CodeWarrior™ Development Studio for PowerPC® ISA, Linux® Application/ Platform Edition

Targeting Manual

Revised: 23 March 2005

metrowerks

How to Contact Metrowerks

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

Table of Contents

1 Introduction

This manual explains how to use the CodeWarrior™ Integrated Development Environment (IDE) to develop software for the embedded Linux® operating system. The CodeWarrior™ IDE software supports the embedded Linux operating system on specific target platforms, such as PowerPC™, ARM®, and ColdFire®.

NOTE You can use the CodeWarrior™ Development Studio for Embedded Linux software on a computer running the Linux Operating System (OS). For system requirements, refer the *Quick Start* document located in the CodeWarrior installation directory.
For a list of target platforms supported by this product, see “Supported Target Processors” on page 126.

This chapter has these sections:

- Read the Release Notes
- Features
- About This Manual
- Conventions Used in This Manual
- Documentation Overview

Read the Release Notes

Before using the CodeWarrior™ IDE, read the release notes. The release notes contain important information about last minute changes, bug fixes, incompatible elements, or topics that may not be included in the documentation.

Release notes are located here:

`*CWInstall*/CodeWarriorIDE/`

If you are new to the CodeWarrior™ IDE, read this chapter and the “Getting Started” on page 15 chapter carefully. This chapter provides references to resources of interest to new users. The Getting Started chapter provides an overview of the CodeWarrior development tool.

NOTE The release notes for specific components of the CodeWarrior™ IDE are located at this location:
`CWInstall/CodeWarriorIDE/Release_Notes`

Features

This CodeWarrior product has the following features:

- Newer version of the CodeWarrior™ IDE.
- Support for creating and debugging applications, shared library, and static library.
- Support for debugging multiple applications, threads, and processes at the same time.
- Support for debugging the Linux kernel. For more information, see “Kernel Debugging” on page 136.
- Support for loading and debugging the kernel modules. For more information, see “Kernel Module Debugging” on page 157.
- Support for debugging Linux kernel using custom target platform or board support packages (BSP). For more information, see “Debugging Kernel Using Custom Target Platform/BSP” on page 155.
- XML files with pre-configured settings for debugging the Linux kernel on supported target platforms/board support packages (BSP’s).
- Support for debugging bootloader (u-boot). For more information, see “Debugging u-boot” on page 132.
- Sample projects for debugging applications and kernel modules.
- Integrated version control system. For details, refer the *IDE User’s Guide*.

NOTE The CodeWarrior Development Studio for Embedded Linux product targeting the ColdFire® target platforms currently does not support shared library and multi-thread debugging.

About This Manual

Table 1.1 describes the contents of this manual.

Table 1.1 Manual Contents

Chapter / Appendix	Description
Introduction	a general description of the development tools, where to find the product-specific release notes and release notes for specific CodeWarrior IDE components, product features, contents of this manual, and where to go next and the documentation formats available with the product
Getting Started	detailed description of the various features of the development tool and how to obtain a license for the product
Application Tutorial	a tutorial teaching how to write and debug programs using the CodeWarrior IDE
Target Settings	detailed information on project target settings common across all the target platforms supported by the CodeWarrior™ Development Studio for Embedded Linux
Common Application Debugging Features	detailed description of the application debugging features that are common across all the target platforms supported by the CodeWarrior™ Development Studio for Embedded Linux. This chapter also describes how to set up remote debugging and how to use MetroTRK to debug embedded Linux projects with the CodeWarrior IDE
Other Common Features	describes how to: generate new makefile projects using the Makefile Importer Wizard, strip binary files using the Post Linker Stripper utility, and use the Shell Tool Post Linker utility to add and open shell scripts after the project is built
Target Platform-Specific Features	describes the debugging features specific to PowerPC-based target platforms. This chapter also lists supported target processors, and describes the files and settings that you may require for running your application successfully on an PowerPC-based target platform.
Third Party Cross Compiler Tools	describes the settings that you need to change before building your project using customized cross compiler tools

Table 1.1 Manual Contents (*continued*)

Chapter / Appendix	Description
Using PCS to Build Kernel	overview of the Platform Creation Suite (PCS) tool and steps on how to build your Linux kernel using the PCS tool
Frequently Asked Questions	frequently asked questions about the CodeWarrior IDE

Conventions Used in This Manual

Table 1.2 describes the notation conventions in the targeting manual.

Table 1.2 Notation Conventions in This Manual

Notations	Description
Bold text	used to indicate the name of GUI elements, such as screens, dialog boxes, and commands
Italics	used to highlight or emphasize a specific term or a manual
Hyperlinks	indicates references to chapter titles or section titles in the manual
`Program Code`	used to indicate program code, file name, file path, and directory structure
Menu Command	On Linux: a menu sequence displayed as **GNOME Menu > System Tools > Terminal** can be interpreted as, to launch the **command line**, you must first click **GNOME Menu**, then select System Tools and then click **Terminal**

Documentation Overview

As part of CodeWarrior installation, we provide documentation in the following formats:

- Adobe Acrobat PDF Files
- CodeWarrior Online Help

Adobe Acrobat PDF Files

The CodeWarrior documentation is available as PDF documents in:

`*CWInstall*/CodeWarriorIDE/CodeWarrior/Help/PDF`

To view these documents, you must have the Adobe® Acrobat® Reader™ software. You can download this software from:

`http://www.adobe.com/acrobat`

CodeWarrior Online Help

On Linux systems, the CodeWarrior manuals are provided as HTML files in the folder:

`*CWInstall*/CodeWarriotIDE/CodeWarrior/Help`

For help with questions about using the CodeWarrior™ IDE and error messages, use the CodeWarrior online help,

To view these documents, start the CodeWarrior™ IDE and select **Help > CodeWarrior Help**.

Other Resources

- See the *IDE User's Guide* for complete information about the CodeWarrior Integrated Development Environment and debugger.
- Read "CodeWarrior Development Tools" on page 16 for an overview of how the CodeWarrior™ IDE is used to develop software for embedded Linux running on a supported-target platform.
- See "Application Tutorial" on page 21 for a tutorial showing how to use the CodeWarrior™ IDE to create software for embedded Linux running on a supported-target platform.
- Look for the CodeWarrior tutorials projects on your product CD-ROM.
- To learn how to write device drivers for Linux systems, see `http://www.xml.com/ldd/chapter/book/`.

2

Getting Started

This chapter provides an overview of the CodeWarrior™ development tool.

This chapter has these sections:

- Licensing
- CodeWarrior Development Tools
- CodeWarrior Development Process

NOTE For system requirements information, refer the *Quick Start Guide* located in the CD-ROM root.

Licensing

Web-based licensing is available. It is a server licensing solution that generates FlexLM v8 or later based license keys automatically over the world wide web through a registration/activation process. You can register and activate permanent, node-locked license keys. Metrowerks products are shipped to customers with registration cards that contain a unique registration number. Products that ship with a one year annual support certificate will also have a unique registration number.

During product installation you will be instructed to register at `http://www.metrowerks.com/mw/register`. You can also reach the registration website by selecting the **Help > Register Product** menu command from the IDE main menu. Registration from the website collects the registration code and verifies it against the correct product and gathers contact information. An email will be sent to you with the License Authorization Code and instructions.

In the IDE you can select **Help > License Authorization** to display the License Authorization dialog box. Enter the License Authorization Code and select an ethernet address from the Node lock ID for license dropdown list, if one exists. After entering the authorization code, the CodeWarrior IDE will make an HTTP call to the Metrowerks licensing server with the activation code and generate the permanent

license keys. If necessary, enter your Proxy Settings to connect to the internet. The resulting license keys are automatically updated into the `license.dat` text file of the CodeWarrior product executing the authorization. You can also manually edit the license.dat file per instructions provided in the `License_Install.txt` file in the root folder of your CodeWarrior installation path. If the IDE evaluation period expires prior to activation, you will have to manually edit the `license.dat` file.

CodeWarrior Development Tools

Programming for embedded Linux® on a supported target platform is much like programming for any other target platform in CodeWarrior™ IDE. If you have never used the CodeWarrior™ IDE, then you should read these sections:

- Overview of the CodeWarrior IDE
- Cross Compilers, Linkers, and Related Tools
- CodeWarrior Debugger
- Metrowerks Target Resident Kernel

Overview of the CodeWarrior IDE

The CodeWarrior™ IDE lets you write, compile, and debug your software. The CodeWarrior IDE has a project manager, source code editor, compilers and linkers, and a debugger.

The project manager may be new to those more familiar with command-line development tools. All files and settings related to your project are organized in the project manager. The project manager lets you see your project at a glance, and eases the organization of and navigation among your source code files. The CodeWarrior IDE also manages all build dependencies.

A project may contain multiple *build targets*. A build target is a separate build (with its own settings) that uses some or all of the files of the project. For example, you can have a debug version and a release version of your software as separate build targets in the same project.

For more information about how the CodeWarrior™ IDE compares to a command-line environment, see “CodeWarrior Development Process” on page 18. That short section discusses how various parts of the CodeWarrior IDE implement the features of a command-line development system based on makefiles.

The CodeWarrior™ IDE has an extensible architecture that uses plug-in compilers and linkers to target various operating systems and microprocessors.

For more information about the CodeWarrior™ IDE, read the *CodeWarrior IDE User’s Guide*.

Cross Compilers, Linkers, and Related Tools

The CodeWarrior™ IDE uses the cross compiler tools created using GNU Compiler Collection (GCC) sources to generate code that runs on the embedded Linux platform.

The CodeWarrior IDE setup program installs the proper cross GCC components. GCC components are cross compiler tools that let you build your project files on a Linux host PC. See "Cross Compiler Tools Location" on page 176 for information on locating the cross compiler tools executable binaries for your target platform.

The **GNU Tools** settings panel lets you select the cross compilers and linkers used by the CodeWarrior IDE. For more information on this settings panel, see "GNU Tools" on page 46.

"Target Settings" on page 35 describes the various embedded Linux linker and compiler settings.

CodeWarrior Debugger

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

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

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

For more information about debugging software for CodeWarrior™ Development Studio for Embedded Linux for different target platforms, see "Common Application Debugging Features" on page 55.

Metrowerks Target Resident Kernel

Metrowerks Target Resident Kernel (MetroTRK) is a highly-modular, reusable debug server that resides on the target system and communicates with the CodeWarrior debugger.

On embedded Linux systems, MetroTRK is packaged as a regular Linux application for use with the CodeWarrior debugger.

The MetroTRK source code is provided to you so that you can modify it to work in custom situations.

For more information on MetroTRK, see "Using MetroTRK" on page 56.

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. For complete information on performing tasks like editing, compiling, debugging, and linking, refer to the *CodeWarrior IDE User's Guide*.

The difference between the CodeWarrior™ IDE and traditional command-line environments is in how the software helps you manage your work more efficiently. 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™ IDE relates to a traditional command-line environment.

- Projects
- Editing Source Code
- Compiling
- Linking
- Debugging
- Viewing Preprocessor Output
- Checking Syntax
- Disassembling

Projects

The CodeWarrior *project* is analogous to a makefile, or a collection of makefiles. A CodeWarrior project can contain multiple *build targets*. For example, a project might be configured to build both a debug version and a release version of your executable file.

A major difference between the CodeWarrior™ IDE and `make` is that `make` works backwards from object files to source code files (*backward chaining)*. In contrast, the CodeWarrior™ IDE works forward from source code files to object files (*forward chaining)*.

Another major difference is that `make` defines each step of the build process (such as source to object, object to library, library to executable file) and there may be an arbitrary number of steps during a build. By contrast, the CodeWarrior™ IDE uses a fixed build model for each target: build sub-targets, precompile, compile, pre-link, link, and post-link.

The CodeWarrior IDE lists all the project's files in the project window. The input files include source code files, third-party object code files, libraries, scripts and sub-project files. Header files and documentation files are sometimes included in a project for the convenience of having all files listed in one place; but these files are ignored during the build process.

The CodeWarrior™ IDE also lets you add source code files with unsupported file extensions to your project. You can use the CodeWarrior IDE to associate the unsupported file extensions to a CodeWarrior plug-in compiler. For details, refer to the *CodeWarrior IDE User's Guide*.

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

The CodeWarrior IDE manages all the dependencies 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.

Editing Source Code

The CodeWarrior™ IDE provides an integral text editor. It reads and writes text files in UNIX, Mac OS, Linux, and MS-DOS/Windows formats.

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

The editor window has excellent navigational (code browsing) features that let you switch between related files, locate a particular function, mark a 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, you simply select it 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**.

In Linux, and other command-line environments, object code compiled from a source code file is stored in a binary file. The CodeWarrior IDE stores and manages object files transparently.

Linking

To link object code into a final binary file, select **Project > Make**. This command brings the current project up to date, then links the resulting object code into a final output file.

You control the linker through the CodeWarrior IDE. There is no need to specify a list of object files. The CodeWarrior IDE keeps track of all object files automatically. Use the CodeWarrior IDE project window **Link Order** view to control link order by arranging files in the order in which you want them to be linked.

Use the **GNU Target** settings panel to set the name of the final output file. See "GNU Target" on page 37. and "GNU Linker" on page 44 for more information.

Debugging

To debug a project, make sure that the source file you want to debug has a debug mark next to it in the debugging column of the project window.

When debugging code on remote target systems you will need to make sure that the **Use third party debugger** option is disabled, and that compiler optimizations is set to 0.

To debug applications on the remote target, make sure that you have set up a remote connection, specified remote debugging options, and launched MetroTRK on the target.

For details, see:

- “Create a Remote Connection” on page 58
- “Specify Remote Debugging Options” on page 60
- “Start MetroTRK on the Remote Target” on page 61.

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 how your preprocessed file looks like. You can use this feature to track down bugs caused by macro expansion or other subtleties of the preprocessor.

Checking Syntax

To check the syntax of a file in your project, select the file in the project window and select **Project > Check Syntax**. If syntax or compilation errors are detected in the selected file, a message window appears and displays the information about the errors.

Disassembling

To disassemble a compiled file in your project, select the file in the project window and select **Project > Disassemble**. After disassembling a file, the CodeWarrior IDE creates a `.dump` file that contains the disassembled file’s object code in stabs format. The `.dump` file appears in a new window.

3

Application Tutorial

The sections in this tutorial take you step-by-step through the CodeWarrior™ IDE programming environment. This tutorial does not teach you programming. It teaches you how to use the CodeWarrior™ IDE to write and debug applications for a target platform.

The program in this tutorial is a simple application that displays a text message in the current terminal window. The project used to build the program is based on the CodeWarrior C/C++ project stationery.

The tutorial has these sections:

- Create the Project
- Compile and Run the Project
- Debug the Application

Create the Project

This section shows you how to create a CodeWarrior project using the EPPC New Project Wizard and how to set up the project to make a standalone application. The CodeWarrior™ Development Studio for PowerPC ISA, Linux® Application/Platform Edition v2.0 software allows you to develop applications for both hardware floating point and software floating point. During installation, all the settings (cross tools and libraries) required for creating an application are automatically copied to the correct location.

The steps required to do this are:

1. Create a New Project
2. Remove the Place-holder File
3. Add a New Source File to the Project

Create a New Project

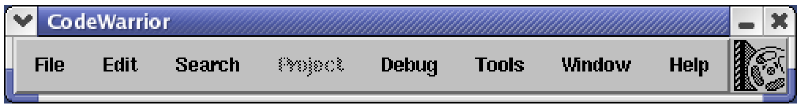
To create a new project using the CodeWarrior™ IDE:

1. Start the CodeWarrior IDE.
   a. Navigate to *`CWInstall`*`/CodeWarriorIDE/`.
   b. Double-click the `cwide` script file to start the CodeWarrior™ IDE. A message box appears asking you to confirm whether to run the application or to display the contents of the script file.

NOTE Alternatively, type `./cwide` or navigate to *`CWInstall`*`/CodeWarriorIDE/CodeWarrior/` and type `sudo ./CodeWarrior_IDE` in the terminal window to start CodeWarrior IDE.

   c. Click **Run**. The CodeWarrior window (Figure 3.1) appears as a floating toolbar on the Linux® desktop.

Figure 3.1 CodeWarrior Window



2. Create a new project using the EPPC New Project Wizard.

TIP Before you start creating a new project, we recommend you to add a remote connection in the **Remote Connections** panel using **Edit > Preferences**. For more information, see “Create a Remote Connection” on page 58.

   a. Select **File > New**. The **New** dialog box appears.
   b. Select **EPPC New Project Wizard**.

NOTE The default location for a new project is the same name as the **Project name** in the drive where the CodeWarrior IDE is installed. For example, if the **Project name** is `test` then the default location of this project will be `{Your_Home_Area}/test`, where `{Your_Home_Area}` is the your home directory.

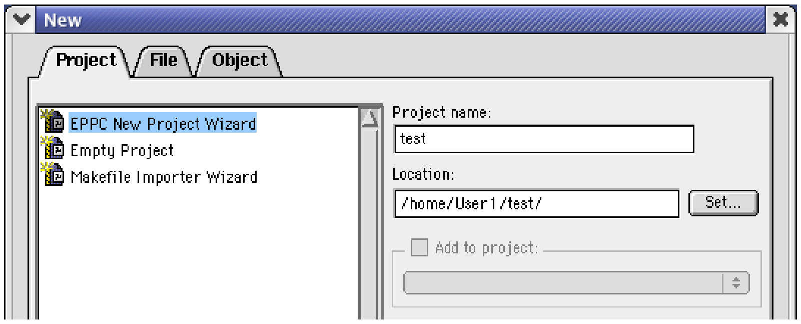
   For this tutorial we will create the tutorial project in a location other than the default location.

   c. Click **Set** to change the location of the new project. The **Create New Project** dialog box appears.
   d. Enter the project name in the **File name** text box. For example, `test`.
   e. Navigate to the directory where you want to store the project.

f. Ensure that the **Create Folder** checkbox at the bottom of the **Create New Project** dialog box is checked. The CodeWarrior IDE lets you create your own project folder that will hold all your project files.

g. Click **Save** to return to the **New** dialog box. The name of the project you specified now appears in the **Project name** text box (Figure 3.2).

Figure 3.2 The New Dialog Box



Project name: test
Location: /home/User1/test/

h. Click **OK** to continue. The first page of the **EPPC New Project Wizard** appears.

3. Select the stationery for which you want to create a project.

   a. Select the **Application level debug** option in the stationery type list.

NOTE For creating a kernel module project from the stationery, select the **Kernel module level debug** option in the stationery type list.

   b. Click **Next**. The second page of the **EPPC New Project Wizard** appears.

4. Select the output type and the programming language of the stationery.

   a. Select the **Application**, **Shared Library**, **Static Library** options in the output type list to display the C/C++ project stationery choices. For this tutorial, let us select the **Application** item in the output type list and the **CPP** item in the languages list (**CPP** stands for C++).

NOTE This wizard page does not appear, if you have selected the **Kernel module level debug** option from the stationery type list.

   b. Click **Next**. The third page of the **EPPC New Project Wizard** appears.

5. Select the target platform and the target processor.

a. Select the board type for which you are creating the stationery from the board type list (left pane). For this tutorial, let us select the **85xx** item.

b. Select the corresponding target processor for the selected board type from the processors list. For this tutorial. let us select the **PowerPC 8560** item.

c. Click **Next**. The fourth page of the **EPPC New Project Wizard** appears.

6. Specify a remote connection for this project.

NOTE This page displays all those remote connection names that you added using the **Remote Connections** panel. See "Create a Remote Connection" on page 58 for steps on how to create a remote connection for your project.
The page display is empty, if you do not have a remote connection set in the **Remote Connections** panel.

a. Select a MetroTRK-based (*EPPC Linux MetroTRK*) remote connection.

b. Click **Next**. The fifth page of the **EPPC New Project Wizard** appears.

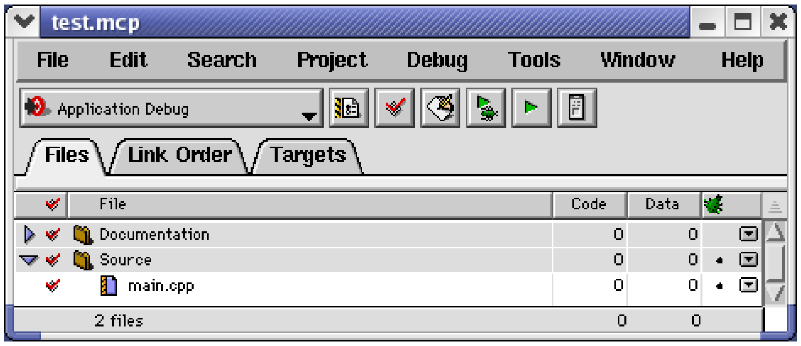
7. Specify the path on the target platform where you want the executable binary to be downloaded during the debugging process.

a. Enter `/home/sample` in the text box.

NOTE This tutorial assumes that you have an account named `sample` on the target platform running the embedded Linux operating system.
Ensure that the specified remote path exists and contains read/write permissions.

b. Click **Finish**. The CodeWarrior IDE creates the new C++ project (Figure 3.3).

Figure 3.3 The Project Window



File	Code	Data
Documentation	0	0
Source	0	0
main.cpp	0	0
2 files	0	0

When generating the new project, the CodeWarrior IDE creates these items in the tutorial folder:

- `test.mcp`—The project file
- `Source`— A folder that contains the generic (place-holder) source file (`main.cpp`). The place-holder filename may vary depending on the language you selected in the wizard.
- `test_Data`—The project data folder
- `bin`—A folder that will contain the executable binary after the project is built successfully

NOTE The project data folder has files that contain information about the project file, various Target Settings, and object code. If the contents of this folder are changed, the CodeWarrior IDE may lose project settings. Do not change the contents of this folder.

Remove the Place-holder File

Part of the stationery used to create your new project is the place-holder source file. For example, `main.cpp`.

Normally, you might use this file as a starting point to write your program. But, as part of this tutorial, you will need to remove the place-holder file from the project and create your own source file.

Select the place-holder file in the project window by clicking it once. With the file selected, select **Edit > Remove**. The CodeWarrior IDE asks you if you are sure you want to remove the file from the project. Click **OK** to confirm removal. The place-holder file is no longer visible in the project window.

NOTE The place-holder file is removed from the project and does not appear in the project window. However, this file is not deleted from the project folder. You may leave the file in the project folder or remove it without affecting your project.

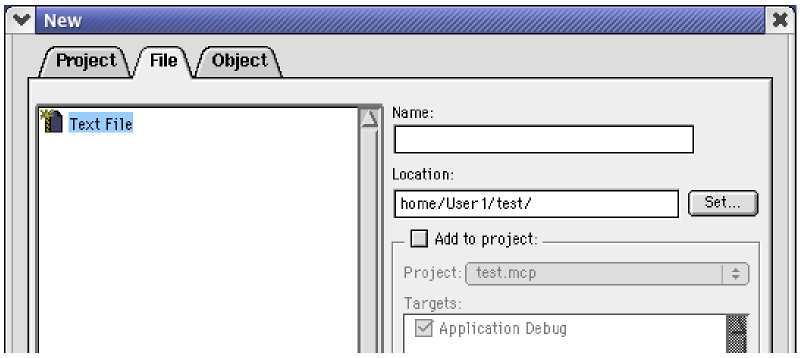
Add a New Source File to the Project

After removing the place-holder file from the project you can create your own source file in place of the place-holder file you removed. You also need to add the new source file to the project you created so that the source code in the new source file is compiled with the rest of the project.

NOTE Before you add a new text file to your project window, ensure that the *Source* item in the project window is selected.

1. Create a new text file.
   a. Select **File > New**. The **New** dialog box appears.
   b. Click the **File** tab in the **New** dialog box. The **File** page (Figure 3.4) appears.

Figure 3.4 The File Page



   c. Select the **Text File** item.

NOTE You may also select **File > New Text File** to create a new text file.

   d. Type `MyHello.cpp` as the file name of the text file in the **File name** text box. The `.cpp` extension enables the CodeWarrior IDE to recognize this file as a source code file.

NOTE If your project is a C project, you must specify `.c` as the extension of your file name.

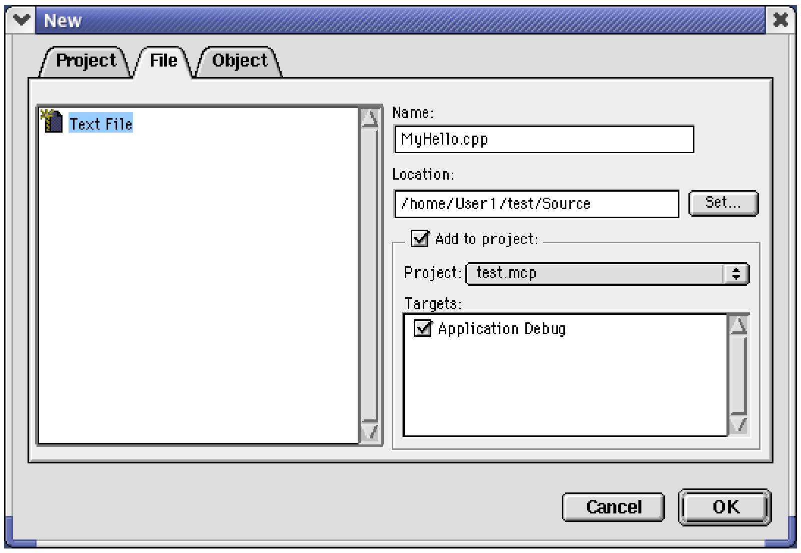
NOTE The **File Mappings** settings panel lets you associate file name extensions with a CodeWarrior plug-in compiler. For details, refer to the *CodeWarrior IDE User's Guide*.

   e. The **Location** text box displays the path of the project folder you created. The `MyHello.cpp` file is stored in this location.

2. Add the source file to the project.

a. Check the **Add to Project** checkbox. The **Project** list box (Figure 3.5) becomes available and the **Targets** box displays the names of the build targets available for this project.

Figure 3.5 Adding Source File to the Project



Name: MyHello.cpp
Location: /home/User1/test/Source
☑ Add to project:
Project: test.mcp
Targets:
☑ Application Debug

b. Select `test.mcp` from the **Project** list box.

3. Select build targets to which the file should be added.

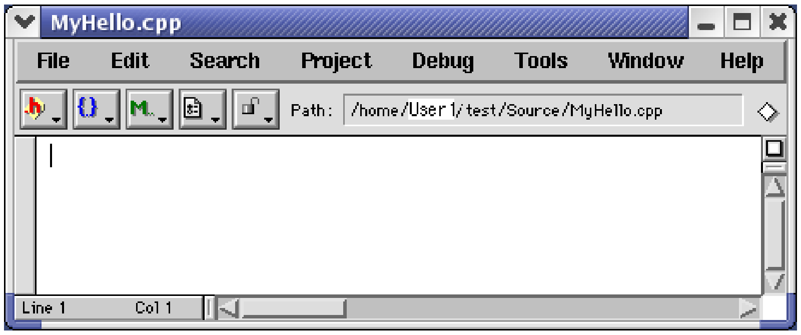
   a. Check the **Application Debug** checkbox, if not checked.

   b. Click **OK** to continue. The CodeWarrior IDE adds the file to the project and displays the file in the project window.

NOTE If you do not see the `MyHello.cpp` file, the **Source** group may be closed. Click the plus sign to the left of folder icon to open the group.

Additionally, the **MyHello.cpp** editor window (Figure 3.6) is displayed.

Figure 3.6 Editor Window



4. Enter the source code.

   Enter the source code of Listing 3.1 into the editor window.

Listing 3.1 New application source code

```
#include <iostream.h>

int main(int argc, char**argv) {

    cout << "My first CodeWarrior application!" << endl << endl;
    return to system; // a deliberate syntax error
}
```

5. Save the new source file.

   To save your source file, select **File > Save**.

   You are now ready to compile and run and debug the project using the CodeWarrior™ IDE.

Compile and Run the Project

Before continuing, you need to set up the project for debugging. This section shows you how to launch the debugger from within the CodeWarrior™ IDE. This section covers the following topics:

- Select the Appropriate Linker
- Set Up Remote Debugging
- Build the Application
- Fix the Error

Select the Appropriate Linker

NOTE By default, the project stationery used to create this project is set up to use the linker specific to the target platform for which you are writing the application.

To check target settings:

1. Select **Edit >** ***Target*** **Settings** (where ***Target*** is the name of the current build target displayed in the project window) to open the **Target Settings** window.
2. Select **Target Settings** from the **Target Settings Panels** list. The **Target Settings** panel appears. For more information about the **Target Settings** panel, see “Target Settings” on page 35.
3. Select the appropriate linker from the **Linker** list box.
4. Click **Save** to save the new linker setting and close the **Target Settings** window.

   The build target is now set up to use the selected linker.

NOTE This change only applies to the current build target. Before using another build target, you must switch to the appropriate linker in that build target as well.

Set Up Remote Debugging

The CodeWarrior™ IDE lets you debug code on the remote target. The CodeWarrior debugger uses a Linux program called Metrowerks Target Resident Kernel (MetroTRK) to control the debug session on the remote system. MetroTRK allows the CodeWarrior IDE to connect to a remote system via serial connections or ethernet connections.

Before you can debug a project remotely, you need to:

- Set up the target board
- Create a remote connection for debugging
- Specify remote debugging options
- Launch MetroTRK on the target board

“Remote Debugging Setup” on page 57 provides detailed information on creating a remote connection, specifying remote debugging options, and launching MetroTRK on the remote target.

Build the Application

Select **Project > Make**. The CodeWarrior IDE calls the GCC cross compiler tools to compile and link your code into the finished application.

While the CodeWarrior IDE is building your project, the check marks to the left of the files in the project window are erased. This indicates that the files no longer need to be built. In the **Code** column, the number of bytes of code is updated.

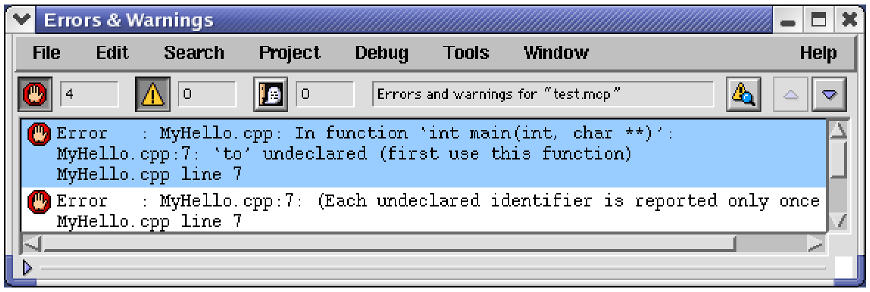
The CodeWarrior IDE displays any errors it finds during the make process in the **Errors & Warnings** window. Syntax errors are usually the result of typos, incorrect variable definitions, or incorrect class information. The tutorial code has one deliberate error, which is discussed in the next section.

Fix the Error

The **Errors & Warnings** window (Figure 3.7) displays a list of errors the compiler found when the CodeWarrior IDE tried to make the project. In this case, there is only one error.

The source view of the **Errors & Warnings** window is editable. You can edit and save your code directly from this window. There is no need to open up the source file and try to find the error manually. This is a great time saver if you have many errors.

Figure 3.7 The Errors & Warnings Window



```
Error   : MyHello.cpp: In function 'int main(int, char **)':
MyHello.cpp:7: 'to' undeclared (first use this function)
MyHello.cpp line 7
Error   : MyHello.cpp:7: (Each undeclared identifier is reported only once
MyHello.cpp line 7
```

The line of code shown in Listing 3.2 contains a syntax error. Fix this line and recompile the project.

Listing 3.2 Syntax error

```
return to system; // a deliberate syntax error
```

1. Correct the problem line.
   a. Change the problem line to:
      ```
      return 0; // fixed!
      ```
2. Save the corrected code.
   a. Select **File > Save** to save the file. Close the **Errors & Warnings** window.
3. Build the project again.

a. Select **Project > Make** to build the project again. The project should build without errors this time.

Debug the Application

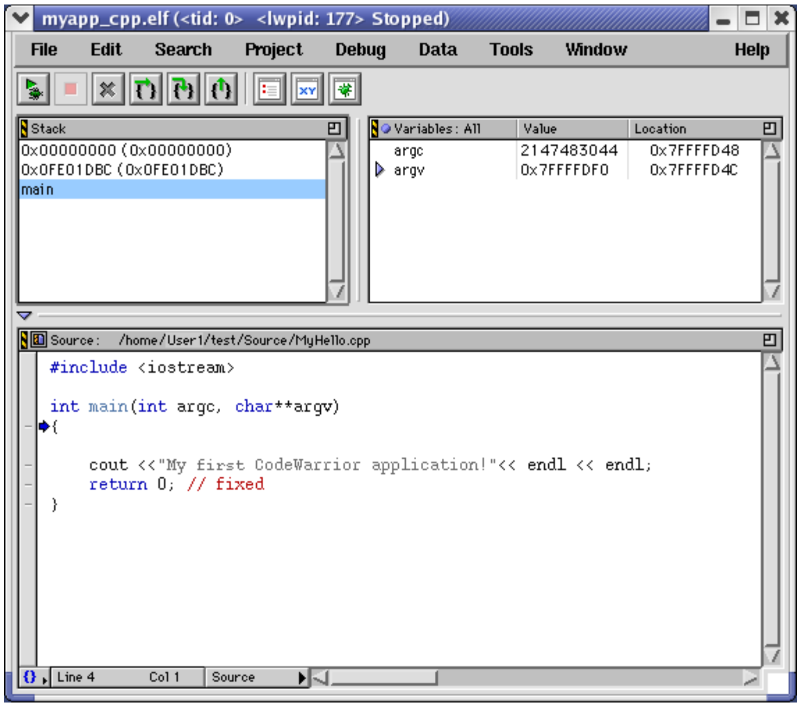
This section explains you how to test whether the application actually runs as expected.

1. Launch the Debugger.

   a. Select **Project > Debug**. The CodeWarrior IDE launches the debugger, passing it the application generated by the build, and hides any open editor windows. The debugger then displays the debugger window (Figure 3.8).

NOTE To be able to successfully debug regular applications, the following library file `ld.so.1` must exist unstripped on the target platform. If the above library is a symbolic link then the file it points to must be unstripped.

Figure 3.8 Debugger Window



```
0x00000000 (0x00000000)
0x0FE01DBC (0x0FE01DBC)
main
```

Variables: All	Value	Location
argc	2147483044	0x7FFFFD48
argv	0x7FFFFDF0	0x7FFFFD4C

```
#include <iostream>

int main(int argc, char**argv)
{

    cout <<"My first CodeWarrior application!"<< endl << endl;
    return 0; // fixed
}
```

NOTE For a detailed description of all of the components of CodeWarrior debugger window, see the *CodeWarrior IDE User’s Guide*.

2. Select **Debug > Step Over**.
3. Step through the rest of the application until you get to the end of the program.
4. Select **Debug > Kill**.

You have successfully completed your first application by using the CodeWarrior IDE.

4 Target Settings

The CodeWarrior™ IDE uses target settings to determine how it compiles, links, edits, and debugs your project's build targets. This chapter discusses those target settings panels that are specific to embedded Linux® programming. See the *CodeWarrior IDE User's Guide* for information about other settings panels.

NOTE For details on Linux kernel debugging target settings panels, see "Target Platform-Specific Target Settings Panels" on page 165.

This chapter has these topics:

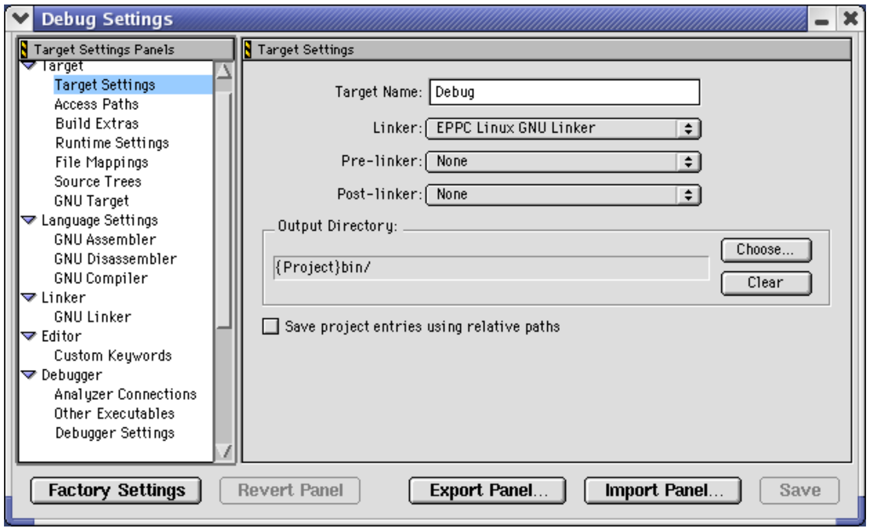
- Overview
- Target Settings Panels

Overview

Target settings are organized into panels that you can display in the CodeWarrior IDE ***Target*** **Settings** window. ***Target*** is the name of the current build target displayed in the project window. Different settings panels control various properties of a build target. Figure 4.1 shows a sample ***Target*** **Settings** panel for a target platform supported by the CodeWarrior™ Development Studio for Embedded Linux.

NOTE The **Linker** option may vary depending on the target platform for which you are writing the application using the CodeWarrior™ Development Studio for Embedded Linux.

Figure 4.1 Target Settings Window



Target Name: Debug
Linker: EPPC Linux GNU Linker
Pre-linker: None
Post-linker: None
Output Directory: {Project}bin/
☐ Save project entries using relative paths

Target Settings Panels

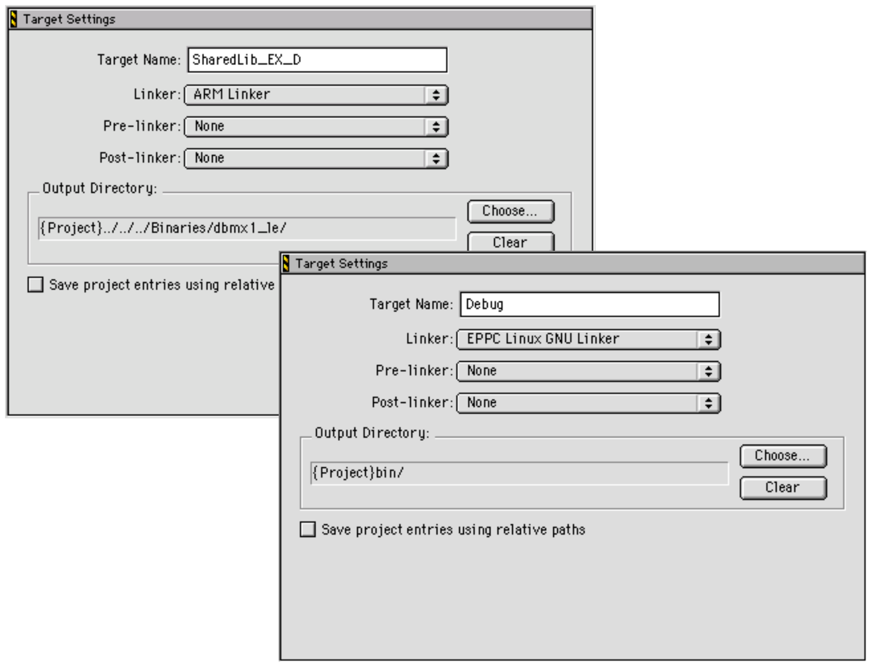
This section discusses only those target settings panels that are specific to embedded Linux programming. These target settings panels are:

- Target Settings
- GNU Target
- GNU Assembler
- GNU Disassembler
- GNU Compiler
- GNU Post Linker
- GNU Linker
- GNU Environment
- GNU Tools
- Debugger Settings
- Console I/O Settings
- Debugger Signals

Target Settings

The **Target Settings** panel is the most critical panel in the CodeWarrior™ IDE. Figure 4.2 shows two **Target Settings** panels for two different target platforms supported by the CodeWarrior™ Development Studio for Embedded Linux.

Figure 4.2 Target Settings Panels for Different Target Platforms



Target Settings

Target Name: SharedLib_EX_D
Linker: ARM Linker
Pre-linker: None
Post-linker: None
Output Directory: {Project}../../Binaries/dbmx1_le/
Choose... Clear
☐ Save project entries using relative

Target Settings

Target Name: Debug
Linker: EPPC Linux GNU Linker
Pre-linker: None
Post-linker: None
Output Directory: {Project}bin/
Choose... Clear
☐ Save project entries using relative paths

This is the panel where you select the operating system and the microprocessor your project runs on. You specify the name of the build target, as well as which linker to use for the target platform. Also, you may select the post linker you want to use for stripping the final output file size. For more information on post linker, see “Stripping Binary Files” on page 120.

When you select a linker, you are specifying the target operating system or CPU. The other panels change to reflect your choice.

The visibility of other related panels is affected by the linker you select. Therefore, you must first select a linker before you can specify other target platform-specific options.

NOTE The **Linker** and the **Post-linker** options may vary depending on the target platform for which you are writing the application using the CodeWarrior™ Development Studio for Embedded Linux.

This settings panel contains the following options:

Target Name

Enter a name (26 or fewer characters) for the selected build target in this text box. The target name you specify here appears in the build target list box in the project window. The CodeWarrior™ IDE assigns a default target name based on the stationery you select in the EPPC New Project Wizard. For example, Application Debug, where *Application* is the output type (application), and *Debug* is the debug version.

NOTE The CodeWarrior Embedded Linux targeting ARM and ColdFire target platforms has `cpp_app_debug` as the default target name, where *cpp* is the C++ language type, *app* is the output type (application), and *debug* is the debug version.

Linker

The **Linker** list box lets you select the linker to use on the current build target. The linkers displayed in this list box vary depending on the target platform for which you are writing your application. For example, if you are writing an application for a PowerPC™-based target platform, the list box displays **EPPC Linux GNU Linker** as choice.

Pre-linker

The **Pre-linker** list box lets you select the pre-linker to use on the current build target. This is currently not implemented for any of the target platforms supported by the CodeWarrior™ Development Studio for Embedded Linux.

Post-linker

The **Post-linker** list box lets you select the post linker to use on the current build target. You can select two types of post linkers:

- GNU Post Linker - Stripper: Used by the linker to generate a stripped version of the binary.
- Shell Tool Post Linker: Used by the linker to display the contents of the shell script file included in the project. For more information, see “Using the Shell Tool Post Linker” on page 123.

NOTE The post linker name may vary depending on the target platform for which you are writing the application using the CodeWarrior™ Development Studio for Embedded Linux.

Output Directory

The **Output Directory** text box displays the selected location where the CodeWarrior™ IDE places the final compiled and linked output file. The default location is the directory that contains the project file (`{Project}`). Click **Choose** to change the location. Click **Clear** to remove the current location.

NOTE All relative paths specified using "`.`" or "`..`" are relative to the output directory.

Save project entries using relative paths

To place the output file in a location other than the output directory, you can specify a relative path after checking the **Save project entries using relative paths** checkbox.

The **Save project entries using relative paths** checkbox causes the CodeWarrior IDE to store the location of a file using a relative path from one of the access paths. The settings include:

- checked—The CodeWarrior IDE stores extra location information to distinctly identify different source files with the same name. The CodeWarrior IDE remembers the location even if it needs to search again for files in the access paths.
- clear—The CodeWarrior IDE remembers project entries only by name. This setting can cause unexpected results if two or more files share the same name. In this case, searching again for files could cause the CodeWarrior IDE to find the project entry in a different access path.

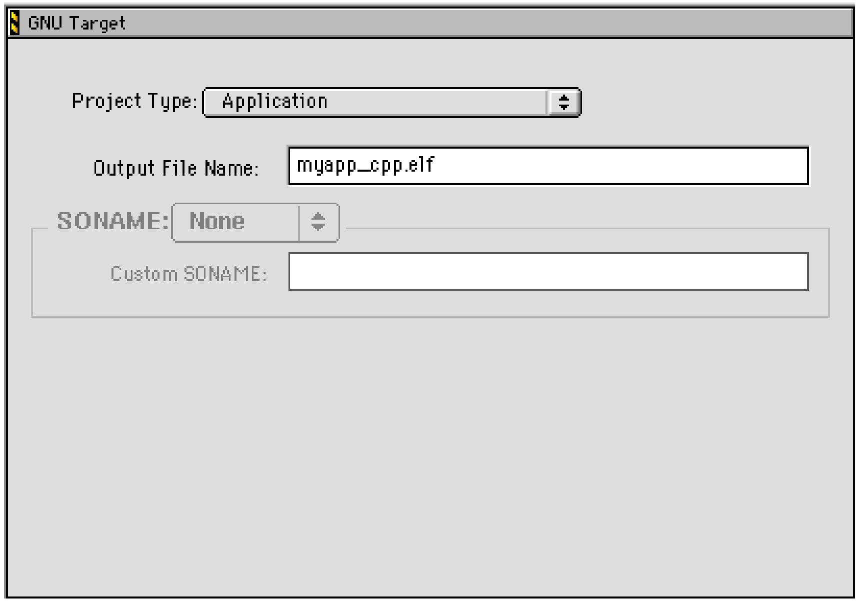
GNU Target

Use the **GNU Target** settings panel (Figure 4.3) to specify these items:

- the project type
- the output file name
- the library SONAME

NOTE This settings panel is similar across all the target platforms supported by the CodeWarrior™ Development Studio for Embedded Linux.

Figure 4.3 GNU Target Settings Panel



Project Type: Application

Output File Name: myapp_cpp.elf

SONAME: None

Custom SONAME:

This settings panel contains the following options:

Project Type

Use the **Project Type** list box to select the project type for the build target. This list box displays *Application*, *Shared Library*, and *Library*.

Table 4.1 defines each project type.

Table 4.1 Project Types

Project Type	Description
Application	A standalone application (such as `cw.elf`)
Shared Library	A library that can be shared by multiple processes (such as `libcw.so`) or dynamically loaded into a process (such as CodeWarrior plug-ins)
Library	A static library (such as `staticlib.a`)

NOTE You can also create Linux kernel modules that can be loaded into the kernel at runtime (such as `printdriver.o`).

Output File Name

Enter the name of the final output file for the build target in this text box.

The CodeWarrior IDE creates this file in the Output Directory. You can place the file somewhere other than the default output directory by specifying a relative path in this field.

Table 4.2 shows the default output file names for different types of projects.

