objects in your small data section as non-small data area relocations. These objects are converted to near absolute relocations, which means that the objects referenced must reside within the first 32 KB of memory. If they do not, the linker emits a "relocation out of range" error. To fix this problem, rewrite your code such that the offending objects use large data relocations.
- `limit` is the maximum size of the small data section to which register `nn` is bound.

  This value is the size of the initialized and uninitialized sections of the small data section combined. If `limit` is not specified, `0x00008000` is used.

NOTE Each small data section you create makes one less register available to the compiler; it is possible to starve the compiler of registers. As a result, create only the number of small data sections you need.

MEMORY

A MEMORY directive is of the form MEMORY : { <memory_spec> ... }, where memory_spec is:

```
<symbolic name> : origin = num, length = num
```

origin may be abbreviated as org or o. length may be abbreviated as len or l. If you do not specify length, the memory_spec is allowed to be as big as necessary. In all cases, the linker warns you if sections overlap. The length is useful if you want to avoid overlapping an RTOS or exception vectors that might not be a part of your image.

You specify that a output_spec or a GROUP goes into a memory_spec with the ">" symbol.

Listing 7.6 shows the MEMORY directive added to the example code shown in Listing 7.5. The results of both examples are identical.

Listing 7.6 Example 2

```
MEMORY {

  text : origin = 0x00010000

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

SECTIONS {

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

  } > text

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

  } > data
GROUP  : {
  .PPC.EMB.sdata0  : {}
  .PPC.EMB.sbss0 : {}
  } > page0
}
```

SECTIONS

A SECTIONS directive has this form:

```
SECTIONS { <section_spec> ... }
```

where section_spec is

```
<output_spec> (<input_type>) <address_modifiers> :
{ <input_spec> ... }
```

output_spec is the section name for the output section.

input_type is one of TEXT, DATA, BSS, CONST and MIXED. CODE is also supported as a synonym of TEXT. One input_type is permitted and must be enclosed in (). If an input_type is present, only input sections of that type are added to the section. MIXED means that the section contains code and data (RWX). The input_type restricts the access permission that are acceptable for the output section, but they also restrict whether initialized content or uninitialized content can go into the output section. Table 7.8 shows the types of input for input_type.

Table 7.8 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.

NOTE Other compiler vendors also support ways that you can specify the ROM Load address with the address_modifiers. With CodeWarrior IDE, this information is specified in the EPPC Linker settings panel. You may also simply specify an address with BIND.

ADDR() takes previously defined output_spec or memory_spec enclosed in () and returns its address.

SIZEOF() takes previously defined output_spec or memory_spec enclosed in () and returns its size.

ALIGN() takes a number and aligns the output_spec to that alignment.

NEXT() is similar to ALIGN. It returns the next unallocated memory address.

BIND() can take a numerical address or a combination of the above pseudo functions.

input_spec can be empty or a file name, a file name with a section name, the wildcard'*' with a section name singly or in combination.

When input_spec is empty, as in

```
.text : {}
```

all .text sections in all files in the project that aren't more specifically mentioned in another input_spec are added to that output_spec.

A file name by itself means that all sections go into the output_spec.

A file name with a section name means that the specified section goes into the output_spec.

A "*" with a section name means that the specified section in all files go into the output_spec.

In all cases, the input_spec is subject to input_type. For example,

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

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

In all cases, if there is more that one input_spec that fits an input file, the more specific input_spec gets the file.

If an archive name is used instead of source file name, all referenced members of that archive are searched. You can further specify a member with red.a(redsky.c). The linker doesn't support grep. If listing just the source file name is ambiguous, enter the full path.

Listing 7.7 shows how you might specify a SECTIONS directive without a MEMORY directive. The .text section starts at 0x00010000 and contains all sections named .text in all input files. The .rodata section starts just after the .text section, and is aligned on the largest alignment found in the input files. The input files are the read only sections (.rodata) found in all files. The .data section starting address is the sum of the starting address of .rodata and the size of .rodata. The resulting address is aligned on a 0x100 boundary. The address contains all sections of .data in all files. The .bss section follows the .data through .sbss2 sections. The .EMB.PPC.sdata0 starts at 0xffff8000 and the .EMB.PPC.sbss0 follows it.

Listing 7.7 Example 3

```
SECTIONS {

.init : {}
  .text BIND(0x00010000) : {}
  .rodata : {}
  extab : {}
  extabindex : {}

  .data BIND(ADDR(.rodata) + SIZEOF(.rodata)) ALIGN(0x100) : {}
  .sdata : {}
  .sbss : {}
  .sdata2 : {}
  .sbss2 : {}
  .bss : {}

  .PPC.EMB.sdata0 BIND(0xffff8000) : {}
  .PPC.EMB.sbss0 : {}

}
```

NOTE extab and extabindex must be in separate sections.

SHORTEN_NAMES_FOR_TOR_101

The directive SHORTEN_NAMES_FOR_TOR_101 instructs the linker to shorten long template names for the benefit of the WindRiver® Systems Target Server. To use this directive, simply add it to the linker command file on a line by itself.

```
SHORTEN_NAMES_FOR_TOR_101
```

WindRiver Systems Tornado Version 1.0.1 (and earlier) does not support long template names as generated for the MSL C++ library. Therefore, the template names must be shortened if you want to use them with these versions of the WindRiver Systems Target Server.

Miscellaneous features

- Memory Gaps
- Symbols

Memory Gaps

You can create gaps in memory by performing alignment calculations such as

```
. = (. + 0x20) & ~0x20;
```

This kind of calculation can occur between `output_specs`, between `input_specs`, or even in `address_modifiers`. A "." refers to the current address. You may assign the . to a specific unallocated address or just do alignment as the example shows. The gap is filled with `0`, in the case of an alignment (but not with `ALIGN()`).

You can specify an alternate fill pattern with `= <short_value>`, as in

```
.text : { . = (. + 0x20) & ~0x20; *(.text) } = 0xAB > text
```

`short_value` is 2 bytes long. Note that the fill pattern comes before the `memory_spec`. You can add a fill to a GROUP or to an individual `output_spec` section. Fills cannot be added between `.bss` type sections. All calculations must end in a "`;`".

Symbols

You can create symbols that you can use in your program by assigning a symbol to some value in your linker command file.

```
.text : { _red_start = .; *(.text) _red_end = .;} > text
```

In the example above, the linker generates the symbols `_red_start` and `_red_end` as 32 bit values that you can access in your source files. `_red_start` is the address of the first byte of the `.text` section and `__red_end` is the byte that follows the last byte of the `.text` section.

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

The CodeWarrior linker automatically generates symbols for the start address, the end address, and the start address for the section if it is to be burned into ROM. For a section `.red`, we create `_f_red`, `_e_red`, and `_f_red_rom`. In all cases, any "`.`" in the name is replaced with a "`_`". Addresses begin with an "`_f`", addresses after the last byte in section begin with an "`_e`", and ROM addresses end in a "`_rom`". See the header file `__ppc_eabi_linker.h` for further details.

All user defined sections follow the preceding pattern. However, you can override one or more of the symbols that the linker generates by defining the symbol in the linker command file.

NOTE BSS sections do not have a ROM symbol.

Using __attribute__ ((aligned(?)))

You can use `__attribute__ ((aligned(?)))` in several situations:

- Variable declarations
- Struct, union, or class definitions
- Typedef declarations
- Struct, union, or class members

NOTE Substitute any power of 2 up to 4096 for the question mark (?).

This section contains these topics:

- Variable Declaration Examples
- Struct Definition Examples
- Typedef Declaration Examples
- Struct Member Examples

Variable Declaration Examples

This section shows variable declarations that use `__attribute__ ((aligned(?)))`.

The following variable declaration aligns `V1` on a 16-byte boundary.

```
int V1[4] __attribute__ ((aligned (16)));
```

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

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

Struct Definition Examples

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

The following struct definition aligns all definitions of `struct` S1 on an 8-byte boundary.

```
struct S1 { short f[3]; }
    __attribute__ ((aligned (8)));
struct S1 s1;
```

The following struct definition aligns all definitions of `struct` S2 on a 4-byte boundary.

```
struct S2 { short f[3]; }
    __attribute__ ((aligned (1)));
struct S2 s2;
```

NOTE You must specify a minimum alignment of at least 4 bytes for structures. Specifying a lower number for the alignment of a structure causes alignment exceptions.

