CodeWarrior™ Development Studio for ColdFire® Architectures, Linux® Edition Targeting Manual

Revised: 14 June 2006

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals", must be validated for each customer application by customer's technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

How to Contact Us

Corporate Headquarters	Freescale Semiconductor, Inc. 7700 West Parmer Lane Austin, TX 78729 U.S.A.
World Wide Web	`http://www.freescale.com/codewarrior`
Technical Support	`http://www.freescale.com/support`

Table of Contents

1

Introduction

This manual explains how to use the CodeWarrior™ Integrated Development Environment (IDE) to develop software for the embedded Linux® operating system running on ColdFire® hardware. This chapter has these sections:

- Overview of This Manual
- Related Documentation
- CodeWarrior Compiler Architecture
- CodeWarrior Development Tools
- CodeWarrior Development Process
- Supported Target Boards

Overview of This Manual

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

Table 1.1 Manual Contents

Chapter	Description
Introduction	(this chapter)
Working With Projects	describes how to create embedded Linux projects with the CodeWarrior IDE
Working With the Debugger	describes how to use the CodeWarrior tools to debug embedded Linux programs on ColdFire hardware
Debugging Boot Loaders, Kernels, Modules, and Threads	describes how to use the CodeWarrior IDE to debug boot loaders, kernels, kernel modules, and kernel threads
Target Settings Reference	describes the various target settings in all CodeWarrior projects

Table 1.1 Manual Contents (*continued*)

Chapter	Description
Working With Hardware Tools	describes how to use the CodeWarrior IDE hardware tools for board bring-up, test, and analysis
Shell Tool Post-Linker	shows how to automatically run shell scripts as part of the IDE's build process
Third Party Cross Compiler Tools	describes how to use third-party compiler tools to build CodeWarrior projects
Debug Initialization Files	describes the syntax of debug initialization files you can use to initialize target boards before the debugger downloads code to them
Memory Configuration Files	describes the syntax of memory initialization files that define the accessible areas of memory for target boards
Frequently Asked Questions	gives answers to common questions about this product

Related Documentation

This section provides information about documentation, web sites, and example source code related to the CodeWarrior IDE and Embedded PowerPC development.

CodeWarrior Information

- 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 here (where, `CWInstall` is the directory where you installed the CodeWarrior IDE software):

  CWInstall`/CodeWarriorIDE/Release_Notes/`

- For system requirements and instructions showing how to install this CodeWarrior product, refer the *Quick Start* located in the `CWInstall/CodeWarriorIDE/` directory, where `CWInstall` is the directory where you installed the CodeWarrior IDE software.
- For general information about the CodeWarrior IDE and debugger, read the *CodeWarrior IDE User's Guide*.

- For information specific to the C/C++ front-end compiler, see the *C Compilers Reference*.
- For information scripting the CodeWarrior IDE, see the *CodeWarrior IDE Automation Guide* manual.
- To learn how to write device drivers for Linux systems, see:

  `http://www.xml.com/ldd/chapter/book/`
- Look for the CodeWarrior tutorials projects on the installation CD.

CodeWarrior Compiler Architecture

A proprietary, multi-language, multi-target compiler architecture is at the heart of the CodeWarrior IDE. Front-end language compilers generate a memory-resident, unambiguous, language-independent intermediate representation (IR) of syntactically correct source code. Back-end compilers generate code from the IR for specific targets. The CodeWarrior IDE manages the whole process.

CodeWarrior plug-in compilers generate object code. CodeWarrior plug-in linkers generate final executable files from the object code. Multiple linkers that support different object code formats are available for some targets.

As a result of this architecture, the same front-end compiler is used to support multiple back-end compilers. In some cases, the same back-end compiler can generate code from a variety of languages.

All CodeWarrior compilers and linkers are built as plug-in modules. The interface between the IDE and compilers and linkers is public; so third parties can create compilers that work with the CodeWarrior IDE.

CodeWarrior Development Tools

With the CodeWarrior Integrated Development Environment (IDE), programming for embedded Linux® on a supported target platform is much like programming for any other target platform. If you have never used the CodeWarrior IDE, then you should read this section.

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 11 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.

The **GNU Tools** settings panel lets you select the cross compilers and linkers used by the CodeWarrior IDE. For more information about this settings panel, see "GNU Tools" on page 137.

"Target Settings Reference" on page 117 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, see "Working With the Debugger" on page 23.

CodeWarrior Target Resident Kernel

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

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

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

For more information about CodeWarrior TRK, see "Using CodeWarrior Target-Resident Kernel" on page 31.

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 about 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

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 Tools” on page 137 and “GNU Linker” on page 127 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 CodeWarrior TRK on the target.

For details, see “Using CodeWarrior Target-Resident Kernel” on page 31.

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.

Supported Target Boards

Table 1.2 lists the target boards supported by this product.

Table 1.2 Supported Target Boards

Manufacturer	Boards
Freescale	MCF5329EVB
	M5208EVB
	M5272C3
	M5282EVB
	M5475EVB
	M5485EVB
	M5474LITEKIT
	M5484LITEKIT

2

Working With Projects

This chapter explains how to create embedded Linux projects with the CodeWarrior IDE. This chapter contains these sections:

- Creating Projects
- Importing Makefile Projects
- Sample Projects

Creating Projects

This section explains how to use the **Linux Stationery Wizard** to create a new project. After you create the project, you can modify project settings, and compile, run, and debug the code in the project.

1. Run the CodeWarrior IDE startup script in this location:

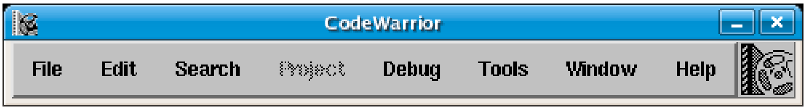
   CWInstall/CodeWarriorIDE/cwide

NOTE *CWInstall* is the location where you installed this product. For example, if you installed the product at /usr/local/Freescale/CW_ColdFire_2.2/, the path to the cwide script would be:

```
/usr/local/Freescale/CW_ColdFire_2.2/CodeWarriorIDE/
cwide
```

   The CodeWarrior menu bar (Figure 2.1) appears.

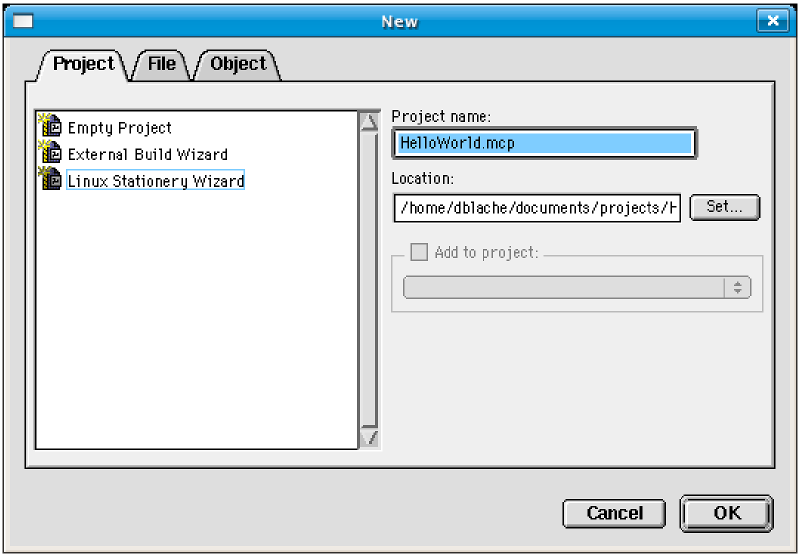
Figure 2.1 CodeWarrior Menu Bar



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

   The **New** dialog box (Figure 2.2) appears.

Figure 2.2 New Dialog Box



Project name: HelloWorld.mcp

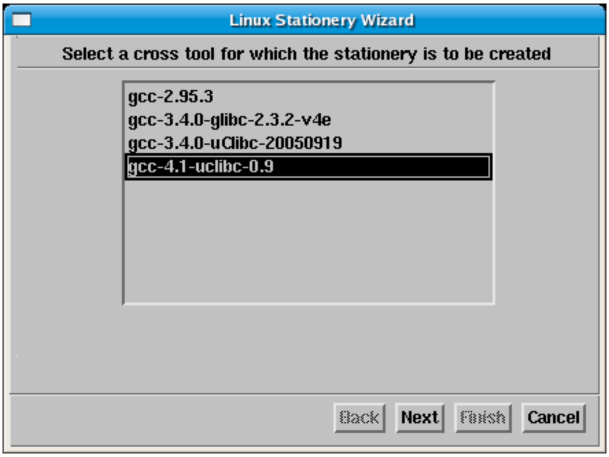
Location: /home/dblache/documents/projects/H

□ Add to project:

3. Select **Linux Stationery Wizard**.
4. In the **Project name** text box, enter a name for the project, such as `MyKillerApplication.mcp`.
5. In the **Location** text box, enter the full path to the folder where you want the IDE to create the new project (or click the **Set** button to navigate to and select a location).
6. Click **OK**.

   The **Linux Stationery Wizard** (Figure 2.3) appears.

Figure 2.3 Wizard — Cross Tool Page



Linux Stationery Wizard

Select a cross tool for which the stationery is to be created

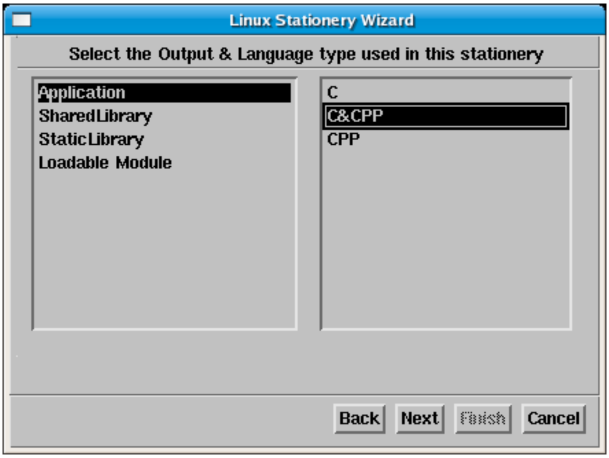
gcc-2.95.3
gcc-3.4.0-glibc-2.3.2-v4e
gcc-3.4.0-uClibc-20050919
gcc-4.1-uclibc-0.9

7. From the cross tools list, select the cross compiler tool the CodeWarrior IDE should use to build the project.

8. Click **Next**.

   The **Output Type and Language** page (Figure 2.4) appears.