Table 4.2 Default File Names

Project Type	Default Output File Name
Application	cw_dbg.elf and cw_rel.elf
Shared Library	libexample_dbg.so and lbexample_rel.so
Library	libexample_dbg.a and libexample_rel.a
Loadable Module	module_dbg.o and module_rel.o

SONAME

The **SONAME** list box is only available for shared library projects. For detailed information on shared library, see “Shared Library Debugging” on page 63. SONAME stands for *shared object name*. This list box lets you specify how the SONAME file will be named. The SONAME you specify is used by the Linux dynamic loader to locate the shared library on the target platform at runtime.

The options you can select from this list box are:

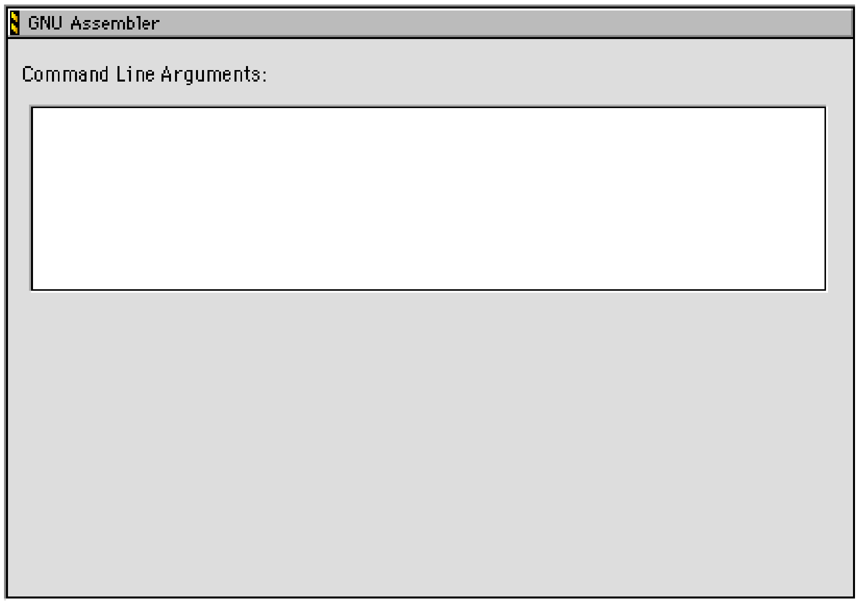
- None — does not assign a SONAME
- Default — assigns the name of the final output file as SONAME
- Custom — assigns a custom SONAME as specified in the **Custom SONAME** text box

NOTE If you specify a custom SONAME, the shared object loader searches for the custom SONAME file on the target while launching the application that uses a shared library. Make sure that the SONAME file exists on the target system and is symbolically linked to the output file.

GNU Assembler

The **GNU Assembler** settings panel (Figure 4.4) lets you specify additional command line options that are passed to the assembler when it is invoked by the CodeWarrior IDE.

Figure 4.4 GNU Assembler Settings Panel



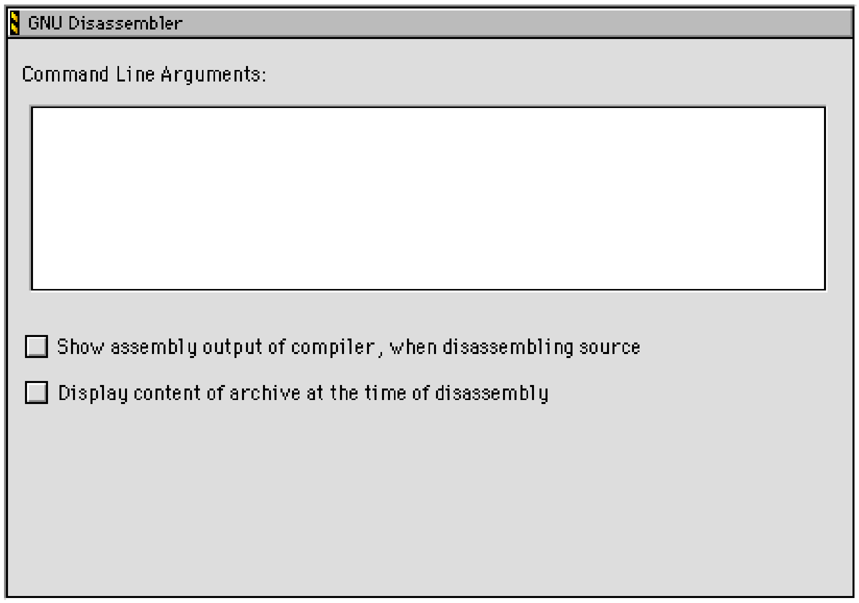
You can enter the command line arguments for the GCC assembler in the **Command Line Arguments** text box. The contents of this text box are passed to the `gcc` command line for each file in your project as they are assembled.

NOTE This settings panel is similar across all the target platforms supported by the CodeWarrior™ Development Studio for Embedded Linux.

GNU Disassembler

The **GNU Disassembler** settings panel (Figure 4.5) lets you display the assembly output of the compiler while disassembling the source. Additionally, this panel lets you display the archiver contents at the time of disassembly.

Figure 4.5 GNU Disassembler Settings Panel


GNU Disassembler

Command Line Arguments:

☐ Show assembly output of compiler, when disassembling source

☐ Display content of archive at the time of disassembly

NOTE This settings panel is similar across all the target platforms supported by the CodeWarrior™ Development Studio for Embedded Linux.

This settings panel contains the following options:

Command Line Arguments

You can specify the command line arguments for the GCC disassembler in the **Command Line Arguments** text box. The contents of this text box are passed to the command line disassembler tool for each file in your project as they are disassembled.

Show assembly output of compiler, when disassembling source

Check this checkbox to use the compiler to disassemble source files. You will also be able to view the assembly output of compiler, while it disassembles the source file.

If this checkbox is cleared, the CodeWarrior IDE uses the disassembler tool to disassemble source files. Additionally, the command line arguments you specify in the **Command Line Arguments** text box are passed to the command line disassembler tool.

Display content of archive at the time of disassembly

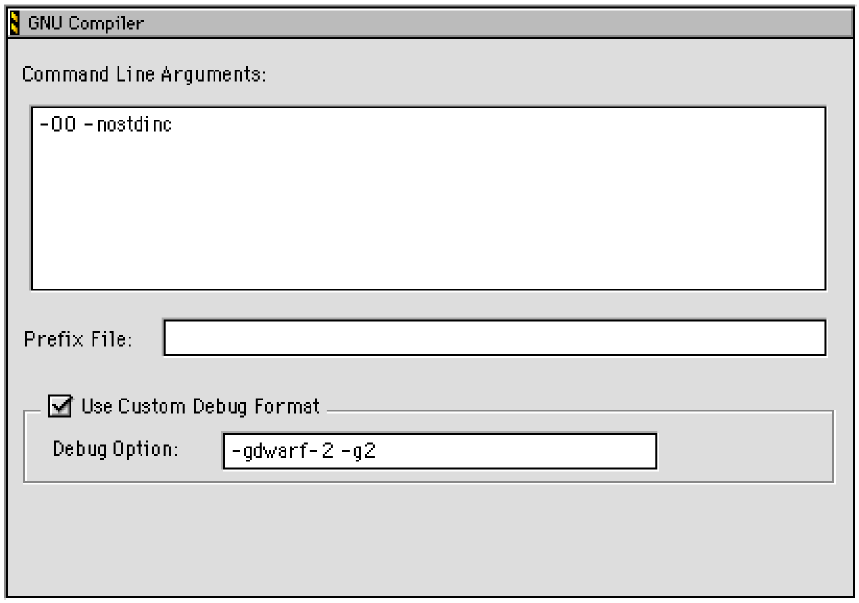
Check this checkbox to use the archiver while disassembling libraries. You will be able to view the list of objects archived in the library while it is being disassembled.

If this checkbox is cleared, the CodeWarrior IDE uses the disassembler tool. Additionally, the command line arguments you specify in the **Command Line Arguments** text box are passed to the command line disassembler tool.

GNU Compiler

The **GNU Compiler** settings panel (Figure 4.6) lets you specify command line arguments, prefix file settings, and the format for generating debugging information.

Figure 4.6 GNU Compiler Settings Panel



Command Line Arguments: -O0 -nostdinc

Prefix File:

☑ Use Custom Debug Format

Debug Option: -gdwarf-2 -g2

NOTE This settings panel is similar across all the target platforms supported by the CodeWarrior™ Development Studio for Embedded Linux.

This settings panel contains the following options:

Command Line Arguments

You can specify the command line arguments for the GCC compiler in the **Command Line Arguments** text box. The contents of this text box are inserted in the `gcc` command line for each file in your project as they are compiled.

Prefix File

The file listed in the **Prefix File** text box corresponds directly to the `-include` parameter of the GCC compiler. The CodeWarrior IDE includes the prefix file before each source file in your project.

Use Custom Debug Format

Check the **Use Custom Debug Format** checkbox to let the compiler use a particular format for generating debugging information.

The format in which you want to generate debugging information is specified in the **Debug Option** text box. The CodeWarrior™ debugger uses the `-gstabs` or `-` Debug With Arbitrary Record Format (`DWARF2`) custom debug format. This format is passed to the GCC cross compiler tools.

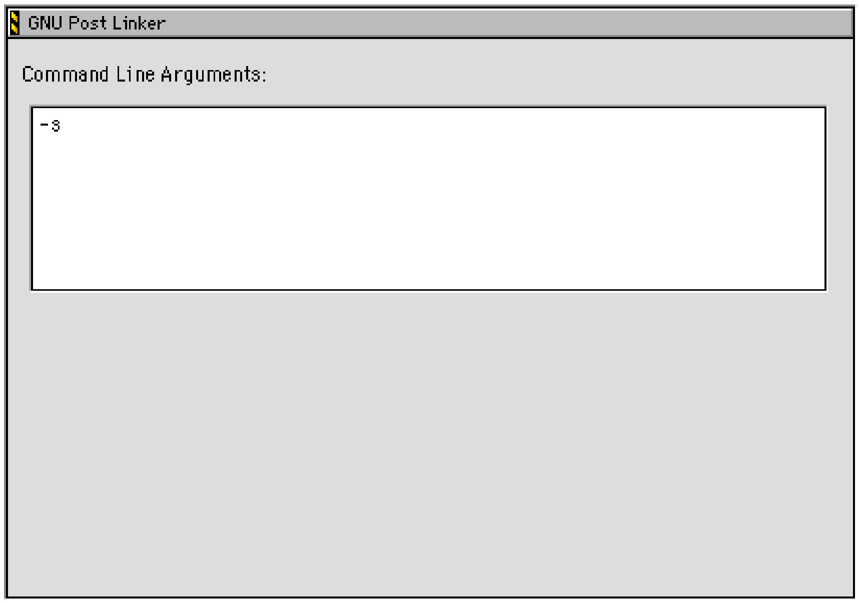
If the **Use Custom Debug Format** checkbox is cleared, debugging information is generated in the default format (`-g`).

GNU Post Linker

The **GNU Post Linker** settings panel (Figure 4.7) lets you specify additional command line option that is passed to the post linker adaptor for generating a binary stripped of the debug section. For detailed information on the post linker stripper feature, see “Stripping Binary Files” on page 120.

NOTE This settings panel is similar across all the target platforms supported by the CodeWarrior™ Development Studio for Embedded Linux.

Figure 4.7 GNU Post Linker Settings Panel



```
-S
```

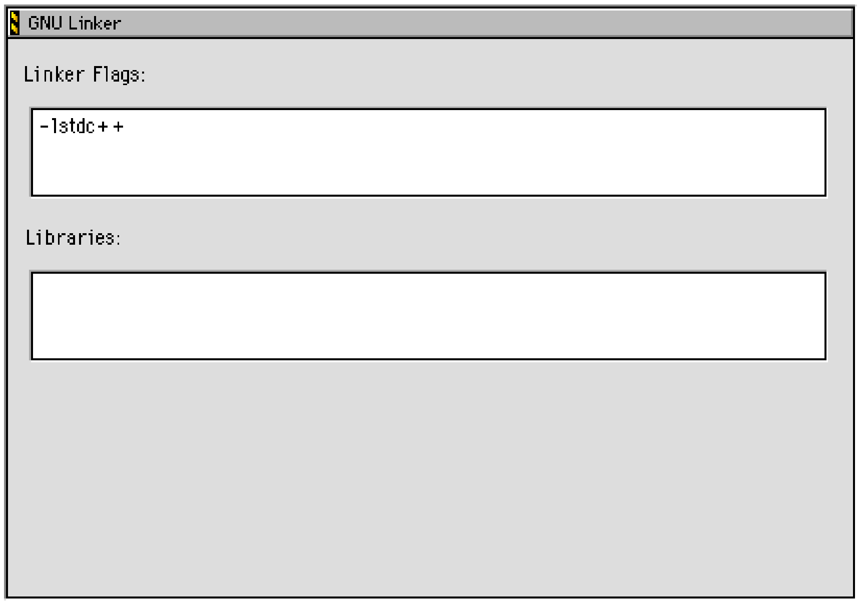
Enter the command line argument in the **Command Line Arguments** text box.

WARNING! Ensure that the command line arguments you specify strips only the debug section from the binary file. For example, if you specify command line argument that removes *ELF Symbol Table* from the binary file, you may not be able to debug the stripped file on the target platform because the *ELF Symbol Table* data is required by the MetroTRK for debugging purpose.

GNU Linker

The **GNU Linker** settings panel (Figure 4.8) lets you specify additional command line options that are passed to the GCC linker when it is invoked by the CodeWarrior IDE.

Figure 4.8 GNU Linker Settings Panel



Linker Flags:

```
-lstdc++
```

Libraries:

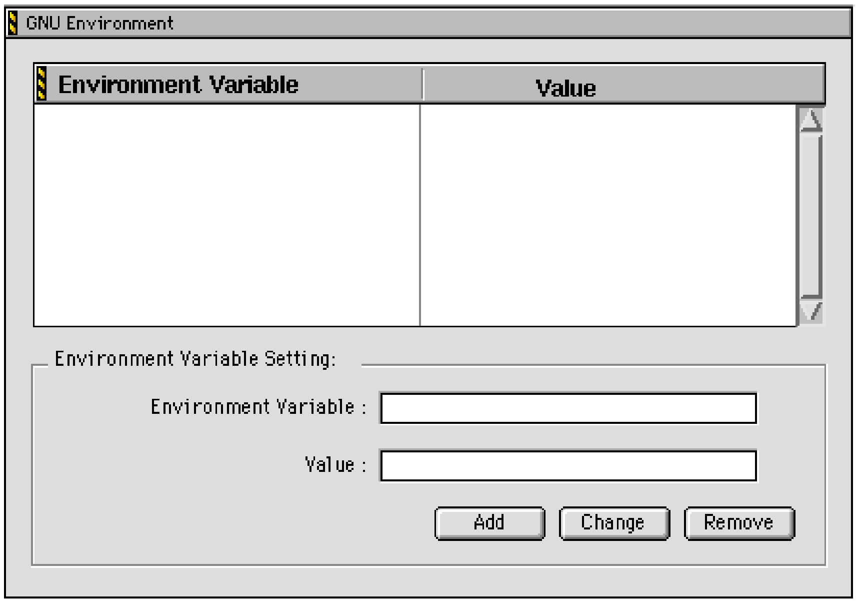
You can enter the command line arguments for the GCC linker in the **Linker Flags** and **Libraries** text boxes. The contents of this text box are passed to the `gcc` command line for each file in your project as they are linked.

NOTE This settings panel is similar across all the target platforms supported by the CodeWarrior™ Development Studio for Embedded Linux.

GNU Environment

The **GNU Environment** settings panel (Figure 4.9) lets you specify environment variables that are passed to the external compiler, linker, assembler, and other build tool processes when they are invoked by the CodeWarrior IDE.

Figure 4.9 GNU Environment Settings Panel



The variables specified in this settings panel are passed to the GCC cross compiler tools. For information on the environment variables, refer to the documentation supplied with the external tools.

NOTE This settings panel is similar across all the target platforms supported by the CodeWarrior™ Development Studio for Embedded Linux.

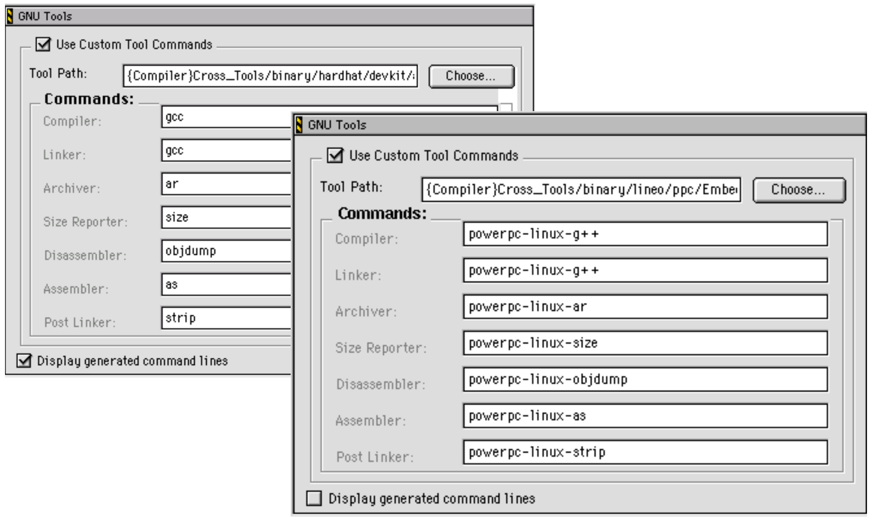
GNU Tools

The **GNU Tools** settings panel contains settings for the cross compiler tool path and executable files that compile, assemble, link, disassemble, archive projects, and report the code and data size of project files. This panel also allows you to specify the executable file used by the post linker. For more information on post linker, see “Stripping Binary Files” on page 120.

Figure 4.10 shows **GNU Tools** settings panel containing different cross compiler tools for two different target platforms.

NOTE The cross compiler tool path and the names of the executable files may vary depending on the target platform for which you are writing the application using the CodeWarrior™ Development Studio for Embedded Linux.

Figure 4.10 GNU Tools Settings Panel



GNU Tools

☑ Use Custom Tool Commands

Tool Path: {Compiler}Cross_Tools/binary/hardhat/devkit/

Choose...

Commands:

Compiler:	gcc
Linker:	gcc
Archiver:	ar
Size Reporter:	size
Disassembler:	objdump
Assembler:	as
Post Linker:	strip

☑ Display generated command lines

GNU Tools

☑ Use Custom Tool Commands

Tool Path: {Compiler}Cross_Tools/binary/lineo/ppc/Embe

Choose...

Commands:

Compiler:	powerpc-linux-g++
Linker:	powerpc-linux-g++
Archiver:	powerpc-linux-ar
Size Reporter:	powerpc-linux-size
Disassembler:	powerpc-linux-objdump
Assembler:	powerpc-linux-as
Post Linker:	powerpc-linux-strip

☐ Display generated command lines

This settings panel contains the following options:

Use Custom Tool Commands

Check the **Use Custom Tool Commands** checkbox to let the CodeWarrior IDE use the GCC cross compiler tools you generated.

Tool Path

The **Tool Path** text box displays the path where cross compiler tools exist on your computer. Click **Choose** to specify the location of the cross compiler tools to be used by the CodeWarrior IDE. The **Choose** button is available only if you check the **Use Custom Tool Commands** checkbox.

NOTE The cross compiler tools path may vary depending on the cross compiler tools you are using. For cross compiler tools path information, see “Cross Compiler Tools Location” on page 176.

Commands Area

The descriptions of the text boxes in the **Commands Area** are as follows:

- In the **Compiler** text box, specify the name of the compiler executable file from the cross compiler tools.

- In the **Linker** text box, specify the name of the linker executable file from the cross compiler tools.
- In the **Archiver** text box, specify the name of the cross compiler tools executable file used for building static libraries.
- In the **Size Reporter** text box, specify the name of the cross compiler tools executable file used for reporting the code and data size of the files in your project after they are compiled. The code and data size of files are displayed in the project window.
- In the **Disassembler** text box, specify the name of the cross compiler tools executable file used for disassembling binary files, libraries, or object files.
- In the **Assembler** text box, specify the name of the cross compiler tools executable file used for assembling the files in your project.
- In the **Post Linker** text box, specify the name of the cross compiler tool executable file used for stripping debug information from the final output file. The file size of the stripped file is reduced. For example, the cross compiler tool executable used across all the target platforms supported by the CodeWarrior™ Development Studio for Embedded Linux is `strip.exe`. For more information, see “Stripping Binary Files” on page 120.

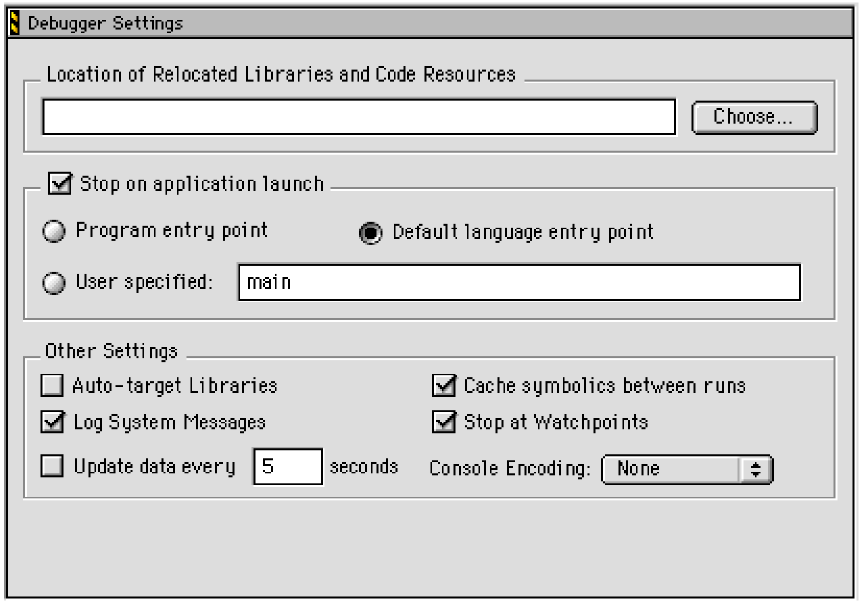
Display generated command lines

Check the **Display generated command lines** checkbox to view command line messages while the project is being built.

Debugger Settings

The **Debugger Settings** panel (Figure 4.11) lets you configure activity logs, data-update intervals, console encoding options, and other debugger-related options.

Figure 4.11 Debugger Settings Panel


Debugger Settings

Location of Relocated Libraries and Code Resources

Choose...

[✓] Stop on application launch

○ Program entry point   ● Default language entry point

○ User specified: main

Other Settings

[ ] Auto-target Libraries   [✓] Cache symbolics between runs

[✓] Log System Messages   [✓] Stop at Watchpoints

[ ] Update data every 5 seconds   Console Encoding: None

NOTE This settings panel is similar across all the target platforms supported by the CodeWarrior™ Development Studio for Embedded Linux.

This settings panel contains the following options:

Location of Relocated Libraries and Code Resources

In this text box, type the path of code resources or relocated libraries required for debugging the project. Alternatively, click **Choose** to select the required files.

Stop on application launch

Check the **Stop on application launch** checkbox to halt program execution at the beginning of a debugging session. You can halt program execution at these entry points:

- **Program entry point —** Select the **Program entry point** option button to halt program execution upon entering the program

- **Default language entry point** — Select the **Default language entry point** option button to halt program execution upon entering a default point defined by the programming language
- **User specified entry point**— Select the **User specified** option button to halt program execution at a specified function. Type the desired function name in the corresponding text box.

Auto-target Libraries

Check the **Auto-target Libraries** checkbox to debug dynamically linked libraries (DLLs) loaded by the target application.

Cache symbolics between runs

Checking the **Cache symbolics between runs** checkbox lets the CodeWarrior IDE save the symbolics information it generates for a project. If you are debugging an application a number of times, checking this checkbox lets the debug sessions start faster.

If this checkbox is cleared, the CodeWarrior IDE discards the symbolics information after each debugging session ends.

NOTE Make sure that the **Cache symbolics between runs** checkbox is checked. Otherwise, the console window will close when the debug session ends.

Log System Messages

Not supported in this release.

Stop at Watchpoints

Not supported in this release.

Update data every n seconds

Type the number of seconds (n) to wait before updating the data displayed in debugging-session windows.

Console Encoding

The **Console Encoding** list box lets you specify the Japanese language encoding for displaying the output of a program in the MetroTRK console window.

The encoding methods you can select are:

- Shift - Japanese Industrial Standard (Shift-JIS)
- Extended Unix Code - Japanese (EUC-JP).

For more information on Japanese encoding methods, see this web site:

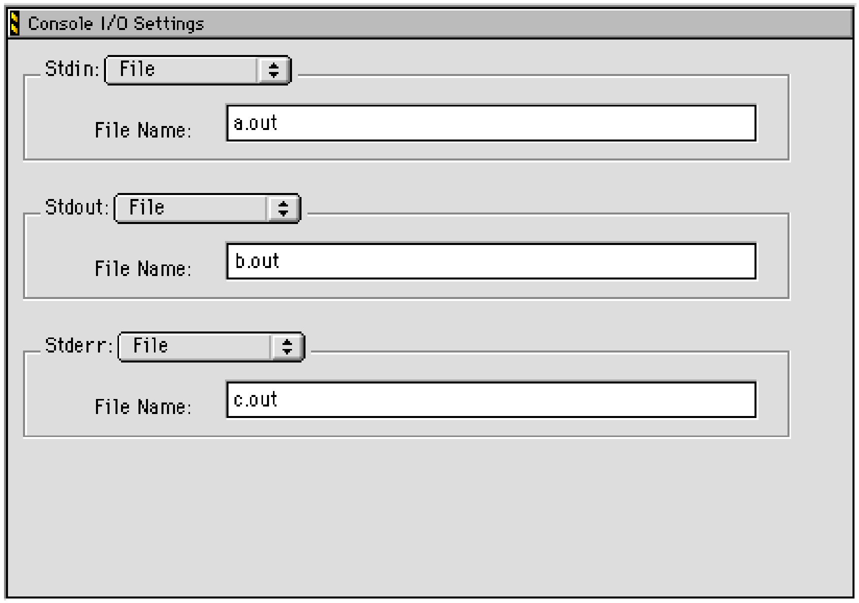
`http://www.tlug.jp`

Console I/O Settings

The **Console I/O Settings** panel (Figure 4.12) lets you specify where to redirect standard input, standard output, and error messages while an application is being debugged.

NOTE This settings panel is similar across all the target platforms supported by the CodeWarrior™ Development Studio for Embedded Linux.

Figure 4.12 Console I/O Settings Panel



Console I/O Settings

Stdin: File
File Name: a.out

Stdout: File
File Name: b.out

Stderr: File
File Name: c.out

You can redirect standard input, standard output, and error messages to:

- a file on the target system
- the debugger console window
- the console window from where you launched MetroTRK.

NOTE Standard input, standard output, and error messages cannot be redirected to the debugger console window when you run an application without the debugger.

If you wish to redirect standard input, standard output, and error messages to a file on the target system, you need to specify the full target-side path of the file. However, if the target-side location of the file is the same as that of MetroTRK, you only need to specify the file name.

This settings panel contains the following options:

Stdin List Box

Use the **Stdin** list box to specify from where you want the CodeWarrior IDE to read standard input.

- Select the **File** option to read standard input from a file.
- Select the **Debugger** option to read standard input from the debugger console window
- Select the **Console I/O** option to read standard input from the console window from where you launched MetroTRK.

Stdout List Box

Use the **Stdout** list box to specify where you want the CodeWarrior IDE to write standard output.

- Select the **File** option to write standard output to a file.
- Select the **Debugger** option to write standard output to the debugger console window
- Select the **Console I/O** option to write standard output to the console window from where you launched MetroTRK.

Stderr List Box

Use the **Stderr** list box to specify where you want the CodeWarrior IDE to send standard error messages.

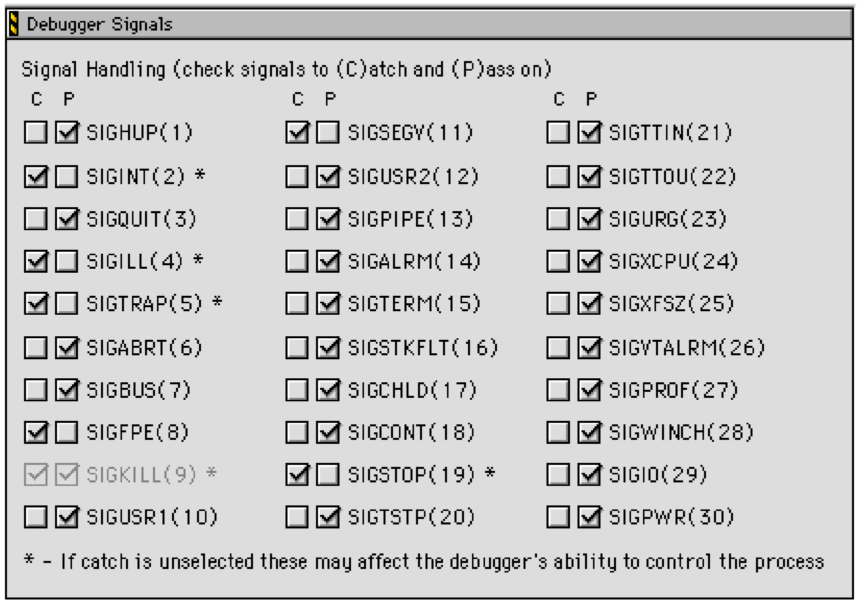
- Select the **File** option to send standard error messages to a file.
- Select the **Debugger** option to send standard error messages to the debugger console window
- Select the **Console I/O** option to send standard error messages to the console window from where you launched MetroTRK.

Debugger Signals

The **Debugger Signals** settings panel (Figure 4.13) is used to specify which signals the CodeWarrior™ debugger should catch and/or pass on to the process being debugged. The settings in this panel are passed to MetroTRK at the beginning of a debug session.

NOTE The **Debugger Signals** settings panel will not be visible in the target settings window unless you create a remote connection to the target platform. For details on creating a remote connection, see “Create a Remote Connection” on page 58.

Figure 4.13 Debugger Signals Settings Panel



Debugger Signals

Signal Handling (check signals to (C)atch and (P)ass on)

C	P		C	P		C	P	
	✓	SIGHUP(1)	✓		SIGSEGV(11)		✓	SIGTTIN(21)
✓		SIGINT(2) *		✓	SIGUSR2(12)		✓	SIGTTOU(22)
	✓	SIGQUIT(3)		✓	SIGPIPE(13)		✓	SIGURG(23)
✓		SIGILL(4) *		✓	SIGALRM(14)		✓	SIGXCPU(24)
✓		SIGTRAP(5) *		✓	SIGTERM(15)		✓	SIGXFSZ(25)
	✓	SIGABRT(6)		✓	SIGSTKFLT(16)		✓	SIGVTALRM(26)
	✓	SIGBUS(7)		✓	SIGCHLD(17)		✓	SIGPROF(27)
✓		SIGFPE(8)		✓	SIGCONT(18)		✓	SIGWINCH(28)
✓	✓	SIGKILL(9) *	✓		SIGSTOP(19) *		✓	SIGIO(29)
	✓	SIGUSR1(10)		✓	SIGTSTP(20)		✓	SIGPWR(30)

* - If catch is unselected these may affect the debugger's ability to control the process

NOTE This settings panel is similar across all the target platforms supported by the CodeWarrior™ Development Studio for Embedded Linux.

If the checkbox in the **C** (catch) column is checked, and a corresponding debugger signal is raised in a process being debugged, MetroTRK sends an event to the CodeWarrior (or host) debugger. Otherwise no event is sent. If the checkbox in the **P** (pass on) column is checked, and the program is continued (either immediately or as a result of a debug command), MetroTRK passes the signal on to the process being debugged.

NOTE Make sure to set the **Debugger Signals** settings panel to factory default by clicking the **Factory Settings** button (not shown in Figure 4.13).

To ensure that the CodeWarrior IDE functions properly and the debugger is able to control the process, make sure that these signals are set to be caught (factory setting):

- SIGINT(2)
- SIGILL(4)
- SIGTRAP(5)
- SIGURG(23)

Not catching these signals may cause certain features of the CodeWarrior IDE to function improperly or to become unavailable.

5

Common Application Debugging Features

This chapter describes how to use the CodeWarrior™ IDE to debug code on the remote target. The debugging features discussed in this chapter are common across all the target platforms supported by the CodeWarrior™ Development Studio for Embedded Linux®.

This chapter has these topics:

Remote Target Debugging - An Overview

The CodeWarrior™ IDE lets you debug code on the remote target system. The CodeWarrior debugger uses a Linux program called Metrowerks Target Resident Kernel (MetroTRK) to control the debug session on the remote target system. MetroTRK allows the CodeWarrior IDE to connect to a remote target system via serial or ethernet connections.

Before you can debug a project remotely, you must also set up certain settings in the IDE Preference panels and target settings panels. “Remote Debugging Setup” on page 57 provides information on how to set a project for remote debugging.

Using MetroTRK

This section describes MetroTRK and provides information related to using MetroTRK with the CodeWarrior™ IDE.

This section contains these topics:

- MetroTRK Overview
- Customizing MetroTRK
- Installing MetroTRK on the Remote Target

MetroTRK Overview

MetroTRK is a user-level application for use with the CodeWarrior debugger. You use MetroTRK to download and debug applications built with the CodeWarrior™ IDE. MetroTRK on the remote target system connects with the host computer via an ethernet link or serial port by using the CodeWarrior IDE remote debugging feature. Refer to "Remote Debugging Setup" on page 57 for an example of how the process works. Detailed information on the remote debugging features of the CodeWarrior™ IDE is available in the *CodeWarrior IDE User's Guide*.

On embedded Linux systems, MetroTRK is packaged as a regular Linux application. MetroTRK resides on the remote target system with the program you are debugging to provide debug services to the CodeWarrior™ debugger.

NOTE The CodeWarrior IDE installation installs the source files for the MetroTRK application. To know the path where the target-specific versions of the MetroTRK source files are located in the CodeWarrior installation directory, see "MetroTRK Project and Binary File Location" on page 176.

Customizing MetroTRK

You may customize the MetroTRK source code and recreate the MetroTRK binary for your specific needs. To know the path where the target platform-specific version of the MetroTRK project file is located in the CodeWarrior installation directory, see "MetroTRK Project and Binary File Location" on page 176.

You can either make a copy of the project (and its associated source files) or directly edit the original source.

To build the MetroTRK project successfully, make sure that these libraries are linked to the project:

- libthread_db.so
- libutil.so

The MetroTRK project has build targets for:

- building a debug version of MetroTRK
- building a release version of MetroTRK
- building all the versions of MetroTRK one after another

NOTE For information on MetroTRK target names, see “MetroTRK Project - Build Targets” on page 177.While we recommend that you build the MetroTRK binary as explained in this section, you can also use the pre-built MetroTRK binary available in your CodeWarrior installation directory. For location information, see “MetroTRK Project and Binary File Location” on page 176.

Installing MetroTRK on the Remote Target

To use MetroTRK for debugging, you must install and launch it on the remote target system.

After you have launched MetroTRK on the remote target, you can use the CodeWarrior debugger to upload your application to the remote target system and debug the application.

To install MetroTRK on the remote target system, you need to download the MetroTRK binary file to a suitable location on the root file system of the remote target system.

You can use any of the available network utilities, such as File Transfer Protocol (FTP), to transfer the MetroTRK binary from the host computer to the root file system of the remote target system.

The procedure for launching MetroTRK is covered in “Start MetroTRK on the Remote Target” on page 61.

Remote Debugging Setup

In order to debug a remote executable file, you must have a CodeWarrior project open in the CodeWarrior™ IDE on the local computer. The project you are using on the local computer must be the same project used to create the executable file that is running on the remote target system.

Perform these steps to debug remote executable files:

- Create a Remote Connection
- Specify Remote Debugging Options
- Start MetroTRK on the Remote Target
- Start the Debugger

Create a Remote Connection

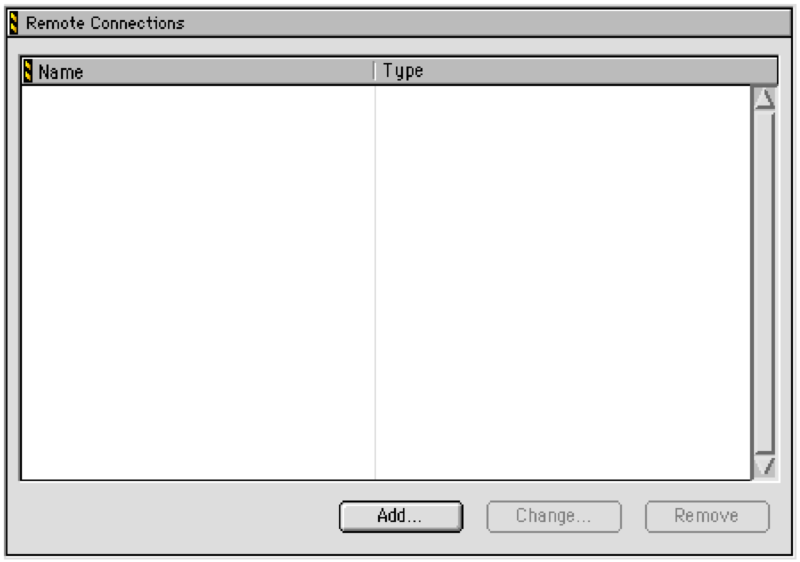
First, you need to define the characteristics of the remote connection so that the CodeWarrior IDE can connect to the remote machine. This example explains how to specify the settings for a remote TCP/IP connection.

NOTE For more detailed information on the **Remote Connections** preference panel, refer to the *CodeWarrior IDE User's Guide*.

The steps to define a remote connection are as follows:

1. Display the **Remote Connections** panel.
   a. Select **Edit > Preferences**. The **IDE Preferences** window appears.
   b. Select **Remote Connections** from the **IDE Preference Panels** list to display the **Remote Connections** panel (Figure 5.1).

Figure 5.1 Remote Connections Preference Panel

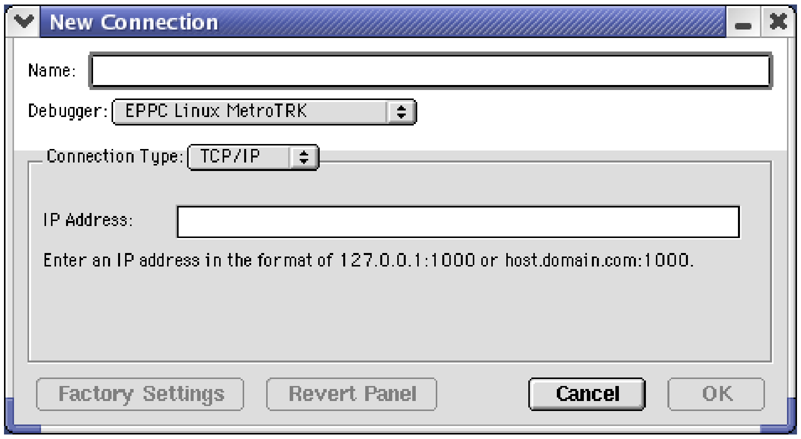


2. Add a new remote connection.
   a. Click **Add**. The **New Connection** dialog box appears. This dialog box is where you specify all information about the remote connection.

NOTE The **New Connection** dialog box displays the options for creating a serial connection by default. For example, if you want to use a serial connection for debugging, specify the connection name in the **Name** text box and select `COM2`, `115200`, `8`, `None`, `1`, and `None` from the **Port**, **Rate**, **Data Bits**, **Parity**, **Stop Bits**, and **Flow Control** list boxes.

b. Select **TCP/IP** from the **Connection Type** list box. The **New Connection** dialog box (Figure 5.2) display changes.

Figure 5.2 New TCP/IP Connection



Enter an IP address in the format of 127.0.0.1:1000 or host.domain.com:1000.

c. Type the remote connection name in the **Name** text box. The remote connection name is used to identify the remote connection in other CodeWarrior IDE windows and dialog boxes.

NOTE The **Debugger** list box displays the target platform-specific MetroTRK name. For example, **EPPC Linux MetroTRK** for PowerPC™-based target platform or **ARM Linux MetroTRK** for ARM®-based target platform.

d. In the **IP Address** text box, type the IP address of the remote target system and the TCP/IP port number used for connecting to MetroTRK. For example, if the IP address is `127.0.0.1` and the port number is `6969`, type `127.0.0.1:6969`.

NOTE Specifying a descriptive remote connection name is helpful. For example, you might use the name "*PPC Linux* - 127.0.0.1:6969", where *PPC Linux* is the name of the target platform you are using. By specifying a descriptive name it gives an indication of how the remote connection is to be used and the IP address and port used by the connection.

e. Save the new remote connection.

f. Click **OK**. The system saves the remote connection and closes the **New Connection** dialog box.

g. Click **Save**.

h. Close the **IDE Preferences** window.

Specify Remote Debugging Options

Once the remote connection is set up, you must specify remote debugging options for the build target.

1. Verify source code file debug settings.

   Ensure that the source code files you want to debug have a mark next to their names in the debug column of the project window.

2. Switch to the debug build target.

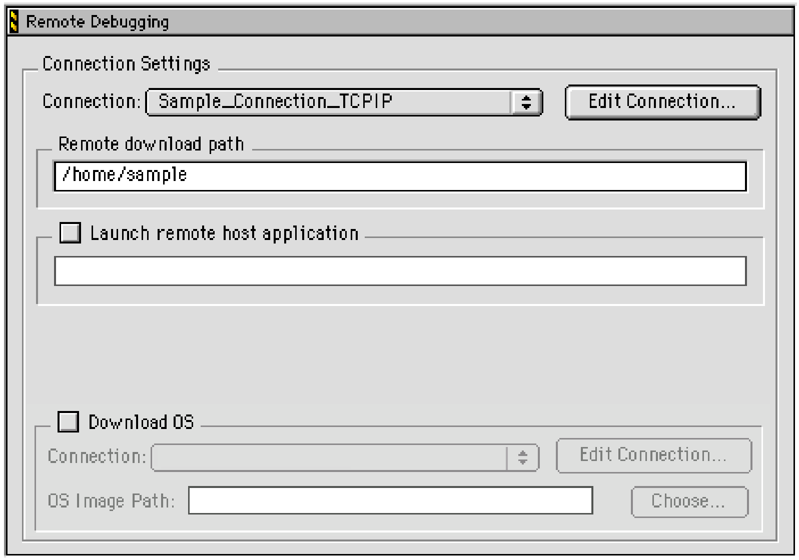
   If the project has a debug build target, switch to the debug build target. Select the target name from the build target list box in the project window.

3. Select a remote connection.

   a. Open the ***Target*** **Settings** window by choosing **Edit >** ***Target*** **Settings**, where ***Target*** is the name of the debug build target displayed in the project window.

   b. Select **Remote Debugging** from the list of settings panels. The **Remote Debugging** settings panel (Figure 5.3) appears.

Figure 5.3 Remote Debugging Settings Panel



Remote Debugging

Connection Settings

Connection: Sample_Connection_TCPIP

Remote download path: /home/sample

☐ Launch remote host application

☐ Download OS

c. Select the remote connection by using the **Connection** list box. The remote connection you select here is the same remote connection you specified in “Create a Remote Connection” on page 58.

NOTE Perform step 3(d) only if you have skipped step 4(a) in Section “Create a New Project” on page 21 in the Application Tutorial chapter.

d. In the **Remote download path** text box, specify the location where the executable binary is to reside on the remote target system. MetroTRK transfers the executable binary to this location immediately before starting the debugger.

NOTE The **Download OS** checkbox lets you specify the location of the compressed kernel image that should be downloaded to the target platform for a specific remote connection. For more information, see “Download and Boot the Kernel” on page 150.

4. Ensure that external debugging is disabled.

   Ensure that the **Use External Debugger** checkbox in the **Build Extras** settings panel is cleared.

Start MetroTRK on the Remote Target

MetroTRK must be running on the remote target system before the debugger can connect to the remote target system. The steps to launch MetroTRK on a remote target system depend on the type of remote connection you are using.

Starting MetroTRK by Using TCP/IP Connection

To launch MetroTRK through a TCP/IP connection:

1. Connect to the remote target system.

   a. Start the Terminal application.

   b. At the command prompt, type `telnet` *`IP address`*, where *`IP address`* is the IP address of the remote target system, and press Enter. Your computer connects to the remote target.

2. Navigate to the target-system directory that contains the MetroTRK binary.

   Type `cd /`*`Directory Name`*, where *`Directory Name`* is the name of the target-system directory where you downloaded the MetroTRK binary, and press Enter. The current directory changes.

3. Launch MetroTRK on the remote target system.

   Type `./MetroTRK binary name.elf :`*`port number`*, where `MetroTRK binary name` is the name of the target-specific MetroTRK binary and *`port number`* is the TCP/IP port number you specified while creating a remote connection. For example, type `./AppTRK.elf :6969`.

4. Press Enter. MetroTRK starts on the remote target system.

NOTE To reuse the console, you may start MetroTRK as a background process. For example, if you want to start MetroTRK as a background process on the TCP/IP port number 6969, the syntax is as follows: `./MetroTRK binary name.elf :6969&`.

Starting MetroTRK by Using Serial Connection

It is recommended that your computer have two serial ports if you want to debug applications through a serial connection. This is because one serial port (for example, COM1) of the host is connected to the first serial port (S0) of the target board while setting up the target board. This connection is used for startup and console log messages from the target board. You need to use another serial port (for example, COM2) of the host for connecting to the second serial port (S1) of the target. This connection will be used by the CodeWarrior™ debugger to communicate with MetroTRK.

To launch MetroTRK on the remote target by using a serial connection:

1. Connect a serial cable between the host computer serial port COM(*x*) and the second serial port (S1) of the board. Here, *x* is the port number.
2. Launch the Terminal application with these settings: 115200, 8, N, 1, N.
3. Navigate to the target-system directory where you downloaded the prebuilt MetroTRK binary.

   In the Terminal serial connection console, type `cd /`*`Directory Name`*, where *`Directory Name`* is the name of the target-system directory where the prebuilt MetroTRK binary exists, and press Enter. The current directory changes.

4. Configure the serial port (S1) on which MetroTRK is to be launched.

   The substeps that follow, enable you to configure the second serial port S1 of the board as per the serial port settings of the host that you specified while creating a new serial connection in the CodeWarrior IDE.

   a. Type `stty -F /dev/ttyS1 raw` command in Terminal window and press Enter. The serial connection is configured for raw mode of operation. If raw mode is not selected, the special characters sent as part of packet will be interpreted (dropped) causing the connection to break.

b. Type `stty -F /dev/ttyS1 ispeed 15200` command in Terminal window and press Enter. The serial input speed is set to 115200-bauds.

c. Type `stty -F /dev/ttyS1 ospeed 15200` command in Terminal window and press Enter. The serial output speed is set to 115200-bauds.

d. Type `stty -F /dev/ttyS1 crtscts` command in Terminal Window and press Enter. Handshake is enabled to make the serial connection more reliable.

e. Type `stty -a -F /dev/ttyS1` to verify the current device settings.