Typedef Declaration Examples

This section shows typedef declarations that use `__attribute__ ((aligned(?)))`.

The following typedef declaration aligns all definitions of T1 on an 8-byte boundary.

```
typedef int T1 __attribute__ ((aligned (8)));
T1 t1;
```

The following typedef declaration aligns all definitions of T2 on an 1-byte boundary.

```
typedef int T2 __attribute__ ((aligned (1)));
T2 t2;
```

Struct Member Examples

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

The following struct member definition aligns all definitions of `struct S3` on an 8-byte boundary, where `a` is at offset 0 and `b` is at offset 8.

```
struct S3 {
   char a;
   int b __attribute__ ((aligned (8)));
};
struct S3 s3;
```

The following struct member definition aligns all definitions of `struct S4` on a 4-byte boundary, where `a` is at offset 0 and `b` is at offset 4.

```
struct S4 {
   char a;
   int b __attribute__ ((aligned (2)));
};
struct S4 s4;
```

NOTE Specifying `__attribute__ ((aligned (2)))` does not affect the alignment of S4 because 2 is less than the natural alignment of int.

8

Inline Assembler

This chapter explains how to use the inline assembler built into the CodeWarrior™ Embedded PowerPC C/C++ compiler.

The chapter does *not* discuss the standalone CodeWarrior EPPC assembler. For information about this tool, refer to the *Assembler Guide*.

Further, the chapter does not document all the instructions in the instruction set of the Embedded PowerPC processor. For complete documentation of this instruction set, see *PowerPC Microprocessor Family: The Programming Environment for 32-Bit Microprocessors,* published by Motorola.

Finally, refer to this web page for documentation of Motorola's entire semiconductor product line, including embedded versions of the PowerPC processor:

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

This chapter contains these topics:

- Working With Assembly Language
- Assembler Directives
- Intrinsic Functions

Working With Assembly Language

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

This section contains these topics:

- 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
- 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 instructions must end with `blr` instruction. For example,

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

Statement-level assembler syntax 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.” 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
    blr
  }
  ```

Listing 8.1 shows an example of an assembly language function.

Listing 8.1 Example Assembly Language Function

```
asm void mystrcpy(char *tostr, char *fromstr)

{
  addi  tostr,tostr,-1
  addi  fromstr,fromstr,-1
@1 lbzu  r5,1(fromstr)
  cmpwi r5,0
  stbu  r5,1(tostr)
  bne  @1
  blr
}
```

Special Embedded PowerPC Instructions

To set the branch prediction (`y`) bit for those branch instructions that can use it, use `+` or `-`. For example:

```
@1 bne+ @2
@2 bne- @1
```

Most integer instructions have four forms:

- normal form — `add r3,r4,r5`

- record form — `add.  r3,r4,r5`

  This form ends in a period. This form sets register `cr0` to whether the result is less, than, equal to, or greater than zero.

- overflow — `addo  r3,r4,r5`

  This form ends in the letter (`o`). This form sets the `SO` and `OV` bits in the `XER` if the result overflows.

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

  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** listbox 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 func1(){
  int i;
    @x: li r0,1 //Invalid !!!
  }

  asm void func2(){
  int i;
    x:  li r0,1 //OK
    @y: add r3, r4, r5 //OK
  }
  ```

- A label name must end with a colon character (`:`) unless it begins with an at-sign character (`@`).

  For example, `red:` and `@red` are valid, but `red` is *not* valid.

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

  For example, this statement is valid:

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

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

```
asm void red(void){
  x1:  add r3,r4,r5
  @x2: add r6,r7,r8
}
```

Using Comments

You cannot begin comments with a pound sign (`#`) because the preprocessor uses the pound sign. For example, this format is invalid:

```
add   r3,r4,r5 # Comment
```

Use C and C++ comments in this format:

```
add   r3,r4,r5 // Comment
add   r3,r4,r5 /* Comment */
```

Using the Preprocessor in Embedded PowerPC Assembly

You can use all preprocessor features, such as comments and macros, in the assembler. In multi-line macros, you must end each assembly statement with a semicolon (`;`) because the (`\`) operator removes newlines. For example:

```
#define remainder(x,y,z) \
divw z,x,y; \
mullw z,z,y; \
subf z,z,x

asm void newPointlessMath(void)
{
remainder(r3,r4,r5)
blr
}
```

Using Local Variables and Arguments

To refer to a memory location, you can use the name of a local variable or argument.

The rule for assigning arguments to registers or memory depends on whether the function has a stack frame.

If function has a stack frame, the inline assembler assigns:

- scalar arguments declared as `register` to `r14` — `r31`
- floating-point arguments declared as `register` to `fp14` — `fp31`
- other arguments to memory locations
- scalar locals declared as `register` to `r14` — `r31`
- floating-point locals declared as `register` to `fp14` — `fp31`
- other locals to memory locations

If a function has no stack frame, the inline assembler assigns arguments that are declared `register` and kept in registers. If you have variable or non-register arguments, the compiler will warn you that you should use `frfree`

NOTE Some opcodes require registers, and others require objects. For example, if you use `nofralloc` with function arguments, you may run into difficulties.

Creating a Stack Frame 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.

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 8.2 shows a block of code for specifying operands.

Listing 8.2 Using Register Variables and Memory Variables

```
asm void red(register int *a)

{
   int b;
   fralloc
   lwz r4,a
   lwz r4,0(a)
   lwz r4,b
   lwz r4, b(SP)
   frfree
   blr
}
```

In Listing 8.2:

- the code at line number five is incorrect because the operand of the operand of register variable is not fully expressed
- the code at line number six is correct because the operand is fully expressed
- the code at line number seven is correct; the inline assembler allows use of locals and arguments that are not declared as register
- the code at line number eight is correct because `b` is a memory variable

Using Registers

For a register operand, you must use one of the register names of the appropriate kind for the instruction. The register names are case-sensitive. You also can use a symbolic name for an argument or local variable that was assigned to a register.

The general registers are `SP`, `r0` to `r31`, and `gpr0` to `gpr31`. The floating-point registers are `fp0` to `fp31` and `f0` to `f31`. The condition registers are `cr0` to `cr7`.

Using Labels

For a label operand, you can use the name of a label. For long branches (such as `b` and `bl` instructions) you can also use function names. For `bla` and `la` instructions, use absolute addresses.

For other branches, you must use the name of a label. For example,

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

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

Using Variable Names as Memory Locations

Whenever an instruction, such as a load instruction, a store instruction, or `la`, requires a memory location, you can use a local or global variable name. You can modify local variable names with struct member references, class member references, array subscripts, or constant displacements. For example, all the local variable references in the following block of code are valid.

```
asm void red(void){
  long myVar;
  long myArray[1];
  Rect myRectArray[3];
  fralloc
  lwz r3,myVar(SP)
  la  r3,myVar(SP)
  lwz r3,myRect.top
  lwz r3,myArray[2](SP)
  lwz r3,myRectArray[2].top
  lbz r3,myRectArray[2].top+1(SP)
  frfree
  blr
}
```

You can also use a register variable that is a pointer to a struct or class to access a member of the struct in this manner:

```
void red(void){
  Rect q;
  register Rect *p = &q;
  asm {
  lwz r3,p->top;
  }
}
```

You can use the `@hiword` and `@loword` directives to access the high and low four bytes of 8 byte long longs and software floating point doubles.

```
long long gTheLongLong = 5;
asm void Red(void);
asm void Red(void)
{
  fralloc
  lwz r5, gTheLongLong@hiword
  lwz r6, gTheLongLong@loword
  frfree
blr
}
```

Using Immediate Operands

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

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

An immediate operand can also be a reference to a member of a struct or class type. You can use any struct or class name from a `typedef` statement, followed by any number of member references. This evaluates to the offset of the member from the start of the struct. For example:

```
lwz   r4,Rect.top(r3)
addi  r6,r6,Rect.left
```

As a side note, `la rD,d(rA)` is the same as `addi rD,rA,d`.