Figure 2.4 Wizard — Output Type and Language Page



Linux Stationery Wizard

Select the Output & Language type used in this stationery

Application
SharedLibrary
StaticLibrary
Loadable Module

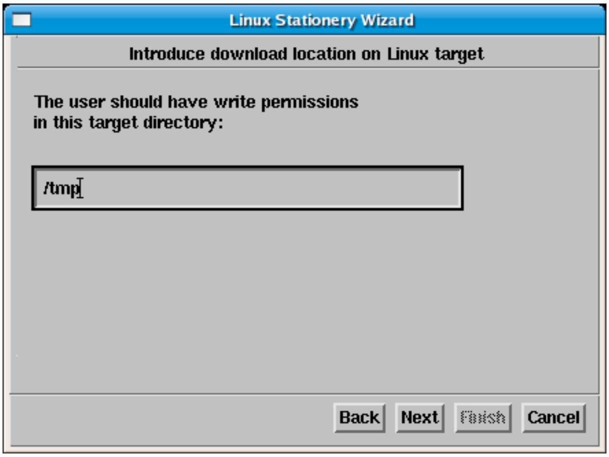
C
C&CPP
CPP

9. Select the output type and the programming language you want to use for this project.

10. Click **Next**.

    The **Download Location** page (Figure 2.5) appears.

Figure 2.5 Wizard — Download Location Page



Introduce download location on Linux target

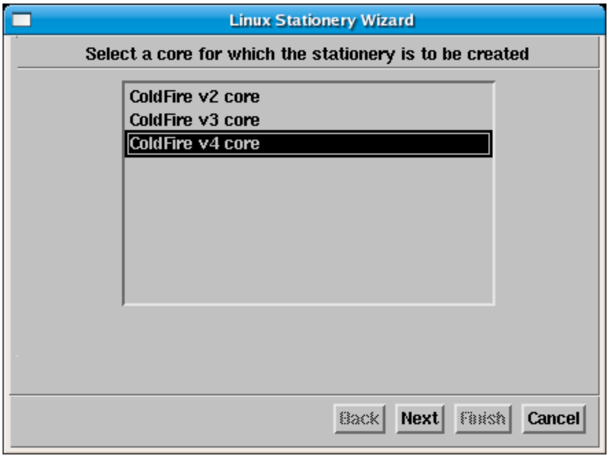
The user should have write permissions in this target directory:

/tmp

11. In the text box, enter the full path, on the target system, to the folder where you want the IDE to place the executable files it generates when you build the project.
12. Click **Next**.

    The **Core Selection** page (Figure 2.6) appears.

Figure 2.6 Wizard — Core Selection Page



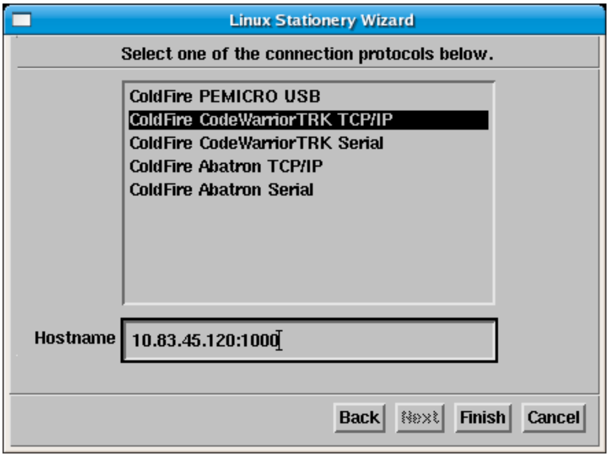
Select a core for which the stationery is to be created

ColdFire v2 core
ColdFire v3 core
ColdFire v4 core

13. From the list, select the core on the target system.
14. Click **Next**.

    The **Connection** page(Figure 2.7) appears.

Figure 2.7 Wizard — Connection Page



Hostname 10.83.45.120:1000

15. From the list, select the method by which the IDE should connect to the target system.

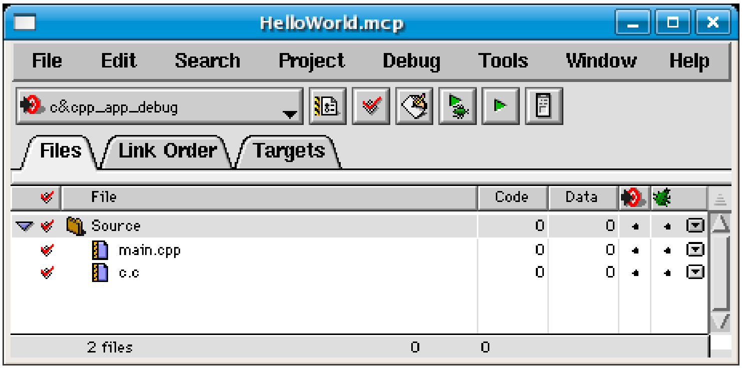
16. In the **Hostname** text box, enter the IP address and listening port of the target system, in this format:

    `IPAddress:PortNumber`

17. Click **Finish**.

    The **Linux Stationery Wizard** window disappears. The IDE generates a new project according to your specifications. The project window (Figure 2.8) appears.

Figure 2.8 Project Window



	File	Code	Data
✓	Source	0	0
✓	main.cpp	0	0
✓	c.c	0	0
	2 files	0	0

Importing Makefile Projects

The **External Build Wizard** lets you import Makefile-based projects into CodeWarrior IDE projects so that you can use the IDE to manage and debug the projects. When you invoke this wizard, it prompts you for information about the makefile you want to import. The wizard then collects data about the make file and creates a CodeWarrior project with a single target configured to build the user-specified make file.

To learn more about the External Build Wizard, read the *CodeWarrior IDE User's Guide* in this folder:

CWInstall`/Help/PDF`

Sample Projects

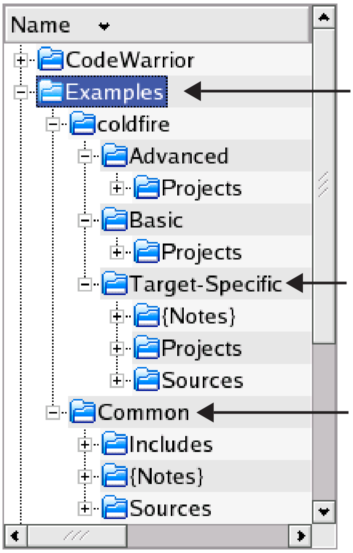
We have provided ready-made projects, containing all the required settings for successfully running and debugging code on ColdFire target systems. These sample projects may help you to understand the features and capabilities of this product.

The examples are located here:

`CWInstall``/CodeWarriorIDE/Examples`

Figure 2.9 shows the directory structure of the `Examples` folder.

Figure 2.9 Examples Directory Structure



application-level example projects for ColdFire-based target platforms

target-specific example projects such as kernel modules for ColdFire-based target platforms

source files, header files, and notes common to the application-level example projects

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:

- all executable files must be downloaded to the `/var` directory
- CodeWarrior TRK should be executed by the user `sample`
- a properly-configured remote connection

Table 2.1 shows some of the sample application projects and the kernel module project available in your CodeWarrior installation directory. For a complete list, examine the `Examples` directory in a file browser.

Table 2.1 Example Projects and Their Location

File Name	Location	Description
Beginners.mcp	*CWInstall*/CodeWarriorIDE/Examples/coldfire/Basic/Projects	application
ForkAndExec.mcp	*CWInstall*/CodeWarriorIDE/Examples/coldfire/Advanced/Projects/	Fork() and Exec ()
KernelModule.mcp	*CWInstall*/CodeWarriorIDE/Examples/coldfire/Target-Specific/Projects/	Kernel module

Table 2.2 shows where you can find the header and source files for the above mentioned sample Linux application projects.

Table 2.2 Header and Source Files for Sample Projects

Source files	*CWInstall*/CodeWarriorIDE/Examples/Common/Sources
Header files	*CWInstall*/CodeWarriorIDE/Examples/Common/Includes

NOTE For information about how to work with the sample projects, read the project notes at: `*CWInstall*/Examples/Common/{Notes}`.

Table 2.3 Source Files for Sample Kernel Module Project

Source files	*CWInstall*/CodeWarriorIDE/Examples/ColdFire/Target-Specific/Sources

NOTE For information about how to work with the sample kernel module project, see the project notes located at: `*CWInstall*/Examples/ColdFire/Target-Specific/{Notes}`.

3

Working With the Debugger

This chapter explains how to use the CodeWarrior tools to debug embedded Linux® programs on ColdFire® hardware.

NOTE The chapter covers those aspects of debugging that are specific to the ColdFire platform. Refer to the *CodeWarrior IDE User's Guide* for debugger information that applies to all CodeWarrior products.

This chapter contains these sections:

Using Remote Connections

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

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

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

Accessing Remote Connections

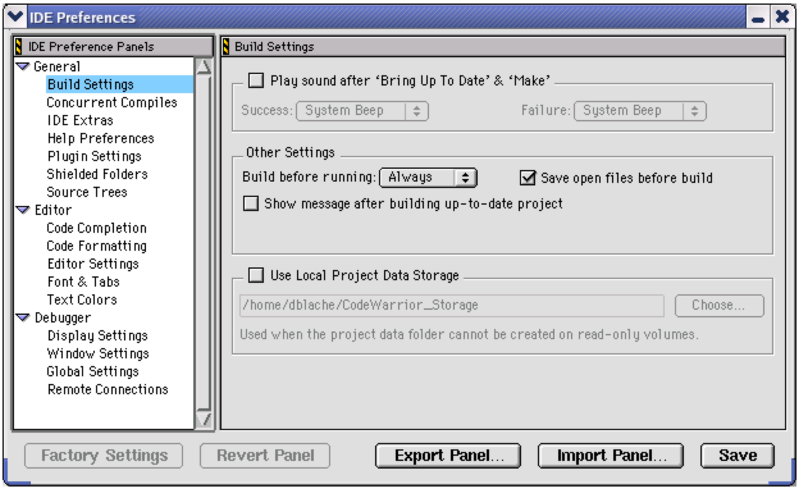
You access remote connections in the CodeWarrior **IDE Preferences** window. Remote connections listed in the preferences window are available for use in all CodeWarrior projects and build targets.