5. Launch MetroTRK on the remote target platform.

   Type `./MetroTRK binary name.elf/dev/ttyS1` command in the Terminal window and press Enter. MetroTRK launches on the remote target board.

Start the Debugger

Select **Project > Debug** to start the CodeWarrior™ debugger. When you start the debugger, the CodeWarrior IDE:

1. builds the target
2. connects to the remote MetroTRK process
3. transfers the executable file to the remote system
4. launches the executable file
5. starts the debugger

Shared Library Debugging

The CodeWarrior™ IDE allows source-level debugging of non-executable files, such as shared libraries. When you debug an executable file with which a shared library interacts, you can step into the shared library code.

The tutorial that follows demonstrates the shared library debugging feature for an implicitly linked shared library.

In this tutorial, you will do the following:

- Create and build an example shared library
- Create and build an example application that implicitly links the example shared library and debug the application

1. As a first step, create a project using the EPPC New Project Wizard and create two new build targets with the following settings (Table 5.1):

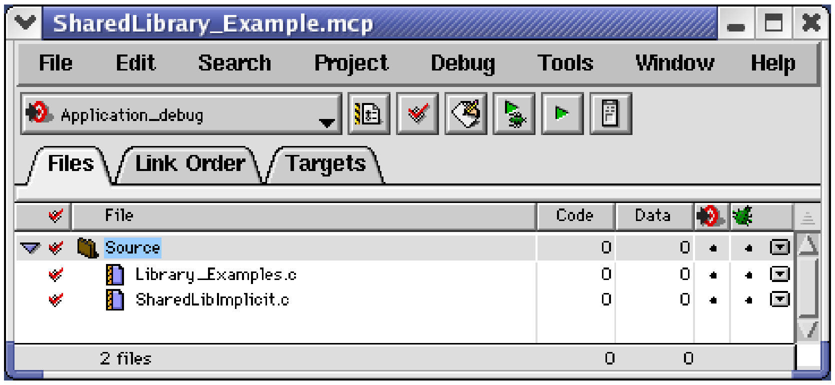
Table 5.1 Shared Library Project Settings

Project Name:	SharedLibrary_Example
Project Location:	/home/usr1/SharedSample
Languages:	C
Build Targets:	- Lib_Example_debug (generates a shared library) - Application_debug (generates an executable binary)
Lib_Example_debug Build Target - **- Output Type:** **- Output File:** **- Output File Location:**	 Shared Library LibExample.so (implements the add_example function) /home/usr1/SharedSample/Output
Application_debug Build Target - **- Output Type:** **- Output File:** **- Output File Location:**	 Application SharedLib_Application.elf (makes a call to the add_example function routine) /home/usr1/SharedSample/Output

NOTE For detailed information on how to create or remove build targets, refer the *IDE User's Guide*.

2. Remove the default `main.c` file and add the source files (`SharedLibImplicit.c` and `Library_Examples.c`) to your project. The project window appears as shown in Figure 5.4. For detailed information on how to add a new source file, see "Add a New Source File to the Project" on page 25.

Figure 5.4 Source Files Added to the SharedLibrary_Example.mcp Project


SharedLibrary_Example.mcp
File Edit Search Project Debug Tools Window Help
Application_debug
Files
Link Order
Targets

File	Code	Data
Source	0	0
Library_Examples.c	0	0
SharedLibImplicit.c	0	0
2 files	0	0

3. Create two header files; `LibExample.h` and `CWExample.h` in your project directory.
4. Enter the source code of Listing 5.1 into the editor window of `LibExample.h` file.

Listing 5.1 Source Code for LibExample.h

```
/* LibExample.h */
int add_example(int x,int y);
int add_example_local(int x,int y)
```

5. Enter the source code of Listing 5.2 into the editor window of `CWExample.h` file.

Listing 5.2 Source Code for CWExample.h

```
/* CWExample.h */
 #define INFINITE_LOOP     while(1);
```

6. Enter the source code of Listing 5.3 into the editor window of `SharedLibImplicit.c` file.

Listing 5.3 Source Code for SharedLibImplicit.c

```
/* SharedlibImplicit.c */
/* Demonstrates implicit linking.*/
/*-----------------------------
User Include files
-----------------------------*/

#include “LibExample.h”
#include “CWExample.h”
```

```
/*---------------------------------
Function Prototype Declaration
---------------------------------*/

int temp(int, int);

/*---------------------------------
Main Program
---------------------------------*/
int main()
{
     int ret;
     int a,b;
     a= 10;
     b= 20;
     ret = temp(a,b);
     ret = add_example(a,b);//Step In here
     return ret;
}

int temp(int i,int j)
{
     return i+j;
}
```

7. Enter the source code of Listing 5.4 into the editor window of `Library_Examples.c` file.

Listing 5.4 Source Code for Library_Examples.c

```
/* LibExample.c */

/*---------------------------
User Include files
---------------------------*/

#include "LibExample.h"

/*---------------------------
Functions Definitions
---------------------------*/
  int add_example(int x,int y)
{
    int p,q;
    p=100;
    q=p+200;
    add_example_local(2,3);//Step In here
    return x+y+q;
}
  int add_example_local(int x,int y)
{
    int p,q;
    p=100;
    q=p+200;
    return x+y+q;
}
```

8. Add the pathname of the header files (`CWExamples.h` and `LibExample.h`) to both the build targets.
   a. Select the **Lib_Example_debug** build target from the build target list box in the project window.
   b. Click ***Target* Settings** button in the project window. The ***Target* Settings** window appears.
   c. Click **Access Paths** in the **Target Settings Panels** list. The **Access Paths** settings panel appears, which displays the current search paths for locating and accessing the build target's system and header files.
   d. Click in the **User Paths** list to select it.
   e. Click **Add**. A file navigation dialog box appears.
   f. Search for the location where the header files (`CWExample.h` and `LibExample.h`) are stored in the project folder.
   g. Select both the header files.
   h. Click **"Select <*project folder*>"** in the **file navigation** dialog box. The header files path location gets added to the **User Paths** list.
   i. Repeat steps b to g for the **Application_debug** build target also.

NOTE Make sure that your project is using the correct cross compiler tools. To verify or change the cross compiler tools path, click the **System Paths** option button in the **Access Paths** settings panel.

Now, let us generate the shared library application and debug it. The following sections describe how to debug a shared library:

- Build the Project
- Configure the Executable Build Target
- Configure the Library Build Target
- Debug the Shared Library

Build the Project

You first need to build the project to generate the shared library file and the executable binary.

1. Build the `SharedLibrary_Example.mcp` project
   a. Select the **Lib_Example_debug** build target from the build target list box in the project window.
   b. Select **Project > Make**. The CodeWarrior IDE builds the project and stores the output file `LibExample.so` in the *Output* directory within the project directory.
   c. Now, select the **Application_debug** build target from the build target list box in the project window.
   d. Select **Project > Make**. The CodeWarrior IDE builds the project and stores the final output file `SharedLib_Application.elf` in the *Output* directory within the project directory.

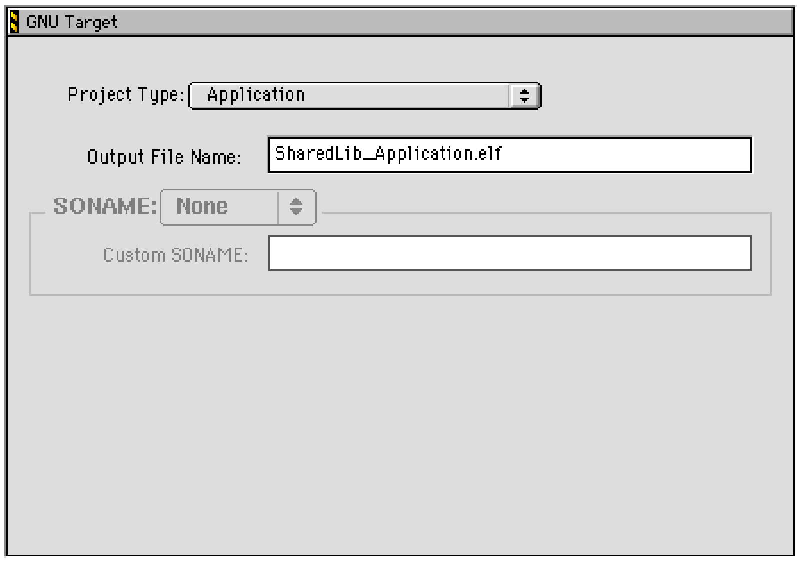
Configure the Executable Build Target

You need to set up the **Application_debug** build target by:

- verifying the final output file name
- adding `LibExample.so` to the **Application_debug** build target
- specifying the linker settings
- specifying the remote download path of the final executable file
- specifying the host-side location and the remote download path of the shared library
- specifying the environment variable that enables the shared object loader to locate the shared library on the remote target at runtime

1. Make the **Application_debug** build target in the project window active, if it not already active.
2. Verify the final output file name.
   a. Select **Edit >** ***Target*** **Settings**, where ***Target*** is the name of the build target. The ***Target*** **Settings** window appears.
   b. Click **GNU Target** in the **Target Settings Panels** list. The **GNU Target** settings panel (Figure 5.5) appears.

Figure 5.5 GNU Target Settings Panel



Project Type: Application

Output File Name: SharedLib_Application.elf

SONAME: None

Custom SONAME:

c. Make sure that the **Output File Name** text box displays the name of the final executable binary as `SharedLib_Application.elf`.

3. Add `LibExample.so` file to the **Application_debug** build target.

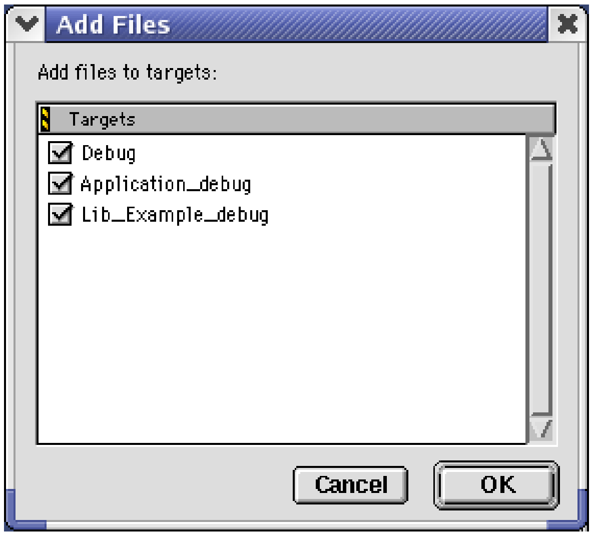
NOTE Before you add the `LibExample.so` file to the build target, make sure that the **File Mapping** settings panel contains an entry for `.so` file type. If it is not there, you need to add it to your shared library project. To add the `LibExample.so` file your project, click **Choose** in the **File Mapping** panel and navigate to the location where the `LibExample.so` file is located in your computer. Select the file. The file gets added to the list. Select **EPPC GNU Obj Importer** from the **Compiler** list box. Click **Save** to save the settings.

a. Right-click on the project window and select **Add Files** from the contextual menu.

b. Navigate to the directory where you have stored the `LibExample.so` file in your project folder. For this tutorial it is: `/home/usr1/SharedSample/Output`.

c. Select the `LibExample.so` file and click **Open**. The **Add Files** dialog box (Figure 5.6) appears.

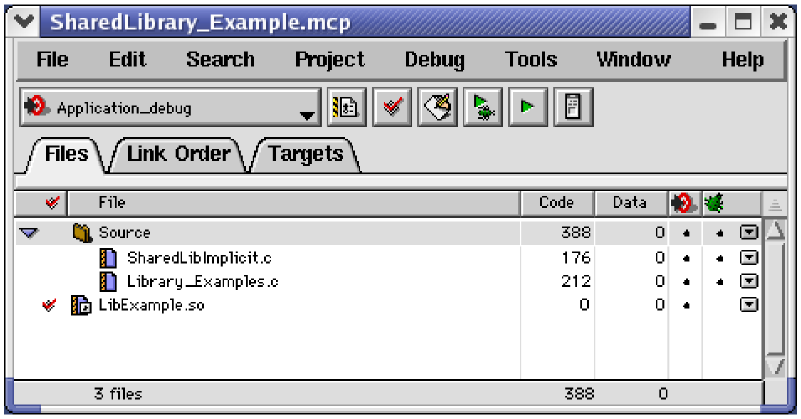
Figure 5.6 Add Files Dialog Box



d. Clear the checkmark adjacent to the **Lib_Example_debug** build target. This will ensure that the `LibExample.so` file is not added to the **Lib_Example_debug** build target.

e. Click **OK**. The `LibExample.so` file gets added to the **Application_debug** build target (Figure 5.7).

Figure 5.7 LibExample.so Added to the Application_debug Build Target

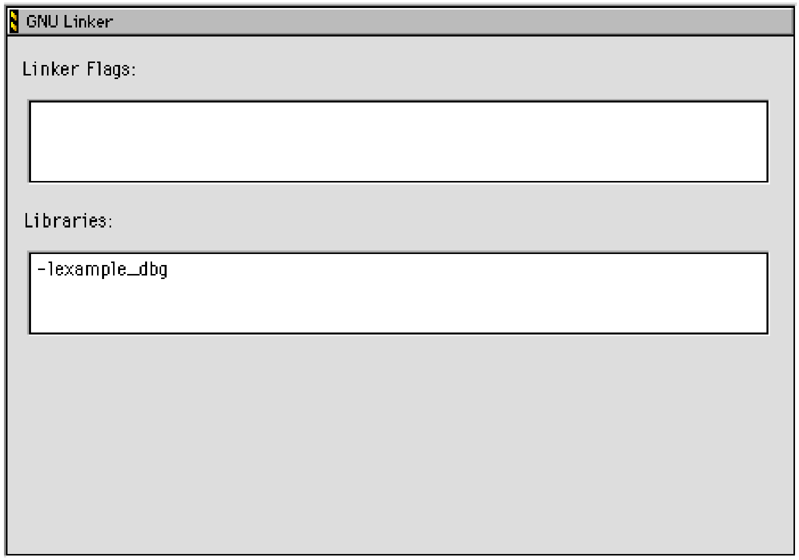


File	Code	Data
Source	388	0
SharedLibImplicit.c	176	0
Library_Examples.c	212	0
LibExample.so	0	0
3 files	388	0

4. Specify the linker settings.

a. Click **GNU Linker** in the **Target Settings Panels** list. The **GNU Linker** settings panel (Figure 5.8) appears.

Figure 5.8 GNU Linker Settings Panel


GNU Linker

Linker Flags:

Libraries:

```
-lexample_dbg
```

b. Type these command line arguments in the **Libraries** text box:

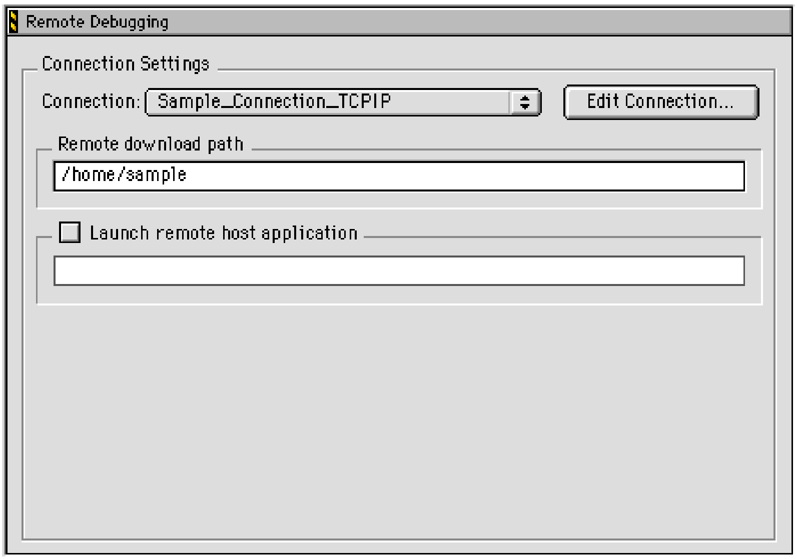
```
-lexample_dbg
```

NOTE The `-lexample_dbg` linker command line argument enables the CodeWarrior™ IDE linker to locate the shared library `LibExample.so`. For detailed information on other linker command line arguments, refer GNU linker manuals. The manuals can be found at `www.gnu.org`.

5. Specify the remote download path of the final executable file.

a. Click **Remote Debugging** in the **Target Settings Panels** list. The **Remote Debugging** settings panel (Figure 5.9) appears.

Figure 5.9 Remote Debugging Settings Panel



Remote Debugging
Connection Settings
Connection: Sample_Connection_TCPIP
Remote download path
/home/sample
Launch remote host application

b. Make sure that the correct remote connection name is selected in the **Connection** list box of the **Remote Debugging** settings panel.

c. Type `/home/sample` in the **Remote Download Path** text box. This specifies that the final executable file will be downloaded to this location on the target platform for debugging.

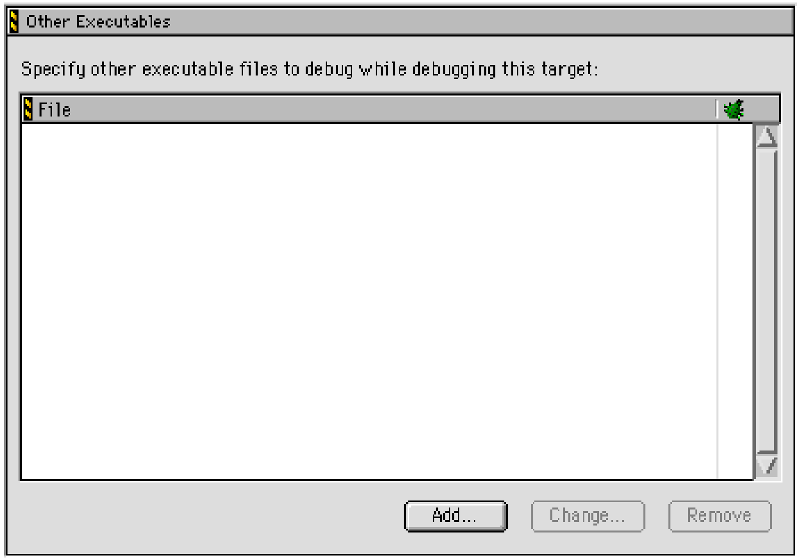
NOTE For this tutorial, the remote download path is specified as `/home/sample`. If you wish, you may specify an alternate remote download path for the executable file.

6. Specify the host-side location and the remote download path of the shared library.

   a. Click **Other Executables** in the **Target Settings Panels** list. The **Other Executables** settings panel (Figure 5.10) appears.

NOTE The **Other Executables** settings panel is displayed in the **Target Settings Panels** list only when you select a MetroTRK-based remote connection from the **Connection** list box in the **Remote Connection** settings panel.

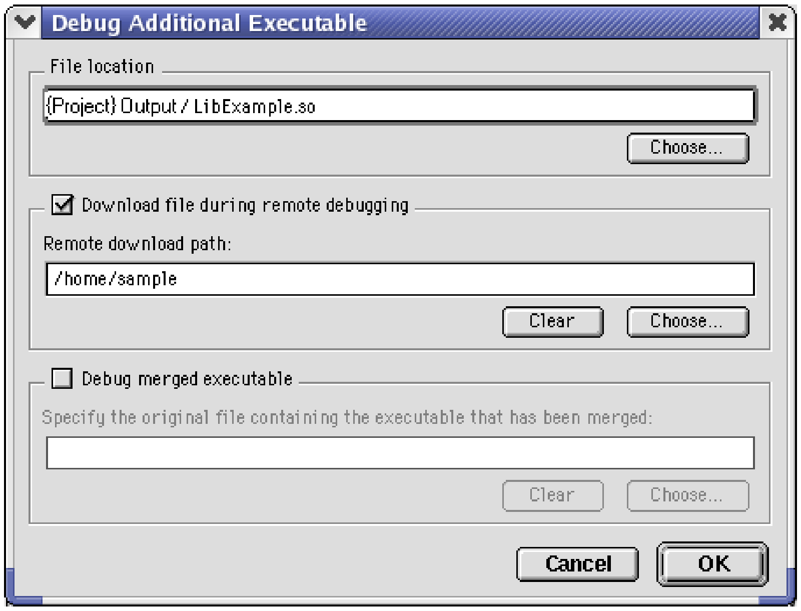
Figure 5.10 Other Executables Settings Panel


Other Executables

Specify other executable files to debug while debugging this target:

b. Click **Add**. The **Debug Additional Executable** dialog box (Figure 5.11) appears.

Figure 5.11 Debug Additional Executable Dialog Box


Debug Additional Executable
File location
{Project} Output / LibExample.so
Choose...
Download file during remote debugging
Remote download path:
/home/sample
Clear
Choose...
Debug merged executable
Specify the original file containing the executable that has been merged:
Clear
Choose...
Cancel
OK

c. Click **Choose** in the **File Location** area. The **Choose an Executable to Debug** dialog box appears.

d. Navigate to the location where you have stored the `LibExample.so` file in your project directory. For this tutorial it is: `/home/usr1/SharedSample/Output`.

e. Select the `LibExample.so` filename.

f. In **Relative To** list box, select **Project**.

g. Click **Open**. The host-side location of the shared library appears in the **File location** text box.

h. Check the **Download file during remote debugging** checkbox.

NOTE If you do not want to download the selected file on the target platform, do not check the **Download file during remote debugging** checkbox.

i. Type `/home/sample` in the **Remote download path** text box. The shared library will be downloaded at this location when you debug or run the executable file.

The default location of shared libraries on the embedded Linux operating system is `/usr/lib`. For this tutorial, the remote download location of `LibExample.so` is `/home/sample`.

j. Click **OK**. The settings are saved.

NOTE For detailed description of the **Other Executables** panel options, see the *IDE User's Guide*.

7. Specify the environment variable that enables the shared object loader to locate the shared library on the remote target at runtime.

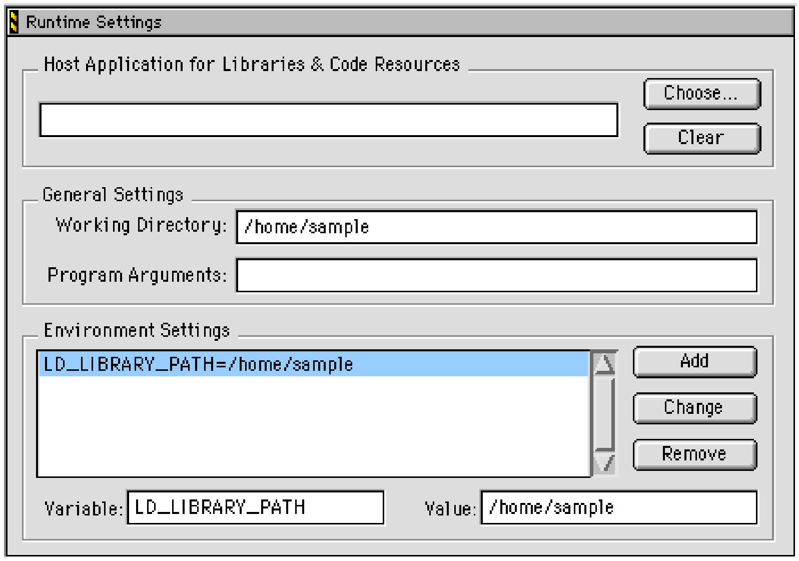
   At runtime, the shared object loader first searches for a shared library in the path specified by the `LD_LIBRARY_PATH` environment variable's value. In this case, the value of this environment variable will be `/home/sample`, which is the remote download path for the shared library you specified in the **Debug Additional Executable** dialog box. If you have not specified the environment variable or have assigned an incorrect value, the shared object loader searches for the shared library in the default location `/usr/lib`.

   a. Click **Runtime Settings** in the **Target Settings Panels** list. The **Runtime Settings** panel appears.

   b. In the **Environment Settings** area, type `LD_LIBRARY_PATH` in the **Variable** text box (Figure 5.12).

   c. Type `/home/sample` in the **Value** text box.

Figure 5.12 Runtime Settings Panel



Working Directory: /home/sample
Program Arguments:
Environment Settings
LD_LIBRARY_PATH=/home/sample
Variable: LD_LIBRARY_PATH
Value: /home/sample

NOTE Make sure you type the same remote download path in the **Value** text box that you specified in the **Debug Additional Executable** dialog box.

d. Click **Add**. The environment variable is added to the build target.

e. Click **Save**. The target settings are saved.

f. Close the **Runtime Settings** panel.

8. Build the project.

Select **Project > Make**. The final executable is built with new target settings.

Configure the Library Build Target

You need to configure the **Lib_Example_debug** build target by:

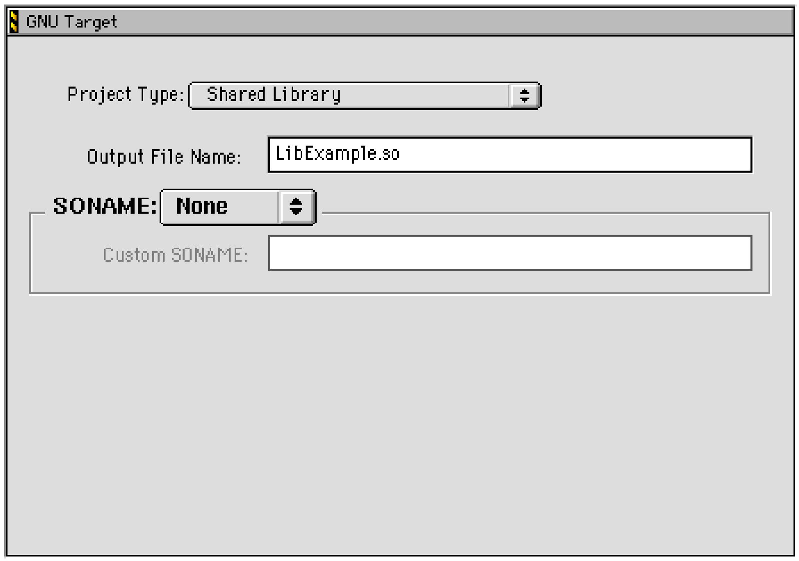
- verifying the final output file name
- specifying the host-side location of the executable file to be used for debugging the shared library
- specifying remote debugging options

1. Make the **Lib_Example_debug** build target in the project window active.
2. Verify the final output file name.

a. Select **Edit >** ***Target*** **Settings**, where ***Target*** is the name of the build target. The ***Target*** **Settings** window appears.

b. Click **GNU Target** in the **Target Settings Panels** list. The **GNU Target** settings panel (Figure 5.13) appears.

Figure 5.13 GNU Target Settings Panel



Project Type: Shared Library

Output File Name: LibExample.so

SONAME: None

Custom SONAME:

c. Make sure that the **Output File Name** text box displays the name of the final executable as `LibExample.so`.

3. Specify the host-side location of the executable file to be used for debugging the shared library.

   a. Click **Runtime Settings** in the **Target Settings Panels** list. The **Runtime Settings** panel appears.

   b. Click **Choose** in the **Host Application for Libraries & Code Resources** section. The **Choose the Host Application** dialog box appears.

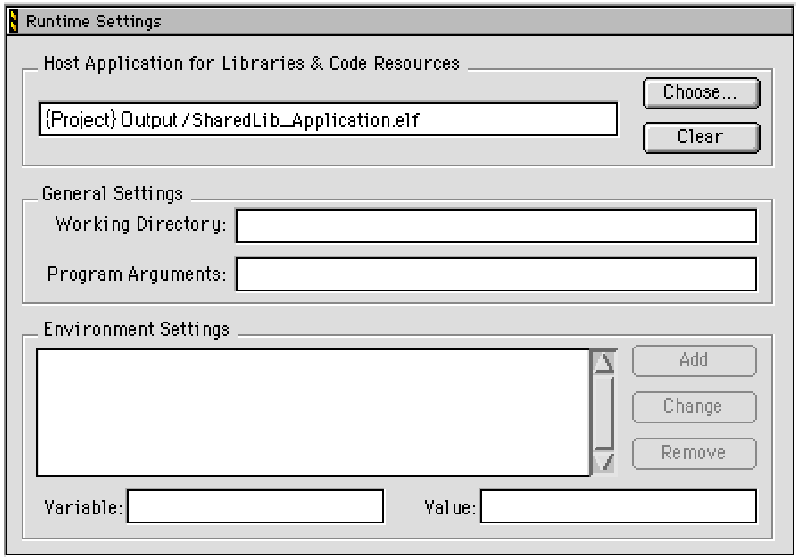
   c. Navigate to the location where you have stored the `SharedLib_Application.elf` file in your project directory. For this tutorial it is: `/home/usr1/SharedSample/Output`.

   d. Select the `SharedLib_Application.elf` filename.

NOTE If the contents of the *Output* folder are not visible in the **Choose the Host Application** dialog box, select **All Files** from the **Files of Type** list box.

   e. Click **Open**. The location of the final executable file appears in the **Host Application for Libraries & Code Resources** text box (Figure 5.14).

Figure 5.14 SharedLib_Application.elf Selected



Host Application for Libraries & Code Resources: {Project} Output/SharedLib_Application.elf
Working Directory:
Program Arguments:
Variable:
Value:

f. In the **Environment Settings** area, type `LD_LIBRARY_PATH` in the **Variable** text box.

g. Type `/home/sample` in the **Value** text box.

h. Click **Add**. The environment variable is added to the build target.

4. Specify remote debugging options.

a. Click **Remote Debugging** in the **Target Settings Panels** list. The **Remote Debugging** settings panel appears.

b. Make sure that the correct remote connection name is selected in the **Connection** list box of the **Remote Debugging** settings panel.

c. Type `/home/sample` in the **Remote download path** text box. This is the location where the shared library will be downloaded on the target for debugging.

d. Check the **Launch remote host application** checkbox.

e. Type `/home/sample/SharedLib_Application.elf` in the text box below the **Launch remote host application** checkbox.

f. Click **Save** to save the target settings.

g. Close the **Remote Debugging** settings panel.

5. Build the project.

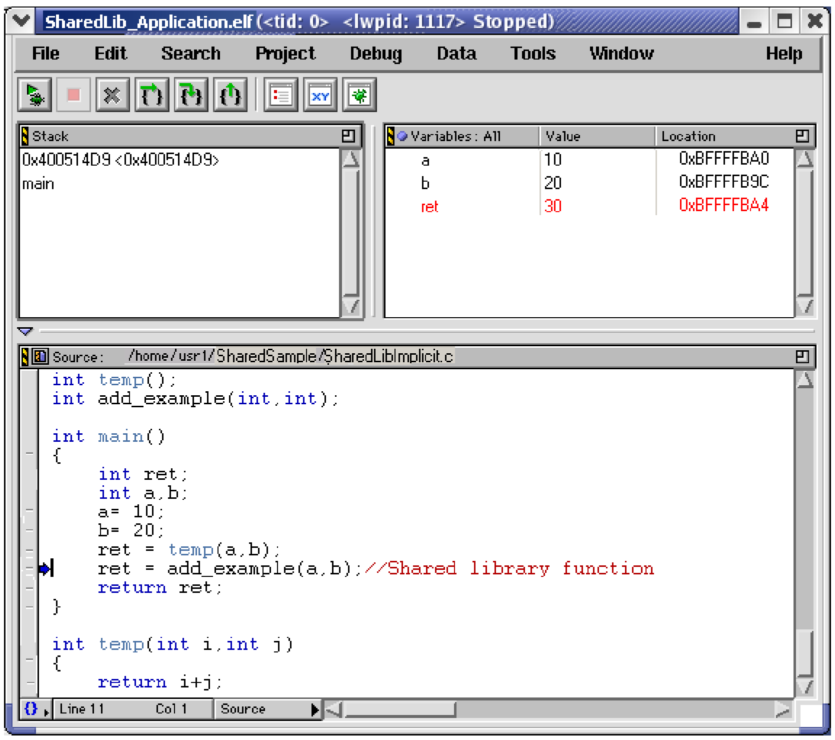
Select **Project > Make**. The library is built with the new settings.

Debug the Shared Library

In the steps that follow, you will launch the debugger. Next, you will step through the code of the executable file `SharedLib_Application.elf` until you reach the code that makes a call to the `add_example` function implemented in the shared library. At this point, you will step into the code of the `add_example` function to debug it.

1. Make the **Application_debug** build target in the project window active.
2. Select **Project > Debug**. The debugger starts and downloads the `SharedLib_Application.elf` and `LibExample.so` files to the specified location on the remote target, one after another. The debugger (Figure 5.15) and symbolics (Figure 5.16) windows appear.

Figure 5.15 Debugger Window


SharedLib_Application.elf (<tid: 0> <lwpid: 1117> Stopped)
File
Edit
Search
Project
Debug
Data
Tools
Window
Help
Stack
Variables : All
Value
Location
Source: /home/usr1/SharedSample/SharedLibImplicit.c
Line 11
Col 1
Source

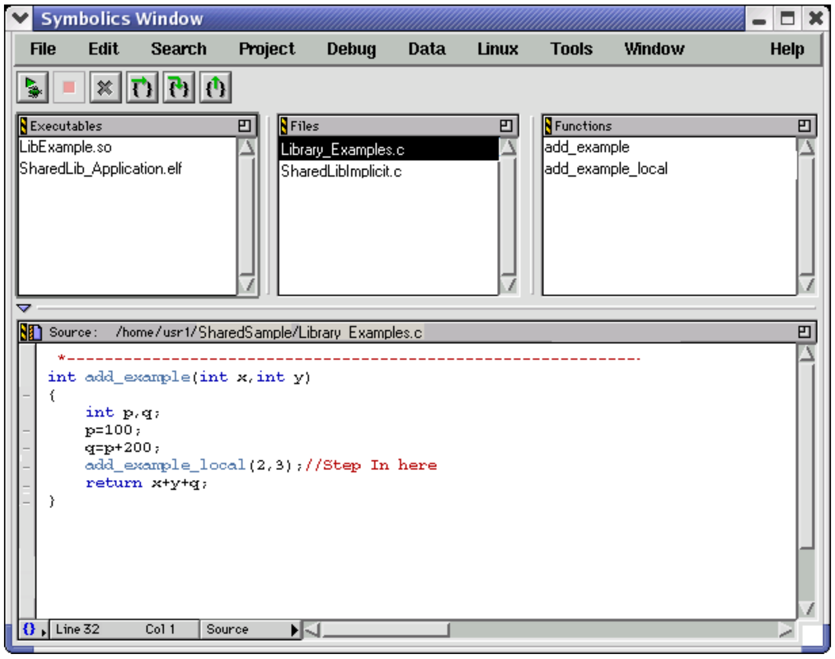
```
0x400514D9 <0x400514D9>
main
```

Variables : All	Value	Location
a	10	0xBFFFFBA0
b	20	0xBFFFFB9C
ret	30	0xBFFFFBA4

```
int temp();
int add_example(int,int);

int main()
{
    int ret;
    int a,b;
    a= 10;
    b= 20;
    ret = temp(a,b);
    ret = add_example(a,b);//Shared library function
    return ret;
}

int temp(int i,int j)
{
    return i+j;
```

NOTE The Thread ID (TID) and Process ID (PID) format may vary across different target platforms supported by the CodeWarrior™ Development Studio for Embedded Linux.

Figure 5.16 Symbolics Window



```
 *-------------------------------------------------------------
int add_example(int x,int y)
{
     int p,q;
     p=100;
     q=p+200;
     add_example_local(2,3);//Step In here
     return x+y+q;
}
```

NOTE For detailed information on symbolics window, see the *IDE User's Guide*.

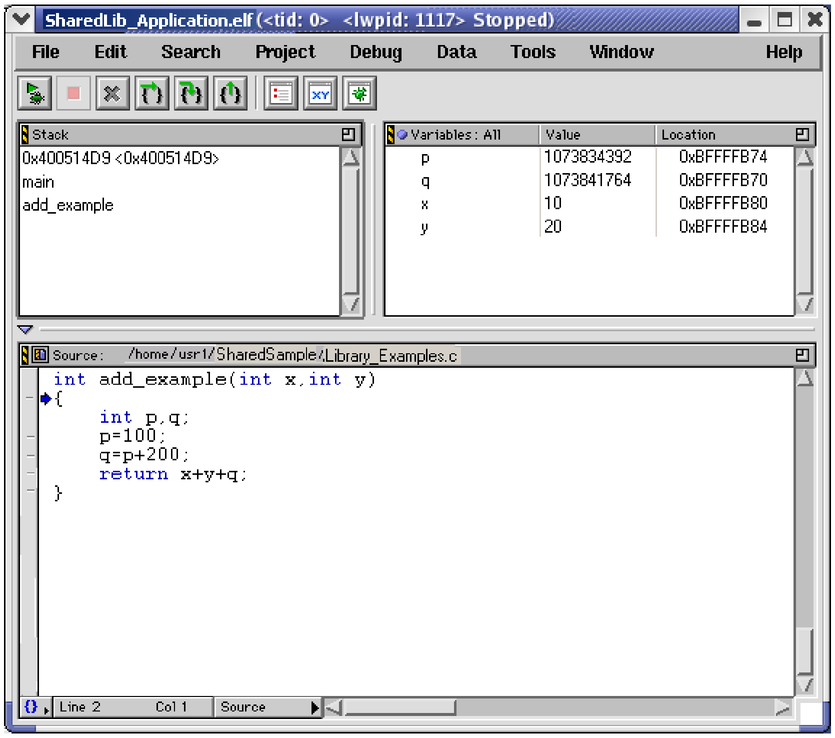
3. Step Over the code.

   Click the **Step Over** button in the debugger window until you reach this line of code: `ret=add_example(a,b);`.

4. Step into the code of the `add_example` function.

   In the debugger window, click the **Step Into** button a couple of times to step into the code of the `add_example` function. The debugger steps into the source code of the `add_example` function in the `Library_Examples.c` file (Figure 5.17).

Figure 5.17 Source Code of Library_Examples.c File



```
0x400514D9 <0x400514D9>
main
add_example
```

Variables : All	Value	Location
p	1073834392	0xBFFFFB74
q	1073841764	0xBFFFFB70
x	10	0xBFFFFB80
y	20	0xBFFFFB84

```
int add_example(int x,int y)
{
      int p,q;
      p=100;
      q=p+200;
      return x+y+q;
}
```

5. Step through rest of the code.

   After stepping in, you can step through the rest of the code.

6. Run the rest of the application.

   Click the **Run** button. The rest of the code is executed and the output appears in the MetroTRK Console window.

You may also use the sample shared library project available in the CodeWarrior installation directory. For location information, see “Sample Projects Location” on page 177.

Multi-threaded Debugging

In multi-threaded debugging, the breakpoints you set in the parent code are valid for all the threads generated by the parent code. Execution of all the generated threads stops at the breakpoint set in the parent code.

You can also set a thread-specific breakpoint (thread point), which is only valid for a particular thread ID. The procedure for setting a thread point is similar to that of setting any other eventpoint. Refer the *IDE User's Guide* for details.

While debugging programs that have multiple threads, the CodeWarrior™ debugger enables you to view separate debug windows for each thread being debugged. Each thread debug window displays its own stack crawl, source, and variable views.

NOTE The CodeWarrior™ debugger also allows you to show all the threads being debugged in a single thread window. For details, see "Viewing Multiple Processes and Threads" on page 110.

The tutorial that follows, demonstrates multi-threaded debugging.

1. Create a new project with the following settings (Table 5.2):

Table 5.2 Multithread Project Settings

Project Name:	multithread
Project Location:	/home/usr1/multithread
Languages:	C
Output Type:	Application
Output File Name:	Multithread_Example.elf
Location of the Output File:	/home/usr1/multithread/Output

   The above step creates two build targets: **c_app_debug** and **c_app_release**. Since this tutorial relates to debugging, only the first target is relevant.

2. Enter the source code of Listing 5.5 into the editor window of `main.c` file.

Listing 5.5 Source Code for main.c File

```
/* main.c */
/*---------------------------
System Include files
-----------------------*/
#include <pthread.h>
#include <stdio.h>
/*------------------------
User Include files
------------------------*/
#include "CWExample.h"
/*----------------------
```

```
Constants and Globals
-------------------------*/
#define MAX_NUM_OF_THREADS         3
int sum; /* this data is shared by the thread(s) */
/*-------------------------
Function Prototypes
-------------------------*/
void *thread(void); // Thread routine
/*------------------------
Main Program
------------------------*/
int main(int argc, char *argv[])
{
   pthread_t tid[MAX_NUM_OF_THREADS]; /* the thread identifier */
   pthread_attr_t attr[MAX_NUM_OF_THREADS];/*set of thread attributes*/
    int i;
    if (argc != 2)
    {
       fprintf(stderr, “Please enter the number of threads you want
       to create!!\n");
     exit();
    }
    if ((atoi(argv[1]) < 0) || (atoi(argv[1]) > MAX_NUM_OF_THREADS))
    {
       fprintf(stderr,"The number of threads(%d) must be > 0 OR < %d
       \n    atoi(argv[1]),MAX_NUM_OF_THREADS);
      exit();
    }
       printf("Number of threads to be created are :%d",atoi(argv[1]));
      fflush(stdout);
    /* get the default attributes */
      for (i=0;i<atoi(argv[1]);i++)
      pthread_attr_init(&attr[i]);
    /* create threads */
      for (i=0;i<atoi(argv[1]);i++)
      pthread_create(&tid[i], &attr[i],(void*)thread,NULL);
    /* now wait for the thread to exit */
      INFINITE_LOOP
      pthread_join(tid[i-1],NULL);
      printf("sum = %d\n", sum);
      fflush(stdout);
      return 0;
}
   /* The thread will begin control in this function */
      void *thread(void)
   {
      int i,j;
      sum=0;
```

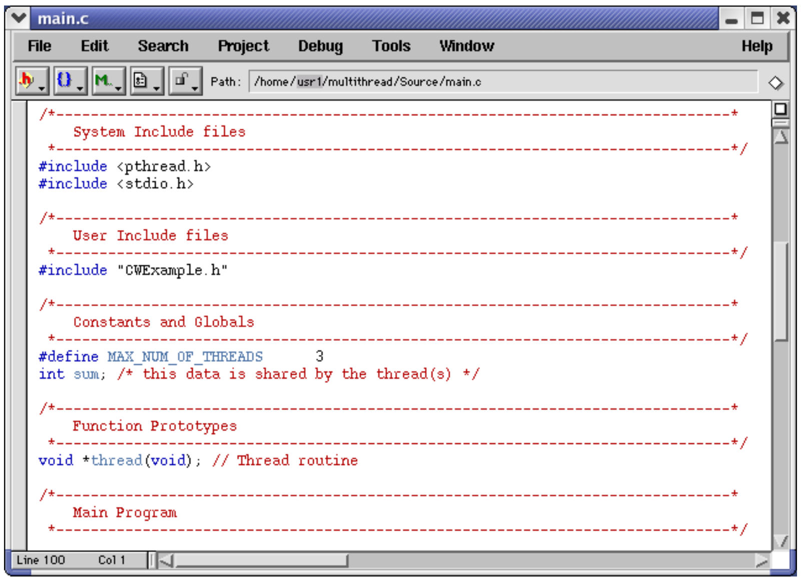
```
        i++; // Set Thread BreakPoint Here
        j++; // Set Thread BreakPoint Here
        sum  = i+j;
        INFINITE_LOOP
        pthread_exit(0);
    }
```

NOTE Make sure that you include the `CWExamples.h` file in your project. You can do this using the **Access Paths** settings panel.

3. Set a breakpoint in the thread code.

   a. Double-click the `main.c` filename in the project window. The source code of the `main.c` file is displayed in the editor window (Figure 5.18).

Figure 5.18 Editor Window



```
/*----------------------------------------------------------------------------*
    System Include files
 *----------------------------------------------------------------------------*/
#include <pthread.h>
#include <stdio.h>

/*----------------------------------------------------------------------------*
    User Include files
 *----------------------------------------------------------------------------*/
#include "CWExample.h"

/*----------------------------------------------------------------------------*
    Constants and Globals
 *----------------------------------------------------------------------------*/
#define MAX_NUM_OF_THREADS      3
int sum; /* this data is shared by the thread(s) */

/*----------------------------------------------------------------------------*
    Function Prototypes
 *----------------------------------------------------------------------------*/
void *thread(void); // Thread routine

/*----------------------------------------------------------------------------*
    Main Program
 *----------------------------------------------------------------------------*/
```

   b. Set a breakpoint at the following line in the editor window:

```
    i++; // Set Thread BreakPoint Here
```

NOTE Setting breakpoints may affect the performance of the debugger. Care should be taken while setting them.

   c. Close the editor window.

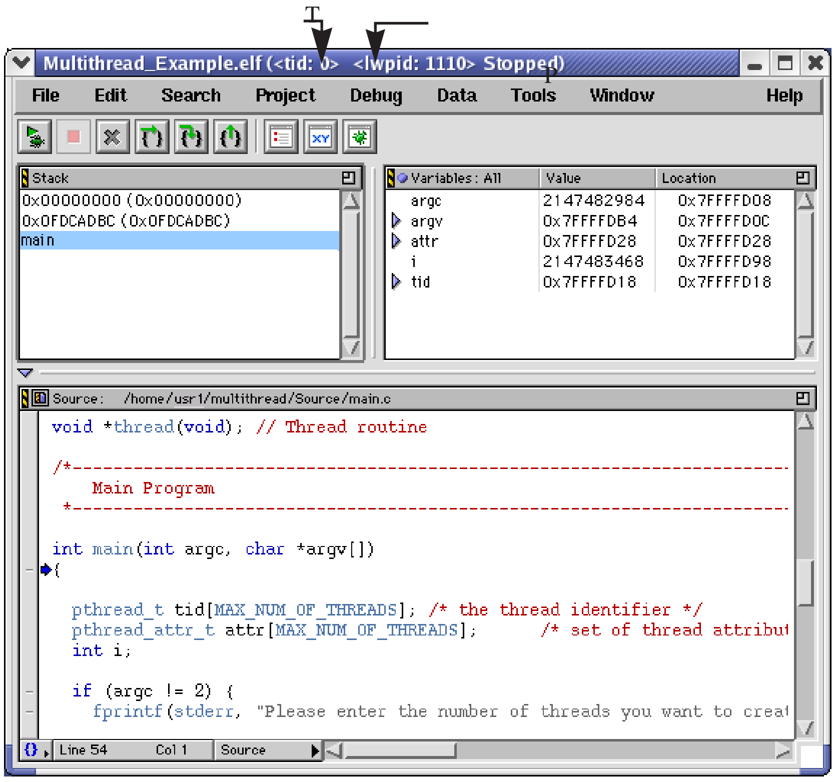
4. Specify program arguments.
   a. Open the **Runtime Settings** panel.
   b. Type 2 as value in the **Program Arguments** text box under the **General Settings** group.
   c. Click **Save** to save the settings.
   d. Close the **Runtime Settings** panel.
5. Specify the linker settings.
   a. Open the **GNU Linker** settings panel.
   b. Type `-lpthread` in the **Libraries** text box.
   c. Click **Save** to save the settings.
   d. Close the **GNU Linker** settings panel.
6. Build the project.