You also can use the top or bottom half-word of an immediate word value as an immediate operand by using one of the @ modifiers.

```
long gTheLong;
asm void red(void)
{
```

```
fralloc
lis r6, gTheLong@ha
addi r6, r6, gTheLong@h
lis r7, gTheLong@h
ori r7, br7, gTheLong@l
frfree
blr
}
```

The access patterns are:

```
lis x,var@ha
la  x,var@l(x)
```

or

```
lis x,var@h
ori x,x,var@l
```

In this example, `la` is the simplified form of `addi` to load an address. The instruction `las` is similar to `la` but shifted. Refer to the Motorola PowerPC manuals for more information.

Using `@ha` is preferred since you can write:

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

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

Assembler Directives

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

- entry
- fralloc
- frfree
- machine
- nofralloc
- opword

entry

entry [extern | static] *name*

Embedded PowerPC assembler directive that defines an entry point into the current function. Use the `extern` qualifier to declare a global entry point; use the `static` qualifier to declare a local entry point. If you leave out the qualifier, `extern` is assumed.

Listing 8.3 shows how to use the `entry` directive.

Listing 8.3 Using the entry directive

```
void __save_fpr_15(void);
void __save_fpr_16(void);
asm void __save_fpr_14(void)
{
    stfd   fp14,-144(SP)
    entry  __save_fpr_15
    stfd   fp15,-136(SP)
    entry  __save_fpr_16
    stfd  fp16,-128(SP)
    // ...
}
```

fralloc

fralloc [*number*]

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

- calls other functions.
- uses more arguments than will fit in the designated parameters `(r3 — r10, fp1 — fp8).`
- declares local registers.
- declares non-registered parameters.

The `fralloc` directive has an optional argument *number* that lets you specify the size in bytes of the parameter area of the stack frame. The compiler creates a 0-byte parameter area. If your assembly-language routine calls any function that requires more parameters than will fit in `r3` — `r10` and `fp1` — `fp8`, you must specify a larger amount.

frfree

`frfree`

Embedded PowerPC assembler directive that frees the stack frame and restores the registers that `fralloc` reserved.

NOTE The `frfree` directive does not generate a `blr` instruction. You must include one explicitly.

machine

`machine` *number*

Embedded PowerPC assembler directive that specifies which CPU the assembly language code is for. The value of *number* must be one of those listed in Table 8.1.

Table 8.1 CPU Identifiers

401	403	505	509
555	56x	601	602
603	604	740	750
801	821	823	850
860	7400	8240	8260
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** listbox in the **EPPC Processor** settings panel. For example, machine altivec.

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

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: *InstallDir*\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);
```

This example shows how to set and restore the FPSCR:

```
double old_fpscr;

oldfpscr = __setflm(0.0); /* Clears all flag/exception/mode
bits and save the original settings */

/* Peform some floating point operations */

__setflm(old_fpscr); /* Restores the FPSCR */
```

Manipulating 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 8.2 map directly to PowerPC assembly instructions

Table 8.2 Data Cache Intrinsics

Intrinsic Prototype	PowerPC Instruction
void __dcbf(void *, int);	dcbf
void __dcbt(void *, int);	dcbt
void __dcbst(void *, int);	dcbst
void __dcbtst(void *, int);	dcbtst
void __dcbz(void *, int);	dcbz

Math Functions

The intrinsics shown in Table 8.3 map directly to PowerPC assembly instructions.

Table 8.3 Math Intrinsics

Intrinsic Prototype	PowerPC Instruction
int __mulhw(int, int);	mulhw
uint __mulhwu(uint, uint);	mulhwu
double __fmadd(double, double, double);	fmadd
double __fmsub(double, double, double);	fmsub
double __fnmadd(double, double, double);	fnmadd
double __fnmsub(double, double, double);	fnmsub
float __fmadds(float, float, float);	fmadds
float __fmsubs(float, float, float);	fmsubs
float __fnmadds(float, float, float);	fnmadds
float __fnmsubs(float, float, float);	fnmsubs

Table 8.3 Math Intrinsics

double __mffs(void);	mffs
float __fabsf(float);	fabsf
float __fnabsf(float);	fnabsf

Buffer Manipulation

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

```
void *__alloca(ulong);
__alloca implements alloca() in the compiler.
char *__strcpy(char *, const char *);
```

`__strcpy()` detects copies of constant size and calls `__memcpy()`. This intrinsic requires that a `__strcpy` function be implemented because if the string is not a constant it will call `__strcpy` to do the copy.

```
void *__memcpy(void *, const void *, size_t);
```

`__memcpy()` provides access to the block move in the code generator to do the block move inline.

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://e-www.motorola.com/collateral/ALTIVECPEMCH.htm`

These are the generic and specific Altivec intrinsics:

vec_abs, vec_adds, vec_cmpgt, vec_xor

vec_abss, vec_ceil, vec_cmple, vec_trunc

vec_and, vec_cmpb, vec_sums

These are the Altivec predicates:

vec_all_eq, vec_all_le

vec_all_ge, 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

9

Libraries and Runtime Code

The CodeWarrior software includes many libraries for use with the CodeWarrior development environment. For example, ANSI-standard libraries for C and C++, as well as runtime libraries and other code. This chapter explains how to use these libraries for Embedded PowerPC development.

With respect to the Metrowerks Standard Libraries (MSL) for C and C++, this chapter is an extension of the *MSL C Reference* and the *MSL C++ Reference*. Consult those manuals for general details on the standard libraries and their functions.

The sections in this chapter are:

- MSL for Embedded PowerPC
- Runtime Libraries for Embedded PowerPC
- Board Initialization Code

MSL for Embedded PowerPC

This section describes the Metrowerks Standard Libraries (MSL) that have been modified for use with Embedded PowerPC. The CodeWarrior IDE includes the source and project files for MSL so that you can modify the libraries if necessary.

- Using MSL for Embedded PowerPC
- Using Console I/O for Embedded PowerPC
- Allocating Memory and Heaps for Embedded PowerPC

Using MSL for Embedded PowerPC

Your CodeWarrior IDE includes a version of the Metrowerks Standard Libraries (MSL). MSL is a complete C and C++ library collection that you can use in your embedded projects. All of the sources necessary to build MSL are included in the CodeWarrior IDE, along with the project files for different configurations of MSL.

If you already have a version of the CodeWarrior IDE installed on your computer, the CodeWarrior installer will include the new files needed for building versions of MSL for Embedded PowerPC.

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 based on your memory configuration, you should make the changes to the runtime libraries.

MSL for Embedded PowerPC supports console I/O through the serial port on the MPC8xx ADS or MBX boards, as well as the MPC5xx EVB board. The standard C library I/O is supported, including `stdio`, `stderr`, and `stdin`. All functions that do not require disk I/O are supported in this version of MSL. The memory functions `malloc()` and `free()` are also supported.

You may be able to use another standard C library with the CodeWarrior IDE. You should check the files `stdarg.h` of both libraries. The CodeWarrior Embedded PowerPC C/C++ compiler will only generate correct variable-argument functions by using the header file supplied 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`.

Other C++ libraries will not be compatible.

If you are working with any kind of embedded OS, you may need to customize MSL to work properly with the OS.

Using Console I/O for Embedded PowerPC

There are a few special considerations when using console I/O with the MSL C or C++ libraries. In order for the console I/O to function, a special serial I/O library must be built into the project. In addition, the hardware must be initialized properly to work with this library. The issues are explained in these topics:

- Including UART libraries
- Configuring the Board for Console I/O (MPC 8xx only)

Including UART libraries

In order for the C or C++ libraries to handle console I/O, a special serial driver library must be included in your project. The particular library you use will depend on the board you are using and the serial port that you want to communicate through.

For each target, there are libraries for unsigned chars and signed chars. The naming convention for these libraries is as follows:

- unsigned char libraries — `<library_name>.UC.a`
- signed char libraries — `<library_name>.a`

Table 9.1 indicates the file you must include based on your setup. You can find all files listed at this location in the CodeWarrior installation directory:

```
PowerPC_EABI_Tools\MetroTRK\
Transport\ppc\{Board Directory}\Bin
```

Table 9.1 Serial I/O Libraries

Board	Filename
56X chip support	UART1_MOT_56X_Chip.a
Axiom 555, 565	UART1_MOT_5xx_Axiom.a
Motorola 555 ETAS	UART1_MOT_555_ETAS.a
phyCORE single-board computer subassembly for MPC5xx family	UART1_Phycore_MPC555.a UART1_Phycore_MPC565.a

If your MBX board is not running at the Processor Speed specified in Table 9.1, you need to modify one of these libraries to work with your hardware.

When making changes, it is important to add a baud-rate divisor table tailored to your processor speed. CodeWarrior projects (the files ending in `.mcp`) are used to modify and build new versions of the library.

The projects are in this directory:

```
InstallDir\PowerPC_EABI_Tools\
MetroTRK\Transport\ppc\{Board directory}
```

Configuring the Board for Console I/O (MPC 8xx only)

If you are using either the 821 or 860 processor, the serial libraries used to implement console I/O depend on the processor running at a certain speed. The libraries included with the CodeWarrior software expect this speed to be either 24 MHz, 40 MHz, or 50 MHz. There are several ways you can ensure that your board is running at a speed compatible with the serial I/O library:

- Run under MetroTRK — MetroTRK for the ADS board attempts to initialize the processor to run at 24 MHz; MetroTRK for the MBX board attempts to initialize it to 40 MHz or 50 MHz, whichever is appropriate for the board

- Use an initialization file specific to your platform target — depending on your target board, use an initialization file (ending in `.asm`) in the directory `*InstallDir*\PowerPC_EABI_Support\Runtime\Src`

  If you are not using one of these boards, you must have a custom initialization routine to set the processor to the right speed.
- Use a custom initialization routine — if you use a custom initialization routine, make sure the processor speed is set to either 24 MHz, 40 MHz, or 50 MHz, depending on which board you are using
- Modify the serial library source code — change the baud rate divisors to match the operating speed of your board

Allocating Memory and Heaps for Embedded PowerPC

The heap you specify in the Heap Address text box in the EPPC Linker panel is the default heap. The default heap needs no initialization. The code responsible for memory management is only linked into your code if you call `malloc` or `new`.

You may find that you do not have enough contiguous space available for your needs. In that case you can initialize multiple memory pools to form a large heap.

You create each memory pool with a call to `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 for Embedded PowerPC

For any C or C++ project, you must include one of following runtime libraries in your project:

- `Runtime.PPCEABI.N.a` or `Run_EC++.PPCEABI.N.a` (No floating point support)
- `Runtime.PPCEABI.H.a` or `Run_EC++.PPCEABI.H.a` (Hardware floating point operations)
- `Runtime.PPCEABI.S.a` or `Run_EC++.PPCEABI.S.a` (Software emulation of floating point operations)
- `Runtime.PPCEABI.A.a` or `Run_EC++.PPCEABI.A.a` (AltiVec and hardware floating point operations)

- `RUN_EC++.PPCEABI.HC.a` or `Runtime.PPCEABI.HC.a` and `RUN_EC++.PPCEABI.NC.a` or `Runtime.PPCEABI.NC.a`

These files are in the directory:
InstallDir`\PowerPC_EABI_Support\Runtime\Lib\`

In addition, you must include one of the following source files, which contains 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 projects)
- `__ppc_eabi_init.cpp` (for C++ projects)

The CodeWarrior IDE includes the source and project files for the runtime libraries so that you can modify them if necessary. All the files are within the directory:
InstallDir`\PowerPC_EABI_Support\Runtime\Src`

The runtime library project files are here:
InstallDir`\PowerPC_EABI_Support\Runtime\Project`

The project names are `Runtime.PPCEABI.mcp and Run_EC++.PPCEABI.mcp`. Each project file has unique targets for each of the configurations of the runtime library.

For more information on how to customize the runtime library for use with your project, you should carefully 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 will be loading and running the programs with the Metrowerks debugger. When your program is ready to run as a stand-alone application, you must add the necessary hardware initialization.

Board Initialization Code

The Metrowerks CodeWarrior software comes with 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, as the debugger or debug kernel already performs the same board initializations.

If your code is running stand-alone (without the debugger), you may want to include the board initialization file. These files are located at the following path, and use the extension `.asm`.

`InstallDir\PowerPC_EABI_Support\Runtime\Src`

These files are included in source form, so you are free to modify them to work with other boards or hardware configurations.

Each of these files includes a function called `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 indicators to have your program perform the hardware initializations.

10

Hardware Tools

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