To access remote connections:

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

   The **IDE Preferences** window (Figure 3.1) appears.

Figure 3.1 IDE Preferences Window



□ Play sound after 'Bring Up To Date' & 'Make'

Success: System Beep   Failure: System Beep

Other Settings

Build before running: Always   ☑ Save open files before build

□ Show message after building up-to-date project

□ Use Local Project Data Storage

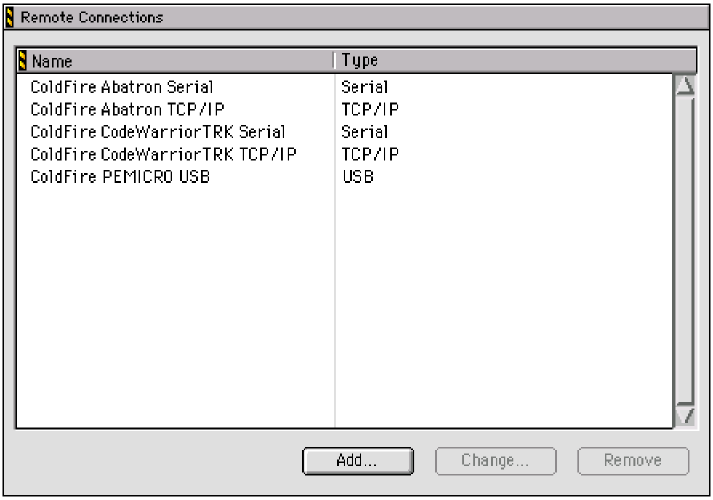
/home/dblache/CodeWarrior_Storage

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

2. From the **IDE Preference Panels** list, select **Remote Connections**.

   The **Remote Connections** preference panel (Figure 3.2) appears.

Figure 3.2 Remote Connections Preference Panel



Name	Type
ColdFire Abatron Serial	Serial
ColdFire Abatron TCP/IP	TCP/IP
ColdFire CodeWarriorTRK Serial	Serial
ColdFire CodeWarriorTRK TCP/IP	TCP/IP
ColdFire PEMICRO USB	USB

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

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

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

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

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

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

Understanding Remote Connections

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

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

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

Table 3.1 describes each of the supported debugger protocols.

Table 3.1 Debugger Protocols

Debugger Protocol	Description
ColdFire Abatron	Select to use serial or TCP/IP connections and an Abatron device with and debug a target system.
CF Linux CodeWarrior TRK	Select to use a serial or TCP/IP connection with CodeWarrior TRK to debug a target system.
ColdFire PEMicro	Select to use a USB connection with a P&E Microcomputer Systems USB device to debug a target system.

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

Editing Remote Connections

Based on the specified debugger protocol and connection type, the IDE makes different settings available to you. For example, if you specify a **Serial** connection type, the IDE presents settings for baud rate, stop bits, flow control, and so on. Table 3.2 describes the supported connection types for each debugger protocol.

Table 3.2 Supported Connection Types

Debugger Protocol	Supported Connection Types
ColdFire Abatron	Serial, TCP/IP
CF Linux CodeWarrior TRK	Serial, TCP/IP
ColdFire PEMicro	USB

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

- In the **Remote Connections** IDE preference panel, select a connection from the list, and click **Edit**. The **Edit Connection** dialog box appears.

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

This section describes the settings for each connection type:

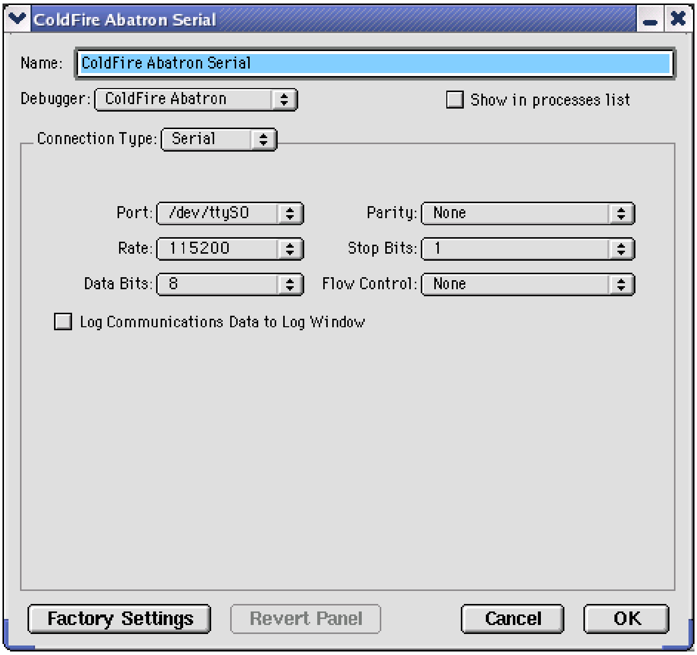
- Serial
- TCP/IP
- USB

Serial

Use this connection type to configure how the IDE uses the serial interface of the host computer to connect with the target system. This connection type is available when the **ColdFire Abatron** or **CF Linux CodeWarrior TRK** debugger protocol is selected.

Figure 3.3 shows the settings that are available to you when you select **Serial** from the **Connection Type** list box in the **Edit Connection** dialog box.

Figure 3.3 Serial Connection Settings



ColdFire Abatron Serial

Name: ColdFire Abatron Serial

Debugger: ColdFire Abatron

□ Show in processes list

Connection Type: Serial

Port: /dev/ttyS0

Parity: None

Rate: 115200

Stop Bits: 1

Data Bits: 8

Flow Control: None

□ Log Communications Data to Log Window

Table 3.3 describes the options in this dialog box.

Table 3.3 Serial Options

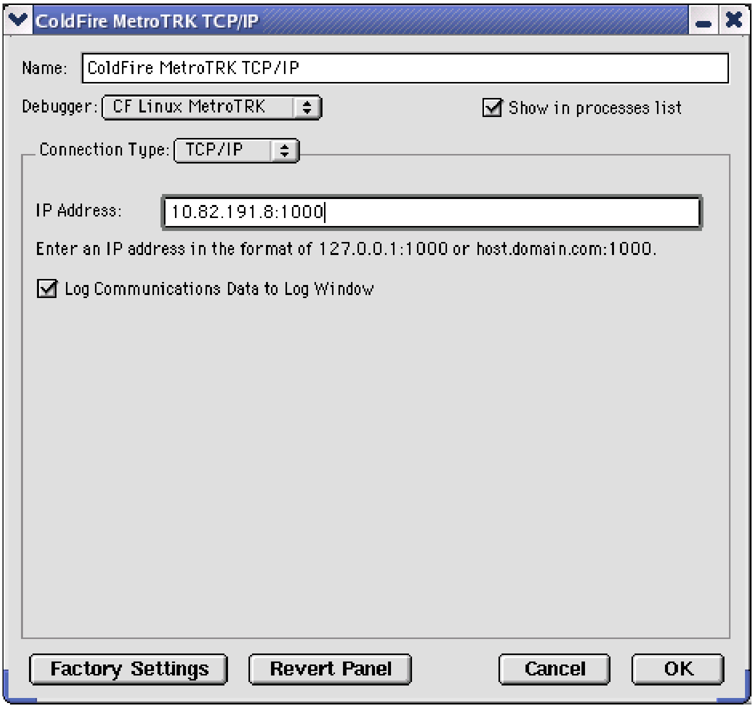
Option	Description
Name	Enter the name you want to use to refer to this remote connection within the CodeWarrior IDE.
Debugger	Select **ColdFire Abatron** or **CF Linux CodeWarrior TRK**.
Connection Type	Select **Serial**.
Port	Select the serial port device to which the target system is connected on the host computer.
Rate	For Abatron device connections, select the communication rate that the device supports. For CodeWarrior TRK connections, select the CodeWarrior TRK communication rate on the target system.
Data Bits	Select the number of data bits the IDE should use when it communicates with the target system.
Parity	Select the parity the IDE should use when it communicates with the target system.
Stop Bits	Select the stop bits the IDE should use when it communicates with the target system.
Flow Control	Select the flow control the IDE should use when it communicates with the target system.

TCP/IP

Use this connection type to configure how the IDE uses the TCP/IP protocol to connect with the target system. This connection type is available when the **ColdFire Abatron** or **CF Linux CodeWarrior TRK** debugger protocol is selected.

Figure 3.4 shows the settings that are available to you when you select **TCP/IP** from the **Connection Type** list box in the **Edit Connection** dialog box.

Figure 3.4 TCP/IP Connection Settings



ColdFire MetroTRK TCP/IP

Name: ColdFire MetroTRK TCP/IP

Debugger: CF Linux MetroTRK

☑ Show in processes list

Connection Type: TCP/IP

IP Address: 10.82.191.8:1000

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

☑ Log Communications Data to Log Window

Table 3.4 describes the options in this dialog box.

Table 3.4 TCP/IP Options

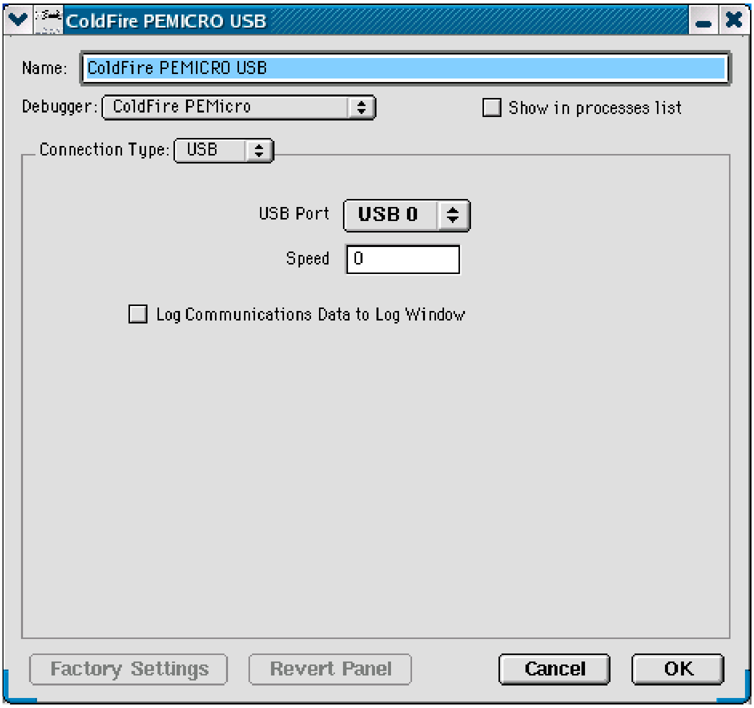
Option	Description
Name	Enter the name you want to use to refer to this remote connection within the CodeWarrior IDE.
Debugger	Select **ColdFire Abatron** or **CF Linux CodeWarrior TRK**.
Connection Type	Select **TCP/IP**.
IP Address	Enter the Internet Protocol (IP) address and listening port number assigned to the target system, in the form: *IPAddress*:*PortNumber*

USB

Use this connection type to configure how the IDE uses the Universal Serial Bus (USB) interface of the host computer to connect with the target system. This connection type is available only when the **ColdFire PEMicro** debugger protocol is selected.