   Select **Project > Make**. The final output file `Multithread_Example.elf` is generated and is placed in the project folder.
7. Start the debugger.

   Select **Project > Debug**. The debugger window (Figure 5.19) appears.

NOTE To be able to successfully debug multithreaded applications, the following library file `libpthread.so.0` must exist unstripped on the target platform. If the above library is a symbolic link then the file it points to must be unstripped.

Figure 5.19 Debugger Window


T
P
Multithread_Example.elf (<tid: 0> <lwpid: 1110> Stopped)
File Edit Search Project Debug Data Tools Window Help
Stack
Variables: All
Value
Location
Source: /home/usr1/multithread/Source/main.c
Line 54
Col 1
Source

```
0x00000000 (0x00000000)
0x0FDCADBC (0x0FDCADBC)
main
```

Variables: All	Value	Location
argc	2147482984	0x7FFFFD08
argv	0x7FFFFDB4	0x7FFFFD0C
attr	0x7FFFFD28	0x7FFFFD28
i	2147483468	0x7FFFFD98
tid	0x7FFFFD18	0x7FFFFD18

```
void *thread(void); // Thread routine

/*-----------------------------------------------------------------------
    Main Program
 *-----------------------------------------------------------------------

int main(int argc, char *argv[])
{

  pthread_t tid[MAX_NUM_OF_THREADS]; /* the thread identifier */
  pthread_attr_t attr[MAX_NUM_OF_THREADS];      /* set of thread attribu
  int i;

  if (argc != 2) {
    fprintf(stderr, "Please enter the number of threads you want to crea
```

The thread window displays the Process ID (PID) and Thread ID (TID) for the currently running process. In this case, the PID is 1110 and the TID is 0.

NOTE The Thread ID (TID) and Process ID (PID) format may vary across different target platforms supported by the CodeWarrior™ Development Studio for Embedded Linux.

In the following steps, you will create multiple threads for the same process.

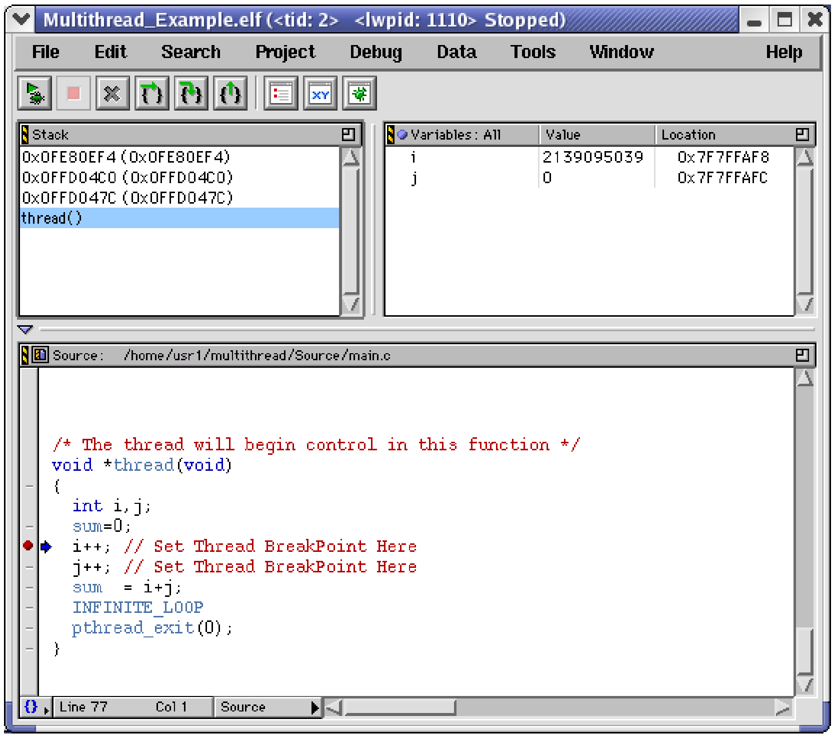
NOTE The Thread ID (TID) on the thread window is the ID assigned by the debugger/ MetroTRK to a particular thread. The debugger uses this ID to identify a thread.

8. Create the first thread.

   Step through the code by clicking the **Step Over** button. When the following code is executed, the first thread is created, thread execution stops at the breakpoint, and the first thread window (Figure 5.20) appears. This thread window has the same PID, but a new TID (2):

```
for (i=0;i<atoi(argv[1]);i++)
    pthread_create(&tid[i], &attr[i],(void*)thread,NULL);
```

Figure 5.20 First Thread Window


Multithread_Example.elf (<tid: 2> <lwpid: 1110> Stopped)
File
Edit
Search
Project
Debug
Data
Tools
Window
Help
Stack
Variables: All
Value
Location
Source: /home/usr1/multithread/Source/main.c
Line 77
Col 1
Source

```
0x0FE80EF4(0x0FE80EF4)
0x0FFD04C0(0x0FFD04C0)
0x0FFD047C(0x0FFD047C)
thread()
```

```
i    2139095039    0x7F7FFAF8
j    0             0x7F7FFAFC
```

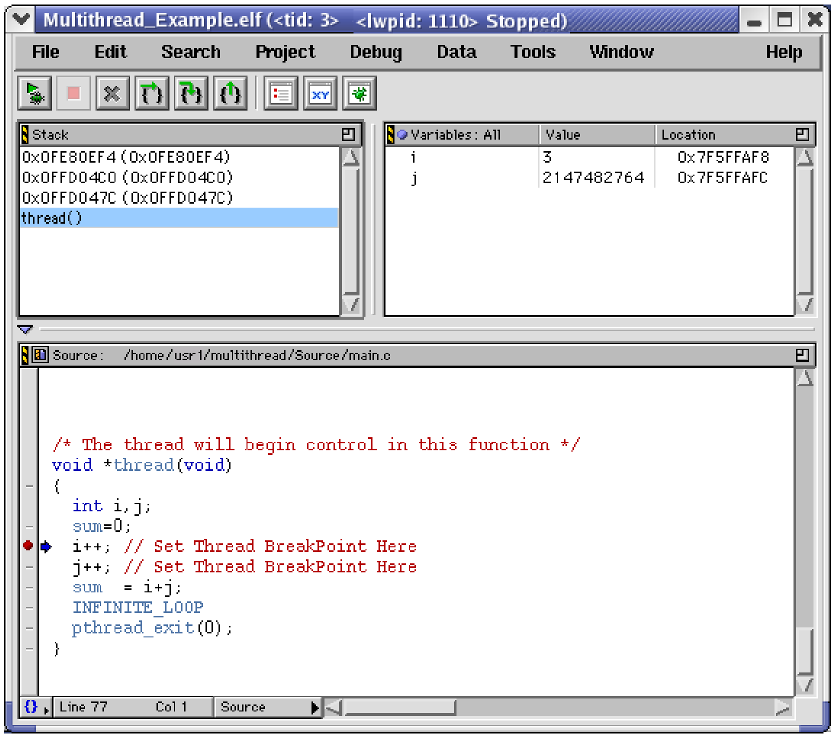
```
/* The thread will begin control in this function */
void *thread(void)
{
  int i,j;
  sum=0;
  i++; // Set Thread BreakPoint Here
  j++; // Set Thread BreakPoint Here
  sum  = i+j;
  INFINITE_LOOP
  pthread_exit(0);
}
```

   Once the thread window appears, you can step through the thread code.

9. Create the second thread.

   Step through the code in the parent debugger window once. When the *for loop* code is executed again, the second thread is created, thread execution stops at the breakpoint, and the second thread window (Figure 5.21) appears.

Figure 5.21 Second Thread Window


Multithread_Example.elf (<tid: 3> <lwpid: 1110> Stopped)
File
Edit
Search
Project
Debug
Data
Tools
Window
Help
Stack
Variables : All
Value
Location
Source :   /home/usr1/multithread/Source/main.c
Line 77
Col 1
Source

```
0x0FE80EF4 (0x0FE80EF4)
0x0FFD04C0 (0x0FFD04C0)
0x0FFD047C (0x0FFD047C)
thread()
```

```
i      3            0x7F5FFAF8
j      2147482764   0x7F5FFAFC
```

```
/* The thread will begin control in this function */
void *thread(void)
{
  int i,j;
  sum=0;
  i++; // Set Thread BreakPoint Here
  j++; // Set Thread BreakPoint Here
  sum  = i+j;
  INFINITE_LOOP
  pthread_exit(0);
}
```

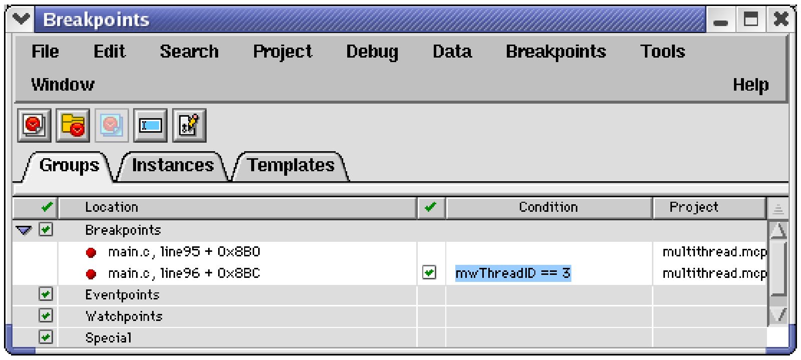
10. Set a breakpoint, which is specific for the second thread.

    a. Set a breakpoint at this line of code in the parent debugger window:

    ```
    j++; // Set Thread BreakPoint Here
    ```

    b. Select **Window > Breakpoints Window**. The **Breakpoints** window (Figure 5.22) appears. For more information, refer the *IDE User's Guide*.

Figure 5.22 Breakpoints Window



✓	Location	✓	Condition	Project
✓	Breakpoints			
	● main.c, line95 + 0x8B0			multithread.mcp
	● main.c, line96 + 0x8BC	✓	mwThreadID == 3	multithread.mcp
✓	Eventpoints			
✓	Watchpoints			
✓	Special			

c. Double-click the **Condition** field corresponding to the breakpoint you have set in the parent debugger window. A cursor appears in the condition field. For this example, it is line 96.

d. Type this condition:

```
mwThreadID == 3.
```

This condition specifies that the breakpoint is valid for the second thread, which has the thread ID 3.

NOTE The thread ID appears on the title bar of the thread window.

e. Close the **Breakpoints** window. A breakpoint specific to the second thread is set.

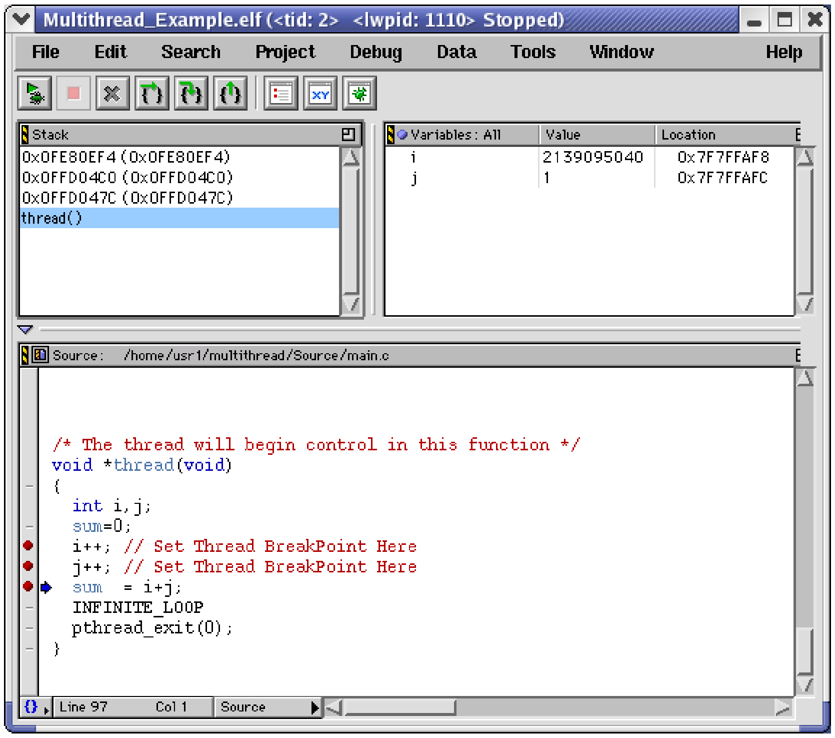
11. Set a breakpoint just after the conditional breakpoint.

    This breakpoint lets you verify that the conditional breakpoint is only valid for the second thread.

12. Execute the first thread.

    Click the **Run** button in the first thread window. The debugger ignores the conditional breakpoint; thread execution stops at the breakpoint just after the conditional breakpoint (Figure 5.23).

Figure 5.23 First Thread Ignores Conditional Breakpoint


Multithread_Example.elf (<tid: 2> <lwpid: 1110> Stopped)
File
Edit
Search
Project
Debug
Data
Tools
Window
Help
Stack
Variables : All
Value
Location
Source : /home/usr1/multithread/Source/main.c
Line 97
Col 1
Source

```
0x0FE80EF4 (0x0FE80EF4)
0x0FFD04C0 (0x0FFD04C0)
0x0FFD047C (0x0FFD047C)
thread()
```

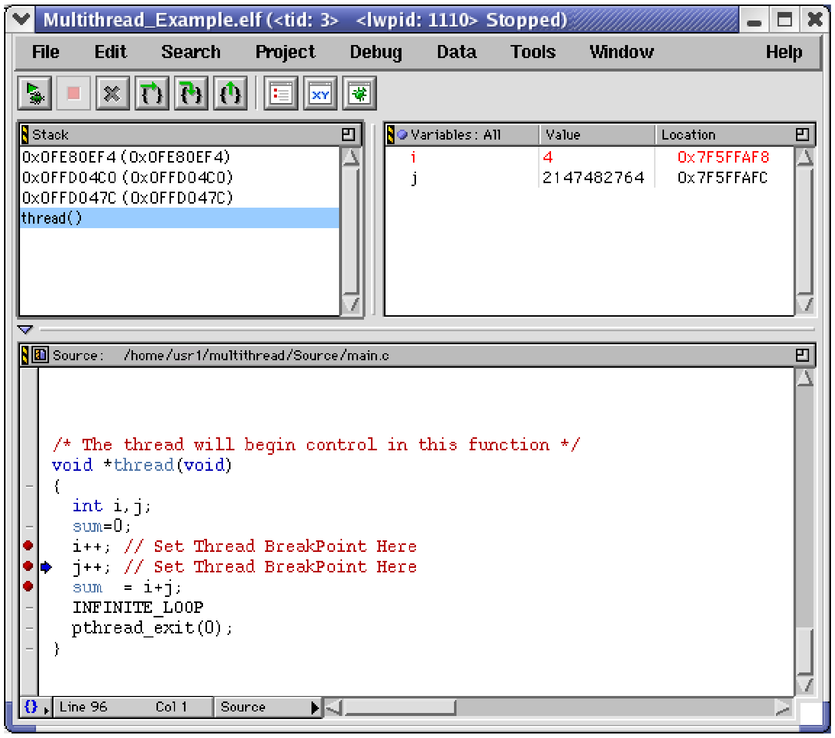
Variables : All	Value	Location
i	2139095040	0x7F7FFAF8
j	1	0x7F7FFAFC

```
/* The thread will begin control in this function */
void *thread(void)
{
  int i,j;
  sum=0;
  i++; // Set Thread BreakPoint Here
  j++; // Set Thread BreakPoint Here
  sum  = i+j;
  INFINITE_LOOP
  pthread_exit(0);
}
```

13. Execute the second thread.

Click **Run** in the second thread window. The thread execution stops at the conditional breakpoint (Figure 5.24) set at the following line of code: `j++; // Set Thread BreakPoint Here`.

Figure 5.24 Execution of Second Thread Stopped at Conditional Breakpoint



```
0x0FE80EF4 (0x0FE80EF4)
0x0FFD04C0 (0x0FFD04C0)
0x0FFD047C (0x0FFD047C)
thread()
```

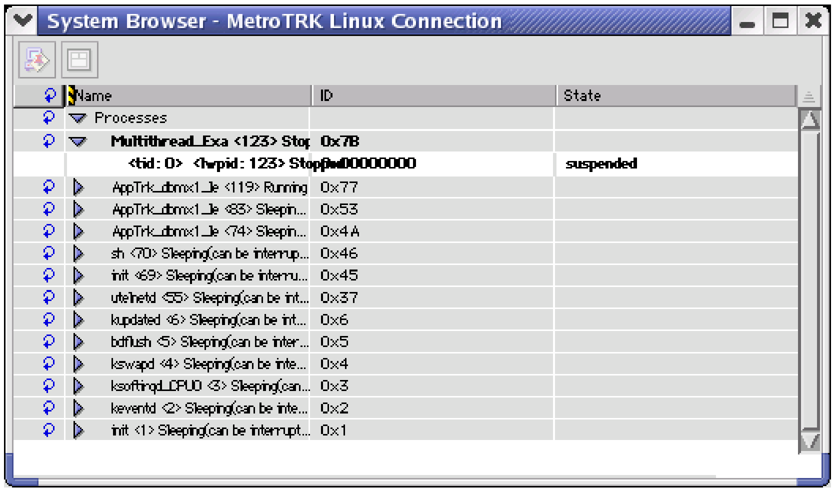
```
i    4             0x7F5FFAF8
j    2147482764    0x7F5FFAFC
```

Source: /home/usr1/multithread/Source/main.c

```
/* The thread will begin control in this function */
void *thread(void)
{
  int i,j;
  sum=0;
  i++; // Set Thread BreakPoint Here
  j++; // Set Thread BreakPoint Here
  sum  = i+j;
  INFINITE_LOOP
  pthread_exit(0);
}
```

14. While debugging, if you wish to view the list of threads associated with a process, select **Window > System Windows**. The **System Browser** window (Figure 5.25) appears. For more information on the **System Browser** window, see *IDE User's Guide*.

Figure 5.25 Example of Multi-thread Processes Window


System Browser - MetroTRK Linux Connection

Name	ID	State
Processes		
Multithread_Exa <123> Stop	0x7B	
<tid: 0> <lwpid: 123> Stopp	0x00000000	suspended
AppTrk_dbmx1_le <119> Running	0x77	
AppTrk_dbmx1_le <83> Sleepin...	0x53	
AppTrk_dbmx1_le <74> Sleepin...	0x4A	
sh <70> Sleeping(can be interrup...	0x46	
init <69> Sleeping(can be interru...	0x45	
utelnetd <55> Sleeping(can be int...	0x37	
kupdated <6> Sleeping(can be int...	0x6	
bdflush <5> Sleeping(can be inter...	0x5	
kswapd <4> Sleeping(can be inte...	0x4	
ksoftirqd_CPU0 <3> Sleeping(can...	0x3	
keventd <2> Sleeping(can be inte...	0x2	
init <1> Sleeping(can be interrupt...	0x1	

You may also use the sample multithreading project available in the CodeWarrior installation directory. For location information, see “Sample Projects Location” on page 177.

Debugging Binary Files With No Source Code

The CodeWarrior™ IDE lets you download and run on the target platform, a binary file (`.elf` or `.so`) whose source code is not available to you. When you open a binary file into the CodeWarrior™ IDE window, the CodeWarrior IDE creates a dummy project for the binary file. You can specify the runtime settings and remote debugging options in the dummy project and download and run the binary file on the target platform.

NOTE For debugging a shared library (.so) file on the target platform, you must associate a host file with the shared library.

To download and run on the target platform, an executable file (.elf) whose source code is not available to you, follow these steps:

1. Create a dummy project.
   a. Select **File > Open** to open the executable file (`.elf`) for which there is no source code available into the CodeWarrior IDE window. A file mapping dialog box appears asking you to map the source files.
   b. Map the source files and click **OK**.

      The CodeWarrior™ IDE creates a dummy project with the same name as the file name of the elf file. For example, if the elf filename is `cw_elf_drop.elf,` the dummy project created will be `cw_elf_drop.mcp`.
2. Change the default output file name to the name of the file you want to run.
   a. Select **Edit >** ***Target*** **Settings**. The target settings window appears.
   b. Click **GNU Target** in the **Target Settings Panels** list. The **GNU Target Settings** panel appears.
   c. Type the name of the executable file in the **Output File Name** text box.

NOTE If the executable file uses a shared library, you need to specify the host-side location and remote download path of the shared library in the **Other Executables** settings panel. Additionally, you need to specify the `LD_LIBRARY_PATH` environment variable in the **Runtime Settings** panel to enable the shared object loader to locate the shared library on the target system.

3. Specify the remote download path of the executable file.
   a. Click **Remote Debugging** in the **Target Settings Panels** list. The **Remote Debugging** settings panel appears.
   b. Select the remote connection name by using the **Connection** list box.
   c. Type the remote download path of the executable file in the **Remote Download Path** text box.
   d. Click **Save** in the **Remote Debugging** settings panel. The target settings are saved.
   e. Close the **Remote Debugging** settings panel.
4. Run the executable file.

   Click **Run** in the project window. The executable file is downloaded to the specified location on the target and executed.

NOTE If the executable file you want to run was compiled with the debug build target selected, you may step through the assembly language code of the executable file by clicking **Debug**.

Using the Attach to Process Feature

The attach to process feature lets you debug a process that is already running on the target system. For example, you may find some problems in a process running on the target system and may want to debug the process. To debug the process, you need not kill the process and start it again. Instead, you can directly attach the debugger to the running process and debug.

In the steps that follow, you will create a sample project where the code causes a process to run in an infinite loop on the target platform. Next, you will attach the debugger to the running process, halt the process, and debug it.

Before you start the tutorial, make sure you have:

- created a TCP/IP connection between the host computer and the remote target
- checked the **Show in processes list** checkbox in the **New Connection** dialog box while creating the new connection
- specified remote debugging options in the **Remote Debugging** settings panel
- launched MetroTRK on the remote target

1. Create a new project using the EPPC New Project Wizard with the following settings:

Table 5.3 Attach to Process Project Settings

Project Name:	ProcessAttach
Location of the Project:	/home/usr1/ProcessAttach
Languages:	C
Output Type:	Application
Output File Name:	AttachToProcess.elf
Location of the Output File:	/home/usr1/ProcessAttach/Output

The above step creates two targets: **c_app_debug** and **c_app_release**. Since this tutorial relates to debugging, only the first target is relevant.

2. Enter the source code of Listing 5.6 into the editor window of `main.c` file.

Listing 5.6 Source Code for main.c File

```
#include <stdio.h>

int main(int argc, char **argv)
{
  int pid;
  int x;
  int i = 10;

  printf("This is a message from the AttachToProcess.elf");
  x=0;

  while(1)
  {
    x++;
    if(x > 500000)
    {
      x=0;
    }
  }
  return 0;
}
```

3. Build the project.
   a. Select the **c_app_debug** build target from the build target list box in the project window, if not selected.
   b. Select **Project > Make**. The final output file `AttachToProcess.elf` is generated and is placed in the specified location in the project folder.
4. Execute the project.

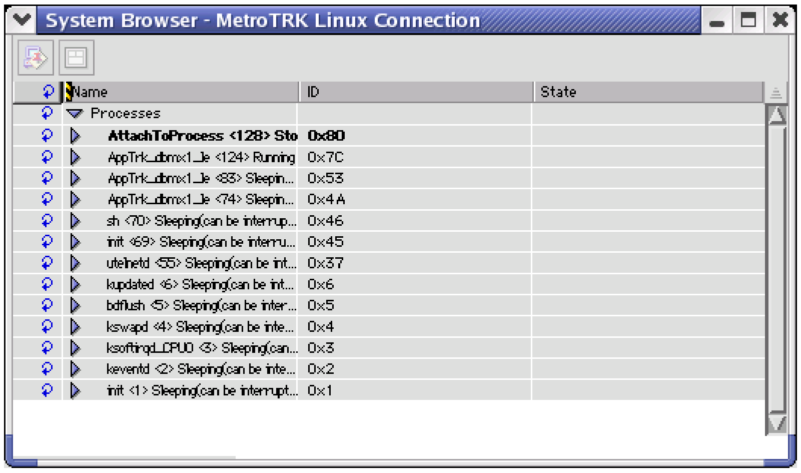
   Select **Project > Run**. The process starts to run in an infinite loop on the target Platform.
5. Establish a connection between the CodeWarrior™ debugger and the remote target system.
   a. Select **Debug > Connect**. The connection window appears.

NOTE The **Connect** command is available only if a project is open. The CodeWarrior IDE uses the current connection selected in the **Remote Debugging** panel, to make a connection to the target system.

   b. Select **Window > System Windows**. The **System Browser** window (Figure 5.26) appears.

NOTE The **System Browser** window view is not continuously refreshed. Any processes that are started immediately after the connection has been established will not be visible in this window. The **System Browser** window view is updated only when there is a change in the state of the process being debugged.

Figure 5.26 System Browser Window



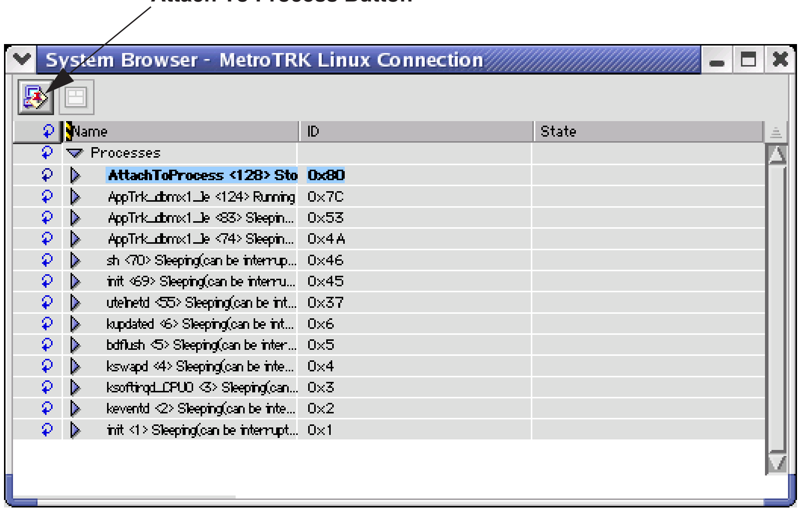
Name	ID	State
Processes		
AttachToProcess <128> Sto	**0x80**	
AppTrk_dbmx1_le <124> Running	0x7C	
AppTrk_dbmx1_le <83> Sleepin...	0x53	
AppTrk_dbmx1_le <74> Sleepin...	0x4A	
sh <70> Sleeping(can be interrup...	0x46	
init <69> Sleeping(can be interru...	0x45	
utelnetd <55> Sleeping(can be int...	0x37	
kupdated <6> Sleeping(can be int...	0x6	
bdflush <5> Sleeping(can be inter...	0x5	
kswapd <4> Sleeping(can be inte...	0x4	
ksoftirqd_CPU0 <3> Sleeping(can...	0x3	
keventd <2> Sleeping(can be inte...	0x2	
init <1> Sleeping(can be interrupt...	0x1	

The **Processes** list in the left pane of the **System Browser** window displays the names of the processes running on the selected target system. Clicking a process name in the **Processes** list displays the threads associated with the process.

TIP You can also view the list of processes on another target system by selecting the corresponding connection name from the **System Windows** submenu. However, the debugger should be connected to the other target system on which you want to view the processes.

c. In the **Processes** list, select the name of the process you want the debugger to attach to. For this tutorial, click the `AttachToProcess` process. The **Attach to Process** button is activated in the **System Browser** window (Figure 5.27).

Figure 5.27 System Browser Window - Attach to Process Button Enabled



Name	ID	State
Processes		
AttachToProcess <128> Sto	0x80	
AppTrk_dbmx1_le <124> Running	0x7C	
AppTrk_dbmx1_le <83> Sleepin...	0x53	
AppTrk_dbmx1_le <74> Sleepin...	0x4A	
sh <70> Sleeping(can be interrup...	0x46	
init <69> Sleeping(can be interru...	0x45	
utelnetd <55> Sleeping(can be int...	0x37	
kupdated <6> Sleeping(can be int...	0x6	
bdflush <5> Sleeping(can be inter...	0x5	
kswapd <4> Sleeping(can be inte...	0x4	
ksoftirqd_CPU0 <3> Sleeping(can...	0x3	
keventd <2> Sleeping(can be inte...	0x2	
init <1> Sleeping(can be interrupt...	0x1	

d. Click the **Attach to Process** button. The **Choose Executable** dialog box appears. This dialog box displays the names of the executable files available for the currently open project.

e. Select the **AttachToProcess.elf** option button.

NOTE If you want to manually search for the executable file, select the **Browse** option button and click **OK**.

f. Click **OK**. The debugger and symbolics windows appear.

If you click the **Cancel** button, a thread window appears with the pointer at the location where the process stopped when the debugger attached to the process. Also, symbolic information is not displayed because no ELF is associated on the host computer. In addition, you can not debug the code in the assembly mode.

NOTE If the debugger is attached to an already running process on the target platform, the console messages appear in the same console window open for the running process.

CAUTION Make sure that you select the correct executable file you want your process to attach to, in the **Choose Executable** dialog box. Otherwise, incorrect symbolics data will be associated with the process.

6. Debug the running process.

   Click the **Break** button in the debugger window. The execution of thread stops and the source code is displayed. You can now perform all the routine debugging operations.

7. Close the Debugger session.

   Select **Debug > Kill** to close the debugger session.

Debugging Applications that use fork() and exec() System Calls

The CodeWarrior™ debugger lets you debug a program that contains `fork()` and `exec()` system calls. Table 5.4 summarizes the descriptions of these system calls.

Table 5.4 fork() and exec() description

System Call	Description
`fork()`	The `fork()` system call is used as a generic call on Linux systems to create a new process. The `fork()` call creates a new process, which is the exact replica of the process that creates it. The only difference is in the PID (Process ID) returned by the fork system call. The value of PID returned in the parent process is the PID of the child, whereas in the child process the PID value returned is zero.
`exec()`	The `exec()` system call launches a new executable in an already running process. The debugger destroys the instance of the previous executable loaded into that address space and a new instance is created.

For debugging applications that use the `fork()` system call, the `fork()` system call is overridden by the `clone()` system call. The `clone()` system call is called with the flag `CLONE_PTRACE` instead of the `fork()` system call. Calling the `clone()` system call with the flag `CLONE_PTRACE` causes:

- the operating system to attach MetroTRK to the child process.
- the child process to stop with a `SIGTRAP` on return from the `clone()` system call.

To call the `clone()` system call transparently while debugging programs that contain the `fork()` system call, you need to add a static library to your project. The source code for building the static library is described later in this section.

NOTE The static library necessary for debugging programs that contain the fork() system call must be added to the project.

Before you start the tutorial, make sure you have:

- created a TCP/IP connection between the host computer and the remote target
- checked the **Show in processes list** checkbox in the **New Connection** dialog box while creating the new connection
- checked the checkbox in the **C** (catch) column corresponding to the SIGCHLD debugger signal in the **Debugger Signals** settings panel
- launched MetroTRK on the remote target

The tutorial that follows demonstrates the functionality for debugging programs that contain `fork()` and `exec()` system calls:

1. As a first step, create a static library project with the following settings (Table 5.5).

Table 5.5 Static Library Project Settings

Project Name:	ForkToCloneLib.mcp
Location of the Project:	/home/usr1/Fork&Exec
Languages:	C
Output Type:	Static Library
Output File Name:	fork2cloneLib.a
Location of the Output File:	/home/usr1/Fork&Exec/Output

The above step creates two targets: **c_lib_static_debug** and **c_lib_static_release**. Since this tutorial relates to debugging, only the first target is relevant.

a. Remove the default main.c file from the project.

b. Add a new `Libstaticfork.c` file to the project. For instructions on how to add a new source file, see “Add a New Source File to the Project” on page 25.

a. Enter the source code of Listing 5.7 into the editor window of `Libstaticfork.c` file.

Listing 5.7 Source Code for Libstaticfork.c

```
/*--------------------------
User Include files
--------------------------*/

#include "db_fork.h"

/*-------------------------
Main Program
```

```
-------------------------*/
   int __libc_fork(void)
 {
   return( __db_fork() );
 }
   extern __typeof (__libc_fork) __fork __attribute__ ((weak, alias
   ("__libc_fork")));
   extern __typeof (__libc_fork) fork __attribute__ ((weak, alias
   ("__libc_fork")));
```

b. Create a header file `db_fork.h` in your project directory and add the code in Listing 5.8 into the header file.

Listing 5.8 Source Code for db_fork.h

```
#include <asm/unistd.h>
#include <errno.h>
#include <signal.h>
#include <sched.h>
#define __NR___db_clone__NR_clone
_syscall2( int, __db_clone, int, flags, int, stack );
```

c. Make the **c_lib_static_debug** build target active.

d. Open the **Access Paths** settings panel and add the pathname of the header file (`db_fork.h`) to the project.

e. Build the `ForkToCloneLib.mcp` project by choosing **Project > Make**. The CodeWarrior IDE builds the project and stores the output file `fork2cloneLib.a` in the *Output* directory within the project directory.

2. Create another project; `Fork&ExecExample.mcp` and create two new build targets with the following settings (Table 5.6):

Table 5.6 Fork and Exec Example Project Settings

Project Name:	Fork&ExecExample
Location of the Project	/home/usr1/Fork&Exec
Languages:	C
Output Type:	Application
Build Targets:	- Parent_debug - ChildA_debug - ChildB_debug

Table 5.6 Fork and Exec Example Project Settings (*continued*)

Parent_debug Build Target - **- Output Type:** **- Output File:** **- Output File Location:**	 Application Parent.elf /home/usr1/Fork&Exec/Output
Child_A_debug Build Target - **- Output Type:** **- Output File:** **- Output File Location:**	 Application Child-A.elf /home/usr1/Fork&Exec/Output
Child_B_debug Build Target - **- Output Type:** **- Output File:** **- Output File Location:**	 Application Child-B.elf /home/usr1/Fork&Exec/Output

3. Add the source files `fork.c`, `ChildA.c`, and `ChildB.c` to the `Fork&ExecExample.mcp` project.

   - `fork.c` — will contain the code of the parent process
   - `ChildA.c` — will generate the executable file `Child-A.elf`
   - `ChildB.c` — will generate the executable file `Child-B.elf`

   The code of the parent process creates a forked process (child process) when the `__db_fork` function executes. The debugger opens a separate thread window for the child process. When the child process finishes executing, the debugger closes the thread window. To debug the code of the child process, you need to set a breakpoint in the child process code or stop the execution of the child process by clicking the **Break** button. You can debug the code of the child process the same way you debug code of any other process.

   The code of both child and parent processes contain `exec()` function calls that execute the `Child-A.elf` and `Child-B.elf` files, respectively.

   As you step through the code of the child process, the `exec()` function call executes and a separate debugger window for the `Child-A.elf` appears. You can perform normal debug operations in this window. Similarly, you step through the code of the parent process to execute the `exec()` system call. The debugger destroys the instance of the previous file (`Parent.elf`) and creates a new instance for the `Child-B.elf` file.

4. Enter the source code of Listing 5.9 into the editor window of `fork.c` file.

Listing 5.9 Source Code for fork.c

```
/*-------------------------
System Include files
-------------------------*/
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/ptrace.h>
#include <sys/errno.h>
#include <sys/types.h>
#include <signal.h>
#include <sched.h>
#include <fcntl.h>
#include <dlfcn.h>
/*-----------------------------
User Include files
-----------------------------*/
#include "CWExample.h"

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

/*-------------------------------
Globals and Constants
-------------------------------*/
int gint;
#define CHILDA_DBG "/home/sample/Child-A.elf"
#define CHILDB_DBG "/home/sample/Child-B.elf"
/*-------------------------------
Main Program
-------------------------------*/
int main(void)
{
   int pid,x;
   int shared_local;
   char *argv[5];
   printf( "Fork Testing!\r\n" );
   fflush( stdout );
   gint = 5;
   shared_local =5;
   pid = fork();
   if(pid == 0)
   {
     x=0;
     gint = 10;
     shared_local =10;
     printf("I am the child,my process ID is %d\n",getpid());
     printf("The child's parent process ID is %d\n",getppid());
     argv[0] = CHILDA_DBG;
     argv[1] = NULL;
     execv(argv[0],argv);
   }
   else
   {
     x=0;
     gint = 12;
     shared_local =12;
     printf("I am the parent,my process ID is %d\n",getpid());
     printf("The parent's parent process ID is %d\n",getppid());
     argv[0] = CHILDB_DBG;
    argv[1] = NULL;
    execv(argv[0],argv);
  }
  return 0;
}
```

NOTE Make sure that you include the `CWExamples.h` file in your project. You can do this using the **Access Paths** settings panel.

5. Enter the source code of Listing 5.10 into the editor window of `ChildA.c` file.

Listing 5.10 Source Code for ChildA.c

```
/*-------------------------------
System Include files
-------------------------------*/
#include <stdio.h>

/*-------------------------------
Main Program
-------------------------------*/
int main(int argc, char **argv)
{
   printf("This is a message from the child-A.elf\n");
   return 0;
}
```

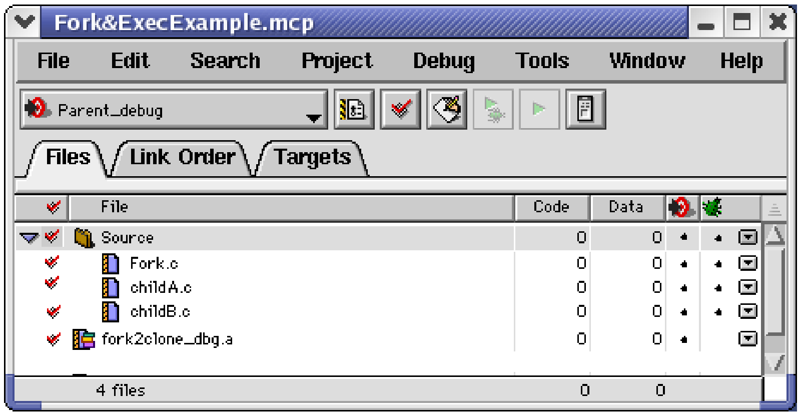
6. Enter the source code of Listing 5.11 into the editor window of `ChildB.c` file.

Listing 5.11 Source code for ChildB.c

```
/*-----------------------------------
System Include files
-----------------------------------*/
#include <stdio.h>

/*-----------------------------------
Main Program
-----------------------------------*/
int main(int argc, char **argv)
{
     printf("This is a message from the child-B.elf\n");
     return 0;
}
```

7. Add `fork2cloneLib.a` file to the `Fork&ExecExample.mcp` project.
   a. Right-click on the project window and select **Add Files** from the context menu.
   b. Navigate to the directory where you have stored the `fork2cloneLib.a` file in your project folder. For this tutorial it is: `/home/usr1/Fork&Exec/Output`.
   c. Select the `fork2cloneLib.a` file and click **Open**. The **Add Files** dialog box appears.
   d. Click **OK**. The `fork2cloneLib.a` file gets added to the project (Figure 5.28).

Figure 5.28 Fork&ExecExample.mcp Project Window



File	Code	Data
Source	0	0
Fork.c	0	0
childA.c	0	0
childB.c	0	0
fork2clone_dbg.a	0	0
4 files	0	0

8. Build `Fork&ExecExample.mcp` project.
   a. Select the **Parent_debug** build target from the build target list box in the project window, if not selected.

b. Select **Project > Make**. The CodeWarrior IDE generates the `Parent.elf`, `Child-A.elf`, and `Child-B.elf` executable files and places them in the project folder. For this tutorial it is: `/home/usr1/Fork&ExecExample/Output`.

9. Specify the host-side location and remote download path of the executable files to be launched by the `exec()` system call.

   a. Select **Edit > Parent_debug Settings**. The **Parent_debug Settings** window appears.

   b. Click **Other Executables** in the **Target Settings Panels** list. The **Other Executables** settings panel appears.

   c. Click **Add** in the **Other Executables** settings panel. The **Debug Additional Executable** dialog box appears.

   d. Click **Choose** in the **Debug Additional Executable** dialog box. The **Choose an Executable to Debug** dialog box appears.

   e. Navigate to the project directory (the `/home/usr1/Fork&ExecExample/Output` directory)

   f. Select **Child-A.elf**.

   g. Click **Open**. The path of the selected file appears in the **File Location** text box.

   h. Check the **Download file during remote debugging** checkbox.

   i. In the **Remote Download Path** text box, type the path where you want to download the executable. For example, you may specify `/home/sample`.

   j. Click **OK**. The **File** list in the **Other Executable** settings panel shows the path of the selected executable file.

   k. Repeat steps c through e.

   l. Select **ChildB.elf**.

   m. Repeat steps g through j.

10. Specify remote debugging options.

   a. Click **Remote Debugging** from the list of settings panels. The **Remote Debugging** settings panel appears.

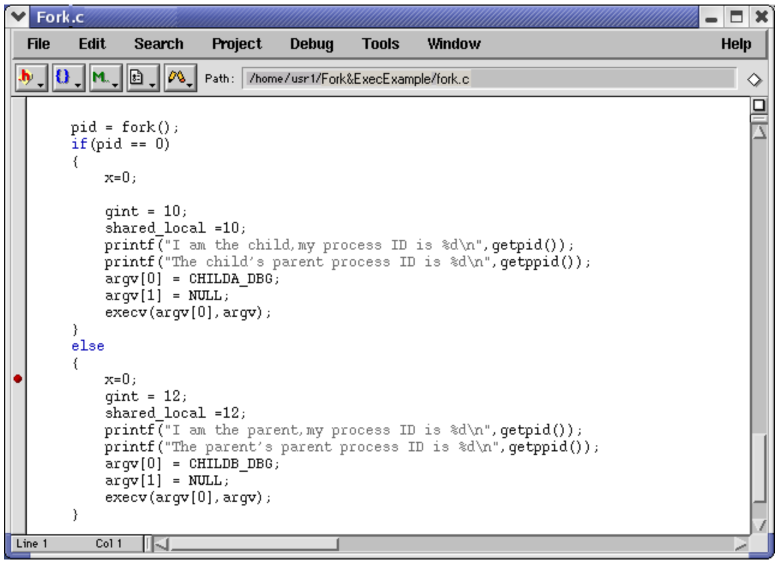
   b. Select the remote connection name by using the **Connection** list box.

   c. In the **Remote download path** text box, specify the location where the executable file `Parent.elf` is to reside on the remote target. For example, you may specify `/home/sample`.

11. Set breakpoints in the child and parent processes.

a. Double-click the `fork.c` filename in the project window. The editor window (Figure 5.29) appears.

Figure 5.29 Source Code of fork.c File



```
pid = fork();
if(pid == 0)
{
    x=0;

    gint = 10;
    shared_local =10;
    printf("I am the child,my process ID is %d\n",getpid());
    printf("The child's parent process ID is %d\n",getppid());
    argv[0] = CHILDA_DBG;
    argv[1] = NULL;
    execv(argv[0],argv);
}
else
{
    x=0;
    gint = 12;
    shared_local =12;
    printf("I am the parent,my process ID is %d\n",getpid());
    printf("The parent's parent process ID is %d\n",getppid());
    argv[0] = CHILDB_DBG;
    argv[1] = NULL;
    execv(argv[0],argv);
}
```

b. Set a breakpoint in the code of the child process at this line: `x=0;`.

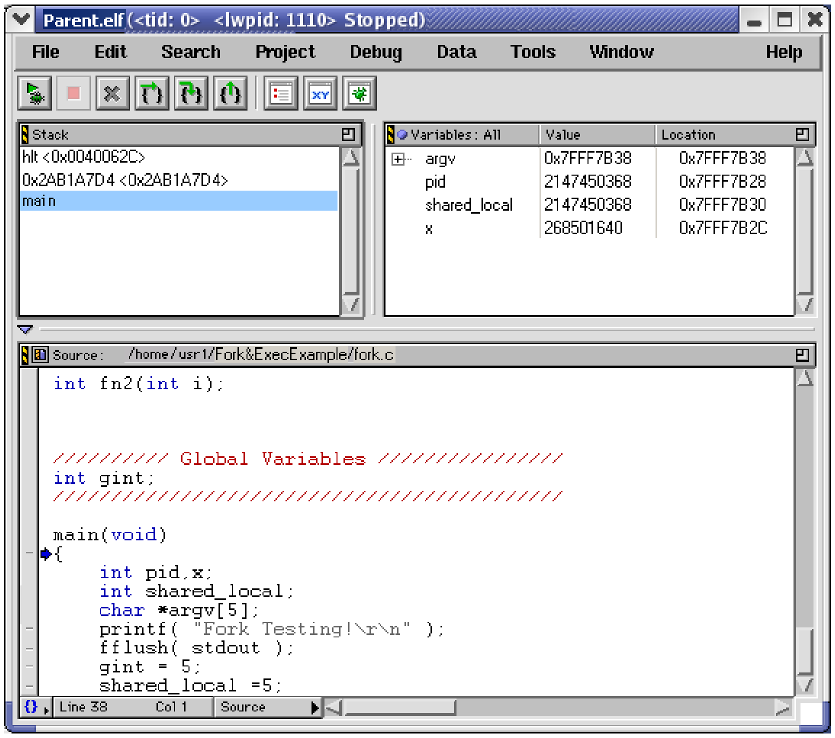
c. Set a breakpoint in the code of the parent process.

d. Close the `fork.c` file.

12. Start the debugger.

Select **Project > Debug**. The debugger window (Figure 5.30) appears. The debugger downloads the `Parent.elf`, `Child-A.elf`, and `Child-B.elf` executable files to the specified location on the remote target one by one.