This chapter contains these sections:

- Flash Programmer
- Hardware Diagnostics
- Logic Analyzer

Flash Programmer

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

- Program
- Erase
- BlankCheck
- Verify
- Checksum

NOTE Certain flash programming features (such as view/modify, memory/register, and save memory content to a file) are provided by the CodeWarrior debugger. As a result, the CodeWarrior flash programmer does not include these features.

The CodeWarrior flash programmer uses the CodeWarrior Debugger Protocol API to communicate with the target boards. The CodeWarrior flash programmer runs as a CodeWarrior plug-in.

The CodeWarrior flash programmer lets you use the same IDE to program the flash of any of the embedded target boards.

Table 10.1 lists the target boards having flash modules that can be programmed by using the CodeWarrior IDE.

Table 10.1 Supported Target Boards and Flash Modules

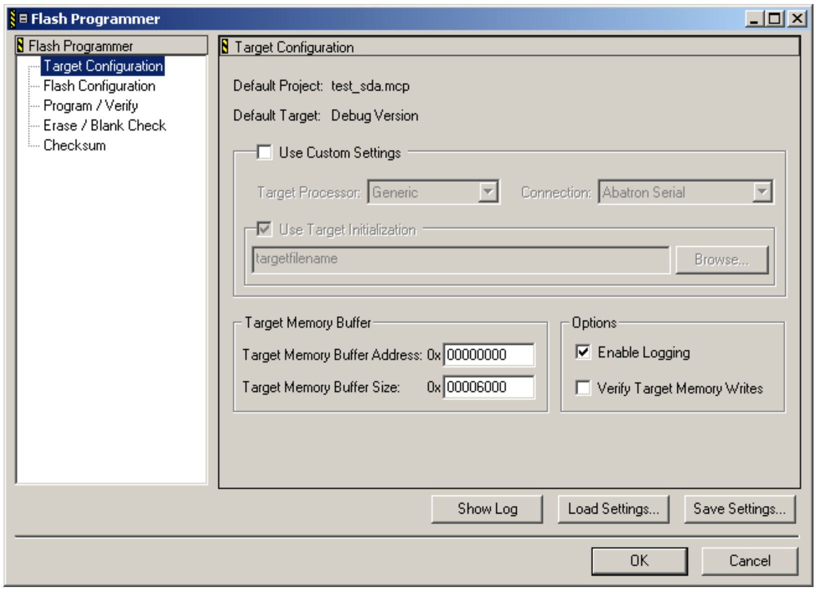
Target Board	Flash Module
Axiom 563, 565	On-chip CMFI Flash Memory
Axiom ETAS 555	On-chip CMFI Flash Memory
phyCORE single-board computer subassembly for MPC5xx family	AMD AM29LVxxx

NOTE * The Sharp flash SIMM LH28F016SCT-Z4 is compatible with the Intel flash SIMM E28F016SCT-Z4

The Motorola MPC 8xx MBX boards and the mezzanine boards of the Motorola Sandpoint X2 and X3 boards have no support for flash memory.

The **Flash Programmer** window (Figure 10.1) lists global options for the flash programmer hardware tool. These preferences apply to every open project file.

Figure 10.1 Flash Programmer Window



Default Project: test_sda.mcp
Default Target: Debug Version

□ Use Custom Settings

Target Processor: Generic
Connection: Abatron Serial

☑ Use Target Initialization
targetfilename

Target Memory Buffer
Target Memory Buffer Address: 0x 00000000
Target Memory Buffer Size: 0x 00006000

Options
☑ Enable Logging
□ Verify Target Memory Writes

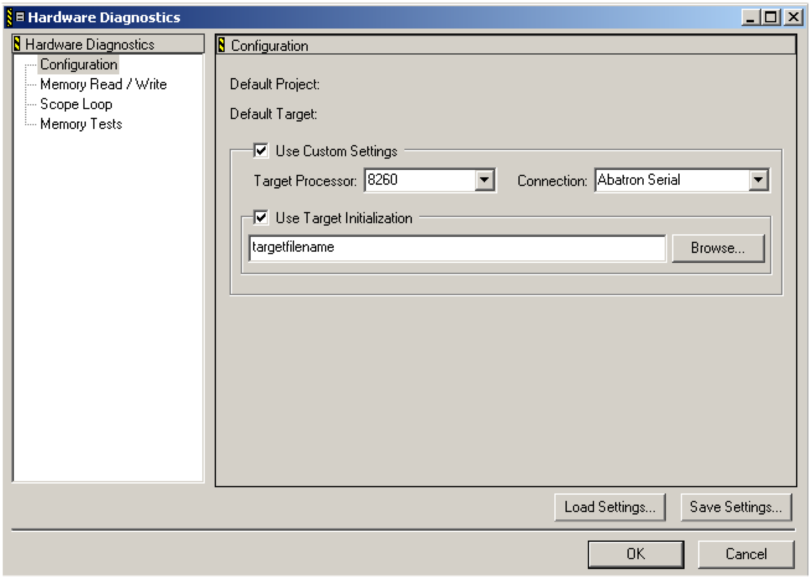
To open the **Flash Programmer** window, select **Tools > Flash Programmer**. The left pane of the **Flash Programmer** window shows a tree structure of panels. Click a panel name to display that panel in the right pane of the **Flash Programmer** window.

Refer to the *IDE User Guide* for information on each panel in the **Flash Programmer** window.

Hardware Diagnostics

The **Hardware Diagnostics** window (Figure 10.2) lists global options for the hardware diagnostic tools. These preferences apply to every open project file. Select **Tools > Hardware Diagnostics** to display the **Hardware Diagnostics** window.

Figure 10.2 Hardware Diagnostics window



Default Project:

Default Target:

☑ Use Custom Settings

Target Processor: 8260

Connection: Abatron Serial

☑ Use Target Initialization

targetfilename

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 *IDE User Guide* for information on each panel in the **Hardware Diagnostics** window.

Logic Analyzer

This section explains how to use the logic analyzer feature of the CodeWarrior IDE. The logic analyzer collects trace data from the target and the debugger correlates the trace data with the currently running source code.

This section has these topics:

- Logic Analyzer Menu
- Logic Analyzer Tutorial

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” 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” for details.

Arm

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

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

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 you how to use the logic analyzer functionality of the IDE to collect target cycles, retrieve trace data, and display trace data.

To explain the logic analyzer functionality, the following setup has been used in the tutorial:

- Agilent logic analyzer
- Motorola MPC 8260 ADS board
- The Agilent 16700B modular frame with three 16717A boards

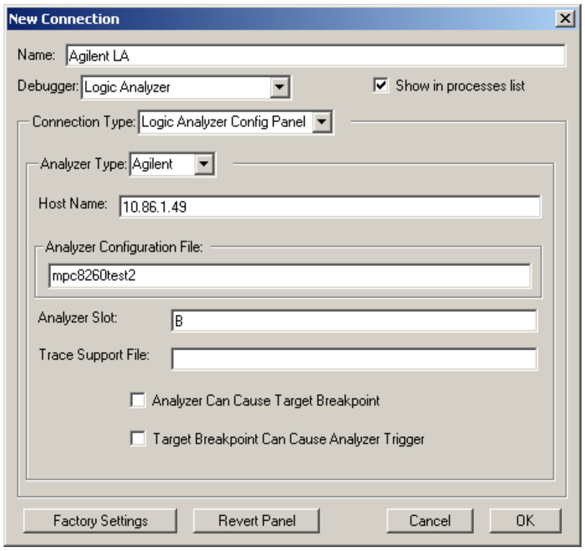
NOTE The 16717A boards in slot A and C are configured as slaves to the master board in Slot B.

- A CodeWarrior IDE project configured to use a WireTAP JTAG remote connection to the MPC8260 ADS target.

The tutorial follows:

1. Open your project.
   a. Start the CodeWarrior IDE.
   b. Select **File > Open**. The **Open** dialog box appears.
   c. Navigate to the directory where you have stored your project.
   d. Select the project file name.
   e. Click **Open**. The project window appears.
2. Create a logic analyzer connection.
   a. Click **Edit > IDE Preferences**. The **IDE Preferences** window appears.
   b. Select the **Remote Connections** item from the **IDE Preference Panels** list. The **Remote Connections** preference panel appears.
   c. In the **Remote Connections** preference panel, click **Add**. The **New Connection** dialog box (Figure 10.3) appears.

Figure 10.3 Logic Analyzer Connection Preferences


New Connection

Name: Agilent LA

Debugger: Logic Analyzer

☑ Show in processes list

Connection Type: Logic Analyzer Config Panel

Analyzer Type: Agilent

Host Name: 10.86.1.49

Analyzer Configuration File: mpc8260test2

Analyzer Slot: B

Trace Support File:

☐ Analyzer Can Cause Target Breakpoint