Figure 3.3 shows the settings that are available to you when you select **USB** from the **Connection Type** list box in the **Edit Connection** dialog box.

Figure 3.5 USB Connection Settings



ColdFire PEMICRO USB

Name: ColdFire PEMICRO USB

Debugger: ColdFire PEMicro

□ Show in processes list

Connection Type: USB

USB Port: USB 0

Speed: 0

□ Log Communications Data to Log Window

Table 3.3 describes the options in this dialog box.

Table 3.5 Serial Options

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

Table 3.5 Serial Options (*continued*)

Option	Description
Connection Type	Select **USB**.
USB Port	Select the USB port device to which the target system is connected on the host computer.
Show In Processes List	Check to have the IDE display processes for this debug session in the **System Browser** window.
Speed	Enter an integer value, in the range 0-31, representing the data stream transfer rate. The debugger calculates the transfer speed, in hertz, using this expression: `1000000 / (Speed + 1) = Hertz` For example, if you specify `1`, the debugger calculates: `1000000 / (1+1) = 500000` (0.5 megahertz) If you specify `31`, the debugger calculates: `1000000 / (31+1) = 31250` (0.031 megahertz, the slowest transfer rate)
Log Communications Data to Log Window	Check to have the IDE display communications data in a log window when you use this connconnection.

Using CodeWarrior Target-Resident Kernel

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

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

CodeWarrior TRK is a user-level application for use with the CodeWarrior debugger. You use CodeWarrior TRK to download and debug applications built with the CodeWarrior IDE. On the host computer, the CodeWarrior debugger connects to CodeWarrior TRK running on the target system via an ethernet link or serial port. For an overview of remote debugging, read the *CodeWarrior IDE User's Guide*. For an example of how the process works, see "Debugging Remote Executable Files" on page 33.

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

Customizing CodeWarrior TRK

You may customize the CodeWarrior TRK source code and recreate the CodeWarrior TRK binary for your specific needs. You can either make a copy of the project (and its associated source files) or directly edit the original source.

The CodeWarrior installer places target-specific versions of the CodeWarrior TRK source files in the CodeWarrior installation directory.

The CodeWarrior TRK project has build targets for:

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

NOTE While we recommend that you build the CodeWarrior TRK binary as explained in this section, you can also use the pre-built CodeWarrior TRK binary available in your CodeWarrior installation directory.

Project and Binary Files

Table 3.6 lists the location where you can find the CodeWarrior TRK project and binary files applicable for your target platform.

Table 3.6 CodeWarrior TRK Project and Binary File Location

Type	Available at

Binary Files	`CodeWarriorIDE/CodeWarrior/ThirdPartyTools/`
	`TargetBoardDir``/AppTrk_BINARY`

Project	`CodeWarriorIDE/CodeWarrior/ColdFire_Tools/`
	`CodeWarriorTRK/Os/unix/linux/cf/`
	`trk_linux_cf.mcp`

Table 3.7 lists the build targets available in the CodeWarrior TRK project.

Table 3.7 CodeWarrior TRK Project Build Targets

Build Target Name	Description
APP_TRK_mcf5272_5282[D]	debug version for MCF5272 and MCF5282
APP_TRK_mcf5272_5282[R]	release version for MCF5272 and MCF5282

Table 3.7 CodeWarrior TRK Project Build Targets (*continued*)

Build Target Name	Description
APP_TRK_mcf5475_5485[D]	debug version for MCF5475 and MCF5485
APP_TRK_mcf5475_5485[R]	release version for MCF5475 and MCF5485
APP_TRK_mcf5208[D]	debug version for MCF5208
APP_TRK_mcf5208[R]	release version for MCF5208
APP_TRK_mcf5329[D]	debug version for MCP5329
APP_TRK_mcf5329[R]	release version for MCP5329
build_all	all versions

Installing CodeWarrior TRK On Remote Systems

To use CodeWarrior TRK for debugging, you must install and launch the compiled binary file on a remote system. After you have launched CodeWarrior TRK 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 CodeWarrior TRK on the remote target system, you need to download the CodeWarrior TRK 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 CodeWarrior TRK binary file from the host computer to the root file system of the remote target system.

The procedure for launching CodeWarrior TRK is covered in "Start CodeWarrior TRK on the Remote Target" on page 37.

Debugging Remote Executable Files

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" on page 34

- “Specify Remote Debugging Options” on page 36
- “Start CodeWarrior TRK on the Remote Target” on page 37
- “Start the Debugger” on page 39

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 about the **Remote Connections** preference panel, refer to the *CodeWarrior IDE User’s Guide*.

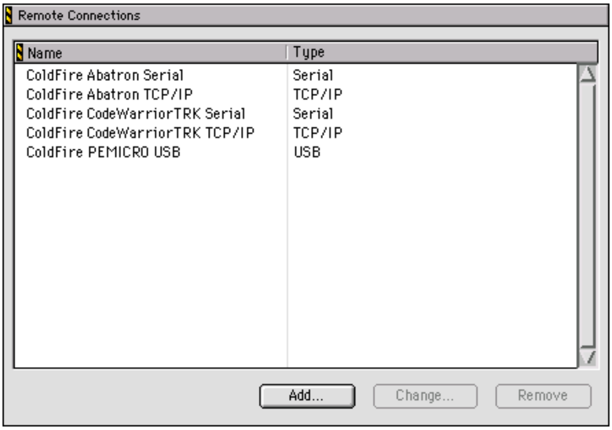
The steps to define a remote connection are as follows:

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 3.6).

Figure 3.6 Remote Connections Preference Panel



Name	Type
ColdFire Abatron Serial	Serial
ColdFire Abatron TCP/IP	TCP/IP
ColdFire CodeWarriorTRK Serial	Serial
ColdFire CodeWarriorTRK TCP/IP	TCP/IP
ColdFire PEMICRO USB	USB

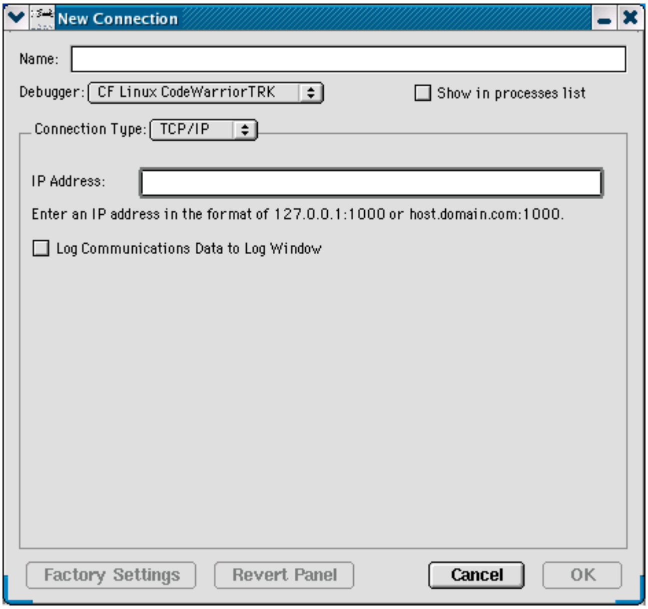
2. Add a new remote connection.

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 3.7) display changes.

Figure 3.7 New TCP/IP Connection



New Connection

Name:

Debugger: CF Linux CodeWarriorTRK

☐ Show in processes list

Connection Type: TCP/IP

IP Address:

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

☐ Log Communications Data to Log Window

NOTE The **Debugger** list box displays the target platform-specific CodeWarrior TRK name. For example, **CF Linux CodeWarrior TRK** for ColdFire target platform.

c. Type the remote connection name in the **Name** text box. You will use this name to identify the remote connection in other CodeWarrior IDE windows and dialog boxes.

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 CodeWarrior TRK. For example, if the IP address is `127.0.0.1` and the port number is `6969`, type `127.0.0.1:6969`.

e. Check the **Show in processes list** checkbox.

f. Save the new remote connection.

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

h. Click **Save**.

i. 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.

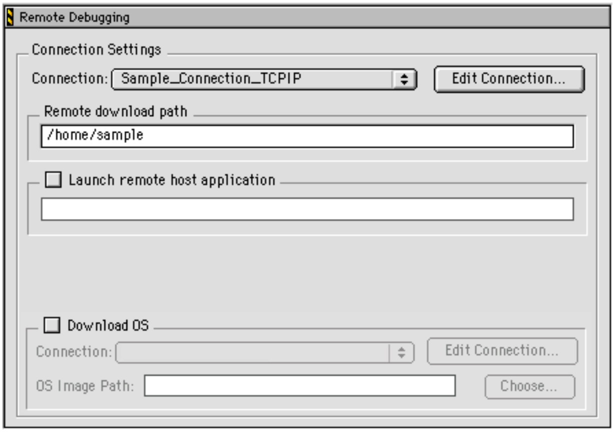
   If the project has a debug build target, switch to the debug build target. Select the target name from the build target list box in the project window.

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 3.8) appears.