Figure 5.30 Debugger Window for Parent Process



```
hlt <0x0040062C>
0x2AB1A7D4 <0x2AB1A7D4>
main
```

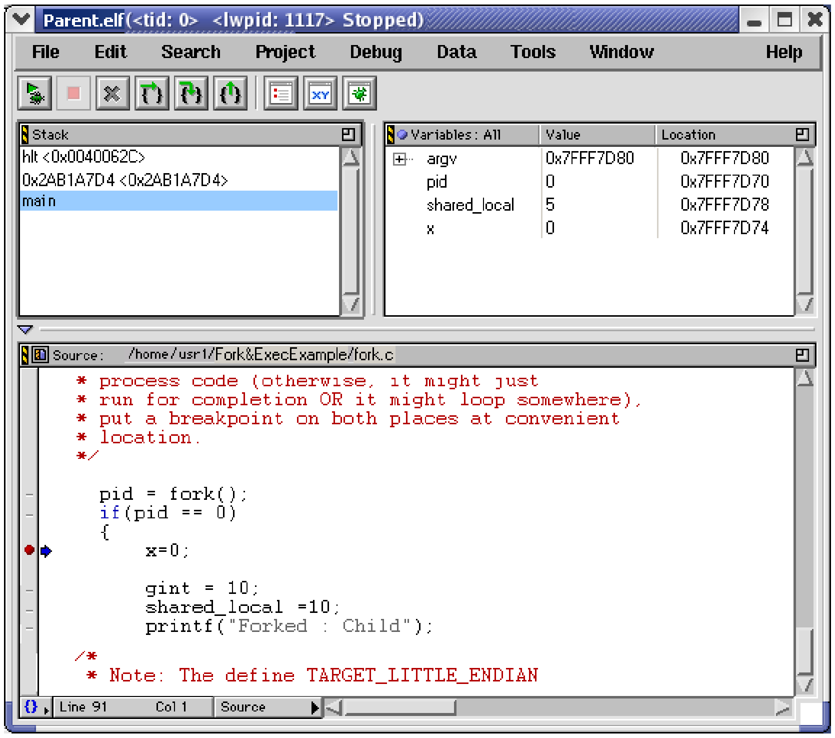
```
argv          0x7FFF7B38    0x7FFF7B38
pid           2147450368    0x7FFF7B28
shared_local  2147450368    0x7FFF7B30
x             268501640     0x7FFF7B2C
```

```
int fn2(int i);


////////// Global Variables ////////////////
int gint;
////////////////////////////////////////////

main(void)
{
    int pid,x;
    int shared_local;
    char *argv[5];
    printf( "Fork Testing!\r\n" );
    fflush( stdout );
    gint = 5;
    shared_local =5;
```

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

    When the `fork()` system call is called, the child process debugger window (Figure 5.31) appears. You can now perform normal debugging operations in this window.

Figure 5.31 Debugger Window for Child Process



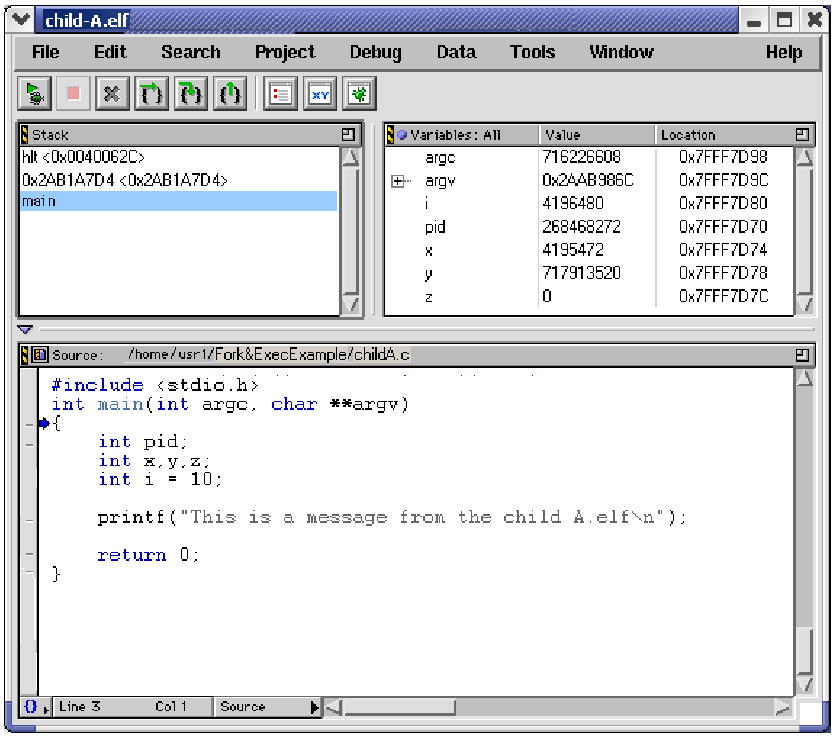
```
hlt <0x0040062C>
0x2AB1A7D4 <0x2AB1A7D4>
main
```

Variables: All	Value	Location
argv	0x7FFF7D80	0x7FFF7D80
pid	0	0x7FFF7D70
shared_local	5	0x7FFF7D78
x	0	0x7FFF7D74

```
 * process code (otherwise, it might just
 * run for completion OR it might loop somewhere),
 * put a breakpoint on both places at convenient
 * location.
 */

  pid = fork();
  if(pid == 0)
  {
      x=0;

      gint = 10;
      shared_local =10;
      printf("Forked : Child");

 /*
  * Note: The define TARGET_LITTLE_ENDIAN
```

14. Step over the code in the child process debugger window a couple of times. When the `exec()` function call in the child process code executes, a new debugger window (Figure 5.32) appears. This window displays the code of the `Child-A.elf` executable file. You can now perform normal debugging operations in this window.

Figure 5.32 Debugger Window for File Executed by Child Process



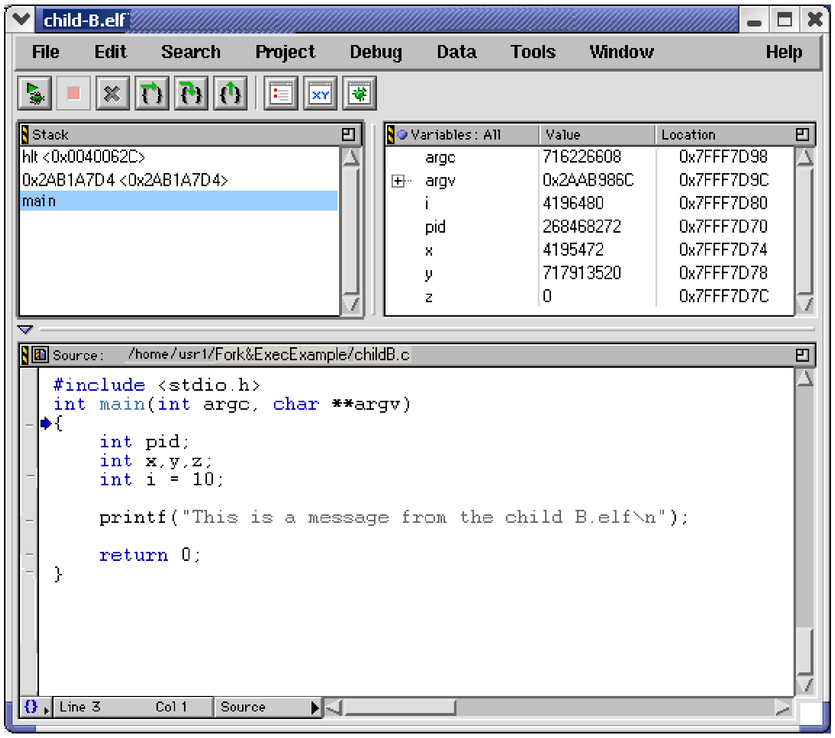
```
hlt <0x0040062C>
0x2AB1A7D4 <0x2AB1A7D4>
main
```

Variables: All	Value	Location
argc	716226608	0x7FFF7D98
argv	0x2AAB986C	0x7FFF7D9C
i	4196480	0x7FFF7D80
pid	268468272	0x7FFF7D70
x	4195472	0x7FFF7D74
y	717913520	0x7FFF7D78
z	0	0x7FFF7D7C

```
#include <stdio.h>
int main(int argc, char **argv)
{
    int pid;
    int x,y,z;
    int i = 10;

    printf("This is a message from the child A.elf\n");

    return 0;
}
```

15. Next, step over the code in the parent process debugger window a couple of times. When the `exec()` function call in the parent process code executes, the debugger destroys the instance of the previous executable file (`Parent.elf`) and creates a new instance for the `Child-B.elf` file (Figure 5.33). You can now perform normal debugging operations in this window.

Figure 5.33 Debugger Window for File Executed by Parent Process



```
hlt <0x0040062C>
0x2AB1A7D4 <0x2AB1A7D4>
main
```

Variables : All	Value	Location
argc	716226608	0x7FFF7D98
argv	0x2AAB986C	0x7FFF7D9C
i	4196480	0x7FFF7D80
pid	268468272	0x7FFF7D70
x	4195472	0x7FFF7D74
y	717913520	0x7FFF7D78
z	0	0x7FFF7D7C

```
#include <stdio.h>
int main(int argc, char **argv)
{
    int pid;
    int x,y,z;
    int i = 10;

    printf("This is a message from the child B.elf\n");

    return 0;
}
```

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

You may also use the sample fork and exec projects available in the CodeWarrior installation directory. For location information, see “Sample Projects Location” on page 177.

Viewing Multiple Processes and Threads

Whenever an application which forks a new process is debugged a new thread window is created and is displayed in the debugger window. If you debug an application that creates many new processes, a number of thread windows appear in the CodeWarrior™ IDE window. Making the CodeWarrior IDE window cluttered with thread windows leading to a lot of confusion about which thread window to debug.

To overcome this problem, a new option is added in the **IDE Preferences** panel that allows you to specify whether you want to display the new processes and associated threads in separate thread windows or in a single thread window.

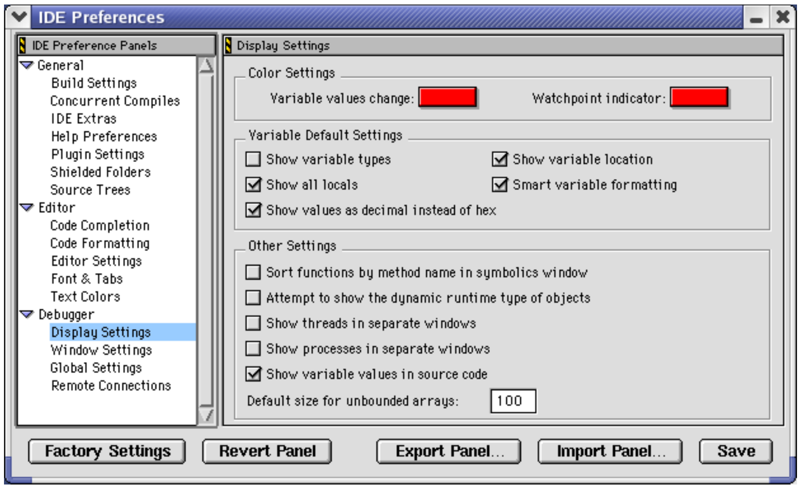
NOTE You can display all the processes in a single thread window for a given remote connection only.

The steps to do this are as follows:

1. Open the **Display Settings** panel.
   a. From the project window, select **Edit > Preferences**. The **IDE Preferences** window appears.
   b. Click **Display Settings** in the list of settings panels in the left pane. The **Display Settings** panel appears in the right pane.
2. Specify the settings to show all the processes and threads in a single debugger window.
   a. Ensure that the **Show processes in separate window** and **Show threads in separate window** checkboxes is cleared in the **Display Settings** panel (Figure 5.34).

NOTE If you check the **Show processes in separate window** and **Show threads in separate window** checkboxes, each process and its associated thread will be displayed in a separate thread window.

Figure 5.34 Display Settings Panel



Variable values change: 🟥 Watchpoint indicator: 🟥

☐ Show variable types
☑ Show variable location
☑ Show all locals
☑ Smart variable formatting
☑ Show values as decimal instead of hex

☐ Sort functions by method name in symbolics window
☐ Attempt to show the dynamic runtime type of objects
☐ Show threads in separate windows
☐ Show processes in separate windows
☑ Show variable values in source code
Default size for unbounded arrays: 100

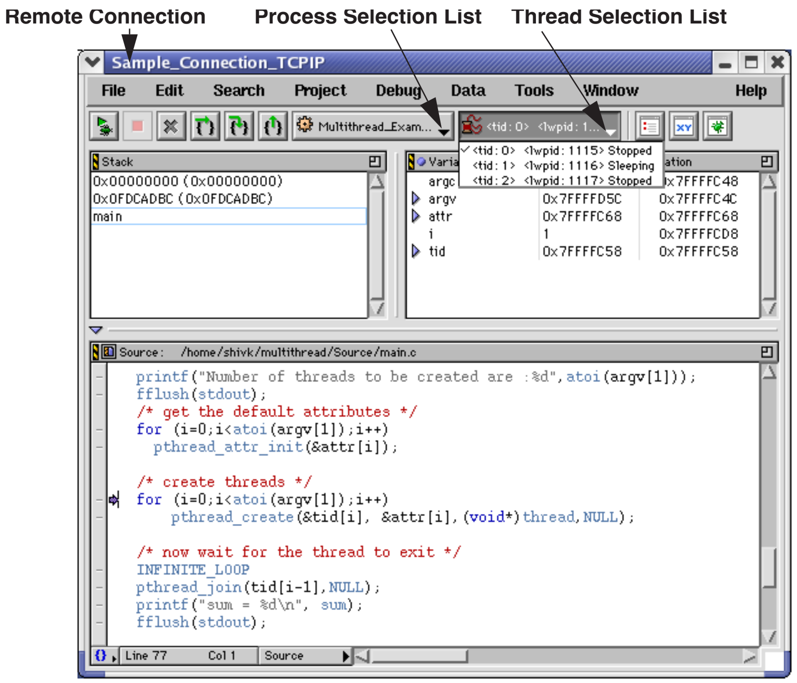
   b. Click **Save** to save the settings.

c. Close the **IDE Preferences** panel.

3. Start the debugger.

   a. Select **Project > Debug**. The thread window (Figure 5.35) for `Multithread_Example.elf` appears.

Figure 5.35 Thread Window for Multithread_Example.elf



```
✓ <tid : 0>  <lwpid : 1115> Stopped
  <tid : 1>  <lwpid : 1116> Sleeping
  <tid : 2>  <lwpid : 1117> Stopped
```

```
0x00000000 (0x00000000)
0x0FDCADBC (0x0FDCADBC)
main
```

```
  argc
▷ argv     0x7FFFFD5C    0x7FFFFC4C
▷ attr     0x7FFFFC68    0x7FFFFC68
  i        1             0x7FFFFCD8
▷ tid      0x7FFFFC58    0x7FFFFC58
```

/home/shivk/multithread/Source/main.c

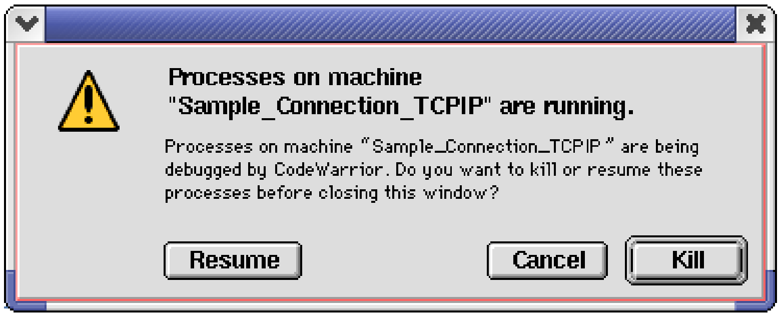
```
printf("Number of threads to be created are :%d",atoi(argv[1]));
fflush(stdout);
/* get the default attributes */
for (i=0;i<atoi(argv[1]);i++)
  pthread_attr_init(&attr[i]);

/* create threads */
for (i=0;i<atoi(argv[1]);i++)
    pthread_create(&tid[i], &attr[i],(void*)thread,NULL);

/* now wait for the thread to exit */
INFINITE_LOOP
pthread_join(tid[i-1],NULL);
printf("sum = %d\n", sum);
fflush(stdout);
```

The thread window shows two list boxes that display the name of the currently debugged process (`Multithread_Example.elf`), the Process ID (PID) of the current process, and the Thread ID (TID) of the current thread. The thread window title bar shows the remote connection name used for debugging the current process. In this example it is `Sample_Connection_TCP/IP`.

b. Click the **Kill** button of the currently active thread window to kill that process.

   If you click the **(X)** button at the top right hand corner of the thread window, a message box (Figure 5.36) appears.

Figure 5.36 Metrowerks CodeWarrior Message Box



Processes on machine "Sample_Connection_TCPIP" are running.

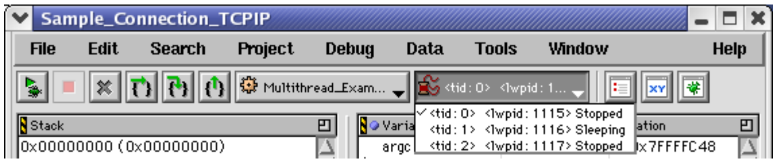
Processes on machine "Sample_Connection_TCPIP" are being debugged by CodeWarrior. Do you want to kill or resume these processes before closing this window?

This message box informs you that currently processes are running on remote connection machine and waits for your instruction. You can perform the following actions:

- Click **Kill** to stop all the currently running processes in the thread window.
- Click **Resume** to close the current debug session and resume it later. All thread windows that are currently open are closed. The project window remains open.
- Click **Cancel** to cancel the action. The thread window remains open and the currently running processes are not affected.

NOTE If you debug a multi-threaded application, any new thread created is listed in the Thread Selection list box (Figure 5.37).

Figure 5.37 Multi threaded Application - Multiple Threads in Same Thread Window



```
✓ <tid: 0> <lwpid: 1115> Stopped
  <tid: 1> <lwpid: 1116> Sleeping
  <tid: 2> <lwpid: 1117> Stopped
```

```
0x00000000 (0x00000000)
x7FFFFC48
```

Viewing Process Information

When you open a debug session (**Project > Debug**) or connect to the target platform (**Debug > Connect**), the CodeWarrior IDE displays the **Linux Info** menu that you may use to view details about the processes running on your target platform.

The **Linux Info** menu (Figure 5.38) contains commands that enable you to view and refresh the processes running on the target platform.

Figure 5.38 Linux Info Menu

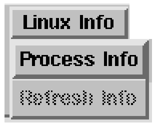


Table 5.7 describes the menu commands provided by the **Linux Info** menu.

Table 5.7 Linux Info Menu - Description of Commands

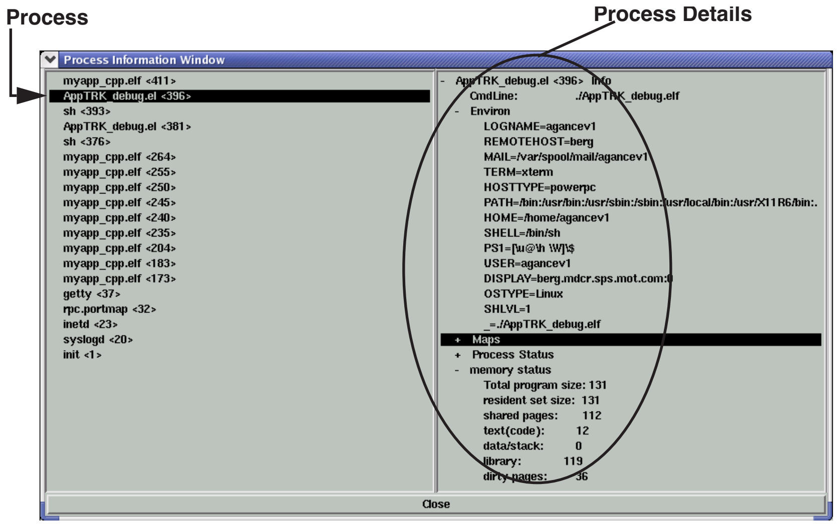
Commands	Description
Process Info	Displays the list of currently running processes on the target platform with a detailed description about each process
Refresh Info	Refreshes the processes list

NOTE The **Linux Info** menu disappears when you close the debugging session.

To view details of the currently running process, the steps are:

1. Start a debug session.
2. Select **Linux Info > Process Info**. The **Process Information Window** (Figure 5.39) appears.

Figure 5.39 Process Information Window



```
myapp_cpp.elf <411>
AppTRK_debug.el <396>
sh <393>
AppTRK_debug.el <381>
sh <376>
myapp_cpp.elf <264>
myapp_cpp.elf <255>
myapp_cpp.elf <250>
myapp_cpp.elf <245>
myapp_cpp.elf <240>
myapp_cpp.elf <235>
myapp_cpp.elf <204>
myapp_cpp.elf <183>
myapp_cpp.elf <173>
getty <37>
rpc.portmap <32>
inetd <23>
syslogd <20>
init <1>
```

```
-  AppTRK_debug.el <396>  Info
     CmdLine:            ./AppTRK_debug.elf
   -  Environ
        LOGNAME=agancev1
        REMOTEHOST=berg
        MAIL=/var/spool/mail/agancev1
        TERM=xterm
        HOSTTYPE=powerpc
        PATH=/bin:/usr/bin:/usr/sbin:/sbin:/usr/local/bin:/usr/X11R6/bin:.
        HOME=/home/agancev1
        SHELL=/bin/sh
        PS1=[\u@\h \W]\$
        USER=agancev1
        DISPLAY=berg.mdcr.sps.mot.com:0
        OSTYPE=Linux
        SHLVL=1
        _=./AppTRK_debug.elf
   +  Maps
   +  Process Status
   -  memory status
        Total program size: 131
        resident set size:  131
        shared pages:       112
        text(code):         12
        data/stack:         0
        library:            119
        dirty pages:        36
```

The **Process Information Window** displays the currently running processes in the left-hand side of the window.

3. Select the process for which you want to view the details from the processes list.

   The left-hand side of the window displays the details for the selected process, such as environment settings, process status, and address mappings.

NOTE You may not be able to view information for processes for which you do not have read/write permissions on `/proc` files for that particular process. For example, the environ (environment) details for a process might not be displayed, if the AppTRK owner on the target platform does not have read/right permissions on the `/proc` files for that process.

4. Select **Linux Info > Refresh Info** to refresh the current state of the processes.
5. Close the **Process Information Window**.

6 Other Common Features

The previous chapter described how to use the debugging features that are common across all the target platforms supported by the CodeWarrior™ Development Studio for Embedded Linux®. This chapter describes the other non-debugging features that you can use across all the target platforms supported by the CodeWarrior™ Development Studio for Embedded Linux.

This chapter describes the:

- Makefile Importer Wizard used for generating new projects using the makefile
- Post Linker Stripper feature used for generating a stripped version of the executable binary
- Shell Tool Post Linker feature used for displaying the contents of the shell script file included in a project

This chapter has these sections:

- Creating New Projects From Makefiles
- Stripping Binary Files
- Using the Shell Tool Post Linker

Creating New Projects From Makefiles

The CodeWarrior™ IDE allows you to convert most GNU makefiles into projects by using the Makefile Importer wizard. The wizard performs the following tasks:

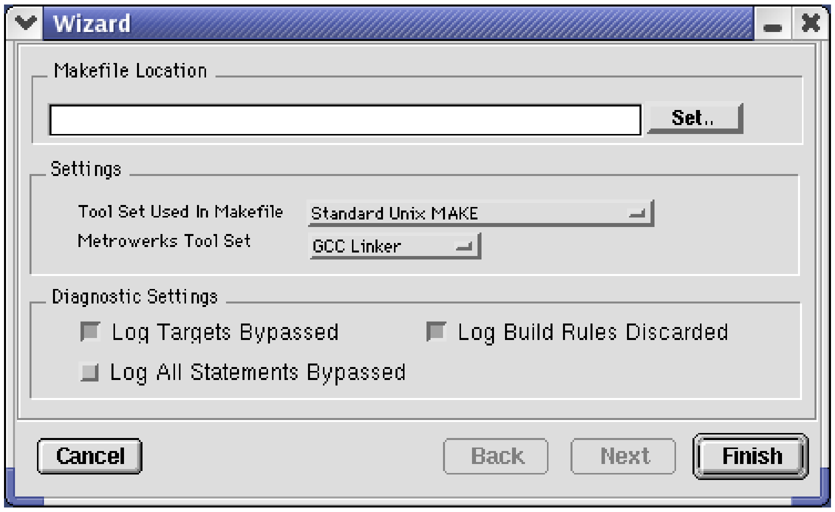
- Parses the makefile to determine source files and build targets
- Creates a project
- Adds the source files and build targets determined during parsing
- Matches makefile information, such as output name, output directory, and access paths, with the newly created build targets.
- Selects a project linker

The steps to use the **Makefile Importer Wizard** to convert a makefile into a project are as follows:

1. Select **File > New**. The **New** dialog box appears.
2. Select **Makefile Importer Wizard**.

3. Enter a project name (include the `.mcp` extension) in the **Project name** field.
4. Set the location for the new project in the **Location** text box. Alternatively, you may click the **Set** button to change the location of the new project.
5. Click **OK**. The **Makefile Importer Wizard** dialog box appears.
6. Enter the path to the makefile in the **Makefile Location** text box (Figure 6.1). Alternatively, you may click **Set** to navigate to the makefile.

Figure 6.1 Makefile Importer Wizard Dialog Box


Wizard

Makefile Location

Settings
Tool Set Used In Makefile: Standard Unix MAKE
Metrowerks Tool Set: GCC Linker

Diagnostic Settings
Log Targets Bypassed
Log Build Rules Discarded
Log All Statements Bypassed

7. Select the tool set used for makefile conversion and linker selection.

   Select **Standard UNIX MAKE** from the **Tool Set Used In Makefile** list box under the **Settings** group (Figure 6.1).

   The **Metrowerks Tool Set** list box under the **Settings** group displays the default linker to be used with the converted project. The linker name may vary depending on the CodeWarrior™ Development Studio for Embedded Linux product you are using.

NOTE The **Tool Set Used In Makefile** list box allows to select the tool set whose build rules form the basis of the makefile. The **Metrowerks Tool Set** list box allows to select the linker tool set to use with the generated project.

8. Select the desired diagnostic settings.

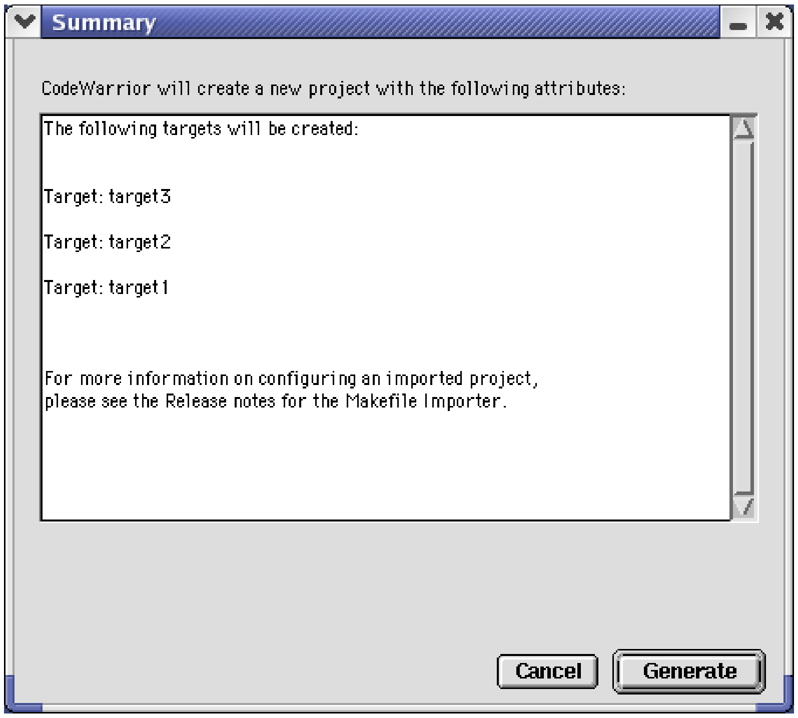
a. Check the **Log Targets Bypassed** checkbox in the **Diagnostic Settings** group to log information about makefile build targets that the CodeWarrior IDE fails to convert to project build targets.

b. Check the **Log Build Rules Discarded** checkbox in the **Diagnostic Settings** group to log information about makefile rules that the CodeWarrior IDE discards during conversion.

c. Check the **Log All Statements Bypassed** checkbox in the **Diagnostic Settings** group to log targets bypassed, build rules discarded, and other makefile items that the CodeWarrior IDE fails to convert.

9. Click **Finish**. The **Summary** dialog box (Figure 6.2) appears, which lists the targets that will be created for the new project.

Figure 6.2 Summary Dialog Box



CodeWarrior will create a new project with the following attributes:

The following targets will be created:

Target: target3

Target: target2

Target: target1

For more information on configuring an imported project,
please see the Release notes for the Makefile Importer.

10. Click **Generate**. The **Makefile Importer Wizard** performs the conversion process and displays the project window for the new project.

NOTE When the new project is generated using the Makefile Importer Wizard, the new project contains the default settings defined in the

`GCCImporterStationery.mcp` project file. You may customize the settings by modifying the `GCCImporterStationery.mcp` file and than generating the project once again using the Makefile Importer Wizard. The generated project will now contain the settings specified by you.

You may also use the sample makefile available at this location in your CodeWarrior installation directory:

`CWInstall``/CodeWarriorIDE/Examples/Makefiles`

Stripping Binary Files

One of the important features of CodeWarrior Development Studio for Embedded Linux products is the *Post Linker Stripper* feature. The *Post Linker Stripper* feature enables you to reduce the file size of an application executable binary (*.elf*) or a shared library (*.so*) by removing the data not required by the target platform to run the application, such as the sections related to debugging and much of the symbolics data. This results in faster download of the binary on the target platform.

NOTE The file size reduction varies depending on the debug format used.

You need to select the *target platform-specific* **Post Linker - Stripper** option in the **Target Settings** panel to perform this task. *target platform-specific* denotes the target platform for which you are writing the application. For example, ARM® or PowerPC™-based target platforms. Then, the post linker adaptor is passed the following information:

- pathname of the binary (.elf or .so) that need to be stripped of the debug information
- options specified in the **GNU Post Linker** settings panel
- command line utility name

When the project is linked, the post linker adaptor calls the command line utility (*strip.exe*) specified in the **GNU Tools** settings panel and passes the pathname of the binary to be stripped of debug information. After a stripped version of the binary is created, the binary can be downloaded on the target platform.

Creating Stripped Binary Files

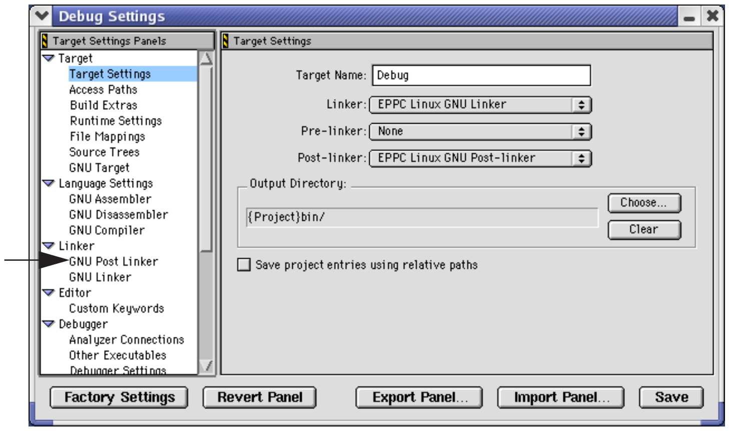
The steps to create a stripped version of an executable binary (`.elf`) are as follows:

1. Create a project that can successfully generate a full-size executable binary (`.elf`).
2. Make the post linker stripper settings.

a. From the project window, open the **Target Settings** panel.

b. Select *target platform-specific* **Post Linker - Stripper** from the **Post-linker** list box. Here, *target platform-specific* denotes the target platform for which you are writing the application. For example, ColdFire®, ARM®, or PowerPC™-based target platforms. Figure 6.3 shows the **Post-linker** option.

Figure 6.3 Selecting *target platform-specific* Post Linker - Stripper Option



Target Name: Debug
Linker: EPPC Linux GNU Linker
Pre-linker: None
Post-linker: EPPC Linux GNU Post-linker
Output Directory: {Project}bin/
☐ Save project entries using relative paths

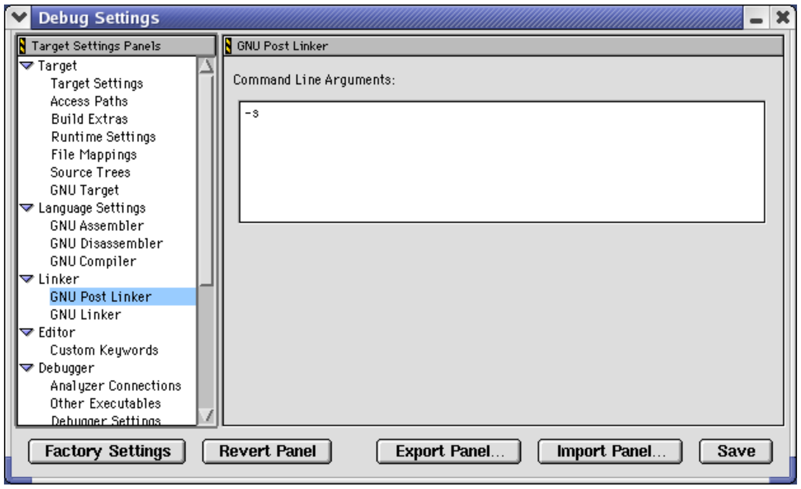
When you select *target platform-specific* **Post Linker - Stripper** option, a new item; **GNU Post Linker** is added to the **Target Settings Panels** tree structure under the **Linker** tree (Figure 6.3).

3. Specify command-line arguments to be passed to the command line utility.

a. Open the **GNU Post Linker** panel.

b. Type `-s` in the **Command Line Arguments** text box (Figure 6.4).

Figure 6.4 Specifying Command Line Arguments



Command Line Arguments:

```
-s
```

4. Specify the name of the post linker command line utility.
   a. Open the **GNU Tools** panel.
   b. Type `strip` in the **Post Linker** text box.

NOTE The post linker stripper executable filename may vary depending on the cross compiler tools you are using.

5. Save the settings and compile the project.
   a. Click **Save** to save the post linker settings.
   b. Close the **GNU Tools** panel.
   c. Select **Project > Make**. The project is compiled and a new file named `<original-name>.elf.strip` or `<original-name>.so.strip` is created in the project folder where `<original-name>.elf` or `<original-name>.so` is the name of the original executable binary file.

NOTE The executable files are generated in the project folder irrespective of whether you use the *target platform-specific* **Post Linker - Stripper** option or not.

NOTE The file extension of the stripped version of the executable binary generated is `.strip` by default and cannot be changed.

If you compare the file size of the original and stripped files, the later is smaller in size. This reduces the download time of the executable binary on the target platform.

NOTE The file size of the stripped file may vary for different debug formats used for different target platforms. For all the target platforms supported by the CodeWarrior™ Development Studio for Embedded Linux, the debug format used in *STABS* or *DWARF 2*.

Downloading Stripped Files

While downloading executable binary (.elf) or shared library files (.so) on the target platform, the debugger first searches for the stripped version of the files mentioned in the:

- **Output Target** text box in the **GNU Target** panel
- **Other Executables** panel
- **Runtime Settings** panel

If a stripped version of the `.elf` or `.so` files exists and is the latest file available, than the debugger downloads the stripped file on the target platform. Otherwise, the original `.elf` or `.so` file is downloaded on the target.

NOTE In case you want to download the stripped version of the files mentioned in the **Other Executables** or **Runtime Settings** panel, make sure that you built these files using the *target platform-specific* **GNU Post Linker - Stripper** option in their respective projects. This will ensure that the debugger finds a stripped version of these files and downloads it on the target platform.

Using the Shell Tool Post Linker

The *Shell Tool Post Linker* displays the contents of the shell script file included in your project after the build process is complete.

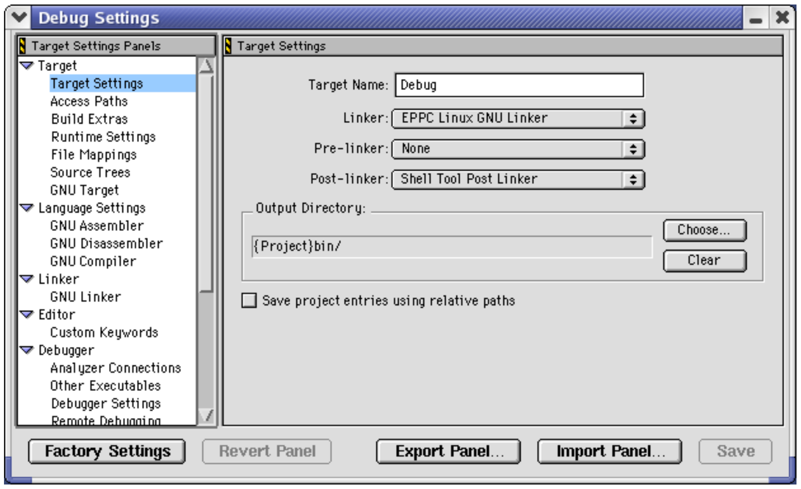
To use the Shell Tool Post Linker, the steps are:

1. Open a project.
2. Make the shell tool post linker settings.

a. From the project window, open the **Target Settings** panel.

b. Select **Shell Tool Post Linker** from the **Post-linker** list box (Figure 6.5).

Figure 6.5 Shell Tool Post Linker Option



Target Name: Debug
Linker: EPPC Linux GNU Linker
Pre-linker: None
Post-linker: Shell Tool Post Linker
Output Directory: {Project}bin/
☐ Save project entries using relative paths

3. Create a shell script (*.sh*) file.
4. Add an entry for the shell script (*.sh*) file in the **File Mappings** settings panel.
   a. Open the **File Mappings** panel.
   b. In the **Mapping Info** section, click **Choose** to navigate to the shell script file location and select it.

      When you select the shell script, the *.sh* extension is automatically added to the **Extension** text box in the **File Mappings** panel.
   c. Select **Shell Tool** from the **Compiler** list box.
   d. Click **Save** to save the settings and close the **File Mappings** settings panel.
5. Build the project.
   a. Select **Project > Make**. The project gets built and a log window appears which displays the contents of the shell script file.

7 Target Platform-Specific Features

This chapter describes the debugging features that are unique to programming for Linux on a PowerPC™ -based target platform. The CodeWarrior™ Development Studio for PowerPC ISA, Linux® Application/Platform Edition v2.0 allows you to debug the Linux kernel and the kernel modules on PowerPC®-based target platforms. This product also allows the PowerPC bareboard application developers to develop applications using the GNU bareboard tools. This chapter describes the hardware debug agents and devices supported by the CodeWarrior™ Development Studio for PowerPC ISA, Linux® Application/Platform Edition v2.0 for debugging the Linux kernel on a PowerPC-based target platform.

This chapter also describes the steps to configure a remote connection using these debug agents. Additionally, this chapter describes the target settings panels that are unique to CodeWarrior™ Development Studio for PowerPC ISA, Linux® Application/Platform Edition v2.0. Finally, the chapter lists the location where you can find the target platform-specific cross compiler tools, MetroTRK project file and binaries, sample projects for debugging purpose, and debug/memory initialization file commands.

This chapter has these topics:

- Supported Target Processors
- Supported Debug Agents
- Supported Remote Connections
- Remote Connections for Kernel-level Debugging
- Target-Platform-Specific Debugging Features
- Target Platform-Specific Target Settings Panels
- Target Platform-Specific Information
- Debug Initialization Files
- Memory Configuration Files
- Using Hardware Tools

Supported Target Processors

This section describes the PowerPC™ -based target processors supported by CodeWarrior™ Development Studio for PowerPC ISA, Linux® Application/Platform Edition v2.0.

Table 7.1 lists the supported target processors and their manufacturers.

Table 7.1 Supported Target Processors

Manufacturer	Processors
Motorola®	PowerQUICC I™ - MPC885, MPC880, MPC875, MPC870, MPC866, MPC862, MPC859T, MPC852T
	PowerQUICC II™ - HIP7 - MPC8272, MPC8271, MPC8248, MPC8247, MPC8280, MPC8275, MPC8270
	PowerQUICC II - MPC8266, MPC8265, MPC8264, MPC8250, MPC8255, MPC8260
	PowerQUICC III™ - MPC8560, MPC8540
	Integrated Host Processors - MPC8245, MPC8241, MPC8240
	MobileGT Processor - MPC5200

Supported Debug Agents

A debug agent performs the actions requested by the debugger, such as:

- setting breakpoints
- reading from and writing to memory
- reading and writing registers

The debug agent is not the program that you are debugging or the debugger itself. It is a software or hardware that processes commands and responses exchanged between the debugger and the target platform.

The debug agents supported by the CodeWarrior™ Development Studio for PowerPC ISA, Linux® Application/Platform Edition v2.0 are:

- Metrowerks Target Resident Kernel (MetroTRK) protocol from Metrowerks®
- PowerTAP® PRO JTAG hardware JTAG debug device from Metrowerks.

- PowerTAP PRO DPI hardware JTAG debug device from Metrowerks.
- PowerTAP Command Converter Server (CCS) protocol from Metrowerks.

Supported Remote Connections

The CodeWarrior™ debugger can connect to a PowerPC-based target platform using one of the following kinds of remote debugging connections:

- Metrowerks Target Resident Kernel (MetroTRK)—Used for application-level debugging only.
- EPPC PowerTAP PRO JTAG—Used for kernel, kernel module, and bareboard debugging only. For steps on how to configure this connection, see "Configuring a PowerTAP PRO JTAG Remote Connection" on page 128.
- EPPC PowerTAP PRO DPI—Used for kernel, kernel module, and bareboard debugging only. For steps on how to configure this connection, see "Configuring a PowerTAP PRO DPI Remote Connection" on page 130.
- EPPC CCS Protocol Plug-in—Used for kernel, kernel module, and bareboard debugging only. This remote connection is similar to the EPPC PowerTAP PRO JTAG connection.

NOTE For more information on MetroTRK and how to configure a MetroTRK remote connection, see "Using MetroTRK" on page 56.

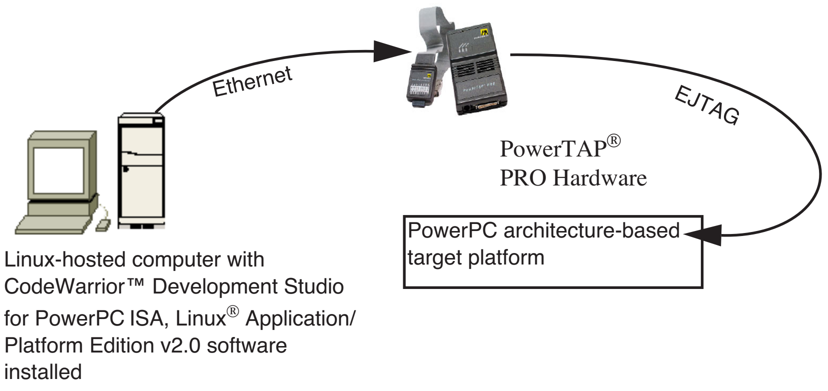
PowerTAP PRO Connection

The PowerTAP® PRO hardware JTAG debug device enables software developers to:

- control and observe program execution on the target platform
- download program at high-speed to the target platform.

Figure 7.1 depicts graphically how the CodeWarrior™ Development Studio for PowerPC ISA, Linux® Application/Platform Edition v2.0 software communicates with a PowerPC target platform using the PowerTAP PRO hardware debug device.

Figure 7.1 CodeWarrior IDE, PowerTAP Pro, andd Target Platform Communication


Ethernet
EJTAG
PowerTAP® PRO Hardware
PowerPC architecture-based target platform
Linux-hosted computer with CodeWarrior™ Development Studio for PowerPC ISA, Linux® Application/ Platform Edition v2.0 software installed

For more details on PowerTAP® PRO device, see information provided at the following URL:

`http://www.metrowerks.com`

Remote Connections for Kernel-level Debugging

The CodeWarrior™ IDE can connect the debugger with the debug image on a remote target. The default remote connections for kernel-level debugging included with the CodeWarrior™ Development Studio for PowerPC ISA, Linux® Application/Platform Edition v2.0 are:

- PowerTAP PRO JTAG
- PowerTAP PRO DPI
- PowerTAP PRO CCS

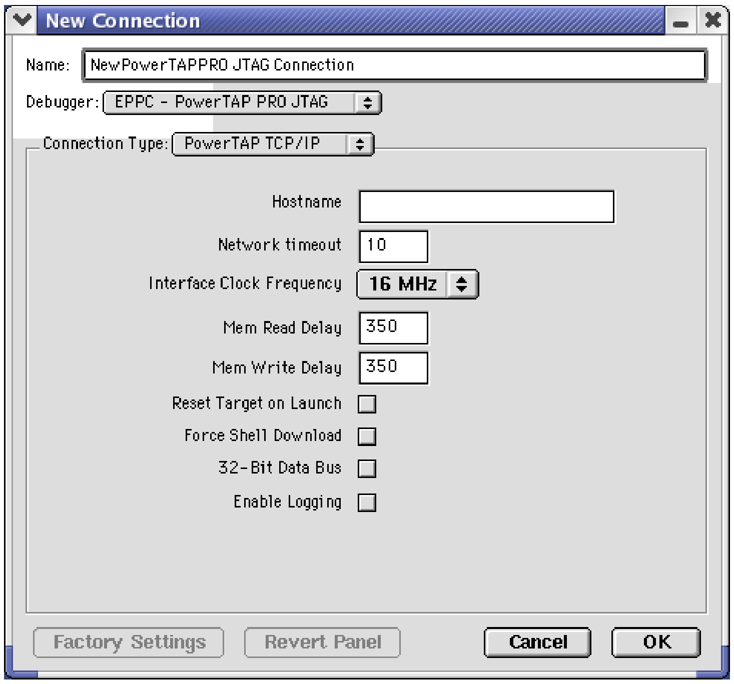
The sections that follow describe the steps to configure these remote connections.

Configuring a PowerTAP PRO JTAG Remote Connection

To configure a PowerTAP PRO JTAG remote connection:

1. Display the **Remote Connections** panel.
   a. Select **Edit > Preferences**. The **IDE Preferences** window appears.
   b. Select **Remote Connections** from the **IDE Preference Panels** list to display the **Remote Connections** panel**.**
2. Add a new remote connection.
   a. Click **Add**. The **New Connection** dialog box appears.
   b. In the **Name** text box, type a name that will identify this connection, such as *NewPowerTAPPRO JTAG Connection*.
   c. Select the remote connection type, that is, **EPPC - PowerTAP PRO JTAG** from the **Debugger** list box. The **New Connection** dialog box display changes to reflect options related to the PowerTAP PRO JTAG connection.