☐ Target Breakpoint Can Cause Analyzer Trigger

d. Enter the connection name in the **Name** text box. For example, 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, Agilent.

h. Enter 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, `mpc8260test2`.

   To find out which configuration file to use for your target, refer to the analyzer trace support package documentation.

NOTE If you only enter the analyzer configuration file name in the **Analyzer Configuration File** text box, the system downloads the file at this location on the analyzer file system: `/logic/config`. If you want to download the configuration file somewhere else on the analyzer file system, enter the full path from root. For example, `/logic/config/myconfig/myconfigfile`. If you leave the **Analyzer Configuration File** text box blank, you must load an analyzer configuration file through the Analyzer GUI. In this case, the logic analyzer connection of the IDE will not load a configuration file.

j. In the **Analyzer Slot** text box, enter the slot name that identifies the logic analyzer location.

k. In the **Trace Support File** text box, enter 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 wish to to allow the logic analyzer to cause a hardware breakpoint.

m. Check the **Target Breakpoint Can Cause Analyzer Trigger** checkbox if you wish to allow a hardware breakpoint to trigger the logic analyzer.

n. Click **OK**. The system saves the connection settings.

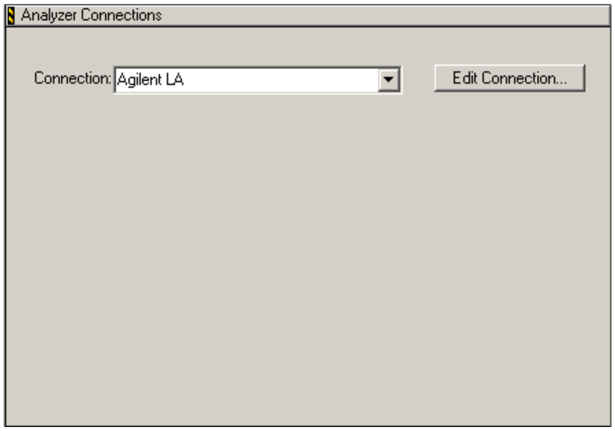
o. In the **IDE Preferences** window click **OK**. The IDE closes the **IDE Preferences** window.

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 10.4) appears.

Figure 10.4 Analyzer Connections Panel



Analyzer Connections

Connection: Agilent LA

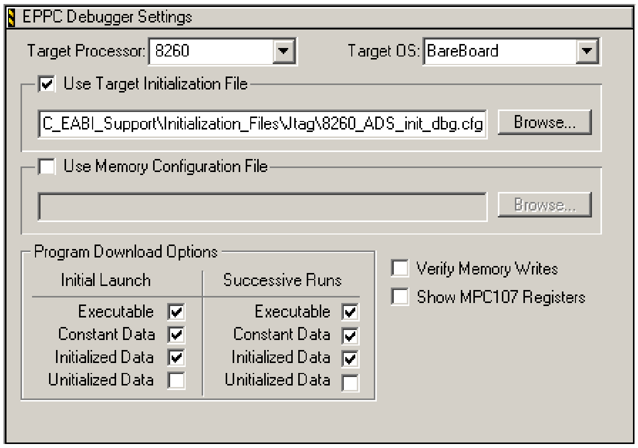
   c. Select the analyzer connection name from the **Connection** list.

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 the **EPPC Debugger Settings** item from the **Target Setting Panels** list. The **EPPC Debugger Settings** panel (Figure 10.5) appears.

Figure 10.5 EPPC Debugger Panel



EPPC Debugger Settings

Target Processor: 8260 Target OS: BareBoard

☑ Use Target Initialization File

C_EABI_Support\Initialization_Files\Jtag\8260_ADS_init_dbg.cfg

☐ Use Memory Configuration File

Program Download Options

Initial Launch		Successive Runs	
Executable	☑	Executable	☑
Constant Data	☑	Constant Data	☑
Initialized Data	☑	Initialized Data	☑
Unitialized Data	☐	Unitialized Data	☐

☐ Verify Memory Writes

☐ Show MPC107 Registers

b. Check the **Use Target Initialization File** checkbox.

c. In the text box associated with the **Use Target Initialization File** checkbox, enter the path of the target initialization file appropriate for your target. Alternatively, click **Browse** to open a dialog box that you can use to specify the initialization file path. Table 10.2 shows the location of generic initialization files for supported target boards.

Table 10.2 Configuration Files

Board	Configuration File Location (
56x chip support	PowerPC_EABI_Support\Initialization_Files\BDM\56X_CHIP_init.cfg
AXIOM 555	PowerPC_EABI_Support\Initialization_Files\BDM\555_AXIOM_init.cfg
AXIOM 555 with shadow flash	PowerPC_EABI_Support\Initialization_Files\BDM\555_AXIOM_vfSH_flash_init.cfg
AXIOM 565	PowerPC_EABI_Support\Initialization_Files\BDM\565_AXIOM_init.cfg
AXIOM 565 with shadow flash	PowerPC_EABI_Support\Initialization_Files\BDM\565_AXIOM_vfSH_flash_init.cfg
Motorola 555 ETAS	PowerPC_EABI_Support\Initialization_Files\BDM\555_ETAS_init.cfg
phyCORE single-board computer subassembly for MPC5xx family	PowerPC_EABI_Support\Initialization_Files\BDM\Phycore_MPC555_BDM_init.cfg

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

5. Connect the analyzer pods. Table 10.3 shows the Agilent analyzer pod connection scheme.

NOTE The pod connections are dependent on the trace support package installed on your analyzer. This package is available from the analyzer vendor. To know about the pod connection scheme for your target board, refer to the package documentation of the analyzer.

Table 10.3 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.

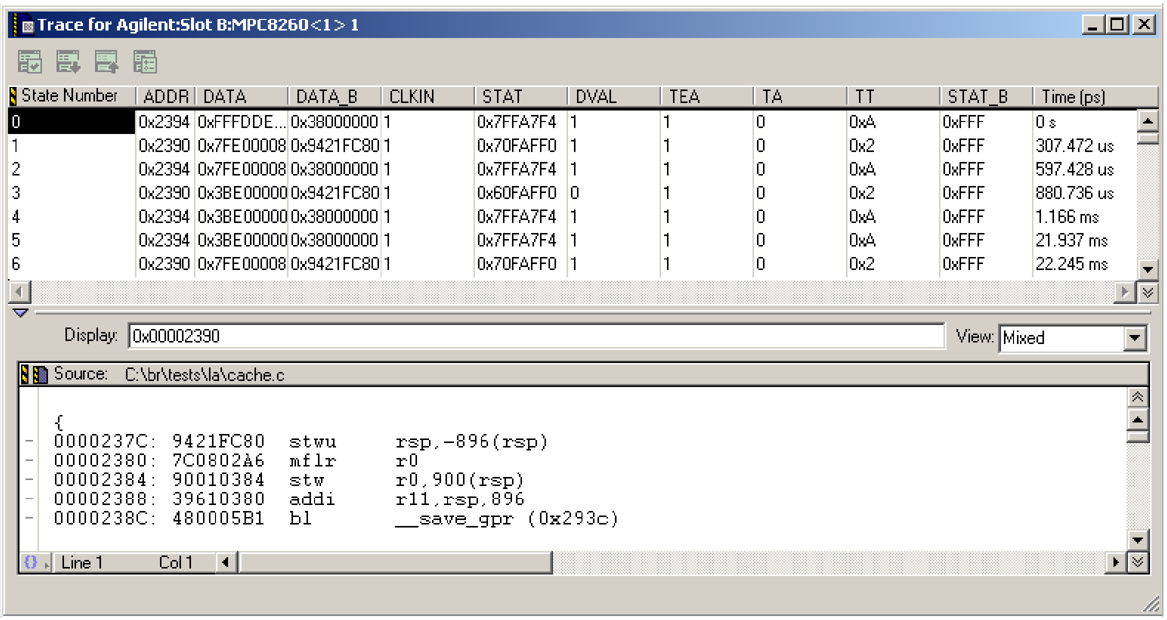
NOTE While the analyzer is armed the debugger periodically queries the analyzer for its Run status.

   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.

f. Select **Data > Trace View**. The trace window (Figure 10.6) appears. The trace window displays the data collected.

Figure 10.6 Trace Window



State Number	ADDR	DATA	DATA_B	CLKIN	STAT	DVAL	TEA	TA	TT	STAT_B	Time (ps)
0	0x2394	0xFFFDDE...	0x38000000	1	0x7FFA7F4	1	1	0	0xA	0xFFF	0 s
1	0x2390	0x7FE00008	0x9421FC80	1	0x70FAFF0	1	1	0	0x2	0xFFF	307.472 us
2	0x2394	0x7FE00008	0x38000000	1	0x7FFA7F4	1	1	0	0xA	0xFFF	597.428 us
3	0x2390	0x3BE00000	0x9421FC80	1	0x60FAFF0	0	1	0	0x2	0xFFF	880.736 us
4	0x2394	0x3BE00000	0x38000000	1	0x7FFA7F4	1	1	0	0xA	0xFFF	1.166 ms
5	0x2394	0x3BE00000	0x38000000	1	0x7FFA7F4	1	1	0	0xA	0xFFF	21.937 ms
6	0x2390	0x7FE00008	0x9421FC80	1	0x70FAFF0	1	1	0	0x2	0xFFF	22.245 ms

Display: 0x00002390

View: Mixed

Source: C:\br\tests\la\cache.c