Figure 3.8 Remote Debugging Settings Panel



Remote Debugging

Connection Settings

Connection: Sample_Connection_TCPIP

Remote download path

/home/sample

☐ Launch remote host application

☐ Download OS

Connection:

OS Image Path:

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 34.

d. In the **Remote download path** text box, specify the location where the executable binary is to reside on the remote target system. CodeWarrior TRK 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.

e. Ensure that external debugging is disabled.

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

Start CodeWarrior TRK on the Remote Target

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

Start CodeWarrior TRK Using TCP/IP Connection

To launch CodeWarrior TRK 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 CodeWarrior TRK binary file.

   Enter the command `cd /`*`TRKDir`* (where *`TRKDir`* is the name of the target-system directory where you downloaded the CodeWarrior TRK binary file). The current directory changes.
3. Launch CodeWarrior TRK on the remote target system.

   Type `./`*`TRKBinary`* `:`*`port`*, where *`TRKBinary`* is the name of the target-specific CodeWarrior TRK binary file and *`port`* is the TCP/IP port number you specified while creating a remote connection. For example, type `./AppTrk.elf :6969`.
4. Press Enter. CodeWarrior TRK starts on the remote target system.

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

Start CodeWarrior TRK 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 CodeWarrior TRK.

To launch CodeWarrior TRK 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 CodeWarrior TRK binary file.

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

4. Launch CodeWarrior TRK on the remote target platform.

   Type `./`*`TRKBinary`*`/dev/ttyS1` command in the Terminal console and press Enter. CodeWarrior TRK 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 CodeWarrior TRK process
3. transfers the executable file to the remote system
4. launches the executable file
5. starts the debugger

Debugging Shared Libraries

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 3.8):

Table 3.8 Shared Library Project Settings

Project Name:	`SharedLibrary_Example`
Project Location:	`/home/usr1/SharedSample`
Languages:	C

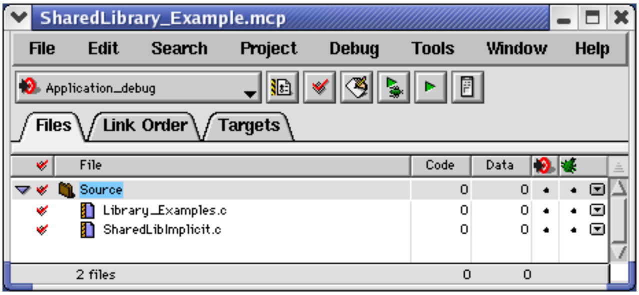
Table 3.8 Shared Library Project Settings (*continued*)

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 about how to create or remove build targets, refer the *CodeWarrior IDE User's Guide*.

2. Remove the default `main.c` file and add the source files (`SharedLibImplicit.c` and `Library_Examples.c`) to the project. The project window appears as shown in Figure 3.9.

Figure 3.9 Source Files Added to the SharedLibrary_Example.mcp Project



	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 3.1 into the editor window of `LibExample.h` file.

Listing 3.1 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 3.2 into the editor window of `CWExample.h` file.

Listing 3.2 Source Code for CWExample.h

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

6. Enter the source code of Listing 3.3 into the editor window of `SharedLibImplicit.c` file.

Listing 3.3 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 3.4 into the editor window of `Library_Examples.c` file.

Listing 3.4 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 path 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.

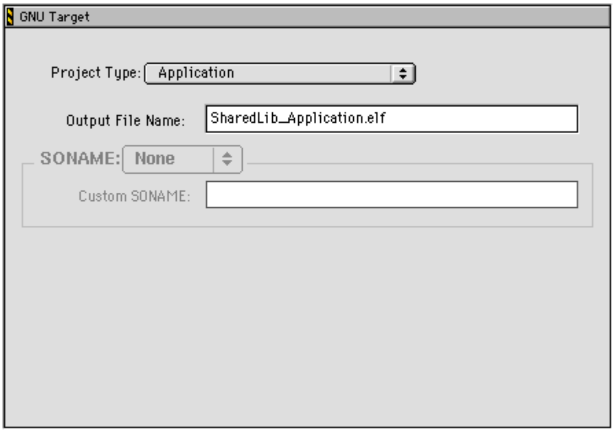
Configure the Executable Build Target

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 3.10) appears.

Figure 3.10 GNU Target Settings Panel



GNU Target

Project Type: Application

Output File Name: SharedLib_Application.elf

SONAME: None

Custom SONAME:

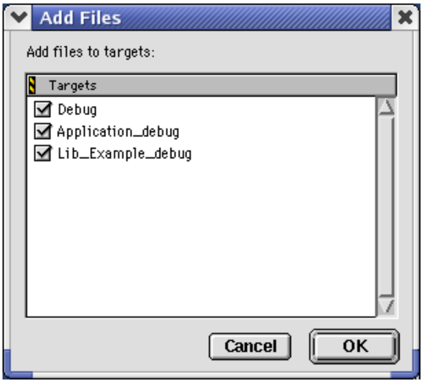
   c. Make sure that the **Output File Name** text box displays the name of the final executable binary as `SharedLib_Application.elf`.
3. Add `LibExample.so` file to the **Application_debug** build target.

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 3.11) appears.

Figure 3.11 Add Files Dialog Box



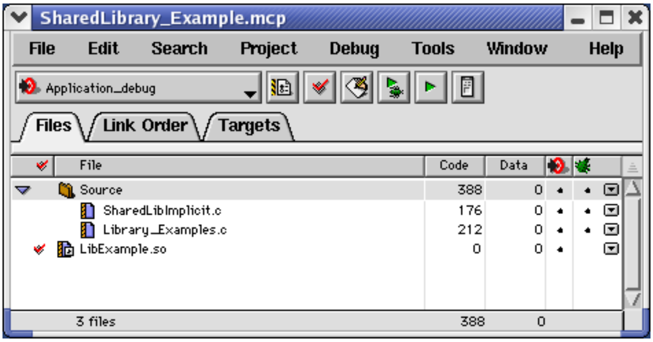
Add files to targets:

Targets
☑ Debug
☑ Application_debug
☑ Lib_Example_debug

d. Clear the checkbox 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 3.12).

Figure 3.12 LibExample.so Added to the Application_debug Build Target

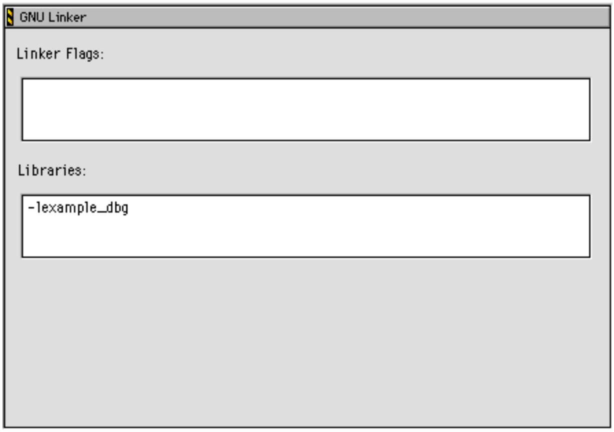


✓	File	Code	Data			
	Source	388	0	•	•	
	SharedLibImplicit.c	176	0	•	•	
	Library_Examples.c	212	0	•	•	
✓	LibExample.so	0	0	•		
	3 files	388	0			

4. Specify the linker settings.

a. Click **GNU Linker** in the **Target Settings Panels** list. The **GNU Linker** settings panel (Figure 3.13) appears.

Figure 3.13 GNU Linker Settings Panel



```
-lexample_dbg
```

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

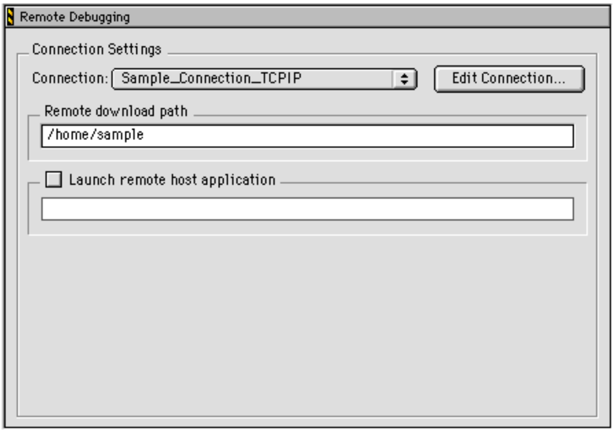
`-lexample_dbg`

NOTE The `-lexample_dbg` linker command line argument enables the CodeWarrior IDE linker to locate the shared library `LibExample.so`. For detailed information about 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 3.14) appears.

Figure 3.14 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.

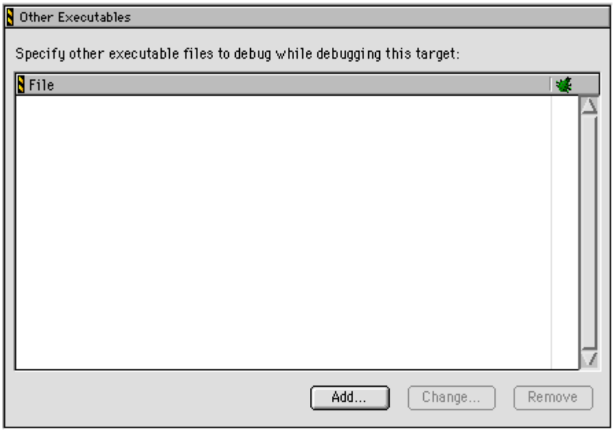
NOTE For this tutorial, the remote download path is specified as `/home/sample`. If you wish, you may specify an alternate remote download path for the executable file.

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 3.15) appears.

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

Figure 3.15 Other Executables Settings Panel

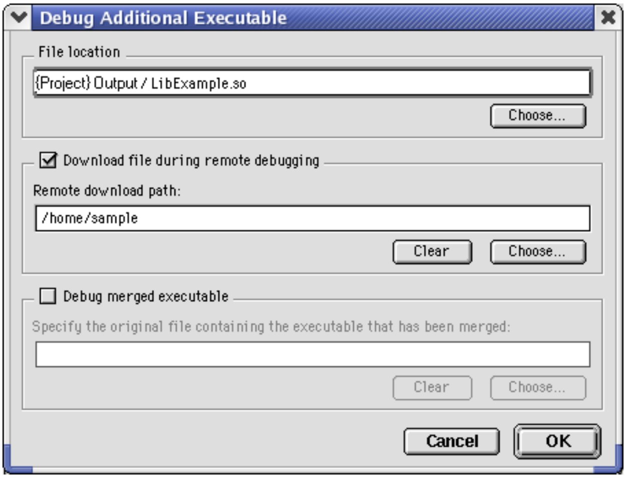


Other Executables

Specify other executable files to debug while debugging this target:

b. Click **Add**. The **Debug Additional Executable** dialog box (Figure 3.16) appears.