Figure 7.2 New Connection Dialog Box - PowerTAP PRO JTAG Preferences



Name: NewPowerTAPPRO JTAG Connection
Debugger: EPPC - PowerTAP PRO JTAG
Connection Type: PowerTAP TCP/IP
Hostname
Network timeout 10
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 ☐

Type the host name or IP address that you assigned to the PowerTAP PRO device during the setup in the **Hostname** text box. For steps on how to set up the PowerTAP PRO device, refer the PowerTAP PRO documentation that came with your PowerTAP PRO device.

d. Use the **Interface Clock Frequency** list box to select the clock frequency for the PowerTAP PRO device. This is the speed at which the data is read and written to the target platform. Higher the frequency, faster will be the speed. Differences in target platform design may affect the maximum reliable device clock rate. The recommended clock frequency is 16 MHz.

e. Use the **Mem Read Delay** text box to specify the number of additional processor cycles inserted as a delay for completion of read memory operations. The range of values that can be entered is 0-65024 cycles. Use the default delay of 350 cycles.

f. Use the **Mem Write Delay** text box to specify the number of additional processor cycles inserted as a delay for memory write operations. The range of values that can be entered is 0-65024 cycles. Use the default delay of 350 cycles.

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

h. Check the **Force shell download on connect** checkbox if you wish to reload the PowerTAP PRO shell at each debugger connection.

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

j. Check the **Enable Logging** checkbox to have the CodeWarrior IDE display a log of all the debug translations. If this checkbox is checked, a protocol logging window appears when the debugger connects to the target platform. If you set the `AMCTAP_LOG_FILE` environment variable, the IDE directs log messages to the specified file.

k. Click **Save**.

   The CodeWarrior IDE saves the connection information that you specified and closes the **New Connection** dialog box. The debugger uses these settings when you select this connection in the **Remote Debugging** settings panel.

NOTE The **EPPC CCS Protocol Plugin** and the **EPPC PowerTAP PRO JTAG** remote connection panels are similar.

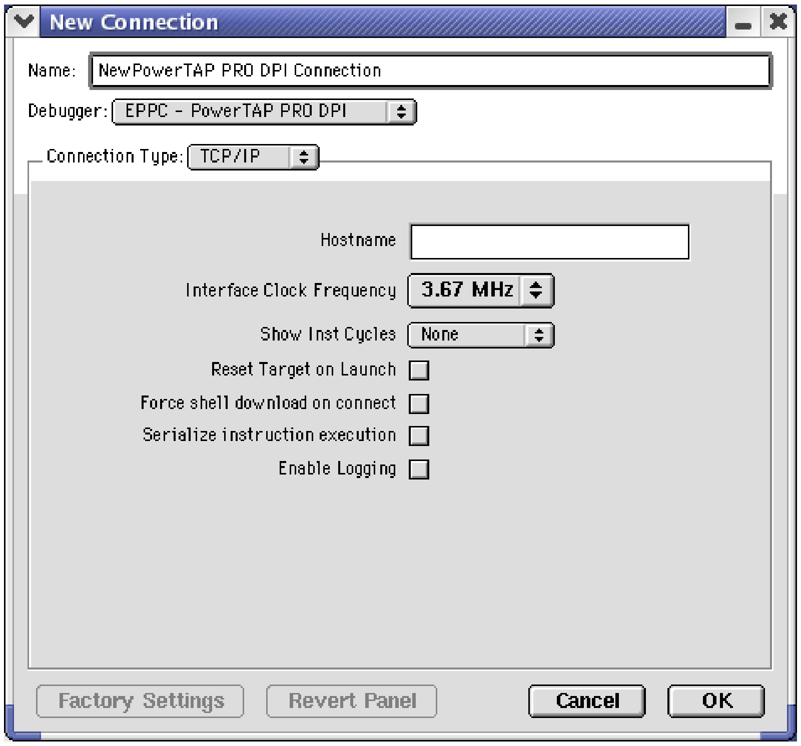
Configuring a PowerTAP PRO DPI Remote Connection

To configure a PowerTAP PRO DPI remote connection:

1. Display the **Remote Connections** panel.
2. Add a new remote connection.
   a. Click **Add**. The **New Connection** dialog box appears.
   b. In the **Name** text field, type a name that will identify this connection, such as *NewPowerTAP PRO DPI Connection*.

c. Select **EPPC - PowerTAP PRO DPI** from the **Debugger** list box,. The **New Connection** dialog box display changes to reflect options related to an PowerTAP PRO connection (Figure 7.3).

Figure 7.3 New Connection Dialog Box - PowerTAP PRO DPI Preferences



New Connection

Name: NewPowerTAP PRO DPI Connection

Debugger: EPPC - PowerTAP PRO DPI

Connection Type: TCP/IP

Hostname

Interface Clock Frequency: 3.67 MHz

Show Inst Cycles: None

Reset Target on Launch ☐

Force shell download on connect ☐

Serialize instruction execution ☐

Enable Logging ☐

d. Type the host name or IP address that you assigned to the PowerTAP PRO device during the setup in the **Hostname** text box.

e. Use the **Interface Clock Frequency** list box to select the clock frequency for the PowerTAP PRO device. This is the speed at which the data is read and written to the target platform. Higher the frequency, faster will be the speed. Differences in target platform design may affect the maximum reliable device clock rate. The recommended setting is **3.65** MHz.

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

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

h. Check the **Force shell download on connect** checkbox if you wish to reload the PowerTAP PRO shell at each debugger connection.

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

j. Check the **Enable Logging** checkbox to have the CodeWarrior IDE display a log of all the debug translations. If this checkbox is checked, a protocol logging window appears when the debugger connects to the target platform.

k. Click **Save**.

The CodeWarrior IDE saves the connection information that you specified and closes the **New Connection** dialog box. The debugger uses these settings when you select this connection in the **Remote Debugging** settings panel.

Target-Platform-Specific Debugging Features

This section describes the debugging features that are unique to CodeWarrior™ Development Studio for PowerPC ISA, Linux® Application/Platform Edition v2.0. These debugging features include:

- Debugging u-boot
- Kernel Debugging
- Kernel Module Debugging
- Boa Server Application Debugging

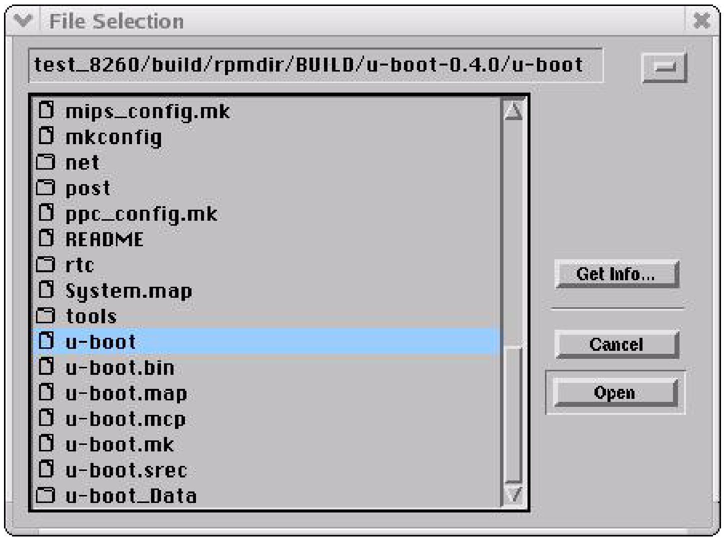
Debugging u-boot

NOTE Before you start debugging u-boot on your target platform, you must install the board support package (BSP) for the target platform. Also, you need to recompile the u-boot with `-g2` flag for debug information using the Metrowerks® Platform Creation Suite (PCS).

The steps to debug u-boot using the CodeWarrior IDE are:

1. Flash program the `u-boot.srec` file from BSP.
2. Select **File > Open** to open the u-boot binary (*u-boot*) in the CodeWarrior IDE window. A warning message box appears asking you whether you want to continue. Click **OK**. A file mapping dialog box (Figure 7.4) appears asking you to map the required source file with a matching file in your project folder.

Figure 7.4 u-boot Source Files Mapping

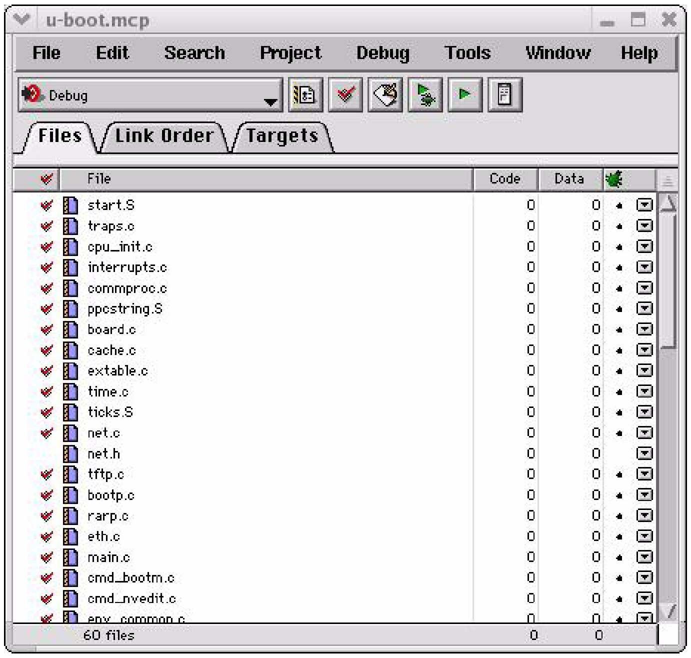


```
test_8260/build/rpmdir/BUILD/u-boot-0.4.0/u-boot
mips_config.mk
mkconfig
net
post
ppc_config.mk
README
rtc
System.map
tools
u-boot
u-boot.bin
u-boot.map
u-boot.mcp
u-boot.mk
u-boot.srec
u-boot_Data
```

The CodeWarrior IDE creates a dummy project with the filename of the binary, that is `u-boot.mcp`.

A progress bar appears showing the status of the source files imported into the project. After the import process is complete, all the u-boot source files appear in your project window (Figure 7.5).

Figure 7.5 u-boot Project Window



	File	Code	Data
✓	start.S	0	0
✓	traps.c	0	0
✓	cpu_init.c	0	0
✓	interrupts.c	0	0
✓	commproc.c	0	0
✓	ppcstring.S	0	0
✓	board.c	0	0
✓	cache.c	0	0
✓	extable.c	0	0
✓	time.c	0	0
✓	ticks.S	0	0
✓	net.c	0	0
	net.h	0	0
✓	tftp.c	0	0
✓	bootp.c	0	0
✓	rarp.c	0	0
✓	eth.c	0	0
✓	main.c	0	0
✓	cmd_bootm.c	0	0
✓	cmd_nvedit.c	0	0
✓	env_common.c	0	0
	60 files	0	0

3. Click **Remote Debugging** in the **Target Settings Panels** list. The **Remote Debugging** settings panel appears.

   a. Make sure that the correct remote connection name (*PowerTAP PRO DPI*, *PowerTAP PRO JTAG*, or *PowerTAP PRO* CCS-based) is selected in the **Connection** list box of the **Remote Debugging** settings panel.

NOTE Make sure that you have specified the IP address of the PowerTAP PRO hardware debug device in the remote connection panel.

   b. Click **Save** to save the settings.

4. Click **EPPC Debugger Settings** in the **Target Settings Panels** list. The **EPPC Debugger Settings** panel appears.

   a. Select the PowerPC processor architecture that you are targeting from the **Target Processor** list box.

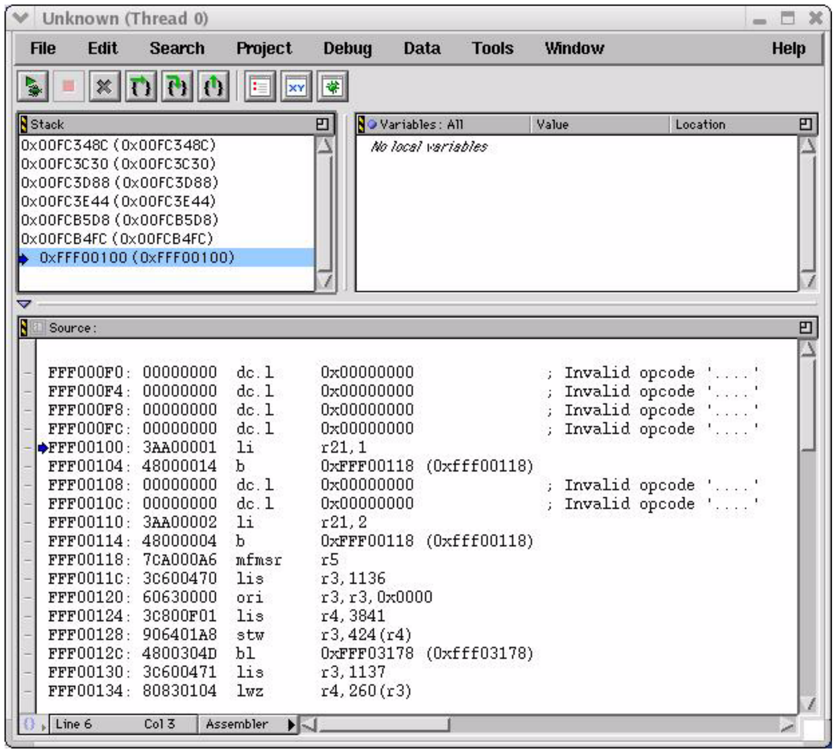
   b. Select **Bareboard** from the **Target OS** list box.

   c. Check the **Use Target Initialization File** checkbox and click the **Browse** button to select the appropriate target initialization file.

NOTE Check the **Reset Target on Launch** checkbox in the PowerTAP PRO-based remote connection panel.

5. Select **Debug > Connect**. The debugger connects to the target platform and displays a thread window (Figure 7.6).

Figure 7.6 u-boot Thread Window



```
0x00FC348C (0x00FC348C)
0x00FC3C30 (0x00FC3C30)
0x00FC3D88 (0x00FC3D88)
0x00FC3E44 (0x00FC3E44)
0x00FCB5D8 (0x00FCB5D8)
0x00FCB4FC (0x00FCB4FC)
0xFFF00100 (0xFFF00100)
```

```
FFF000F0: 00000000  dc.l    0x00000000              ; Invalid opcode '....'
FFF000F4: 00000000  dc.l    0x00000000              ; Invalid opcode '....'
FFF000F8: 00000000  dc.l    0x00000000              ; Invalid opcode '....'
FFF000FC: 00000000  dc.l    0x00000000              ; Invalid opcode '....'
FFF00100: 3AA00001  li      r21,1
FFF00104: 48000014  b       0xFFF00118 (0xfff00118)
FFF00108: 00000000  dc.l    0x00000000              ; Invalid opcode '....'
FFF0010C: 00000000  dc.l    0x00000000              ; Invalid opcode '....'
FFF00110: 3AA00002  li      r21,2
FFF00114: 48000004  b       0xFFF00118 (0xfff00118)
FFF00118: 7CA000A6  mfmsr   r5
FFF0011C: 3C600470  lis     r3,1136
FFF00120: 60630000  ori     r3,r3,0x0000
FFF00124: 3C800F01  lis     r4,3841
FFF00128: 906401A8  stw     r3,424(r4)
FFF0012C: 4800304D  bl      0xFFF03178 (0xfff03178)
FFF00130: 3C600471  lis     r3,1137
FFF00134: 80830104  lwz     r4,260(r3)
```

You can now debug the u-boot application like any other application.

NOTE To debug the relocated section of the u-boot in RAM, you need to specify the RAM address of the relocated code in the **Alternate Load Address** text box in the **Debugger PIC Settings** panel.
To find out the RAM address, you need to build u-boot (bootloader) with DEBUG option and look at the runtime messages. Note that, for each debug session, you can debug on source only the RAM sections or the ROM sections depending on whether the Debugger PIC settings is enabled or not. Because

when checking the alternate load address, the CodeWarrior IDE assumes that all sections from the `elf` file are relocated.

Kernel Debugging

The CodeWarrior IDE allows you to debug the Linux kernel on your host computer running Linux OS.

Linux Kernel - An Introduction

The Linux operating system (OS) operates in two modes—*kernel mode* (kernel space) and *user mode* (user space). The kernel works at the top level where it performs the function of a mediator for all the currently running programs and the hardware. The kernel manages the memory for all the programs (processes) currently running, and ensures that each program gets a fair share of the available memory. In addition, the kernel also provides a portable interface for programs to talk to the hardware.

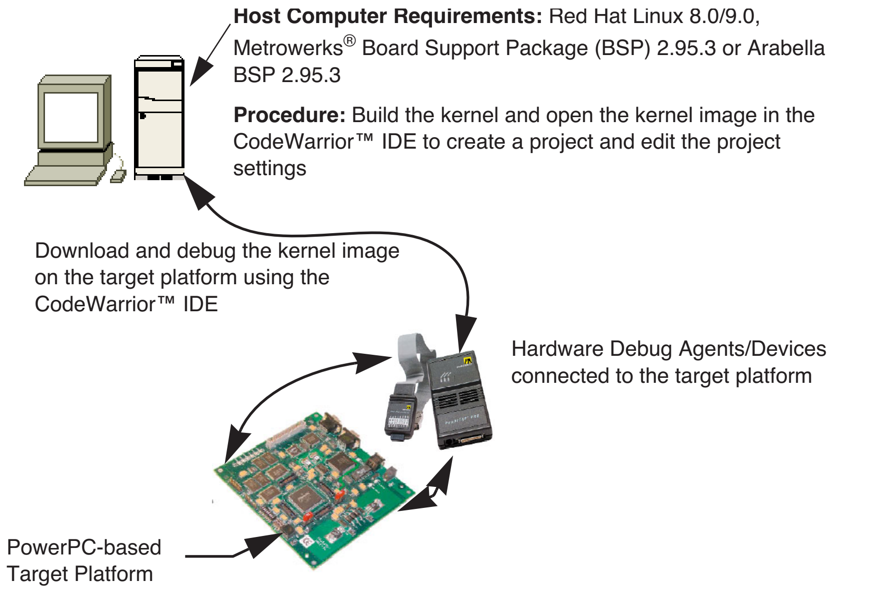
The User mode (user space) works at the lowest level or the application level where you do not have the permission to directly access the memory or the hardware. You can access the hardware resources through the system calls.

Kernel Debugging Using CodeWarrior IDE - The Prerequisites

Before you can debug the kernel using the CodeWarrior™ IDE, you must ensure that a remote connection is already created for the hardware debug agent (PowerTAP PRO JTAG®, PowerTAP PRO DPI, or PowerTAP PRO CCS) that you are using for connecting to the target platform.

Figure 7.7 graphically illustrates the setup environment used by CodeWarrior™ Development Studio for PowerPC ISA, Linux® Application/Platform Edition v2.0 to debug the kernel on a PowerPC-based target platform.

Figure 7.7 Setup for Kernel Debugging Using the CodeWarrior IDE


Host Computer Requirements: Red Hat Linux 8.0/9.0, Metrowerks® Board Support Package (BSP) 2.95.3 or Arabella BSP 2.95.3
Procedure: Build the kernel and open the kernel image in the CodeWarrior™ IDE to create a project and edit the project settings
Download and debug the kernel image on the target platform using the CodeWarrior™ IDE
Hardware Debug Agents/Devices connected to the target platform
PowerPC-based Target Platform

Kernel Debugging - the Methods

There are three methods for debugging the Linux kernel:

- Using CodeWarrior IDE Target Initialization File - Download and start the kernel using the CodeWarrior IDE based on the initialization done by CodeWarrior IDE. This method depends only on the initialization file and does not require a bootloader to be present in the flash at the reset address.
- Using u-boot Initialization - Download and start the kernel using CodeWarrior IDE based on the initialization done by the u-boot on the target platform.
- Attaching to the Running Kernel - Start the kernel using any of the above methods and attach to the running kernel.

Using CodeWarrior IDE Target Initialization File

To debug the kernel using this method, you need to perform the following steps:

- Build the Kernel
- Create a CodeWarrior Project for the Kernel
- Set Up the Kernel Project for Debugging

- Download and Boot the Kernel

Build the Kernel

The first step is to build the kernel using the CodeWarrior patch available in your CodeWarrior installation directory. When you build the kernel, the kernel image (*vmlinux*) is generated and placed in the base directory where the kernel source files are located on your computer. Usually, when you build the kernel, two kernel images are generated—a compressed image and a full image. You can find the full kernel image in the following folder of the base directory on your computer:

`{Linux_Install_Dir}\My_dir`

where, `My_dir` is the directory where your kernel sources reside.

You can find the compressed kernel image at the following location on your computer:

`{Linux_Install_Dir}\My_dir\arch\ppc\boot\compressed`

Create a CodeWarrior Project for the Kernel

The next step is to create a project for the kernel in your CodeWarrior™ IDE.

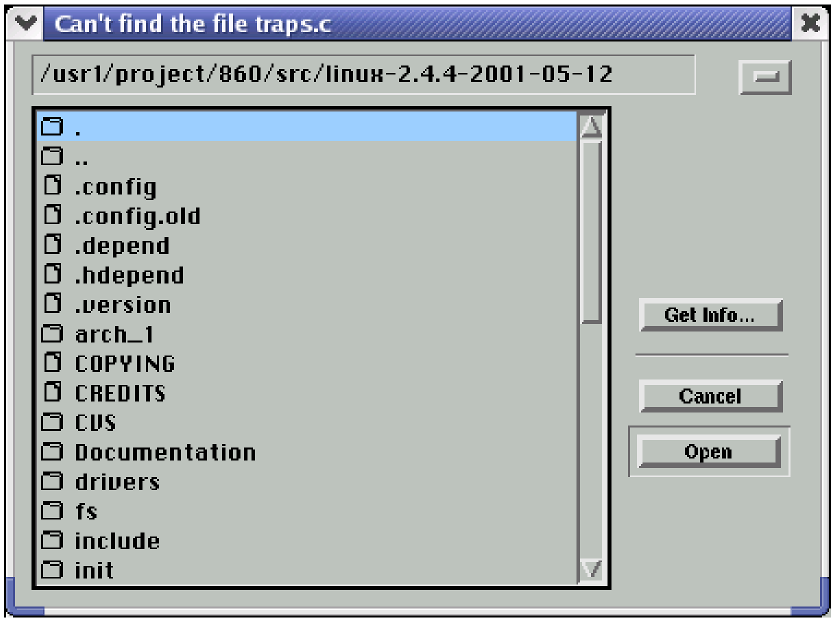
The steps to create a kernel project are:

1. Select **File > Open** to open the uncompressed kernel image file (*vmlinux*) built with full debug information in the CodeWarrior™ IDE window. A warning message box appears asking you whether you want to continue. Click **OK**.

   The CodeWarrior IDE creates a dummy project with the filename of the image file, that is `vmlinux.mcp`. A file mapping dialog box (Figure 7.8) appears asking you to map the required source file with a matching file in your project folder on your host computer.

NOTE The CodeWarrior IDE requires an uncompressed kernel image having the debug symbolics information to create a project.

Figure 7.8 Kernel Source Mapping



NOTE The title bar of the file mapping dialog box displays the build path of the file sources that CodeWarrior™ IDE is trying to locate on your computer.

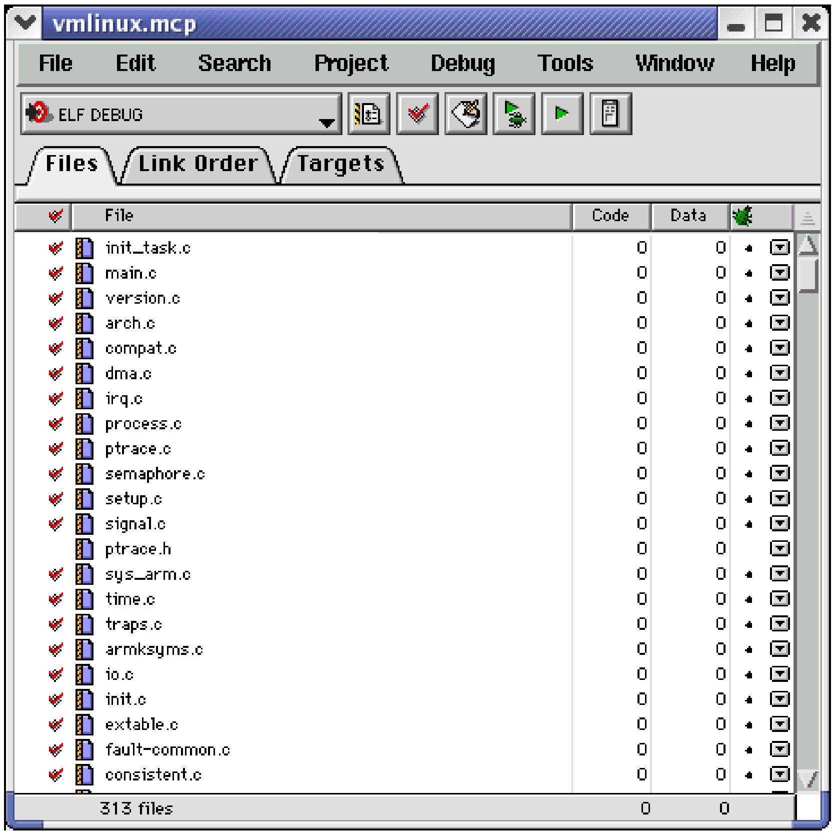
2. Type or browse to the location where you have the source files for this project located in your project directory on your host computer.
3. Click **Open**. After you specify the relative path for the first source file, the same relative path is applied to the rest of the source files.

NOTE After you select a file, the CodeWarrior IDE sets up a mapping between the build location and the current source location on your computer. The mapping is automatically stored in the Source Folder Mapping settings panel and you may edit it if either source folder (build or current) location changes. For example, if the directory was mapped to `d:` when project was created, but is now mapped to `e:`, you can specify the new directory settings.

A progress bar appears showing the status of the source files imported into the project.

After the import process is complete, all the kernel source files appear in your project window (Figure 7.9).

Figure 7.9 Linux Kernel Project



✓	File	Code	Data	
✓	init_task.c	0	0	•
✓	main.c	0	0	•
✓	version.c	0	0	•
✓	arch.c	0	0	•
✓	compat.c	0	0	•
✓	dma.c	0	0	•
✓	irq.c	0	0	•
✓	process.c	0	0	•
✓	ptrace.c	0	0	•
✓	semaphore.c	0	0	•
✓	setup.c	0	0	•
✓	signal.c	0	0	•
	ptrace.h	0	0	
✓	sys_arm.c	0	0	•
✓	time.c	0	0	•
✓	traps.c	0	0	•
✓	armksyms.c	0	0	•
✓	io.c	0	0	•
✓	init.c	0	0	•
✓	extable.c	0	0	•
✓	fault-common.c	0	0	•
✓	consistent.c	0	0	•
	313 files	0	0	

NOTE You cannot re-build the kernel in the CodeWarrior™ IDE. The kernel can only be re-built on your Linux computer where the kernel source files are located. The ELF drop feature creates the new kernel project with *Build - Never* settings.

After you have created a kernel project in the CodeWarrior™ IDE, the next step is to set up the project for debugging. There are some settings that you need to specify for debugging the kernel.

Set Up the Kernel Project for Debugging

NOTE For quick start steps on how to set up your kernel project for debugging on a PowerQUICC III™ MPC8560 ADS target platform, refer the kernel debug quick start guide.

1. Set a program entry point in the kernel code.
   a. Select **Edit >** ***Target*** **Settings** (where ***Target*** is the name of the current build target displayed in the project window) to open the ***Target*** **Settings** window.
   b. Select **Debugger Settings** from the **Target Settings Panels** list. The **Debugger Settings** panel appears.
   c. Check the **Stop on application launch** checkbox to activate its options.
   d. Click the **Program entry point** radio button to instruct the debugger to stop program execution upon entering the program.

NOTE You can also set your own program entry point where you want the debugger to stop the program execution. For example, you can specify the debugger to stop program execution on entering the `setup_arch()` function in the kernel code. For doing this, click the **user specified** option and type `setup_arch()` in the text box.

   e. Click **Save** to save the settings.
2. Specify the remote download options for the kernel image.
   a. Click **Remote Debugging** in the **Target Settings Panels** list. The **Remote Debugging** settings panel appears.
   b. Make sure that the correct remote connection name (*PowerTAP PRO JTAG, PowerTAP PRO DPI, or PowerTAP PRO* CCS-based) is selected in the **Connection** list box of the **Remote Debugging** settings panel.

NOTE If you wish to modify the remote connection preferences, select a connection name from the **Connection** list box and click **Edit Connection**.

NOTE You do not have to specify the remote target path for downloading the kernel because the kernel is downloaded to target platform RAM.

   c. Click **Download OS** checkbox to activate the compressed kernel image download options.

NOTE If you change from a MetroTRK-based remote connection to *PowerTAP PRO JTAG, PowerTAP PRO DPI, or PowerTAP PRO* CCS-based remote connection, you must click Save to view the **Download OS** checkbox.

---

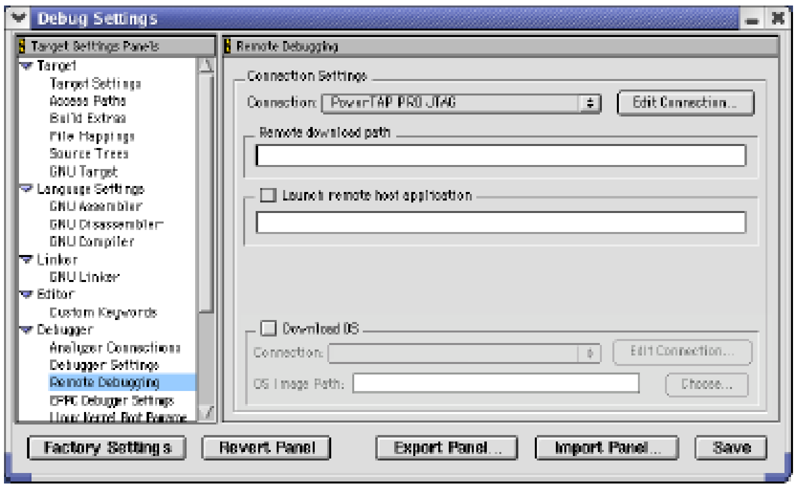
d. Select the correct remote connection for downloading the compressed kernel image to the target platform from the **Connection** list box.

NOTE Make sure that the **Connection** specified in the **Download OS** section and the **Communication Settings** section are same.

---

e. Enter or click **Choose** to specify the host-side path of the compressed kernel image to be downloaded to the target platform (Figure 7.10).

Figure 7.10 Specifying Kernel Download Options



Connection: PowerTAP PRO JTAG

f. Click **Save** to save the settings.

NOTE When you check the **Download OS** checkbox, make sure that you also check the **Set residual data after decompress** checkbox in the Linux Kernel Residual Data settings panel. For details, see “Linux Kernel Residual Data” on page 173.

---

3. Specify a target processor, operating system, and an initialization file for the debugger.

a. Click **EPPC Debugger Settings** in the **Target Settings Panels** list. The **EPPC Debugger Settings** panel appears.

NOTE The **EPPC Debugger Settings** panel is displayed in the **Target Settings Panels** list only when you select an *PowerTAP PRO JTAG, PowerTAP PRO DPI,* or *PowerTAP PRO CCS* -based remote connection from the **Remote Connections** settings panel.
An *xml* file with pre-configured settings for this panel is provided for the target platforms/BSP's supported by CodeWarrior™ Development Studio for PowerPC ISA, Linux® Application/Platform Edition v2.0. Import the XML file settings for the target platform/BSP you are using by clicking the **Import Panel** button at the bottom of this panel and selecting the desired XML file from this location on your CodeWarrior installation directory:
`CodeWarriorIDE/CodeWarrior/PowerPC_EABI_Tools/`
`KernelDebug_Settings/<target_platform_name>`
where `<target_platform_name>` can be MPC8560ADS, MPC8260ADS,MPC5200Lite, PC8280FADS-ZU, and MPC860FADS.

b. Select the PowerPC processor architecture that you are targeting from the **Target Processor** list box.

c. Select **Linux** from the **Target OS** list box.

NOTE Ensure that **Bareboard** is selected in the **Target OS** list box when you are performing application-level debugging on the target platform. Otherwise, the debugger will not be able to debug your applications.

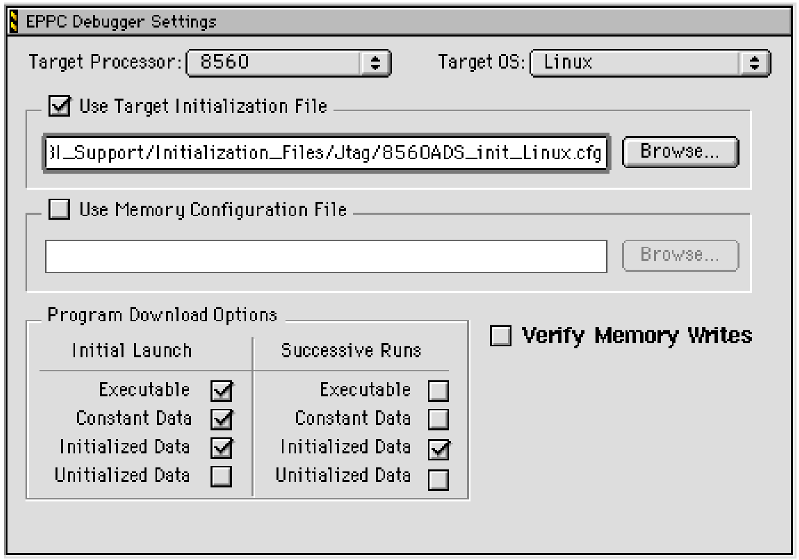
d. Check the **Use Target Initialization File** checkbox and specify the filename in the text box, if you want the debugger to execute an initialization file before downloading the kernel image to the platform target (Figure 7.11).

A set of standard initialization files is available in the following directory of your CodeWarrior installation directory:

```
CodeWarriorIDE/CodeWarrior/PowerPC_EABI_Support/
Initialization_Files/
```

NOTE For more information, see “Debug Initialization Files” on page 179 and “Memory Configuration Files” on page 187.

Figure 7.11 EPPC Debugger Settings - Target Processor and OS Preferences



Target Processor: 8560
Target OS: Linux

[✓] Use Target Initialization File

3I_Support/Initialization_Files/Jtag/8560ADS_init_Linux.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

TIP You must customize the supplied initialization file to debug the Linux kernel on your custom target platform/board support package. For more information, see “Debugging Kernel Using Custom Target Platform/BSP” on page 155.

e. Check the **Executable**, **Constant Data**, **Initialized Data**, and **Uninitialized Data** checkboxes under the **Program Download Options** section to specify what portions of the project to download on the initial launch of debugger and successive launches. For example, you can download the entire executable or only certain sections of the program to the target platform.

For a complete description of the **EPPC Debugger Settings** panel, see “EPPC Debugger Settings” on page 165.

f. Click **Save** to save the settings.

4. Specify the kernel boot parameters and the RAM disk parameters.

a. Click **Linux Kernel Boot Parameters** in the **Target Settings Panels** list. The **Linux Kernel Boot Parameters** settings panel appears (Figure 7.12).

NOTE The **Linux Kernel Boot Parameters** settings panel is displayed in the **Target Settings Panels** list only when you select a *PowerTAP PRO JTAG, PowerTAP PRO DPI,* or *PowerTAP PRO CCS* -based remote connection from the **Remote Connections** settings panel.

It is recommended that you use the XML file which contains pre-configured settings for this panel for your target platform/BSP.

b. Check the **Enable Command Line Settings** checkbox to set the command line and base address options.

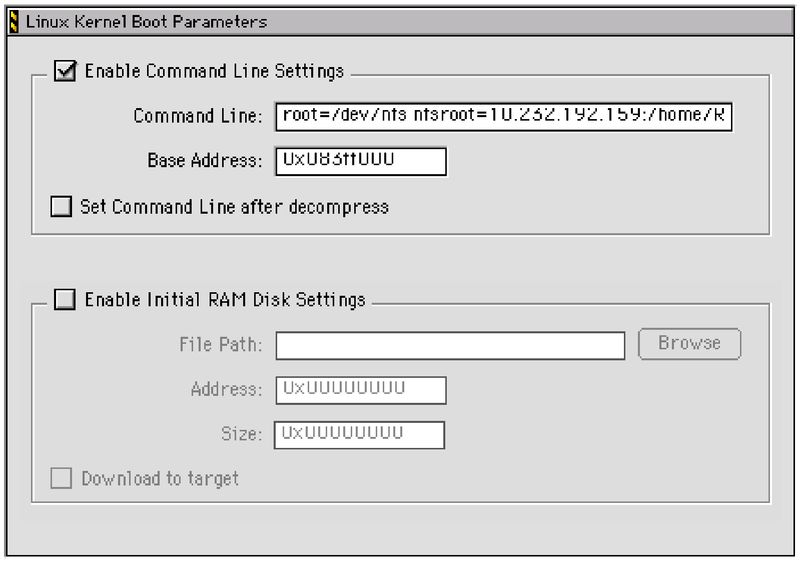
c. You can modify the command line parameters that you have specified during building the kernel by entering a new set of parameters in the **Command Line** text box. These parameters are passed to the kernel during the booting of kernel.

NOTE The **Command Line** text box does not display the values that were set while building the kernel. You can view these values in your Terminal window.

d. Enter the address in the **Base Address** text box where you want the command line parameters to be written in the memory. For examples, *0x100000* for PowerQUICC II MPC8260 target platform and *0x400000* for PowerQUICC III target platforms.

e. Check the **Enable Initial RAM Disk Settings** checkbox to enable the initial RAM Disk support in the kernel (Figure 7.12).

Figure 7.12 Linux Kernel Boot Parameters - Command Line and initrd Settings



Linux Kernel Boot Parameters

[✓] Enable Command Line Settings

Command Line: root=/dev/nfs nfsroot=10.232.192.159:/home/R

Base Address: 0x083ff000

[ ] Set Command Line after decompress

[ ] Enable Initial RAM Disk Settings

File Path:

Address: 0x00000000

Size: 0x00000000

[ ] Download to target

f. Enter or browse for the location on the host computer from where the debugger should pick up the initial RAM disk (`initrd`) file. After you navigate to the directory where the RAM disk file (`.gz`) exists, you need to manually enter the file name in the **File Path** text box.

Sample RAM Disk parameters:

```
"root=/dev/ram rw"
```

Sample NFS parameters:

```
"root=/dev/nfs ip=10.171.77.26
nfsaddr=10.171.77.26:10.171.77.21 nfsroot=/tftpboot/
10.171.77.26"
```

```
"root=/dev/nfs rw nfsroot=10.171.77.21:/tftpboot/
10.171.77.26
ip=10.171.77.26:10.171.77.21:10.171.77.254:255.255.255.0:82
80x:eth0:off"
```

where,

`10.171.77.21` is the IP address of the NFS server and `10.171.77.26` is the IP address of the target platform.

`"/tftpboot/10.171.77.26"` is a directory on the host computer where the target platform file system is located.

`"8280x"` is the host name.

Sample flash parameters:

`root=/dev/mtdblock0` or `root=/dev/mtdblock2` (depending on your configuration)

For more information on NFS root, see `Documentation/nfsroot.txt`.

For more information on intial RAM Disk, see `Documentation/ramdisk.txt` file in your kernel sources.

g. Enter the address in the **Address** text box where you want the RAM disk to be written in the memory.

CAUTION Make sure that the address you specify does not overwrite the kernel in memory with the RAM disk.

h. Enter the size of the RAM Disk file in the **Size** text box.

NOTE To copy all the contents of the RAM disk file, enter `0` in the **Size** text box.

i. Check the **Download to target** checkbox to download the RAM Disk file to the target platform.

NOTE When you enable the RAM disk support, the debugger copies the initial RAM disk to the target platform only when the **Download to target** checkbox is checked.

j. Click **Save** to save the settings.

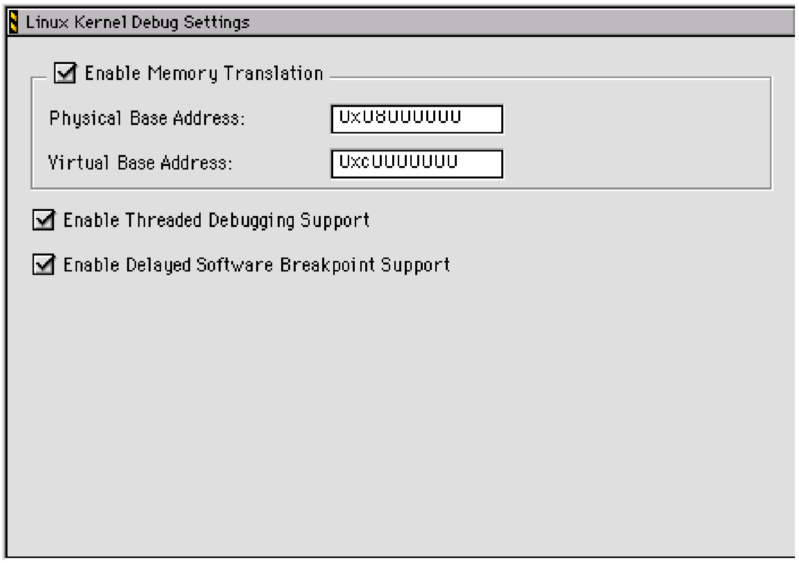
   For a complete description of the **Linux Kernel Boot Parameters** settings panel, see "Linux Kernel Boot Parameters" on page 170.

5. Specify the settings for debugging the kernel on the target platform.

   a. Click **Linux Kernel Debug Settings** in the **Target Settings Panels** list. The **Linux Kernel Debug Settings** panel (Figure 7.13) appears.

NOTE The **Linux Kernel Debug Settings** panel is displayed in the **Target Settings Panels** list only when you select a *PowerTAP PRO JTAG, PowerTAP PRO DPI,* or *PowerTAP PRO CCS* -based remote connection from the **Remote Connections** settings panel.
It is recommended that you use the XML file which contains pre-configured settings for this panel for your target platform/BSP.

Figure 7.13 Linux Kernel Debug Settings



Linux Kernel Debug Settings

✓ Enable Memory Translation

Physical Base Address: 0x08000000

Virtual Base Address: 0xc0000000

✓ Enable Threaded Debugging Support

✓ Enable Delayed Software Breakpoint Support

   b. Check the **Enable Memory Translation** checkbox to enable the memory translation. The debugger maps the physical base memory and virtual base memory.

c. Enter the physical base address for memory translation in the **Physical Base Address** text box.

d. Enter the virtual base address for memory translation in the **Virtual Base Address** text box.

e. Check the **Enable Threaded Debugging Support** checkbox, if not checked already to enable multithreading support in the debugger so that you can view and debug multiple kernel threads on the target platform. For more information, see "Kernel Thread Debugging" on page 155.

f. Check the **Enable Delayed Software Breakpoint Support** checkbox, if not checked already to delay the setting of software breakpoints till the MMU is enabled.

NOTE For a complete description of the Linux Kernel Debug Settings panel, see "Linux Kernel Debug Settings" on page 171.

g. Click **Save** to save the settings.

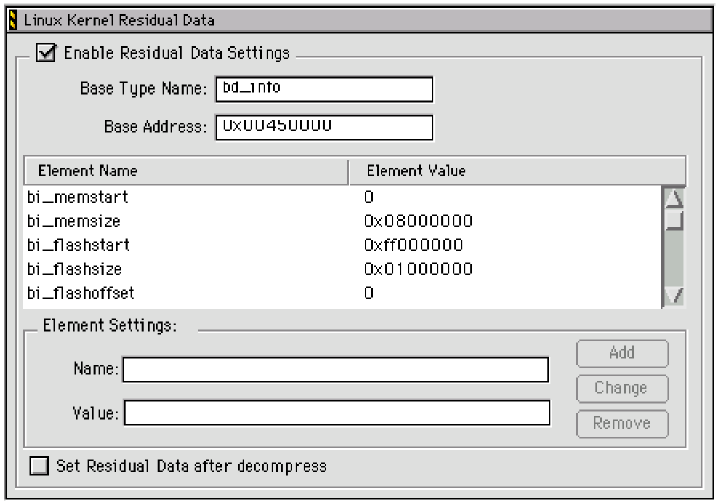
6. Set a structure for the Linux kernel in the memory.

WARNING! If you want to boot the kernel successfully using the CodeWarrior IDE, you must set the *bd_info* structure for the kernel in the memory. The kernel searches this structure to retrieve information on location of DRAM, SRAM, FLASH memory, IMMPR base, IP address to use, bus speed, and ethernet MAC addresses for the target platform ports.

a. Click **Linux Kernel Residual Data** in the **Target Settings Panels** list. The **Linux Kernel Residual Data** panel (Figure 7.14) appears.

NOTE The **Linux Kernel Residual Data** panel is displayed in the **Target Settings Panels** list only when you select a *PowerTAP PRO JTAG, PowerTAP PRO DPI,* or *PowerTAP PRO CCS* -based remote connection from the **Remote Connections** settings panel.
It is recommended that you use the XML file which contains pre-configured settings for this panel for your target platform/BSP.

Figure 7.14 Linux Kernel Residual Data


Linux Kernel Residual Data

☑ Enable Residual Data Settings

Base Type Name: bd_info

Base Address: 0x00450000

Element Name	Element Value
bi_memstart	0
bi_memsize	0x08000000
bi_flashstart	0xff000000
bi_flashsize	0x01000000
bi_flashoffset	0

Element Settings:

Name:

Value:

☐ Set Residual Data after decompress

b. Check the **Enable Residual Data Settings** checkbox to set a structure for the kernel in memory.

c. Enter the structure name in the **Base Type Name** text box. In this case, *bd_info*.

d. Enter the physical address in the **Base Address** text box, which will be the starting point of the structure.

e. Enter the structure members name and value in the **Element Settings** group.

f. Check the **Set Residual Data after decompress** checkbox to download the structure again after the kernel image is decompressed on the target platform. This option is necessary only if you have checked the **Download OS** checkbox in the **Remote Debugging** panel.

NOTE For a complete description of the Linux Kernel Residual Data panel, see "Linux Kernel Residual Data" on page 173.

g. Click **Save** to save the settings.