```
{
0000237C: 9421FC80 stwu   rsp,-896(rsp)
00002380: 7C0802A6 mflr   r0
00002384: 90010384 stw    r0,900(rsp)
00002388: 39610380 addi   r11,rsp,896
0000238C: 480005B1 bl     __save_gpr (0x293c)
```

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

A

Debug Initialization Files

A debug initialization file is used to initialize the target board before the debugger downloads your program's code. The primary purpose of an initialization file is to ensure that the target memory is initialized properly before it is accessed.

This appendix contains these topics:

- 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 when 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 when debugging with MetroTRK.

Several examples of initialization files are provided for the supported evaluation boards and can be found in these locations in the CodeWarrior installation directory:

- `PowerPC_EABI_Support\Initialization_Files\BDM\`
- `PowerPC_EABI_Support\Initialization_Files\JTAG\`

Debug Initialization File Commands

This section explains debug initialization file commands, and has these sections:

- Debug Initialization File Command Syntax
- Descriptions and Examples of Commands

Debug Initialization File Command Syntax

The following list shows the rules for the syntax of debug initialization file commands.

- Any white spaces and tabs are ignored.
- Character case is ignored in all commands.
- You can enter a number in hex, octal, or decimal:
  - Hex - preceded by 0x (0x00002222 0xA 0xCAfeBeaD)
  - Oct - preceded by 0   (0123 0456)
  - Dec - starts with 1-9 (12 126 823643)
- Comments start with a “;” or “#”, and continue to the end of the line.

Descriptions and Examples of Commands

This section has the descriptions and examples of these commands:

- reset
- setMMRBaseAddr
- sleep
- writemem.b
- writemem.w
- writemem.l
- writemmr
- writereg
- writespr
- writeupma
- writeupmb

Each subsection explains these individual command lists:

- The command name

- A brief description of the command
- Command usage (prototype)
- Command examples
- Any important notes about the command

reset

The `reset` command is specific to debugging through the CCS protocol.

Description	This command determines a target reset depending on its parameter.
Usage	`reset <value>`, where `<value>` can be 0 or 1. Value 0 determines a reset to user and value 1 determines a reset to debug.
Example	`reset 0`

setMMRBaseAddr

The `setMMRBaseAddr` command works only with target boards that use the 825x/826x processors.

Description	The debugger requires the base address of the memory mapped registers on the 825x/826x since this register is memory mapped itself. This command must be in all debug initialization files for the 825x/826x processors. This command informs the debugger plug-in of the base address, which allows you to send any `writemmr` commands from the debug initialization file, as well as read the memory mapped registers for the register views.
Usage	`setMMRBaseAddr<value>`, where `<value>` is the base address for the memory mapped registers.
Example	`setMMRBaseAddr 0x0f00000`

sleep

Description	Causes the processor to wait the specified number of milliseconds before continuing to the next command.
Usage	`sleep <value>`
Example	`sleep 10   # sleep for 10 milliseconds`

writemem.b

Description	Writes data to a memory location using a byte as the size of the write.
Usage	`writemem.b <address> <value>`, where: • `<address>` — the hex, octal, or decimal address in memory to modify • `<value>` — the hex, octal, or decimal value to write at the address
Example	`writemem.b 0x0001FF00  0xFF   # Write 1 byte to memory`

writemem.w

Description	Writes data to a memory location using a word as the size of the write.
Usage	`writemem.w <address> <value>`, where: • `<address>` — the hex, octal, or decimal address in memory to modify • `<value>` — the hex, octal, or decimal value to write at the address
Example	`writemem.w 0x0001FF00  0x1234   # Write 2 bytes to memory`

writemem.l

Description	Writes data to a memory location using a long as the size of the write.
Usage	`writemem.l <address> <value>`, where: • `<address>` — the hex, octal, or decimal address in memory to modify • `<value>` — the hex, octal, or decimal value to write at the address
Example	`writemem.l 0x00010000 0x00000000 # Write 4 bytes to memory`

writemmr

Description	Writes a value to the specified MMR (Memory Mapped Register). All memory mapped register names for the supported processors should be accepted by this command. If any registers are found to not be supported, writemem commands can be used to accomplish the register modification.

Usage	`writemmr < register name> <value>`

Example	`writemmr SYPCR   0xffffffc3`
	`writemmr RMR     0x0001`
	`writemmr MPTPR   0x3200`

writereg

Description	Writes data to the specified register on the target. All register names that are part of the core processor are supported including GPRs and SPRs.
Usage	`writereg <registerName> <value>`
Example	`writereg MSR 0x00001002`

writespr

Description	Writes the value to the SPR with number regNumber, which is the same as writereg SPRxxxx but allows you to enter the SPR number in other bases (hex/octal/decimal).
Usage	`writespr <regNumber> <value>`, where: • `<regNumber>` — a hex/octal/decimal SPR number (0-1023) • `<value>` — a hex/octal/decimal value to write to SPR
Example	`writespr 638 0x02200000`

writeupma

Description	Maps the user-programmable machine (UPM) registers to define characteristics of the memory array.
Usage	`writeupma <offset> <ram_word>`, where: • `<offset>` — 0-3F, as defined in the UPM transaction type table in the Memory Controller section of the Motorola manual • `<ram_word>` — UPM RAM word for that offset
Example	`writeupma 0x08 0xffffcc24`

writeupmb

Description	Maps the user-programmable machine (UPM) registers to define characteristics of the memory array.
Usage	`writeupma <offset> <ram_word>`, where: • `<offset>` — 0-3F, as defined in the UPM transaction type table in the Memory Controller section of the Motorola manual • `<ram_word>` — UPM RAM word for that offset
Example	`writeupma 0x08 0xffffcc24`

B

Memory Configuration Files

A memory configuration file contains commands that define the accessible areas of memory for your specific board.

This appendix contains these topics:

- Command Syntax
- Memory Configuration File Commands

Command Syntax

The syntax rules for configuration file commands are:

- All syntax is case insensitive.
- Any white spaces and tabs are ignored.
- Comments can be standard C or C++ style comments.
- A number may be entered in hex, octal, or decimal.
  - Hex - preceded by 0x (0x00002222 0xA 0xCAfeBeaD)
  - Oct - preceded by 0 (0123 0456)
  - Dec - starts with 1-9 (12 126 823643)

Memory Configuration File Commands

This section lists the command name, its usage, a brief explanation of the command, examples of how the command may appear in configuration files, and any important notes about the command.

Sample configuration files can be found at this location in CodeWarrior installation directory: `PowerPC_EABI_Support\Intialization_Files\Memory`

range

Description	Allows you to specify a memory range for reading and/or writing, and its attributes.
Usage	`range <loAddr> <hiAddr> <sizeCode> <access>`, where: • `<loAddr>` — start of memory range to be defined • `<hiAddr>` — ending address in the memory range to be defined • `<sizeCode>` — specifies the size, in bytes, to be used for memory accesses by the debug monitor or emulator. • `<access>` — can be `Read`, `Write`, or `ReadWrite`. This parameter allows you to make certain areas of your memory map read-only, write-only, or read/write only to the debugger.

Example	`range    0xFF000000 0xFF0000FF 4 Read`
	`range    0xFF000100 0xFF0001FF 2 Write`
	`range    0xFF000200 0xFFFFFFFF 1 ReadWrite`

reserved

Description	Allows you to specify a reserved range of memory. Any time the debugger tries to read from this location, the memory buffer is filled with the reservedchar. Any time the debugger tries to write to any of the locations in this range, no write will take place.
Usage	`reserved <loAddr> <hiAddr>`, where: • `<loAddr>` — start of memory range to be defined • `<hiAddr>` — ending address in memory range to be defined
Example	`reserved 0xFF000024 0xFF00002F`

reservedchar

Description	Allows you to specify a reserved character for the memory configuration file. This character is seen when you try to read from an invalid address. When an invalid read occurs, the debugger fills the memory buffer with this reserved character.
Usage	`reservedchar <char>`, where `<char>` can be any character (one byte).
Example	`reservedchar 0xBA`

C
Command-Line Tool Options

This appendix describes the command-line tool options that are available for CodeWarrior for EPPC.