Figure 3.16 Debug Additional Executable Dialog Box



Debug Additional Executable

File location

{Project} Output / LibExample.so

☑ Download file during remote debugging

Remote download path:

/home/sample

□ Debug merged executable

Specify the original file containing the executable that has been merged:

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.

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

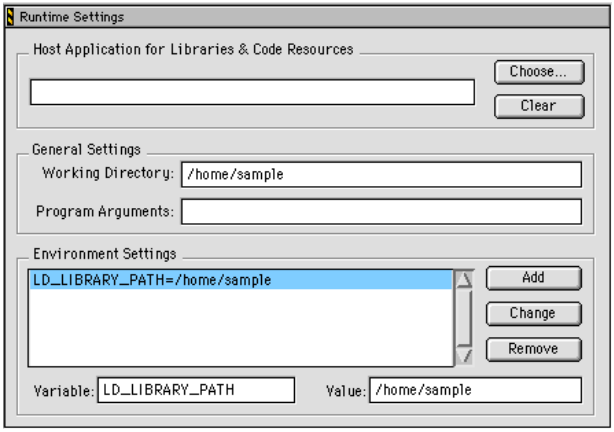
At runtime, the shared object loader first searches for a shared library in the path specified by the `LD_LIBRARY_PATH` environment variable's value. In this case, the value of this environment variable will be `/home/sample`, which is the remote download path for the shared library you specified in 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 3.17).

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

Figure 3.17 Runtime Settings Panel



Working Directory: /home/sample

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:

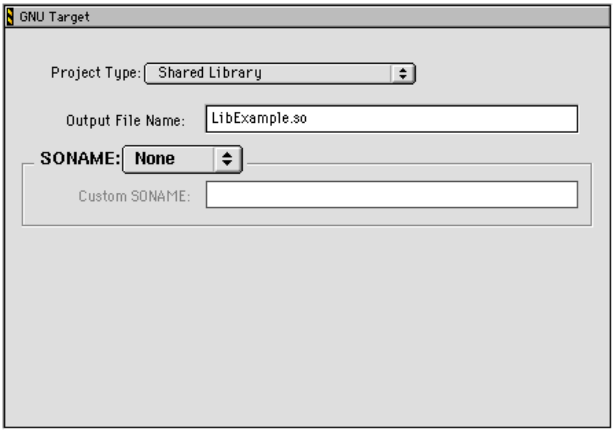
- verifying the final output file name
- specifying the host-side location of the executable file to be used for debugging the shared library
- specifying remote debugging options

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 3.18) appears.

Figure 3.18 GNU Target Settings Panel



GNU Target

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.

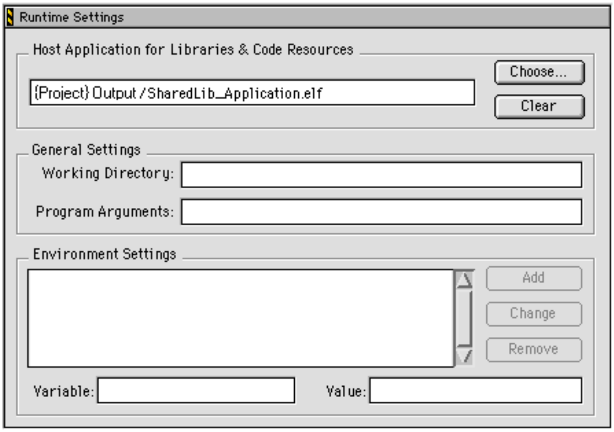
   c. Navigate to the location where you have stored the `SharedLib_Application.elf` file in your project directory. For this tutorial it is: `/home/usr1/SharedSample/Output`.

   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 3.19).

Figure 3.19 SharedLib_Application.elf Selected



Host Application for Libraries & Code Resources: {Project} Output/SharedLib_Application.elf

   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.

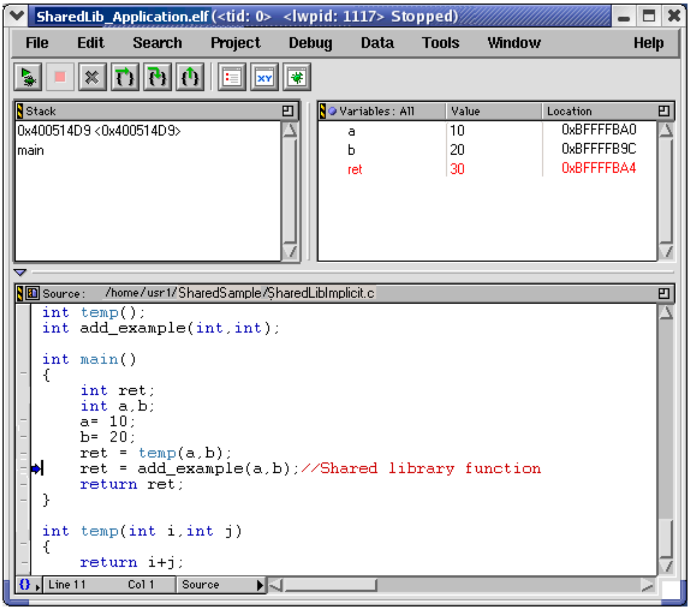
   Select **Project > Make**. The library is built with the new settings.

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 3.20) and symbolics (Figure 3.21) windows appear.

Figure 3.20 Debugger Window



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

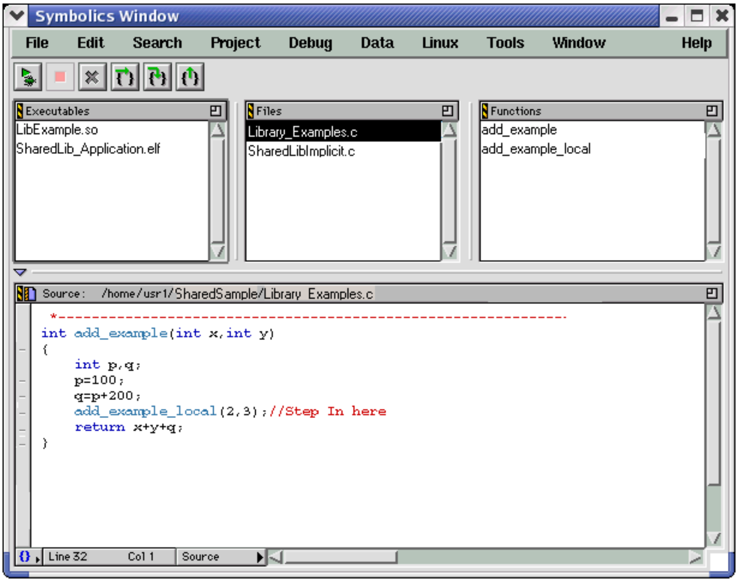
Variables: All	Value	Location
a	10	0xBFFFFBA0
b	20	0xBFFFFB9C
ret	30	0xBFFFFBA4

Source: /home/usr1/SharedSample/SharedLibImplicit.c

```
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 3.21 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 about symbolics window, see the *CodeWarrior IDE User's Guide*.

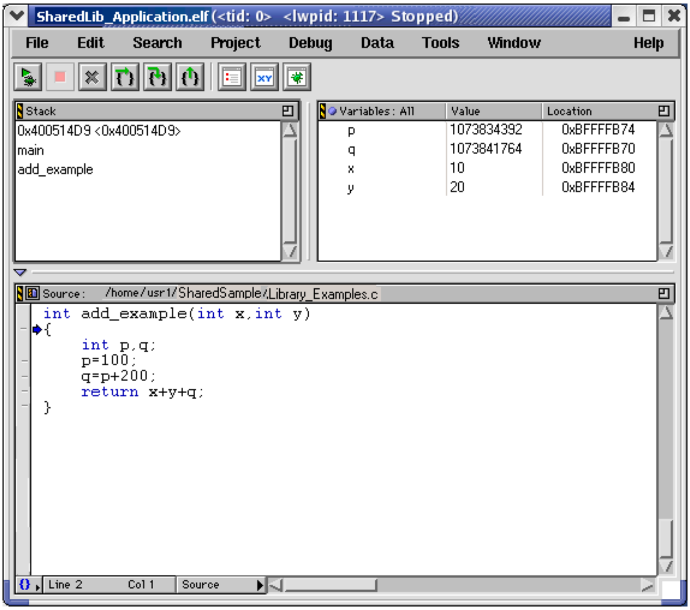
3. Step Over the code.

   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 3.22).

Figure 3.22 Source Code of Library_Examples.c File


SharedLib_Application.elf (<tid: 0> <lwpid: 1117> Stopped)
File
Edit
Search
Project
Debug
Data
Tools
Window
Help
Stack
Variables : All
Source : /home/usr1/SharedSample/Library_Examples.c
Line 2
Col 1
Source

```
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.

7. Click the Run button. The rest of the code is executed and the output appears in the CodeWarrior TRK Console window. You may also use the sample shared library project available in the CodeWarrior installation directory. For more information about sample projects, see “Sample Projects” on page 20.

Debugging Multiple Threads

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 *CodeWarrior 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 81.

The tutorial that follows, demonstrates multi-threaded debugging.

1. Create a new project with the following settings (Table 3.9):

Table 3.9 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 3.5 into the editor window of `main.c` file.