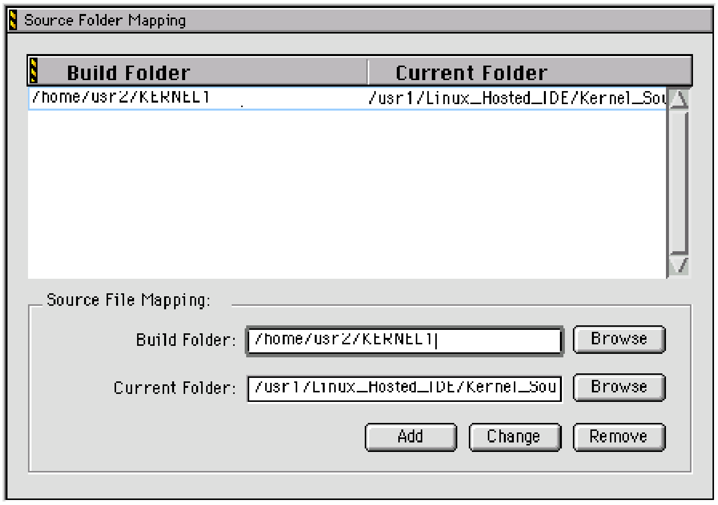
NOTE You can also set the residual data by typing *bdinfo* in the u-boot console. This displays the required parameters that you can use for filling the **Linux Kernel Residual Data** panel.

7. Verify the mapping of kernel sources on the Linux-hosted computer to your host computer.

a. Click **Source Folder Mapping** in the **Target Settings Panels** list. The **Source Folder Mapping** settings panel (Figure 7.15) appears.

If you have already mapped the kernel sources on the Linux-hosted computer to a folder on your host computer, the current mapping is displayed in the **Source Folder Mapping** settings panel. You may also edit the current settings.

Figure 7.15 Source Folder Mapping


Source Folder Mapping

Build Folder	Current Folder
/home/usr2/KERNEL1	/usr1/Linux_Hosted_IDE/Kernel_Sou

Source File Mapping:

Build Folder: /home/usr2/KERNEL1

Current Folder: /usr1/Linux_Hosted_IDE/Kernel_Sou

b. Click **Save** to save the settings.

c. Close the ***Target*** **Settings** window.

8. Configure the build settings for the kernel.

a. Select **Edit > Preferences** to open the **IDE Preferences** window.

b. Click **Build Settings** in the **IDE Preference Panels** list. The **Build Settings** panel appears.

c. Select **Never** in the **Build before running** list box in the **Settings** section.

d. Click **Save** to save the settings

e. Close the **IDE Preferences** window.

Download and Boot the Kernel

After you have specified the settings for debugging the kernel on the target platform, you can now download the kernel to the target platform and boot it. To do this:

1. Launch the debugger.

NOTE Before you download the kernel image to the target platform, make sure that you *switch off* and then *switch on* the target platform. Otherwise, the kernel image does not get downloaded to the target platform.

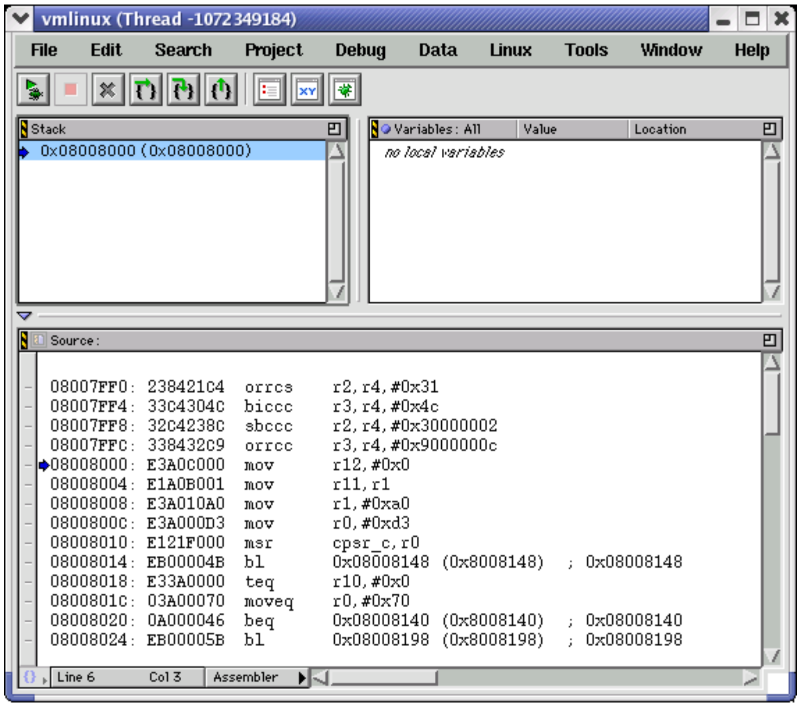
CAUTION Do not allow the kernel flashed on the target platform (if any) to boot from the bootloader. To stop the kernel flashed on the target platform from booting, press any special key on the keyboard, like the Spacebar key.

   a. Select **Project > Debug**.

      The CodeWarrior IDE launches the debugger and the kernel image (`vmlinux`) is downloaded to the target platform. The debugger then displays the debugger window (Figure 7.16).

NOTE When you download the kernel image to the target platform, two progress bars appear, one after the other, which display the progress of the kernel image download to the target platform and the name of the initialization file that you specified in the **EPPC Debugger Settings** panel.

Figure 7.16 Kernel Debugger Window



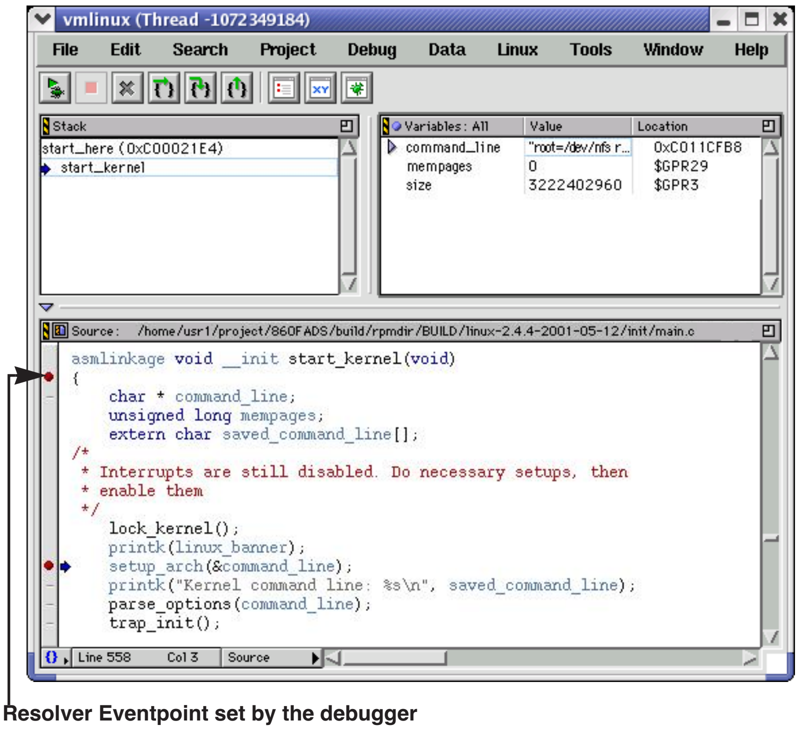
```
08007FF0: 238421C4  orrcs    r2, r4, #0x31
08007FF4: 33C4304C  biccc    r3, r4, #0x4c
08007FF8: 32C4238C  sbccc    r2, r4, #0x30000002
08007FFC: 338432C9  orrcc    r3, r4, #0x9000000c
08008000: E3A0C000  mov      r12, #0x0
08008004: E1A0B001  mov      r11, r1
08008008: E3A010A0  mov      r1, #0xa0
0800800C: E3A000D3  mov      r0, #0xd3
08008010: E121F000  msr      cpsr_c, r0
08008014: EB00004B  bl       0x08008148 (0x8008148)   ; 0x08008148
08008018: E33A0000  teq      r10, #0x0
0800801C: 03A00070  moveq    r0, #0x70
08008020: 0A000046  beq      0x08008140 (0x8008140)   ; 0x08008140
08008024: EB00005B  bl       0x08008198 (0x8008198)   ; 0x08008198
```

b. Click **Run**. The debugger stops execution of the program at `setup_arch()` only if a breakpoint is set in this function (Figure 7.17).

If you checked the **Enable Delayed Software Breakpoint Support** checkbox, the debugger sets a hardware breakpoint (Resolver Eventpoint) at `{` in the `start_kernel()` (Figure 7.17).

When the debugger encounters this Resolver Eventpoint, all the subsequent software breakpoints you have set are enabled. For more information on Resolver Eventpoint, see *IDE User's Guide*.

Figure 7.17 Program Entry Point in Kernel Code


vmlinux (Thread -1072349184)
File Edit Search Project Debug Data Linux Tools Window Help
Stack
start_here (0xC00021E4)
start_kernel
Variables: All
Value
Location
Source: /home/usr1/project/860FADS/build/rpmdir/BUILD/linux-2.4.4-2001-05-12/init/main.c
Line 558
Col 3
Source

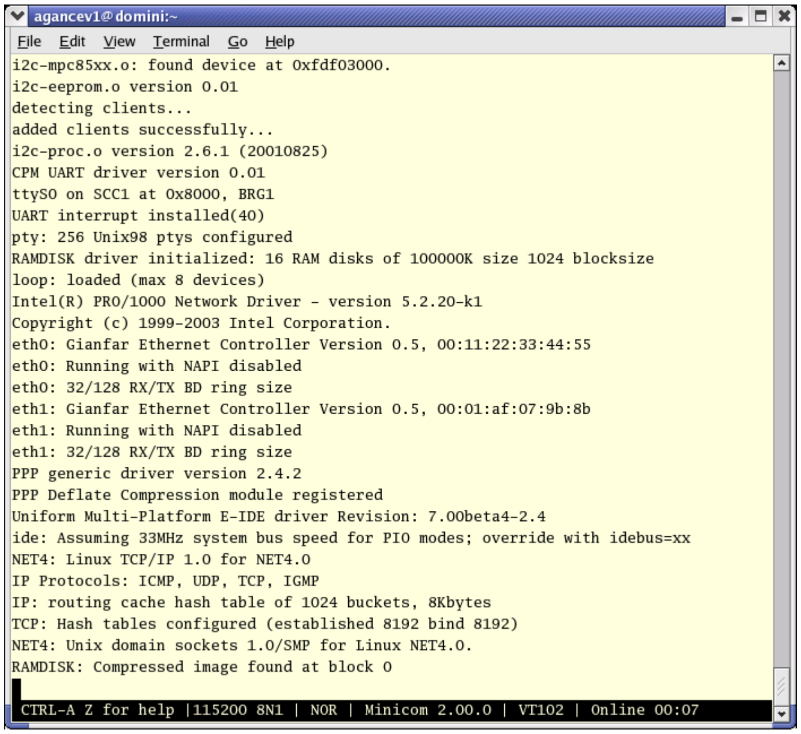
Variables: All	Value	Location
command_line	"root=/dev/nfs r...	0xC011CFB8
mempages	0	$GPR29
size	3222402960	$GPR3

```
asmlinkage void __init start_kernel(void)
{
    char * command_line;
    unsigned long mempages;
    extern char saved_command_line[];
/*
 * Interrupts are still disabled. Do necessary setups, then
 * enable them
 */
    lock_kernel();
    printk(linux_banner);
    setup_arch(&command_line);
    printk("Kernel command line: %s\n", saved_command_line);
    parse_options(command_line);
    trap_init();
```

Resolver Eventpoint set by the debugger

c. Run through the rest of the code until the kernel starts booting. When the kernel boots up you can see the bootup messages in your Terminal window (Figure 7.18).

Figure 7.18 Terminal Window Showing the Kernel Bootup Messages



```
i2c-mpc85xx.o: found device at 0xfdf03000.
i2c-eeprom.o version 0.01
detecting clients...
added clients successfully...
i2c-proc.o version 2.6.1 (20010825)
CPM UART driver version 0.01
ttyS0 on SCC1 at 0x8000, BRG1
UART interrupt installed(40)
pty: 256 Unix98 ptys configured
RAMDISK driver initialized: 16 RAM disks of 100000K size 1024 blocksize
loop: loaded (max 8 devices)
Intel(R) PRO/1000 Network Driver - version 5.2.20-k1
Copyright (c) 1999-2003 Intel Corporation.
eth0: Gianfar Ethernet Controller Version 0.5, 00:11:22:33:44:55
eth0: Running with NAPI disabled
eth0: 32/128 RX/TX BD ring size
eth1: Gianfar Ethernet Controller Version 0.5, 00:01:af:07:9b:8b
eth1: Running with NAPI disabled
eth1: 32/128 RX/TX BD ring size
PPP generic driver version 2.4.2
PPP Deflate Compression module registered
Uniform Multi-Platform E-IDE driver Revision: 7.00beta4-2.4
ide: Assuming 33MHz system bus speed for PIO modes; override with idebus=xx
NET4: Linux TCP/IP 1.0 for NET4.0
IP Protocols: ICMP, UDP, TCP, IGMP
IP: routing cache hash table of 1024 buckets, 8Kbytes
TCP: Hash tables configured (established 8192 bind 8192)
NET4: Unix domain sockets 1.0/SMP for Linux NET4.0.
RAMDISK: Compressed image found at block 0

 CTRL-A Z for help |115200 8N1 | NOR | Minicom 2.00.0 | VT102 | Online 00:07
```

Using u-boot Initialization

Use a u-boot-based initialization file in the kernel project and follow the steps in the previous section.

Use u-boot to Start Linux/Debug Boot Stage

To use u-boot to start Linux, the steps are:

1. Start u-boot.
2. Attach your kernel project to u-boot using **Debug > Attach**. For detailed steps on how to attach to a process, see “Using the Attach to Process Feature” on page 94.
3. Start the boot process from the bootloader console

Attaching to the Running Kernel

To debug the kernel using this method, perform the following steps:

1. Start the kernel on your target platform using any of the above mentioned methods.
2. Attach to the running kernel.

CAUTION Ensure that the ELF file corresponding to the running kernel is present on the target platform.

After the kernel is booted on the target platform, you can now install, load, and debug the kernel modules. See "Kernel Module Debugging" on page 157.

Debugging Kernel Using Custom Target Platform/BSP

You can customize the CodeWarrior IDE to debug kernel using a target platform or board support package (BSP) not supported by CodeWarrior™ Development Studio for PowerPC ISA, Linux® Application/Platform Edition v2.0. For doing this, you must make the minimum changes to the supplied initialization files (also called *minimal initialization*) allowing u-boot to make the initial initialization (by resetting the board, letting u-boot run for a while, and stop) than just specify the MMR Base in order to make CodeWarrior IDE capable to read memory mapped registers. For initialization commands and their description, see "Initialization File Commands" on page 181. Modify the initialization files located at: `CodeWarriorIDE/CodeWarrior/PowerPC_EABI_Support/Initialization_Files/` in your CodeWarrior installation directory according to your target platform requirements. This directory contains initialization files for bareboard and Linux kernel download and debug. Select the initialization file that best suits your requirements.

The initialization file sets up your target platform for downloading and booting the kernel. The **Linux Kernel Residual Data** panel provides the kernel with boot-time dynamic configuration parameters (the `bd_info` structure). For more information on **Linux Kernel Residual Data** panel, see "Linux Kernel Residual Data" on page 173.

Kernel Thread Debugging

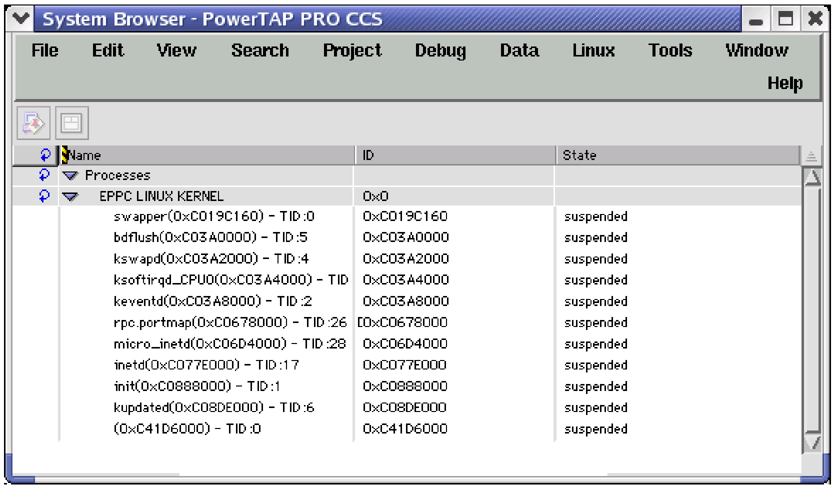
The CodeWarrior debugger enables you to view kernel threads in separate thread windows. Each kernel thread debug window displays its own stack crawl, source, and variable views.

The steps to open multiple kernel thread windows for debugging are:

1. From the kernel debugger window, select **Window > System Windows**. The **System Browser** window (Figure 7.19) appears. The **System Browser** window

displays the currently running processes and the tasks for a particular remote connection.

Figure 7.19 System Browser Window - Process and Tasks Display



Name	ID	State
Processes		
EPPC LINUX KERNEL	0x0	
swapper(0xC019C160) - TID:0	0xC019C160	suspended
bdflush(0xC03A0000) - TID:5	0xC03A0000	suspended
kswapd(0xC03A2000) - TID:4	0xC03A2000	suspended
ksoftirqd_CPU0(0xC03A4000) - TID	0xC03A4000	suspended
keventd(0xC03A8000) - TID:2	0xC03A8000	suspended
rpc.portmap(0xC0678000) - TID:26	[0xC0678000	suspended
micro_inetd(0xC06D4000) - TID:28	0xC06D4000	suspended
inetd(0xC077E000) - TID:17	0xC077E000	suspended
init(0xC0888000) - TID:1	0xC0888000	suspended
kupdated(0xC08DE000) - TID:6	0xC08DE000	suspended
(0xC41D6000) - TID:0	0xC41D6000	suspended

2. Select any of the task that you want to debug for a particular process from the list.
3. Double-click the selected task. A new thread window appears displaying the symbolics for the selected task.

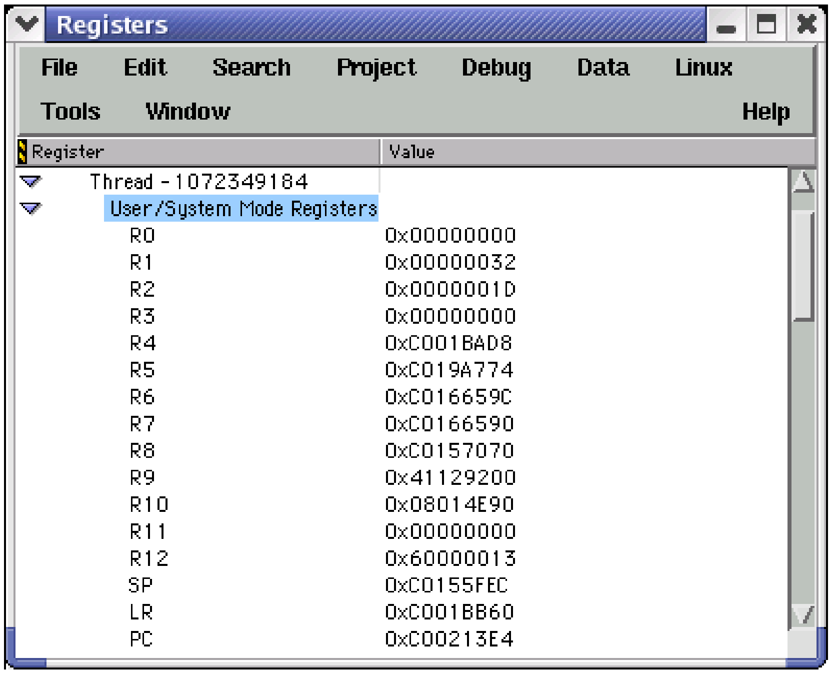
   You can open multiple tasks in separate thread windows.

NOTE Multiple tasks can be displayed in separate thread windows only when the **Show threads in separate window** checkbox is checked in the **Display Settings** panel. For more information, see "Viewing Multiple Processes and Threads" on page 110.

NOTE You may open multiple thread windows for multiple tasks simultaneously, but you can perform debug operations only in the main thread window.

You can view the register values for the open thread windows. Select **Window > Registers Window** to open the **Registers** window. The **Registers** window shown in Figure 7.20 displays the register values for thread windows.

Figure 7.20 Registers Window Displaying the Currently Open Thread Windows



```
Register                               Value
    Thread - 1072349184
        User/System Mode Registers
          R0                           0x00000000
          R1                           0x00000032
          R2                           0x0000001D
          R3                           0x00000000
          R4                           0xC001BAD8
          R5                           0xC019A774
          R6                           0xC016659C
          R7                           0xC0166590
          R8                           0xC0157070
          R9                           0x41129200
          R10                          0x08014E90
          R11                          0x00000000
          R12                          0x60000013
          SP                           0xC0155FEC
          LR                           0xC001BB60
          PC                           0xC00213E4
```

Kernel Module Debugging

This section describes the steps for debugging the kernel modules on your Linux computer.

Linux Kernel Modules - An Introduction

The Linux kernel is a *monolithic kernel*, that is, a single large program where all the functional components of the kernel have access to all of its internal data structures and routines. Alternatively, you may have a micro kernel structure where the functional components of the kernel are broken into pieces with a set communication mechanism between them. This makes adding new components into the kernel using the configuration process very difficult and time consuming. One of the most reliable and robust way is to dynamically load and unload the components of the operating system using *Linux kernel modules*.

The *Linux kernel modules* are pieces of codes, which can be dynamically linked to the kernel according to your requirements. You may unlink and remove the Linux kernel modules from the kernel when you no longer need them. The Linux kernel modules are used for device drivers or pseudo-device drivers such as network drivers or file system.

When a kernel module is loaded, it becomes a part of the kernel as the normal kernel code and functionality and it posses the same rights and responsibilities as the kernel code.

The tutorial that follows demonstrates the kernel module debugging feature. The steps to debug a kernel module are:

- Create a kernel module project
- Build the project
- Physically upload the kernel module binary to the target platform
- Install the binary into the booted kernel
- Display the kernel modules that are loaded in the kernel
- Load the symbolics for the kernel module you want to debug

The first step is to create a kernel module project using the EPPC New Project Wizard.

1. Create a project using the EPPC New Project Wizard with the following settings:

Table 7.2 Kernel Module Project Settings

Project Name:	MyKernel_Module.mcp
Project Location:	/home/usr1/KernelModule
Build Target (Loadable Module Debug) - Output File: **- Output File Location:**	hello.o /home/usr1/KernelModule/Output

NOTE You must select the *Kernel module level debug* item in the EPPC New Project Wizard.

After you create a kernel module project, let us now generate the kernel module application and debug it. The following sections describe how to debug a kernel module.

2. Build the project.

   a. Select **Project > Make**. The kernel module binary (`hello.o`) is built in the specified location in your project directory.

3. Upload the kernel module (`hello.o`) to the target platform.

   After you build the project, you must physically upload the kernel module binary (`hello.o`) to the target platform using any of the following methods (FTP, NFS mount, or copy on the RAM disk image depending on how the file system was mounted).

The next step is to install the kernel module into the kernel.

NOTE Before you install the kernel module into kernel, make sure that the kernel is booted up on the target platform.

4. Install the Kernel Module (`hello.o`) into the running kernel.

   Type `insmod -f <hello.o>` in your Terminal window. The kernel module (`hello.o`) is successfully installed into the booted kernel.

   To verify whether the kernel module was successfully installed, you can type `lsmod` command in your Terminal window. This displays a list of kernel modules currently installed into the kernel.

Display the Kernel Modules List

You can view a list of kernel modules that are currently installed into the kernel by using the CodeWarrior™ IDE. The CodeWarrior IDE provides the **Linux** menu that allows you display the kernel modules that are currently installed.

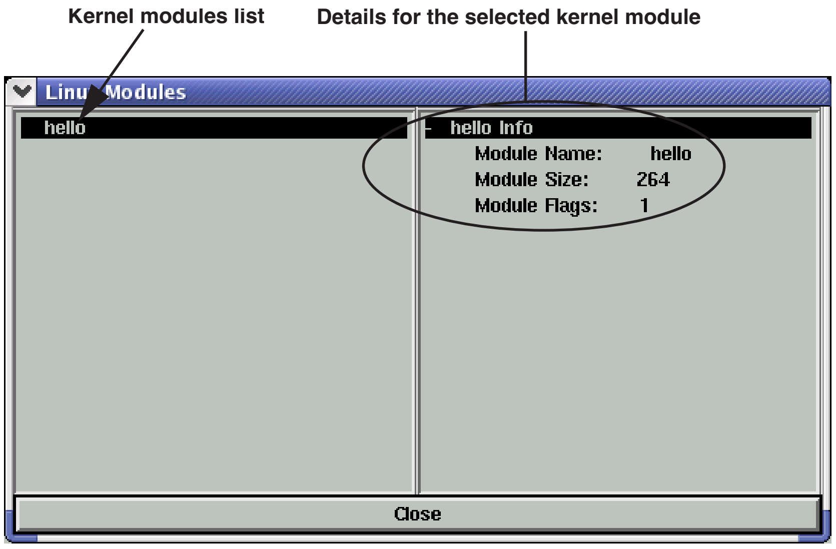
NOTE The **Linux** menu appears only if you are connected to the target platform. For a complete description of the **Linux** menu, see “The Linux Menu” on page 162.

CAUTION To view a list of kernel modules currently installed into the kernel, you must first stop the booted kernel by selecting **Debug > Stop**.
To display the kernel modules list on non-PowerQUICC III target platforms, you need to stop the kernel at a place where MMU is enabled. To do this, set a breakpoint at `if (current->need_resched)` in `idle.c` and run the program until the program hits this breakpoint.

To display a list of kernel modules currently installed into the kernel:

1. Select **Linux > Display Modules**. The **Linux Modules** window appears (Figure 7.21), which displays all the kernel modules (in the left pane) that are currently installed in the kernel.

Figure 7.21 Linux Modules Window - List of Kernel Modules Installed



hello Info
Module Name: hello
Module Size: 264
Module Flags: 1

The **Linux Modules** window displays the module name, file size, and flags set for the selected kernel module. For example, if you select `hello.o` the details of this kernel module is displayed in the right pane.

NOTE The kernel module list is displayed only when the kernel is built with debug symbols. The debug symbols are required by the debugger to read the kernel module list.

Load the Symbolics of the Module

After you select the kernel module that you want to debug, the next step is to load the symbolics for the selected kernel module.

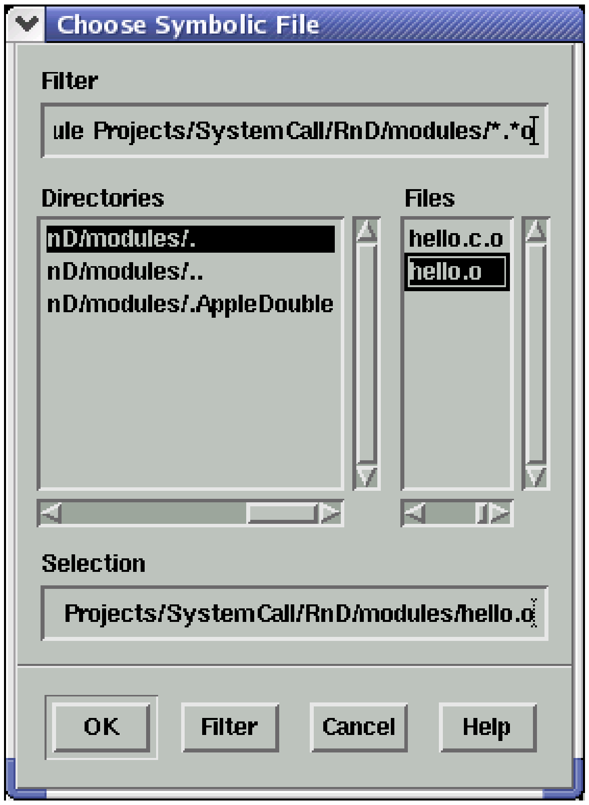
To load the symbolics for a kernel module (`hello.o`):

1. Select **Linux > Load Symbolics**. The **Choose File** dialog box appears.

NOTE Before you load the symbolics for a kernel module, make sure that the symbolics are not already loaded for the kernel module.

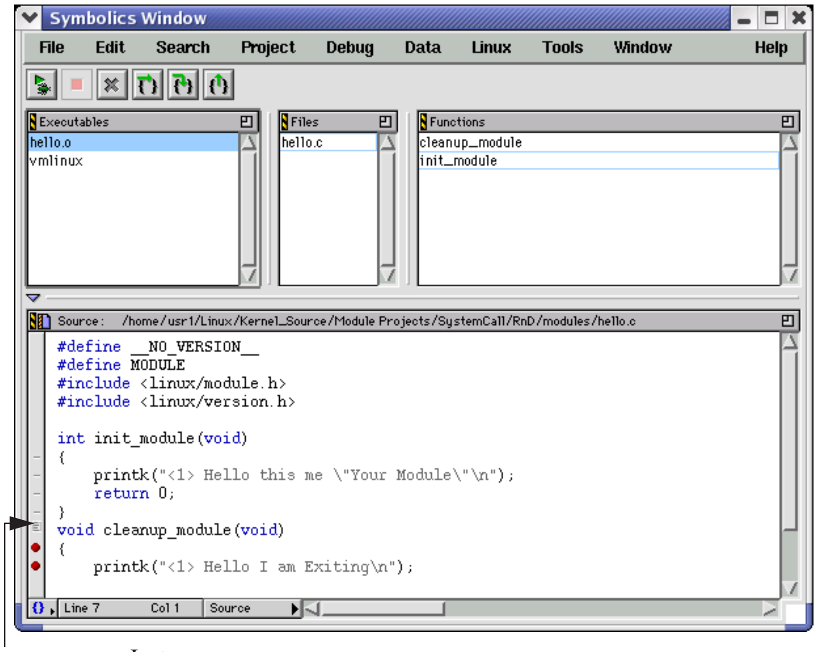
2. Select the kernel module file (`.o`) for which you want to view the symbolics in the **Choose File** dialog box (Figure 7.22).

Figure 7.22 Choose File Dialog Box



3. Click **Open** in the **Choose File** dialog box. The symbolics for the selected kernel module are displayed in the **Symbolics Window** (Figure 7.23).

Figure 7.23 Symbolics Window - Symbolics for the Loaded Kernel Module



Source: /home/usr1/Linux/Kernel_Source/Module Projects/SystemCall/RnD/modules/hello.c

```
#define __NO_VERSION__
#define MODULE
#include <linux/module.h>
#include <linux/version.h>

int init_module(void)
{
    printk("<1> Hello this me \"Your Module\"\n");
    return 0;
}
void cleanup_module(void)
{
    printk("<1> Hello I am Exiting\n");
```

NOTE For detailed information on **Symbolics Window**, see the *IDE User's Guide*.

4. Now, you can perform the regular debugging operations in the Symbolics window.

   If you want to unload the symbolics information for the currently loaded kernel module, select **Linux > Unload Symbolics**.

   If you want to remove the kernel module, type `rmmod` in your Terminal window.

   To verify whether the kernel module is uninstalled, select **Linux > Refresh Module List**. The kernel module is not displayed in the **Linux Modules** window.

The Linux Menu

The CodeWarrior™ Development Studio for PowerPC ISA, Linux® Application/Platform Edition v2.0 contains a menu item on the menu bar called **Linux**. The **Linux** menu contains commands that enables you to view and refresh the currently loaded kernel modules. This menu also has commands to load and unload symbolic information for a kernel module during a debugging session. Clicking the **Linux** menu (Figure 7.24) heading displays a pull-down menu containing the menu commands.

Figure 7.24 Linux Menu

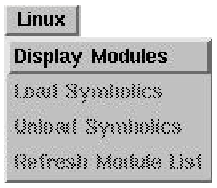


Table 7.3 describes the menu commands provided by the **Linux** menu.

Table 7.3 Linux Menu - Description of Commands

Commands	Description
Display Modules	Displays the list of currently loaded kernel modules
Load Symbolics	Loads the symbolics information for the currently loaded kernel module
Unload Symbolics	Unloads the symbolics information for the currently loaded kernel module
Refresh Module List	Refreshes the kernel modules list and displays the updated list of currently loaded kernel modules

Boa Server Application Debugging

The Boa Server is a high performance HTTP Server for computers running the UNIX operating system. For more information on Boa Server, see boa.html located at the following location in the CodeWarrior installation directory: `/CodeWarriorIDE/Examples/ppc/Target-Specific/boa-0.94.12/docs.`

For the updated copy of this documentation, see

`http://www.boa.org.`

Boa Server Debugging Example

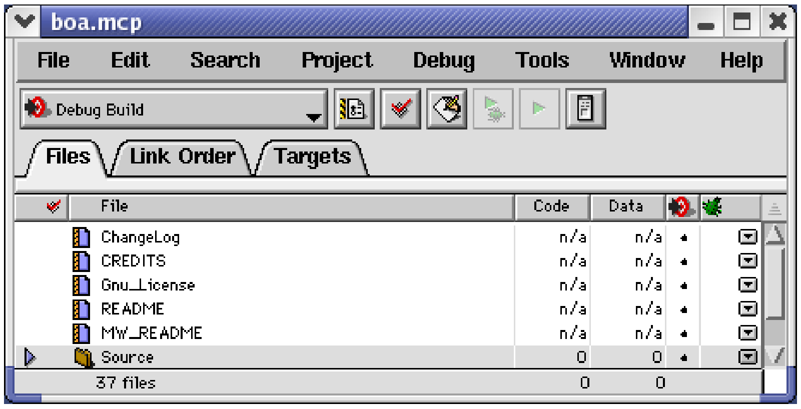
The example in this section debugs a Boa Server project called `boa.mcp`.

NOTE Before you start debugging the Boa Server sample program, make sure that you create *mime.types* and *boa* directories on your target platform. The full path for creating these directories is: `/etc/mime.types` and `/var/log/boa.` Additionally, you need to copy the `boa.conf` file located at: `Examples/boa-0.94.12` to the `/tmp` folder on the target platform. For steps on downloading the files to the target board, see "Installing MetroTRK on the Remote Target" on page 57.

Follow these steps:

1. Open the `boa.mcp` project.
   a. Select **File > Open**.
      The **File Selection** dialog box appears.
   b. Navigate to the following directory in the CodeWarrior installation directory:
      `CodeWarriorIDE/Examples/boa-0.94.12`
   c. Select `boa.mcp`.
   d. Click the **Open** button.
      The project window (Figure 7.25) appears.

Figure 7.25 Project Window - boa.mcp



File	Code	Data		
ChangeLog	n/a	n/a	•	
CREDITS	n/a	n/a	•	
Gnu_License	n/a	n/a	•	
README	n/a	n/a	•	
MW_README	n/a	n/a	•	
Source	0	0	•	
37 files	0	0		

The **Debug Build** target is the default build target selected in the build target list box.

NOTE For this sample application, all the required preference panel settings have already been set to successfully compile, link, and debug the application.

2. Select **Project > Make** to build the application.
   The `boa_debug.elf` file is created in the project folder.
3. Debug the application.

a. Choose **Project > Debug** to start the debugger. The `boa_debug.elf` file is downloaded to the target platform and the initial thread window appears.

   You can set breakpoints, step through the code and perform other routine debugging operations in this window.

4. Select **Debug > Kill** to close the debugger session.

Target Platform-Specific Target Settings Panels

This section discusses only those target settings panels that are unique for CodeWarrior™ Development Studio for PowerPC ISA, Linux® Application/Platform Edition v2.0. These target settings panels are:

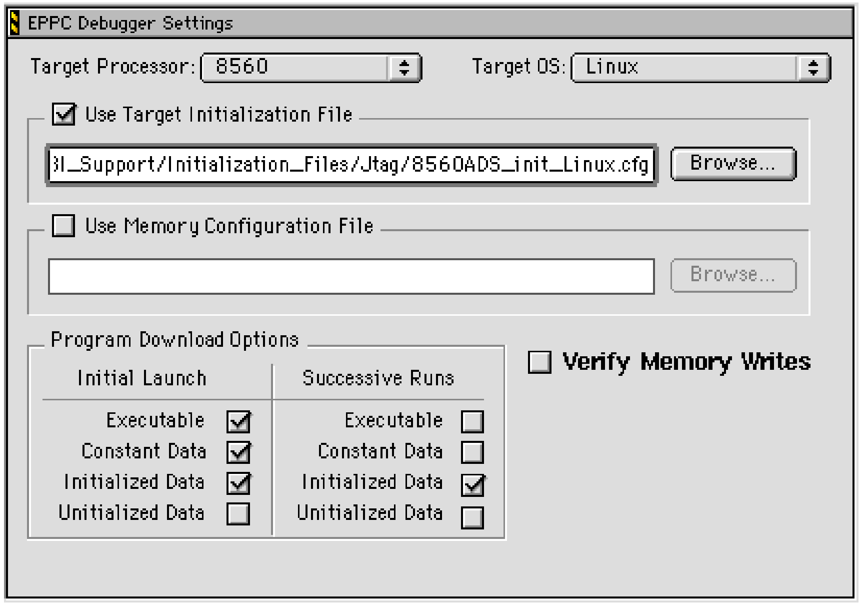
- EPPC Debugger Settings
- Debugger PIC Settings
- EPPC Exceptions
- Linux Kernel Boot Parameters
- Linux Kernel Debug Settings
- Linux Kernel Residual Data
- Source Folder Mapping

EPPC Debugger Settings

The **EPPC Debugger Settings** panel (Figure 7.26) allows you to specify the target platform processor, target operating system, initialization file, and a memory configuration file. You may also configure other debugger-related options for your projects.

An *.xml* file with pre-configured settings for this panel is provided for the target platforms/ BSP's supported by CodeWarrior™ Development Studio for PowerPC ISA, Linux® Application/Platform Edition v2.0. The XML files are located in the following CodeWarrior installation directory: `CodeWarriorIDE/CodeWarrior/PowerPC_EABI_Tools/KernelDebug_Settings/<target_platform_name>`, where `<target_platform_name>` is the target platform name.

Figure 7.26 EPPC Debugger Settings Panel


EPPC Debugger Settings

Target Processor: 8560   Target OS: Linux

[✓] Use Target Initialization File
3I_Support/Initialization_Files/Jtag/8560ADS_init_Linux.cfg Browse...

[ ] Use Memory Configuration File
Browse...

Program Download Options

Initial Launch		Successive Runs	
Executable	✓	Executable	
Constant Data	✓	Constant Data	
Initialized Data	✓	Initialized Data	✓
Unitialized Data		Unitialized Data	

[ ] Verify Memory Writes

This settings panel contains the following options:

Target Processor

Use the **Target Processor** list box to select the processor architecture that you are targeting. This option instructs the debugger to display the appropriate register views.

Target OS

Use the **Target OS** list box to specify the operating system running on your target platform. For kernel debugging, you must select the **Linux** option from the list box. For debugging Linux applications, you must select the **Bareboard** option from the list box.

Use Target Initialization File

Check the **Use Target Initialization File** check box, if you want the debugger to execute an initialization file before downloading the kernel image to the target platform. If you check this option, specify the filename in the text box. Some standard initialization files are available at this location:

`*CWInstall*/CodeWarriorIDE/CodeWarrior/PowerPC_EABI_Support/Initialization_Files/`

For initialization commands and their syntax, see "Debug Initialization Files" on page 179.

Use Memory Configuration File

Check the **Use Memory Configuration File** check box, if you want to use a configuration file. This file contains information about the location of read-only and read/write memory regions in the target platform. If you check this option, specify the filename in the text box. If you do not provide memory configuration information, the debugger assumes that all memory locations are accessible.

For memory configuration commands and their syntax, see "Memory Configuration Files" on page 187.

Program Download Options

These options specify what portions of the project to download on the initial launch of the debugger and successive launches. When debugging code in ROM, this download can be turned off and only the symbols will be loaded.

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 IDE.

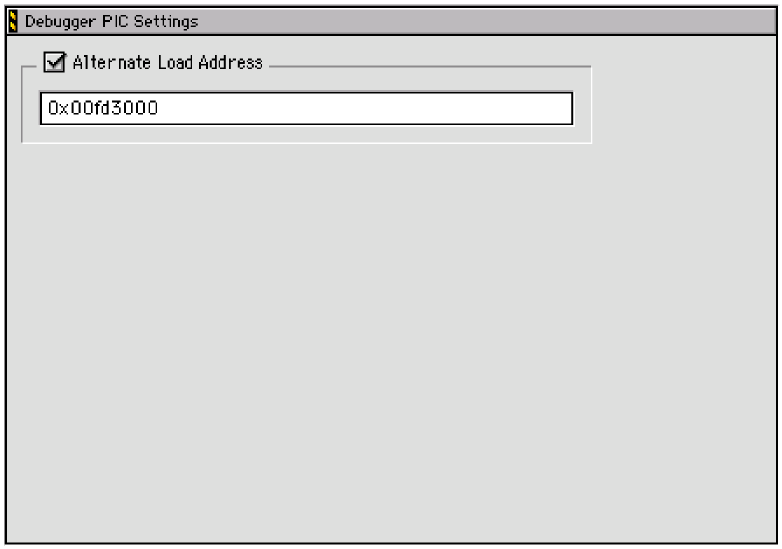
Verify Memory Writes

Check the **Verify Memory Writes** check box to instruct the debugger to verify the memory writes that are performed during the download process.

Debugger PIC Settings

Use the **Debugger PIC Settings** panel (Figure 7.27) to specify an alternate address where you want your ELF image loaded on the target.

Figure 7.27 Debugger PIC Settings Panel



☑ Alternate Load Address

0x00fd3000

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.

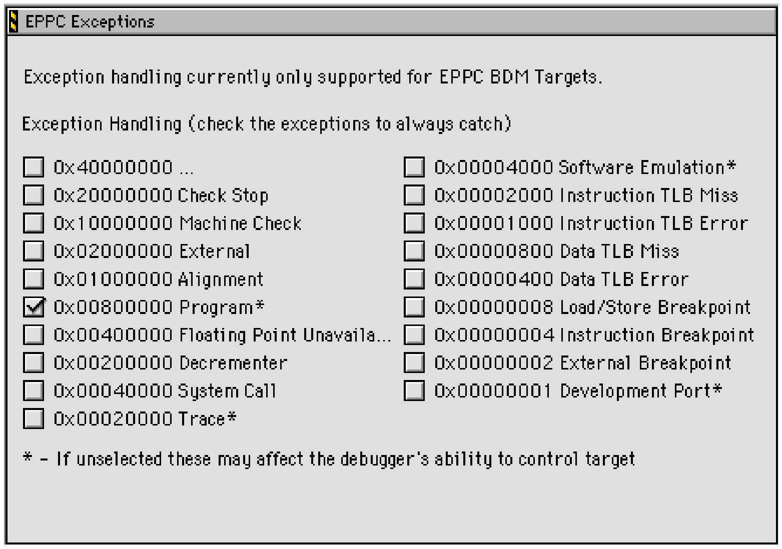
NOTE You need to set this panel before you debug programs that relocate themselves, such as u-boot.

EPPC Exceptions

The **EPPC Exceptions** settings panel (Figure 7.28) lists all the exceptions that the debugger is able to catch.

NOTE It is recommended you use the supplied pre-configured XML file for configuring the settings in this panel for your BDM target platforms, such as, *MPC860FADS*.

Figure 7.28 EPPC Exceptions Panel



EPPC Exceptions

Exception handling currently only supported for EPPC BDM Targets.

Exception Handling (check the exceptions to always catch)

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

* - 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 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:

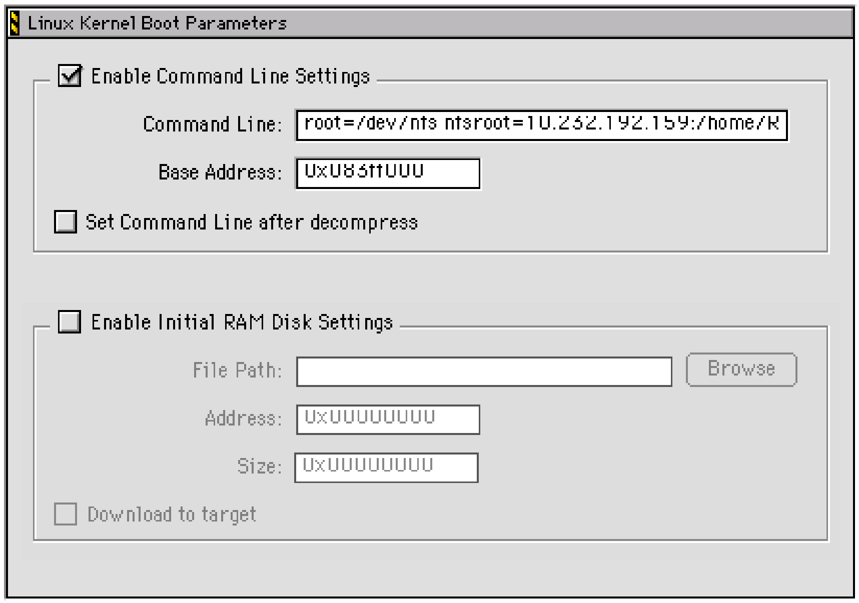
- 0x00800000 Program — for software breakpoints on some boards
- 0x00020000 Trace — for single stepping
- 0x00004000 Software Emulation — for software breakpoints on some boards
- 0x00000001 Development Port — for halting the target processor.

Linux Kernel Boot Parameters

The **Linux Kernel Boot Parameters** settings panel (Figure 7.29) allows you to specify or edit the command line parameter required for booting the kernel.

NOTE It is recommended you use the supplied pre-configured XML file for configuring the settings in this panel for your target platform/BSP.

Figure 7.29 Linux Kernel Boot Parameters Settings Panel



Linux Kernel Boot Parameters

[✓] Enable Command Line Settings

Command Line: root=/dev/nfs nfsroot=10.232.192.159:/home/R

Base Address: 0x083ff000

[ ] Set Command Line after decompress

[ ] Enable Initial RAM Disk Settings

File Path:

Browse

Address: 0x00000000

Size: 0x00000000

[ ] Download to target

This settings panel contains the following options:

Enable Command Line Settings

Check the **Enable Command Line Settings** checkbox to enable passing of command line parameters to the kernel while the kernel boots up.