This appendix contains these topics:

- Embedded PowerPC Project Options
- Embedded PowerPC Options
- Embedded PowerPC Disassembler Options

Embedded PowerPC Project Options

Table C.1 shows the embedded PowerPC project command-line options.

Table C.1 Embedded PowerPC Project Command-line Options

Option	Description
`-big`	Generates code and links for a big-endian target; this option is the default.
`-little`	Generates code and links for a little-endian target.

Table C.1 Embedded PowerPC Project Command-line Options

<table>
<tr><th>Option</th><th colspan="2">Description</th></tr>
<tr><td rowspan="4"><code>-proc[essor]</code> <em>keyword</em></td><td colspan="2">Specifies the processor for scheduling and inline assembler.</td></tr>
<tr><td><strong>Parameter</strong></td><td><strong>Description</strong></td></tr>
<tr><td><code>401, 403, 405, 505, 509, 555, 56x, 601, 602, 603, 603e, 604, 604e, 740, 750, 801, 821, 823, 850, 860, 5100, 5200, 7400, 7450, 8240, 8260, 827x, 8280, 8540, 8560</code></td><td>These are the processor numbers.</td></tr>
<tr><td><code>generic</code></td><td>This is the default option.</td></tr>
<tr><td rowspan="7"><code>-fp</code> <em>keyword</em></td><td colspan="2">Specifies floating-point code generation options.</td></tr>
<tr><td><strong>Parameter</strong></td><td><strong>Description</strong></td></tr>
<tr><td><code>efpu|spfp</code></td><td>e500 SPE-EFPU hardware FP plus software double FP emulation. This option is only applicable to e500 family of processors, which are not supported by this product.</td></tr>
<tr><td><code>none | off</code></td><td>Indicates not to use floating point.</td></tr>
<tr><td><code>soft[ware]</code></td><td>Indicates software floating-point emulation; this option is the default.</td></tr>
<tr><td><code>hard[ware]</code></td><td>Hardware floating-point codegen.</td></tr>
<tr><td><code>fmadd</code></td><td>Same as the following items:
<code>-fp hard</code>
<code>-fp_contract</code></td></tr>
<tr><td><code>-sdata[threshold]</code> <em>short</em></td><td colspan="2">Sets the maximum size in bytes for mutable data objects before being spilled from a small data section into a data section; the default is 8.</td></tr>
<tr><td><code>-sdata2[threshold]</code> <em>short</em></td><td colspan="2">Sets the maximum size in bytes for constant data objects before being spilled from a constant section into a data section; the default is 8.</td></tr>
</table>

Table C.1 Embedded PowerPC Project Command-line Options

<table>
<tr><th>Option</th><th colspan="2">Description</th></tr>
<tr><td rowspan="4"><code>-model</code> <i>keyword</i></td><td colspan="2">Specifies the code model.</td></tr>
<tr><td><b>Parameter</b></td><td><b>Description</b></td></tr>
<tr><td><code>absolute</code></td><td>Specifies absolute code and data addressing; this is the default option.</td></tr>
<tr><td><code>sda_pic_pid</code></td><td>SDA PIC/PID</td></tr>
<tr><td rowspan="7"><code>-abi</code> <i>keyword</i></td><td colspan="2">Specifies the ABI to use.</td></tr>
<tr><td><b>Parameter</b></td><td><b>Description</b></td></tr>
<tr><td><code>eabi</code></td><td>Specifies EABI; this is the default option.</td></tr>
<tr><td><code>SysV</code></td><td>Specifies SysV ABI without gnu-ism</td></tr>
<tr><td><code>SuSE</code></td><td>Specifies SuSE Linux with gnu-ism</td></tr>
<tr><td><code>YellowDog</code></td><td>Specifies YellowDog Linux with gnu-ism</td></tr>
<tr><td><code>sda_pic_pid</code></td><td>Specifies SDA PIC/PID</td></tr>
<tr><td><code>-g[dwarf]</code></td><td colspan="2">Generates DWARF 1.x debugging information.</td></tr>
<tr><td><code>-gdwarf-2</code></td><td colspan="2">Generates DWARF 2.x debugging information.</td></tr>
</table>

Embedded PowerPC Options

Table C.2 shows the embedded PowerPC command-line options.

Table C.2 Embedded PowerPC Command-line Options

<table>
<tr><th>Option</th><th colspan="2">Description</th></tr>
<tr><td rowspan="5"><code>-align keyword[,...]</code></td><td colspan="2">Specifies structure and array alignment options.</td></tr>
<tr><td><b>Parameter</b></td><td><b>Description</b></td></tr>
<tr><td><code>power[pc]</code></td><td>Specifies PowerPC alignment; this option is the default.</td></tr>
<tr><td><code>mac68k</code></td><td>Specifies Macintosh 680x0 alignment.</td></tr>
<tr><td><code>mac68k4byte</code></td><td>Specifies Mac 680x0 4-byte alignment.</td></tr>
<tr><td></td><td><code>array[members]</code></td><td>Specifies to align members of arrays.</td></tr>
<tr><td><code>-common on|off</code></td><td colspan="2">Specifies whether to move all uninitialized data into a common section; the default is off.</td></tr>
<tr><td><code>-fp_contract | -maf on|off</code></td><td colspan="2">Specifies whether to generate fused multiply-add instructions; the default is off.</td></tr>
<tr><td rowspan="8"><code>-func_align</code> <i>keyword</i></td><td colspan="2">Specifies function alignment.</td></tr>
<tr><td><b>Parameter</b></td><td><b>Description</b></td></tr>
<tr><td><code>4</code></td><td>Specifies four-byte alignment; this is the default.</td></tr>
<tr><td><code>8</code></td><td>Specifies eight-byte alignment.</td></tr>
<tr><td><code>16</code></td><td>Specifies 16-byte alignment.</td></tr>
<tr><td><code>32</code></td><td>Specifies 32-byte alignment.</td></tr>
<tr><td><code>64</code></td><td>Specifies 64-byte alignment.</td></tr>
<tr><td><code>128</code></td><td>Specifies 128-byte alignment.</td></tr>
<tr><td><code>-pool[data] on|off</code></td><td colspan="2">Specifies whether to pool like data objects; the default is on.</td></tr>
<tr><td><code>-profile on|off</code></td><td colspan="2">Specifies whether to generate calls at function entry and exit for use with a profiler.</td></tr>
</table>

Table C.2 Embedded PowerPC Command-line Options

<table>
<tr><th>Option</th><th colspan="2">Description</th></tr>
<tr><td>-rostr | -readonlystrings</td><td colspan="2">Specifies to make string constants read-only.</td></tr>
<tr><td>-schedule on|off</td><td colspan="2">Specifies whether to schedule instructions; the default is off.</td></tr>
<tr><td>-use_lmw_stmw on|off</td><td colspan="2">Specifies whether to use multiple-word load/store instructions for structure copies; the default is on.</td></tr>
<tr><td rowspan="5">-vector keyword[,...]</td><td colspan="2">Specifies AltiVec vectorization options.</td></tr>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>on</td><td>Enables support for vector types / codegen.</td></tr>
<tr><td>off</td><td>Disables vectorization.</td></tr>
<tr><td>[no]vrsave</td><td>Specifies to use VRSAVE prologue/epilogue code.</td></tr>
</table>

Embedded PowerPC Disassembler Options

Table C.3 shows the embedded PowerPC disassembler options.

Table C.3 Embedded PowerPC Disassembler Options

<table>
<tr><th>Option</th><th colspan="2">Description</th></tr>
<tr><td rowspan="3">-fmt | -format *keyword*</td><td colspan="2">Specifies formatting options; this option exists for compatibility reasons.</td></tr>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>[no]x</td><td>Specifies whether to show extended mnemonics; the default is to not show the extended mnemonics.</td></tr>
</table>

Table C.3 Embedded PowerPC Disassembler Options

Option	Description	
`-show` *keyword*`[,...]`	Specifies display options.	
	Parameter	**Description**

	`only | none`	Examples:
		`-show none`
		`-show only,code,data`

	`all`	Specifies to show everything.
	`[no]binary`	Specifies whether to show binary information, such as addresses and opcodes, for object code; the default is to show the binary information.

	`[no]code |`
	`[no]text`
		Specifies whether to show `.text` sections; the default is to show the `.text` sections.

	`[no]data`	Specifies whether to show data; the default is to show data.
	`[no]detail`	Specifies whether to show detailed dump information.
	`[no]extended`	Specifies whether to show extended mnemonics; the default is to show extended mnemonics.

	`[no]exceptions |`
	`[no]xtab[les]`
		Specifies whether to show exception tables; these options also imply the following item:
		`-show data`

	`[no]headers`	Specifies whether to show object headers; the default is to show the object headers.

Table C.3 Embedded PowerPC Disassembler Options

Option	Description	

	`[no]debug |`
	`[no]dwarf`
		Specifies whether to show DWARF information.

	`[no]tables`	Specifies whether to show string and symbol tables; the default is to show the string and symbol tables.
	`[no]xtables`	Specifies whether to show exception tables.
`-[no]relocate`	For DWARF information, specifies whether to relocate addends in `.rela.text` and `.rela.debug`.	
`-xtables on|off`	Specifies whether to show exception tables; the default is off. This option exists for compatibility reasons.	

Index