Listing 3.5 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
-------------------------*/
```

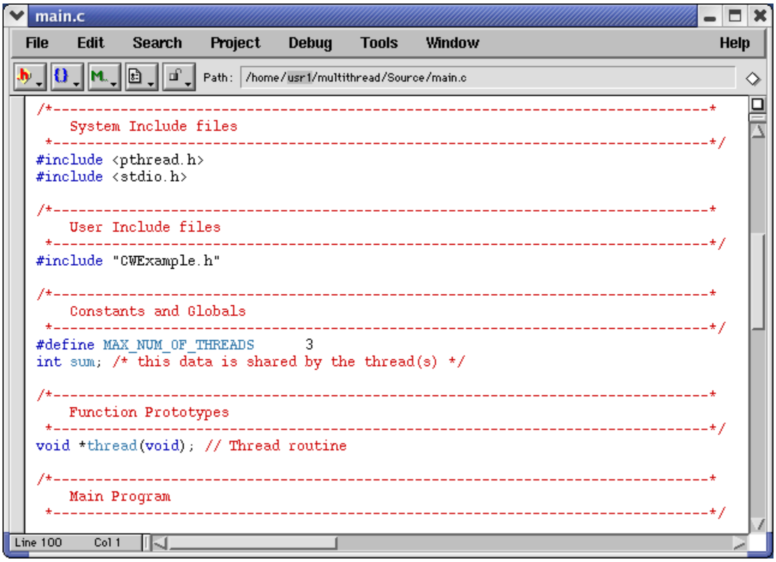
```
void *thread(void); // Thread routine
/*------------------------
Main Program
------------------------*/
int main(int argc, char *argv[])
{
   pthread_t tid[MAX_NUM_OF_THREADS]; /* the thread identifier */
   pthread_attr_t attr[MAX_NUM_OF_THREADS];/*set of thread attributes*/
    int i;
    if (argc != 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 3.23).

Figure 3.23 Editor Window



Path: /home/usr1/multithread/Source/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
 *----------------------------------------------------------------------------*/
```

   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.

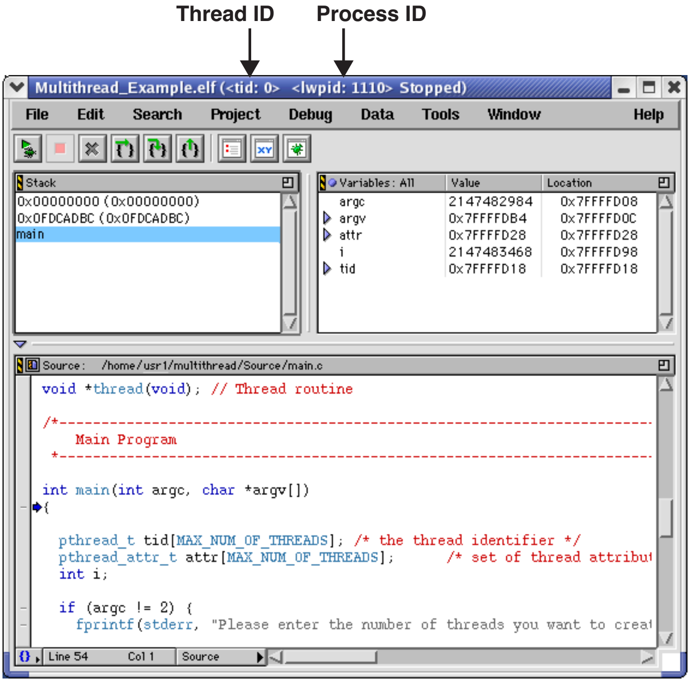
5. Specify the linker settings.
   a. Open the **GNU Linker** settings panel.
   b. Type `-lpthread` in the **Libraries** text box.
   c. Click **Save** to save the settings.
   d. Close the **GNU Linker** settings panel.
6. Build the project.

   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 3.24) appears.

NOTE To be able to successfully debug multi-threaded 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 3.24 Debugger Window


Thread ID
Process ID
Multithread_Example.elf (<tid: 0> <lwpid: 1110> Stopped)
File Edit Search Project Debug Data Tools Window Help
Line 54 Col 1 Source

Stack

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

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

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

```
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 attribut
  int i;

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

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.

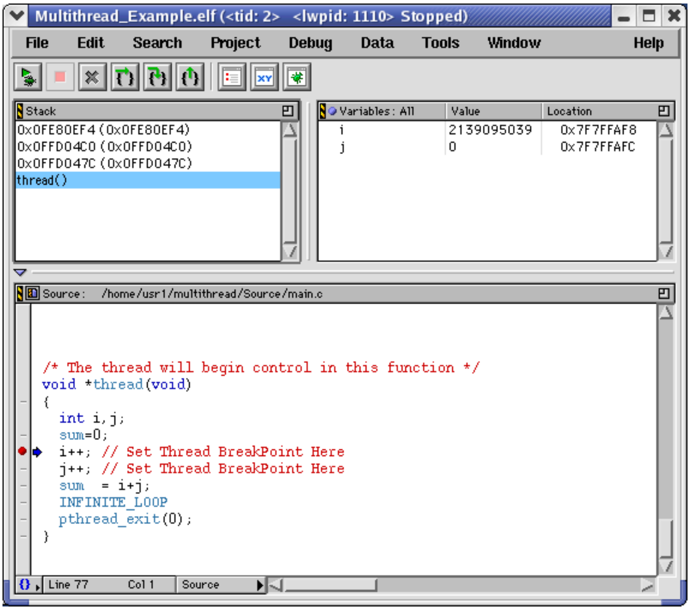
NOTE The Thread ID (TID) on the thread window is the ID assigned by the debugger/ CodeWarrior TRK to a particular thread. The debugger uses this ID to identify a thread.

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 3.25) 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 3.25 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()
```

Variables : All	Value	Location
i	2139095039	0x7F7FFAF8
j	0	0x7F7FFAFC

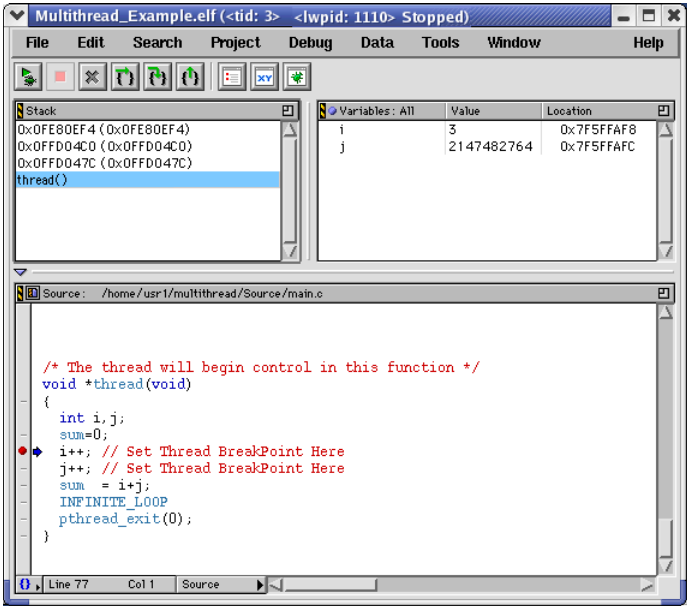
```
/* The thread will begin control in this function */
void *thread(void)
{
  int i,j;
  sum=0;
  i++; // Set Thread BreakPoint Here
  j++; // Set Thread BreakPoint Here
  sum  = i+j;
  INFINITE_LOOP
  pthread_exit(0);
}
```

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 3.26) appears.

Figure 3.26 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()
```

Variables : All	Value	Location
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);
}
```

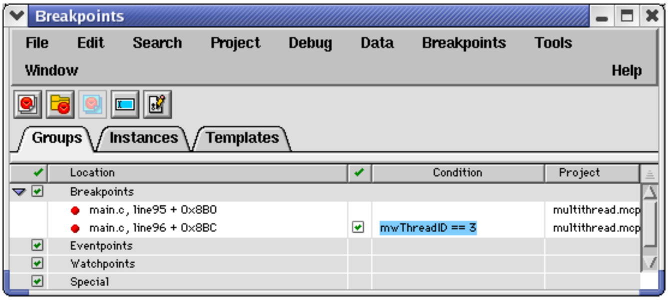
10. Set a breakpoint, which is specific for the second thread.

   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 3.27) appears. For more information, refer the *CodeWarrior IDE User's Guide*.

Figure 3.27 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.

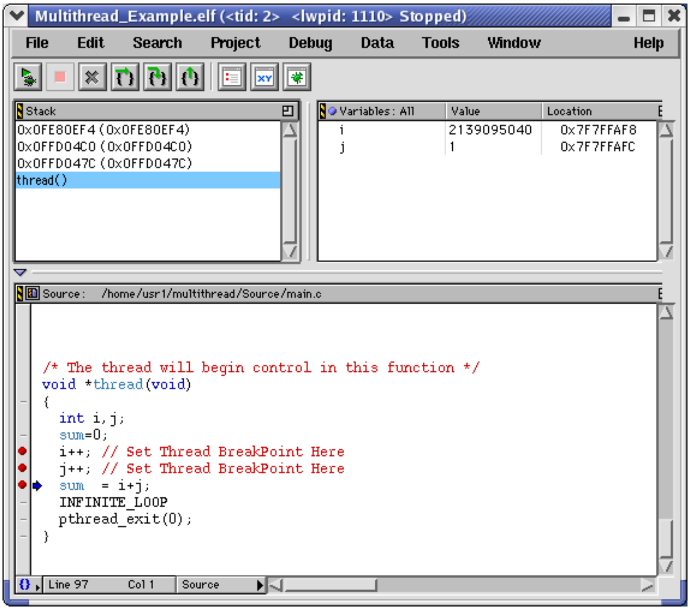
11. Set a breakpoint just after the conditional breakpoint.

    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 3.28).

Figure 3.28 First Thread Ignores Conditional Breakpoint



Multithread_Example.elf (<tid: 2> <lwpid: 1110> Stopped)

Stack

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

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

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

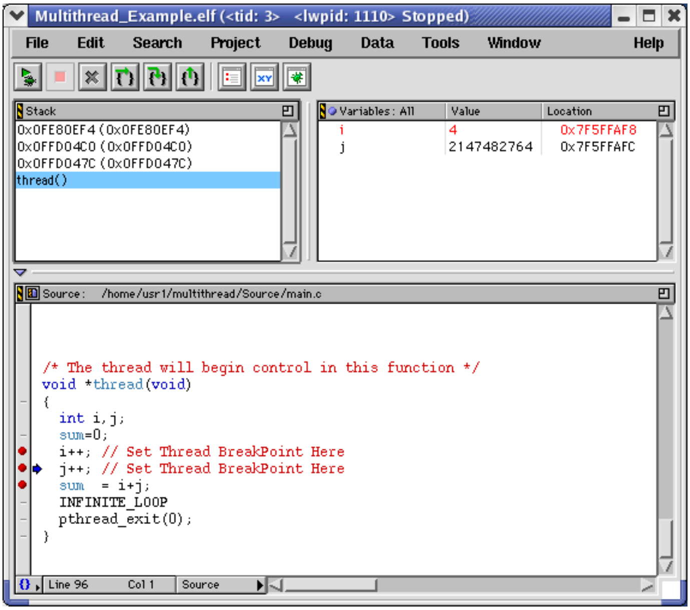
```
/* The thread will begin control in this function */
void *thread(void)
{
  int i,j;
  sum=0;
  i++; // Set Thread BreakPoint Here
  j++; // Set Thread BreakPoint Here
  sum  = i+j;
  INFINITE_LOOP
  pthread_exit(0);
}
```

Line 97 Col 1 Source

13. Execute the second thread.

    Click **Run** in the second thread window. The thread execution stops at the conditional breakpoint (Figure 3.29) set at the following line of code: `j++; // Set Thread BreakPoint Here`.

Figure 3.29 Execution of Second Thread Stopped at Conditional Breakpoint


Multithread_Example.elf (<tid: 3> <lwpid: 1110> Stopped)
File
Edit
Search
Project
Debug
Data
Tools
Window
Help
Stack
Variables: All
Source: /home/usr1/multithread/Source/main.c
Line 96
Col 1
Source

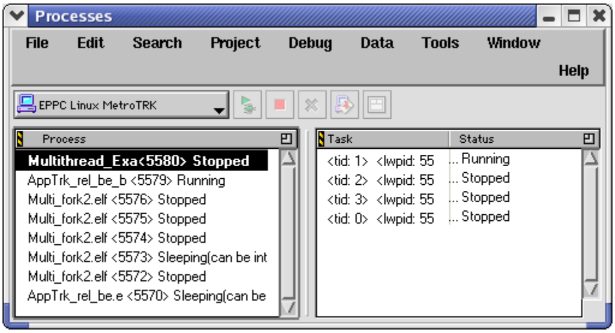
```
0x0FE80EF4 (0x0FE80EF4)
0x0FFD04C0 (0x0FFD04C0)
0x0FFD047C (0x0FFD047C)
thread()
```

Variables: All	Value	Location
i	4	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);
}
```