The **Command Line** text box allows you to specify the command line strings that you want to pass to the kernel.

Enter the address in the **Base Address** text box where you want the command line parameters to be written in the memory.

Enable Initial RAM Disk Settings

Check the **Enable Initial RAM Disk Settings** checkbox to enable the initial RAM Disk settings.

Enter or click **Browse** to specify the host-side path of the initial RAM disk image (*initrd*) to be downloaded to the target platform. If you build the kernel using the RAM file system, the RAM file system must be downloaded to the target platform along with the kernel image.

NOTE It is not mandatory to use a RAM file system for debugging the kernel. A kernel built using the NFS file system or any other file system can also be debugged in a similar manner.

Enter the memory address in the **Address** text box where you want the `initrd` file contents to be written in the memory.

Enter the size of the `initrd` file in the **Size** text box.

Check the **Download to target** checkbox if you want to download the initial RAM disk image to the target platform.

NOTE For more information on NFS root, see `Documentation/nfsroot.txt`.
For more information on intial RAM Disk, see `Documentation/ramdisk.txt` file in your kernel sources.
For more information on kernel boot parameters, see the `bootcmd man` pages.

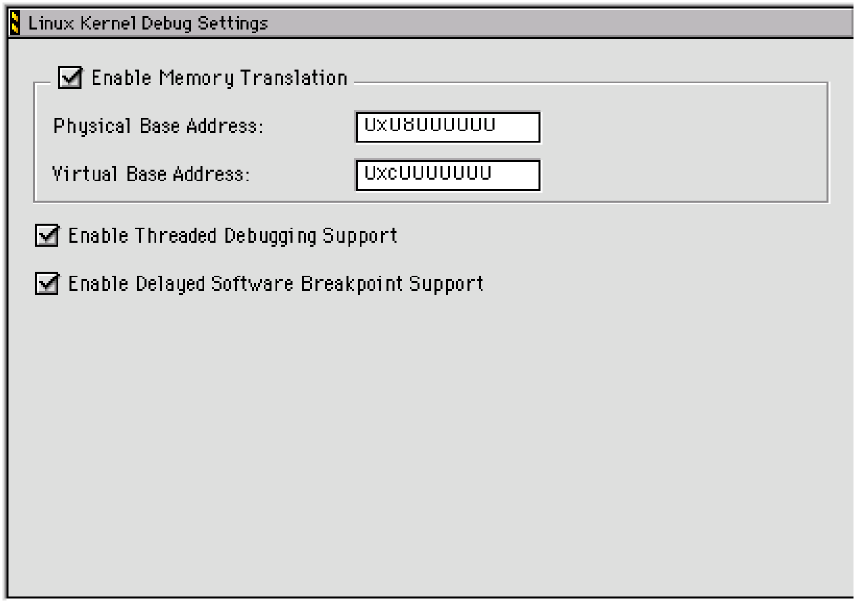
Linux Kernel Debug Settings

The **Linux Kernel Debug Settings** panel (Figure 7.30) allows you to enable:

- memory translation so that the debugger can map the physical address and the virtual address.
- support for debugging multiple kernel threads on the target platform.
- support for delaying setting of software breakpoints in the kernel code.

NOTE It is recommended you use the supplied pre-configured XML file for configuring the settings in this panel for your target platform/BSP.

Figure 7.30 Linux Kernel Debug Settings Panel



Linux Kernel Debug Settings

[✓] Enable Memory Translation

Physical Base Address: 0x08000000

Virtual Base Address: 0xc0000000

[✓] Enable Threaded Debugging Support

[✓] Enable Delayed Software Breakpoint Support

This settings panel contains the following options:

Enable Memory Translation

Check the **Enable Memory Translation** checkbox to enable memory translation so that the debugger can map the physical base memory and the virtual base memory.

Enter the physical base address for memory translation in the **Physical Base Address** text box.

Enter the virtual base address for memory translation in the **Virtual Base Address** text box.

Enable Threaded Debugging Support

Check the **Enable Threaded Debugging Support** checkbox if you want to debug multiple kernel threads in separate thread windows. If you uncheck this checkbox, you will not be able to view newly created kernel threads in a separate thread window. Also, the **Process** window will not display any kernel tasks in the Task list

NOTE To view kernel threads in separate thread windows, the **Show threads in separate window** checkbox must be checked in the **Display Settings** panel.

Enable Delayed Software Breakpoint Support

Check the **Enable Delayed Software Breakpoint Support** checkbox if you want to delay setting of software breakpoints until the MMU is enabled. When the debugger stops at `{` in the `start_kernel()`(hardware breakpoint), the debugger sets all the delayed software breakpoints, which were set at virtual address locations.

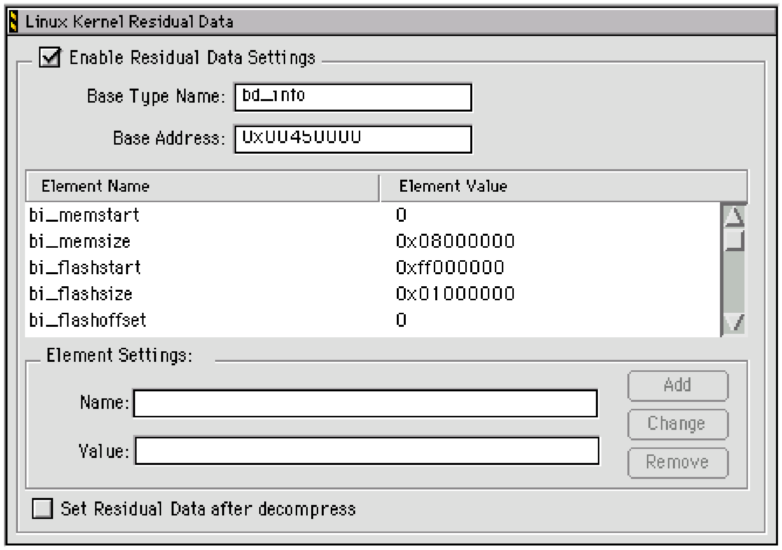
Linux Kernel Residual Data

The **Linux Kernel Residual Data** settings panel (Figure 7.31) allows you to set a *structure* for a executable binary with symbolics in the memory. If the **Enable Residual Data Settings** checkbox is checked and you download the binary to the target platform, the CodeWarrior debugger searches for the structure details in the symbolics for the structure specified in the **Base Type Name** text box. The members of this structure can be set in the memory with values relative to the base address specified in the **Base Address** text box. If the structure name or member name specified in the **Linux Kernel Residual Data** panel does not match with the structure details in the executable binary symbolics, they are ignored.

You must set the *bd_t (bd_info)* structure for the kernel in the memory to boot the kernel successfully. When you run the kernel, the debugger searches for the bd_t structure in the symbolics to retrieve information on location of DRAM, SRAM, FLASH memory, IMMPR base, IP address to use, bus speed, and ethernet MAC addresses for the board ports.

NOTE It is recommended you use the supplied pre-configured XML file for configuring the settings in this panel for your target platform/BSP.

Figure 7.31 Linux Kernel Residual Data Settings Panel



[✓] Enable Residual Data Settings

Base Type Name: bd_info

Base Address: 0x00450000

Element Name	Element Value
bi_memstart	0
bi_memsize	0x08000000
bi_flashstart	0xff000000
bi_flashsize	0x01000000
bi_flashoffset	0

Element Settings:

Name:

Value:

[ ] Set Residual Data after decompress

This settings panel contains the following options:

Enable Residual Data Settings

Check the **Enable Residual Data Settings** checkbox to enable setting the structure in memory after the kernel is downloaded on the target platform.

Base Type Name

Enter the structure name you want to set in the **Base Type Name** text box. For example, *bd_t* (*bd_info*).

Base Address

Enter the physical base address in the **Base Address** text box. This is the starting point for the structure.

Element Settings

Enter the structure members name to be set in the **Name** text box under the **Element Settings** group.

Enter the structure members value in the **Value** text box under the **Element Settings** group.

Set Residual Data after decompress

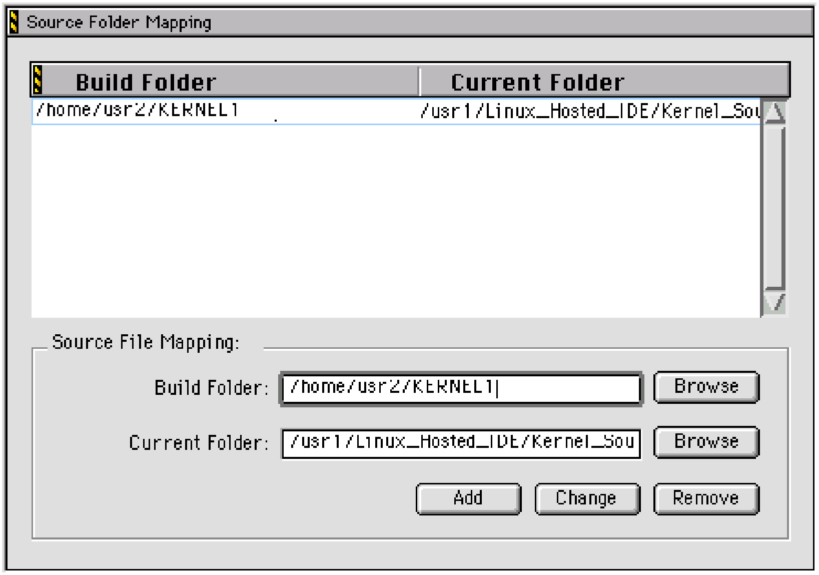
Check the **Set Residual Data after decompress** checkbox to download the structure again after the kernel image is decompressed on the target platform. This option is necessary, if you have checked the **Download OS** checkbox in the **Remote Debugging** settings panel.

Source Folder Mapping

The **Source Folder Mappings** settings panel (Figure 7.32) allows you to map the location of the kernel source files on a Linux computer. If you using a kernel built on another Linux computer, you must map the location of the kernel sources in that computer to the a folder in your Linux computer

If you have already done the mapping, the **Source Folder Mapping** settings panel displays the current mapping.

Figure 7.32 Source Folder Mappings Settings Panel



Build Folder	Current Folder
/home/usr2/KERNEL1	/usr1/Linux_Hosted_IDE/Kernel_Sou

Source File Mapping:

Build Folder: /home/usr2/KERNEL1

Current Folder: /usr1/Linux_Hosted_IDE/Kernel_Sou

This settings panel contains the following options:

Build Folder

Enter or browse for the location path of the folder on the Linux computer, which contains the kernel sources used for building the kernel.

Current Folder

Enter or browse for the location path of the folder on your computer, which you want to map to the folder containing kernel sources on the another Linux computer.

Target Platform-Specific Information

This section has these topics:

- Cross Compiler Tools Location
- MetroTRK Project and Binary File Location
- MetroTRK Project - Build Targets
- Sample Projects Location

Cross Compiler Tools Location

The CodeWarrior™ Development Studio for PowerPC ISA, Linux® Application/Platform Edition v2.0 packages the cross compiler tools for all the supported target platforms.

Table 7.4 lists the location where you can find the target platform-specific cross compiler tools.

Table 7.4 Cross Compiler Tools and Their Locations

Cross Compiler Tools For	Available at
EPPC Target Platforms	`CWInstall/CodeWarriorIDE/CodeWarrior/Cross_Tools/binary/Metrowerks/ppc/tools`

MetroTRK Project and Binary File Location

Table 7.5 lists the location where you can find the MetroTRK project and binary files applicable for your target platform.

Table 7.5 MetroTRK Project and Binary File Location

Type	Available at
MetroTRK binaries	*CWInstall*/CodeWarriorIDE/CodeWarrior/PowerPC_EABI_Tools/ MetroTRK/Os/unix/linux/ppc/Bin
MetroTRK project	*CWInstall*/CodeWarriorIDE/CodeWarrior/PowerPC_EABI_Tools/ MetroTRK/Os/unix/linux/ppc/trk_linux_ppc.mcp

MetroTRK Project - Build Targets

Table 7.6 lists the build targets available in the MetroTRK project.

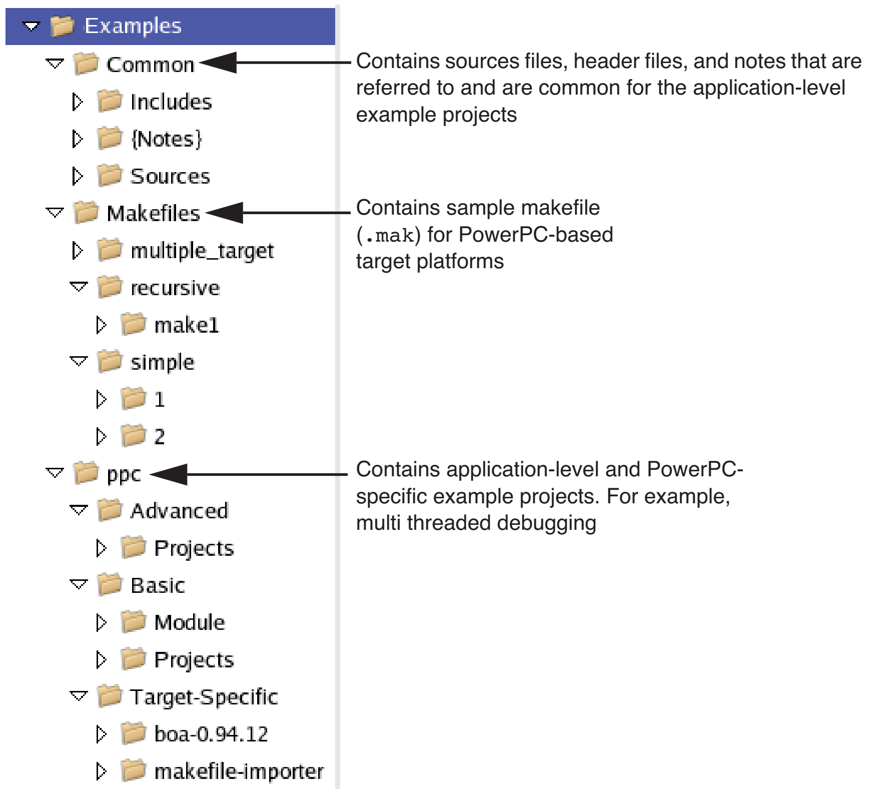
Table 7.6 MetroTRK Project Build Targets

Build Target Name	Description
Linux AppTRK for e500	Builds release version of MetroTRK for e500 target platforms
Linux AppTRK for e500 (Debug)	Builds debug version of MetroTRK for e500 target platforms
Linux AppTRK for e603	Builds release version of MetroTRK for e603 target platforms
Linux AppTRK for e603 (Debug)	Builds debug version of MetroTRK for e603 target platforms
Build All	Builds all versions of the MetroTRK one after another

Sample Projects Location

CodeWarrior™ IDE provides ready-made projects, which contain all the required settings for running it successfully on a PowerPC-based target platform. The sample projects will help you to understand the features and capabilities of the CodeWarrior™ Development Studio for PowerPC ISA, Linux® Application/Platform Edition v2.0 software. Figure 7.33 shows the directory structure of the *Examples* folder.

Figure 7.33 Examples Directory Structure


Examples
Common
Includes
{Notes}
Sources
Makefiles
multiple_target
recursive
make1
simple
1
2
ppc
Advanced
Projects
Basic
Module
Projects
Target-Specific
boa-0.94.12
makefile-importer
Contains sources files, header files, and notes that are referred to and are common for the application-level example projects
Contains sample makefile (.mak) for PowerPC-based target platforms
Contains application-level and PowerPC-specific example projects. For example, multi threaded debugging

Each top-level directory has a `Readme.txt` file that explains the intent of each example in that directory.

To use any of the examples, the following are required:

1. An account on the Linux target system with user name `sample`
2. All the executables must be downloaded to the `/home/sample` directory
3. The MetroTRK should be executed by the user *`sample`*
4. Appropriate settings in the **Remote Debugging** settings panel (for remote connection).

A list of some of the sample application projects and the kernel module project available in your CodeWarrior installation directory is listed in Table 7.7.

Table 7.7 Example Projects and Their Location

Project File Name	Available at	Description
SharedLibrary.mcp	Examples/ppc/Basic/Projects	Shared library sample project
THREAD.mcp	Examples/ppc/Basic/Projects	Multithreading sample project
ForkAndExec.mcp	Examples/ppc/Advanced/ Projects	Fork( ) and Exec( ) sample project
module.mcp	Examples/ppc/Basic/Module	Kernel module sample project

Table 7.8 lists the location where you can find the header and source files for the above mentioned sample Linux application projects.

Table 7.8 Header and Source Files for Sample Projects

Source files	Examples/Common/Sources
Header files	Examples/Common/Includes

NOTE For information on how to work with the sample projects, see the project notes located at: Examples/Common/{Notes}.

Debug Initialization Files

A debug initialization file is used to initialize the target platform before the debugger downloads the code. The main purpose is to ensure that the target platform is initialized properly before being accessed.

This section has 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 JTAG to ensure that the target platform 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 Debugger 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 target platforms and can be found at this location of the CodeWarrior installation directory:

```
/CodeWarriorIDE/CodeWarrior/PowerPC_EABI_Support/
Initialization_Files/
```

Debug Initialization File Commands

This section explains debug initialization file commands, including:

- Debug initialization file command syntax
- Description and examples of individual commands

Each section explains these individual command lists:

- The command name
- A brief description of the command
- Command usage (prototype)
- Command examples
- Hardware debug device supporting the command
- Any important notes about the command

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)
  - Octal - preceded by 0 (0123 0456)
  - Dec - starts with 1-9 (12 126 823643)
- Comments start with a “;” or “#”, and go till the end of the line.

Initialization File Commands

The following commands are found in configuration files:

- alternatePC
- ANDMem.I
- ANDmmr
- InCormmr
- ORMem.I
- reset
- run
- setMMRBaseAddr
- sleep
- stop
- writemem.b
- writemem.w
- writemem.l
- writemmr
- writereg
- writespr
- writeupma
- writeupmb

alternatePC

Table 7.9 alternatePC

Descriptio n	Sets the PC register to the specified value. Can be used when program entry point differs from elf image entry point.
Usage	`alternatePC <value>`
Example	`alternatePC 0xc28737a3`

ANDMem.l

Table 7.10 ANDMem.l

Descriptio n	Reads 4 bytes starting with address, makes a bit AND of this value with 32bitMASK and writes it back at the same address.
Usage	`ANDMem.I address 32bitMASK`

ANDmmr

Table 7.11 ANDmmr

Descriptio n	Reads the value of the MMR register on which executes an AND with the specified value. The result is the written back into the same register.
Usage	`ANDmmr <registerName> <value>`
Example	`ANDmmr ACFG 0x00002000`

InCormmr

Table 7.12 InCormmr

Description	Reads the value of the MMR register on which executes an OR with the specified value. The result is the written back into the same register.
Usage	`IncORmmr <registerName> <value>`
Example	`IncOrmmr ACFG 0x00002000`

ORMem.l

Table 7.13 ORMem.l

Description	Reads 4 bytes starting with address, makes a bit OR of this value with 32bitMASK and writes it back at the same address. Does not perform read/write/verify.
Usage	`ORMem.I address 32bitMASK`

reset

The `reset` command is specific to debugging through the CCS protocol.

Table 7.14 reset

Descripti on	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`

run

Table 7.15 run

Description	Causes the processor to enter in running mode.
Usage	`Run`
Example	`Run`

setMMRBaseAddr

The `setMMRBaseAddr` command works only with target platforms that use the 825x/826x processors.

Table 7.16 setMMRBaseAddr

Descripti on	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.

Table 7.16 setMMRBaseAddr (*continued*)

Usage	`setMMRBaseAddr<value>`, where `<value>` is the base address for the memory mapped registers.
Example	`setMMRBaseAddr 0x0f00000`

sleep

Table 7.17 sleeep

Descriptio n	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`

stop

Table 7.18 stop

Description	Causes the processor to enter in suspend mode.
Usage	`Stop`
Example	`Stop`

writemem.b

Table 7.19 writemem.b

Descripti on	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

Table 7.20 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

Table 7.21 writemeem.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

Table 7.22 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.

Table 7.22 writemmr (*continued*)

Usage	`writemmr < register name> <value>`

Example	`writemmr SYPCR    0xffffffc3`
	`writemmr RMR      0x0001`
	`writemmr MPTPR    0x3200`

writereg

Table 7.23 writereg

Descripti on	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

Table 7.24 wrritespr

Descriptio n	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

Table 7.25 writeupma

Descripti on	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

Table 7.26 writeupmb

Descripti on	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`

Memory Configuration Files

A memory configuration file contains commands that define the accessible areas of memory for your specific target platform.

This section has these topics:

- Command Syntax
- Memory Configuration File Commands

Command Syntax

The list of rules for syntax of commands in a configuration file is as follows:

- All syntax is case insensitive
- Any white spaces and tabs are ignored

- Comments can be standard C or C++ style comments
- A number may be entered in hexadecimal, octal, or decimal:
  - Hexadecimal—preceded by 0x (0x00002222, 0xA, 0xCAfeBeaD)
  - Octal—preceded by 0 (0123, 0456)
  - Decimal—starts with 1-9 (12, 96, 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 the configuration files, and any important notes about the command.

The following commands are found in the memory configuration files:

- range
- reserved
- reservedchar

range

Table 7.27 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

Table 7.28 reserved

Descripti on	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

Table 7.29 reservedchar

Descripti on	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`

Using Hardware Tools

This section explains the CodeWarrior IDE hardware tools. Use these tools for board bring-up, test, and analysis.

Flash Programmer

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

- Program
- Erase
- BlankCheck
- Verify
- Checksum

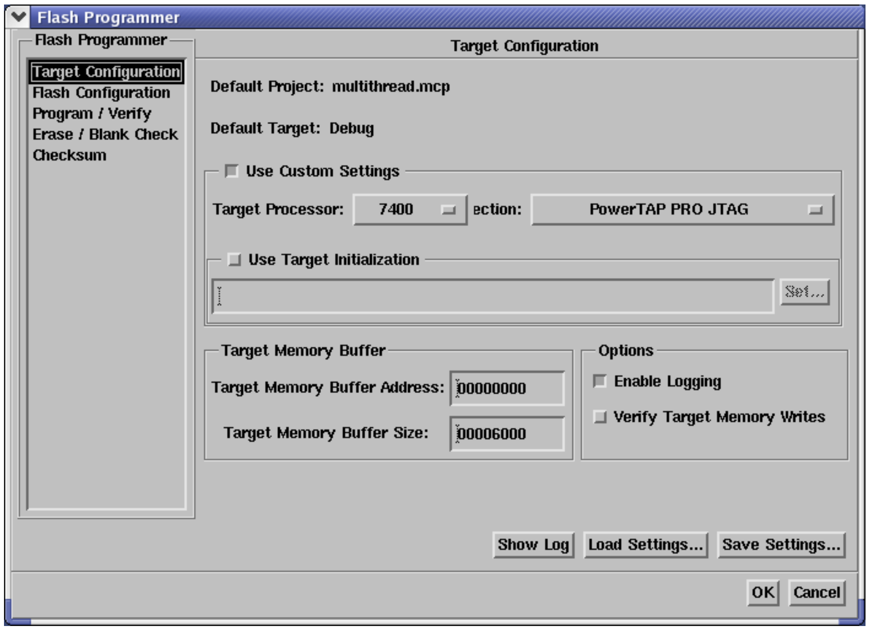
NOTE Certain flash programming features, such as view/modify, memory/register, or save memory content to a file, are provided by the CodeWarrior debugger. Therefore, these features are not a part of the CodeWarrior flash programmer.

The CodeWarrior flash programmer 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.

The **Flash Programmer** window (Figure 7.34) lists global options for the flash programmer hardware tool. These preferences apply to every open project file.

Figure 7.34 Flash Programmer Window



Default Project: multithread.mcp

Default Target: Debug

Use Custom Settings

Target Processor: 7400   ection: PowerTAP PRO JTAG

Use Target Initialization

Target Memory Buffer
Target Memory Buffer Address: 00000000
Target Memory Buffer Size: 00006000

Options
Enable Logging
Verify Target Memory Writes

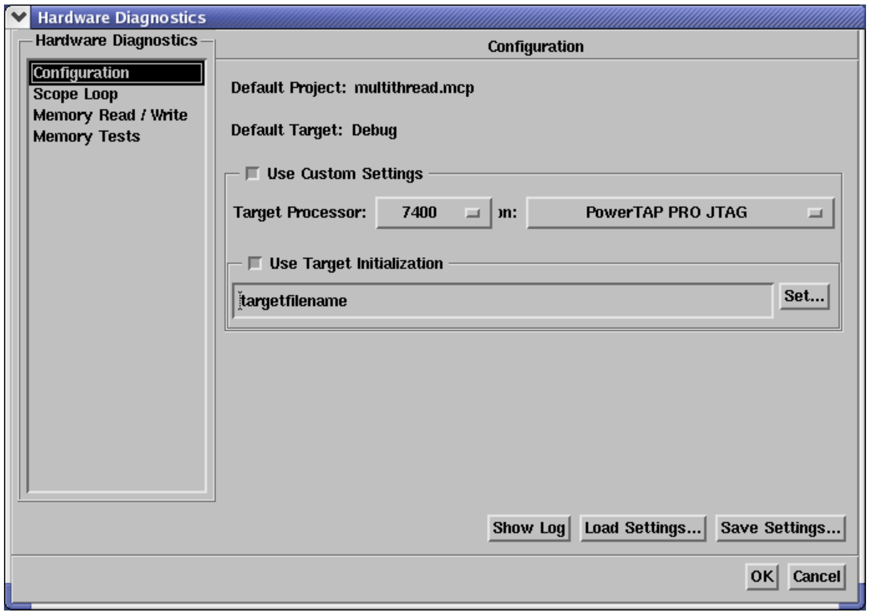
To open the **Flash Programmer** window, click **Tools > Flash Programmer**. The left pane of the **Flash Programmer** window shows a tree structure of panels. Click a panel name to display that panel in the right pane of the **Flash Programmer** window.

Refer to the *IDE User's Guide* for information on each panel in the **Flash Programmer** window.

Hardware Diagnostics

The **Hardware Diagnostics** window (Figure 7.35) 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 7.35 Hardware Diagnostics Window



Default Project: multithread.mcp

Default Target: Debug

Use Custom Settings

Target Processor: 7400 on: PowerTAP PRO JTAG

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's Guide* for information on each panel in the **Hardware Diagnostics** window.

A

Third Party Cross Compiler Tools

You may want to use/build cross compiler tools from other sources that are not installed during CodeWarrior™ IDE installation. This appendix describes how to use these third party cross compiler tools to build your project.

NOTE This Appendix assumes that you already have a CodeWarrior project open. Ensure that the source files for the cross compiler tools are available in your host computer.

You need to rebuild the CodeWarrior project with the new third party cross compiler tools. Before you rebuild the project, you need to make changes in the **GNU Tools** and **Access Paths** setting panels. The steps are:

1. Select **Edit >** ***Target*** **Settings**. The ***Target*** **Settings** panel appears. ***Target*** is the build target name.
2. Select **Target Settings** from the **Target Settings Panels** list. The **Target Settings** panel appears.
3. Select the appropriate linker from **Linker** list box.
4. Click **Save** to save the settings.
5. Specify the path where the third party cross compiler tools are installed/copied.
   a. Select **GNU Tools** from the **Target Settings Panels** list. The **GNU Tools** settings panel appears.
   b. Check the **Use Custom Tool Commands** checkbox to specify new third party cross compiler tools.
   c. Specify the path where cross compiler tools exist on your computer in the **Tool Path** text box. For example, if the third party cross compiler tools are located at `/usr/local/ppc,` then the Tool Path will be at `/usr/local/ppc/bin`
   d. Update the **Commands** section with the commands. These commands are located at third party cross compiler tools installation directory. For example, **Compiler** gcc, **Linker** gcc, **Archiver** ar, **Size Reporter** size, **Disassembler** objdump, **Assembler** as, and **Post Linker** strip.

6. Change the access path settings for kernel and gcc-lib include files in the Access Paths settings panel.
   a. Select **Access Paths** from the **Target Settings Panels** list. The **Access Paths** settings panel appears.
   b. Click **Change** to modify the access path settings for kernel and gcc-lib-specific include files. A file mapping dialog box appears.
   c. Select the kernel include files from the list and click **Select "directory_name"**, where "**directory_name"** is the directory where the kernel source files are located.
   d. Similarly, select gcc-lib include files from the list.
   e. Click **Save** to save the settings.
   f. Close the ***Target* Settings** panel.

   Now, you are ready to rebuild your project using the third party cross compiler tools.

NOTE If you want to use third party cross compiler tools to build your projects/build targets, you must perform the above steps for each project/build target.

B

Using PCS to Build Kernel

This appendix describes how to build kernel using the Platform Creation Suite (PCS).

This appendix has these sections:

- Platform Creation Suite - Overview
- Build the Kernel Using PCS

Platform Creation Suite - Overview

Metrowerks® Platform Creation Suite for Linux® OS provides a complete development environment supporting each step of an embedded product's development cycle from concept to market. The Platform Creation Suite (PCS) consists of three core components:

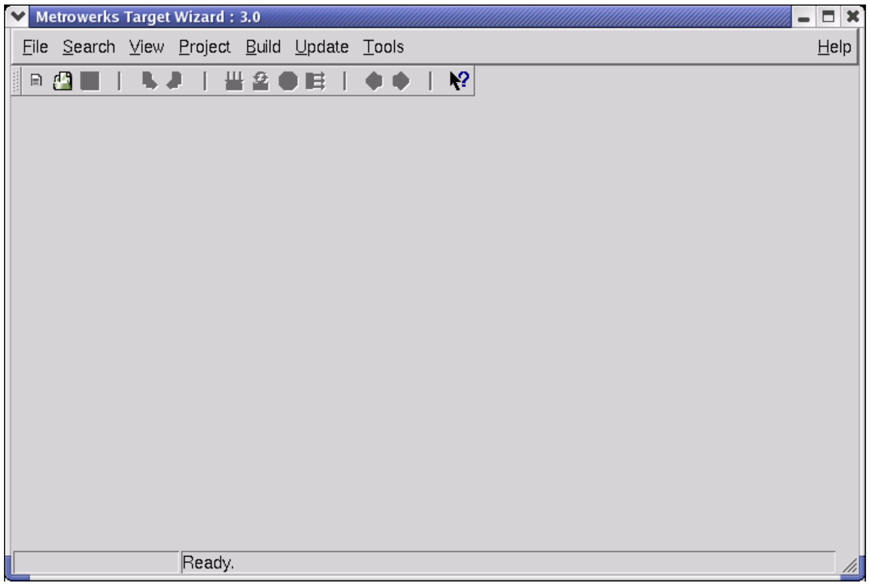
- Target Wizard Tools— used to manage, configure, extend, build and deploy Linux software elements
- CodeWarrior IDE—used to project manage, create, build and debug your value added software which sits on top of the open source elements
- Board Support Package(s)—includes all of the host and target elements that are required to build and deploy the operating system, device drivers and applications to specific target hardware

Build the Kernel Using PCS

NOTE It is assumed that Target Wizard is installed and loaded on your system with Lite500.

Target Wizard (Figure B.1) uses projects to organize your work.

Figure B.1 Metrowerks Target Wizard Opening Screen


Metrowerks Target Wizard : 3.0
File Search View Project Build Update Tools Help
Ready.

Each project is stored in its own directory, which contains a number of files and subdirectories.

- Create a Project
- Build the Project
- Debug the Kernel

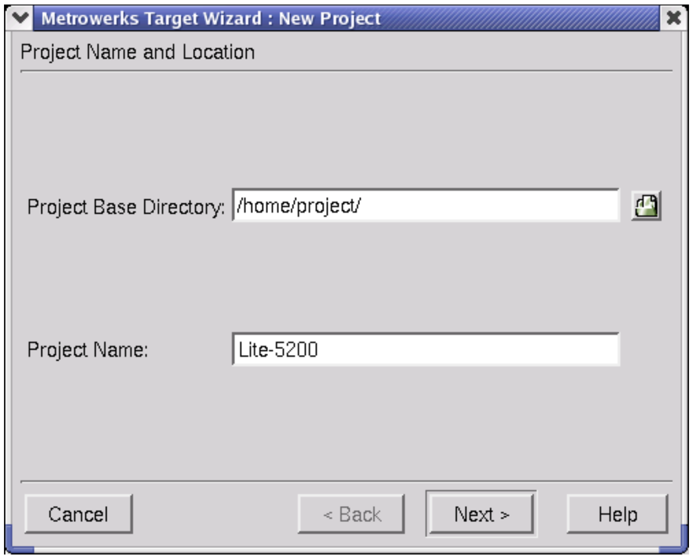
Create a Project

To create a project, first ensure that Target Wizard is running and that no project is open. Then use the following procedure.

NOTE Start Target Wizard by entering the command `tw` at command prompt, before beginning with the Target Wizard.

1. From the Target Wizard interface, select **Project > New**. Page 1 of the Target Wizard (Figure B.2)) appears.

Figure B.2 Target Wizard - Page 1

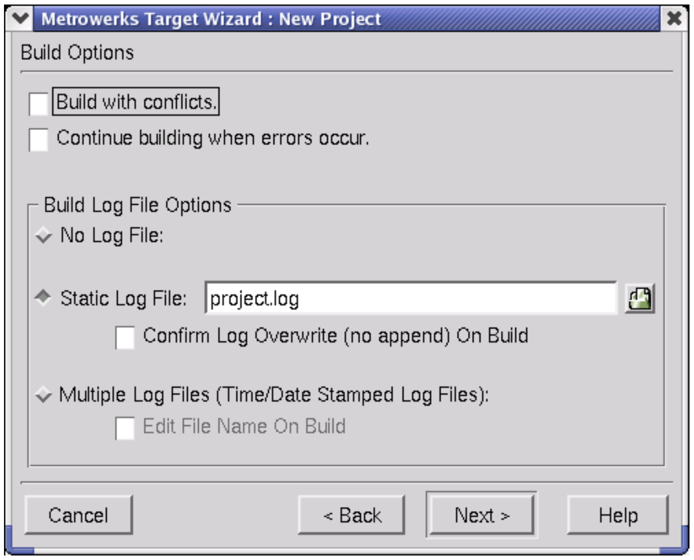

Metrowerks Target Wizard : New Project
Project Name and Location
Cancel
< Back
Next >
Help

Project Base Directory: /home/project/

Project Name: Lite-5200

2. Enter project base directory location in **Project Base Directory** text box.
3. Enter project name in **Project Name** text box.
4. Click **Next**. Page 2 of the Target Wizard (Figure B.3) appears.

Figure B.3 Target Wizard - Page 2



Metrowerks Target Wizard : New Project

Build Options

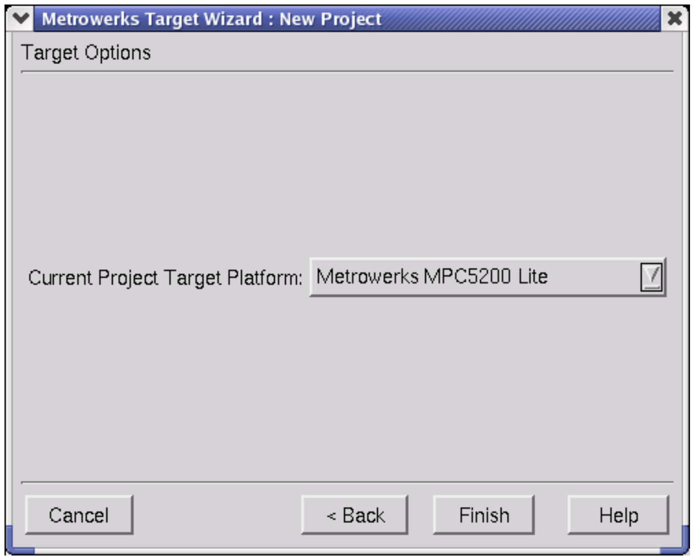
- [ ] Build with conflicts.
- [ ] Continue building when errors occur.

Build Log File Options

- No Log File:
- [✓ Static Log File:] project.log
  - [ ] Confirm Log Overwrite (no append) On Build
- Multiple Log Files (Time/Date Stamped Log Files):
  - [ ] Edit File Name On Build

5. Check **Build with conflicts** checkbox to allow build to continue when conflicts exist.
6. Check **Continue building when errors occur**, if you want to continue build when errors occur.
7. Select type of log file from **Build Log File Options**.
8. Click **Next**. Page 3 of the Target Wizard (Figure B.4) appears.

Figure B.4 Target Wizard - Page 3



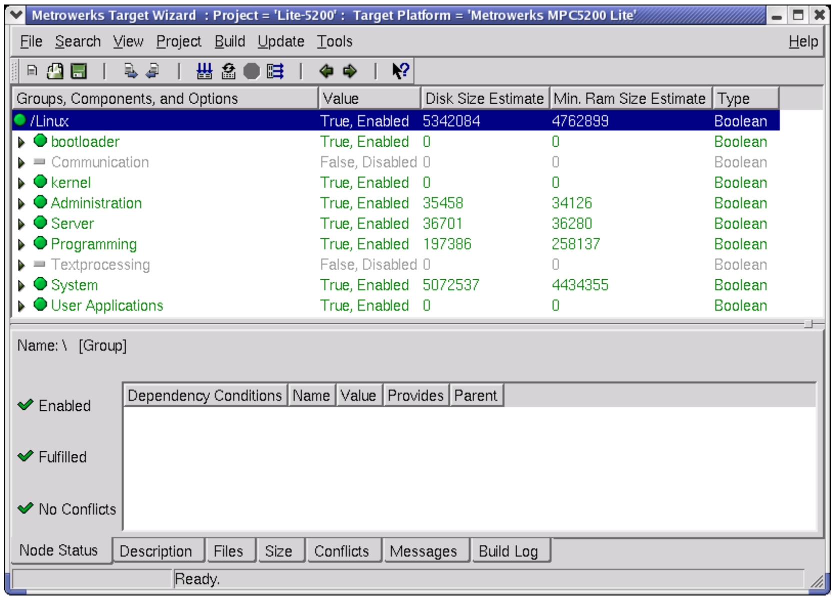
Current Project Target Platform: Metrowerks MPC5200 Lite

9. Select **Current Project Target Platform** from the list box of board support package (BSP) options. For this tutorial, it is Metrowerks MPC5200 Lite.

NOTE The options provided in the drop-down list will vary depending on which BSPs you have installed.

10. Click **Finish**. Target Wizard interface screen (Figure B.5) appears.

Figure B.5 Target Wizard Interface


Metrowerks Target Wizard : Project = 'Lite-5200' : Target Platform = 'Metrowerks MPC5200 Lite'
File Search View Project Build Update Tools
Help

Groups, Components, and Options	Value	Disk Size Estimate	Min. Ram Size Estimate	Type
/Linux	True, Enabled	5342084	4762899	Boolean
bootloader	True, Enabled	0	0	Boolean
Communication	False, Disabled	0	0	Boolean
kernel	True, Enabled	0	0	Boolean
Administration	True, Enabled	35458	34126	Boolean
Server	True, Enabled	36701	36280	Boolean
Programming	True, Enabled	197386	258137	Boolean
Textprocessing	False, Disabled	0	0	Boolean
System	True, Enabled	5072537	4434355	Boolean
User Applications	True, Enabled	0	0	Boolean

Name: \ [Group]

✓ Enabled

✓ Fulfilled

✓ No Conflicts

Dependency Conditions	Name	Value	Provides	Parent

Build the Project

1. If your project of choice doesn’t open by default, from the menu bar, choose **Project > Open Recent** or **Project > Open** and browse for and select the project.
2. Select either **Build > Build** or **Build > Force Rebuild** from menu bar.

NOTE It is recommended to use **Force build.** This will build all packages (regardless of whether or not they have changed since the last build) and will include Build Options that are enabled.

Debug the Kernel

To debug PCS kernel, the steps are.

1. Compile kernel with "-O0 -ggdb" or -O1 -ggdb” options. Change the Makefile.
2. If you want to pass command line parameters to kernel, open the **Linux Kernel Boot Parameters** panel and check the **Enable Command Line Settings** checkbox. Set the command line parameters and Base Address. For more information, see “Set Up the Kernel Project for Debugging” on page 141.
3. Rebuild the kernel.

4. The next step is to create a project for the kernel in your CodeWarrior™ IDE. See "Create a CodeWarrior Project for the Kernel" on page 138.

C
Frequently Asked Questions

This appendix discusses the frequently asked questions about the CodeWarrior™ Development Studio for PowerPC ISA, Linux® Application/Platform Edition v2.0.

This appendix has these topics:

- Settings FAQs
- Debugging FAQs
- The CodeWarrior IDE FAQs
- Kernel Debugging FAQs
- Kernel Module Debugging FAQs

Settings FAQs

Question: What is the purpose of the Cache symbolics between runs setting in the Debugger Settings Panel?

Answer: If you check this option, the debugger keeps the symbolics data loaded across debug sessions. Hence, the debugger will not need to load the symbolic data every time in repeat debug sessions provided the symbolic data has not changed. Also, the Console Window will not close between runs.

Question: I have specified the location of a shared library in the Access Paths. But it is not working. Why?

Answer: The path to a shared library needs to be specified in the Libraries settings in the **GNU Linker** settings panel using the -L option. The access paths specified in the **Access Paths** settings panel are not applicable for shared libraries.

Debugging FAQs

Question: The CodeWarrior debugger does not stop at a Log Point set up in a function but stops at a Pause Point. Is this correct?

Answer: The CodeWarrior™ debugger does not stop at a Log Point unless you check the Stop in Debugger setting when setting the Log Point. A Pause Point suspends program execution just long enough to refresh debugger data.

Question: How do I load an executable for exec() system call?

Answer: You need to specify the name of the executable file in the **Other Executables** settings panel. Then, the executable will be downloaded to the target platform and debugged.

However, if the executable is already present on the target platform and its project is not open in the CodeWarrior IDE, you can still use it by selecting it in the **Choose Executables** dialog box that appears when the exec() system call is executed.

Question: What will happen if the executable specified in the exec() system call is not present on the target platform?

Answer: The CodeWarrior debugger will ignore the exec() system call if the corresponding executable in not present on the target platform. You need to build the required sanity checking and messaging in your application.

Question: I am unable to launch an executable using exec() system call from a thread program. The debugger displays the 'MetroTrkProtocolPlugin: Failed to continue thread' message on running my application.

Answer: This issue has been fixed with one limitation that the exec() system call must be in the main thread only.

Question: I am debugging a shared library and pressed F5 to continue inside the shared library code. CodeWarrior IDE crashes. How do I work around it?

Answer: You can avoid the situation as follows:

1. You have the project for the shared library.

   Check the **Cache symbolics between runs** option in the **Debugger Settings** panel. This setting must be done for all build targets being debugged by the process.

   Also, make sure that you do not close the Symbolics Window till the debugging session is complete.

2. You do not have the project for the shared library and it is specified from the **Other Executables** settings panel.

   Make sure that you do not close the Symbolics Window till the debugging session is complete.

The CodeWarrior IDE FAQs

Question: While attaching to a process, why does the System Browser window show the threads as separate processes?

Answer: This is a known limitation of the Linux™ operating system, as the thread information is not available till the debugger attaches to it and reads the Thread Data structures. For a multi-threaded application, you need to separately attach to each thread to debug the threads in the process. This is because the threads are implemented through a user library (Pthreads), and threads run actually as lightweight processes on the system.

Question: Why does a thread creation call fails after a fork() system call?

Answer: When a fork() is called from a thread, the process is created with a single thread. The new process contains a replica of the calling thread and its entire address space, possibly including the states of mutexes and other resources. Consequently, the child process may only execute async-signal-safe operations until exec() is called. Therefore, a thread creation call will fail after a fork() system call.

Question: Sometimes, the console messages are not flushed to stdout. Why?

Answer: This may occur as the proper permissions may not be present for tty* device. You can either provide work permission for tty* device or ensure that you login into the Linux target platform with "su" or root privileges. This is required to exploit the correct functionality of Console I/O, in absence of which you require external fflush() call to display the output on the console window and input echo will not take place.

Question: Why cannot I step out after stepping into a function without symbolic info?

Answer: This is not a bug but is the expected behavior.

Question: Do I need to do anything with AppTRK while restarting the CodeWarrior IDE after a crash?

Answer: When the CodeWarrior IDE crashes due to any reason, we recommend that you restart the AppTRK session on the target platform before restarting the CodeWarrior IDE.

Kernel Debugging FAQs

Question: I want to create kernel project using the ELF drop feature of the CodeWarrior IDE. I have an uncompressed and a compressed kernel image. Which image of the kernel should I drag and drop into the CodeWarrior IDE?

Answer: The uncompressed kernel image because the ELF drop feature of CodeWarrior IDE requires an uncompressed kernel image having the debug symbolics information to create a project.

Question: I have modified the kernel sources on my host computer. Can I rebuild the kernel using the CodeWarrior IDE?

Answer: No. You can rebuild the kernel only on a Linux-hosted computer.

Question: I want to configure the settings for my kernel project in the CodeWarrior IDE. But, the EPPC Debugger Settings panel, Linux Kernel Boot Parameters, and Linux Kernel Debug Settings panel are not displayed in the Target Settings Panels list. What should I do?

Answer: Make sure that you have selected an PowerTAP PRO JTAG or PowerTAP PRO DPI or PowerTAP PRO CCS-based remote connection in the **Remote Debugging** settings panel.

Question: I created a kernel project using the ELF drop feature. I want to open a file in the kernel project directory. I type the name of the file in the Find and Open file dialog box. But, the CodeWarrior IDE opens a file with a similar name existing in some other architecture subdirectory.

Answer: Make sure that you check the **Save project entries using relative path** checkbox in the **Target Settings** panel before you add any new files to the kernel project directory.

Kernel Module Debugging FAQs

Question: I have created a kernel module project with two build targets - one generates an executable binary and the other generates a kernel module file. I want to download and install the kernel module on the target platform. When I select Project > Run, the kernel module does not get download and installed. Why?

Answer: The reasons may be:

- The kernel module project that you created is not using a MetroTRK-based remote connection to connect to the target platform
- The kernel is not booted on the target platform
- The host-side location of the kernel module is not specified in the **Other Executables** settings panel

Question: The Other Executables settings panel is not appearing in the Target Settings Panels list. What shall I do?

Answer: You may have selected a PowerTAP PRO JTAG or PowerTAP PRO DPI, or PowerTAP PRO CCS-based connection in the **Remote Debugging** settings panel. You must select a MetroTRK-based connection for displaying the **Other Executables** settings panel.

Index