14. While debugging, if you wish to view the list of threads associated with a process, select **Window > Processes Window**. The **Processes** window (Figure 3.30) as in the following example appears. For more information about the **Processes** window, see *CodeWarrior IDE User's Guide*.

Figure 3.30 Example of Multi-thread Processes Window



Process
Multithread_Exa<5580> Stopped
AppTrk_rel_be_b <5579> Running
Multi_fork2.elf <5576> Stopped
Multi_fork2.elf <5575> Stopped
Multi_fork2.elf <5574> Stopped
Multi_fork2.elf <5573> Sleeping(can be int
Multi_fork2.elf <5572> Stopped
AppTrk_rel_be.e <5570> Sleeping(can be

Task	Status
<tid: 1> <lwpid: 55 ...	Running
<tid: 2> <lwpid: 55 ...	Stopped
<tid: 3> <lwpid: 55 ...	Stopped
<tid: 0> <lwpid: 55 ...	Stopped

You may also use the sample multithreading project available in the CodeWarrior installation directory. For more information about sample projects, see “Sample Projects” on page 20.

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 drag 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.

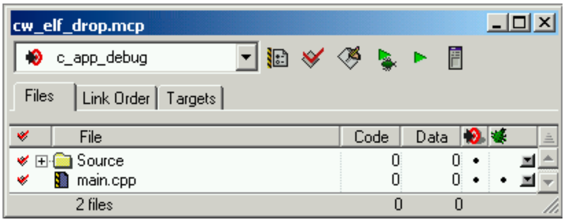
NOTE For debugging a shared library (.so) file on the target platform, you must associate a host file with the shared library.

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.

   Drag an executable file (`.elf`) for which there is no source code available into the CodeWarrior IDE window. 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` `(`Figure 3.31`).`

Figure 3.31 Dummy Project Window



✓	File	Code	Data		
✓	Source	0	0	•	
✓	main.cpp	0	0	•	•
	2 files	0	0		

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**.

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 3.10 summarizes the descriptions of these system calls.

Table 3.10 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 CodeWarrior TRK 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. A pre-built version of the static library is available at this location:
CWInstall`/CodeWarriorIDE/Examples)/arm/Binaries/arch`
where, `arch` is the platform architecture you are using (for example, `dbmx1_le` for DragonBallMX1 platform architecture).

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 CodeWarrior TRK 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 3.11).

Table 3.11 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.

a. Enter the source code of Listing 3.6 into the editor window of `Libstaticfork.c` file.

Listing 3.6 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 3.7 into the header file.

Listing 3.7 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 path 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 3.12):

Table 3.12 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 3.12 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 3.8 into the editor window of `fork.c` file.

Listing 3.8 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 3.9 into the editor window of `ChildA.c` file.

Listing 3.9 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;
}
```

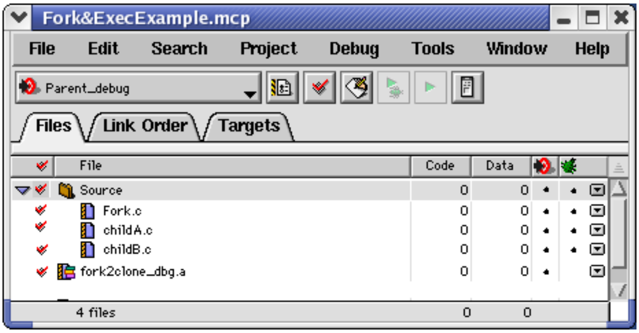
6. Enter the source code of Listing 3.10 into the editor window of `ChildB.c` file.

Listing 3.10 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 3.32).

Figure 3.32 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		

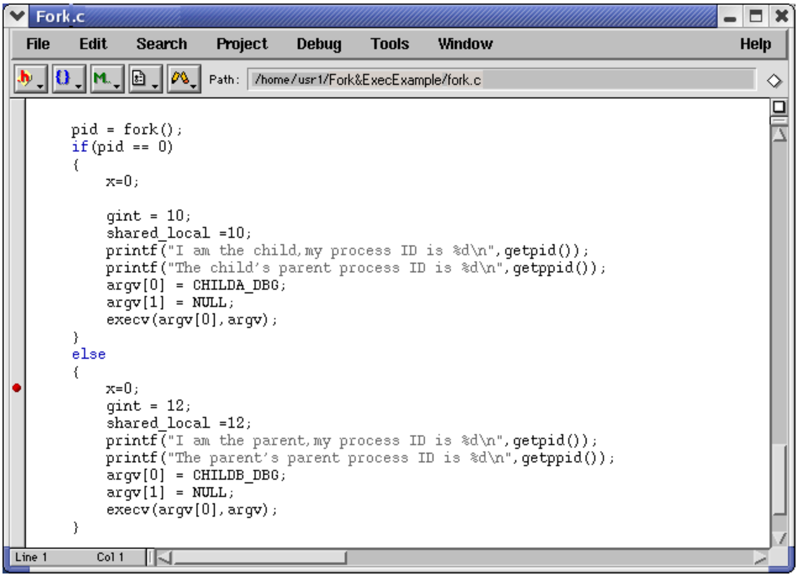
8. Build `Fork&ExecExample.mcp` project.
   a. Select the **Parent_debug** build target from the build target list box in the project window, if not selected.
   b. Select **Project > Make**. The CodeWarrior IDE generates the `Parent.elf`, `Child-A.elf`, and `Child-B.elf` executable files and places 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 3.33) appears.

Figure 3.33 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;`.

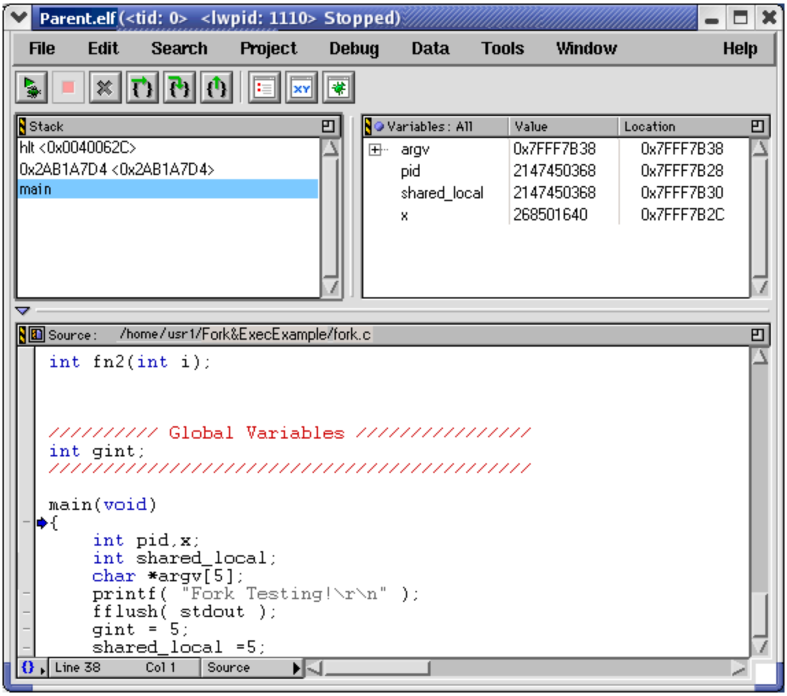
c. Set a breakpoint in the code of the parent process.

d. Close the `fork.c` file.

12. Start the debugger.

Select **Project > Debug**. The debugger window (Figure 3.34) 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 3.34 Debugger Window for Parent Process


Parent.elf (<tid: 0> <lwpid: 1110> Stopped)
File
Edit
Search
Project
Debug
Data
Tools
Window
Help
Stack
Variables: All
Source: /home/usr1/Fork&ExecExample/fork.c
Line 38
Col 1
Source

Stack:

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

Variables: All	Value	Location
⊞ argv	0x7FFF7B38	0x7FFF7B38
pid	2147450368	0x7FFF7B28
shared_local	2147450368	0x7FFF7B30
x	268501640	0x7FFF7B2C

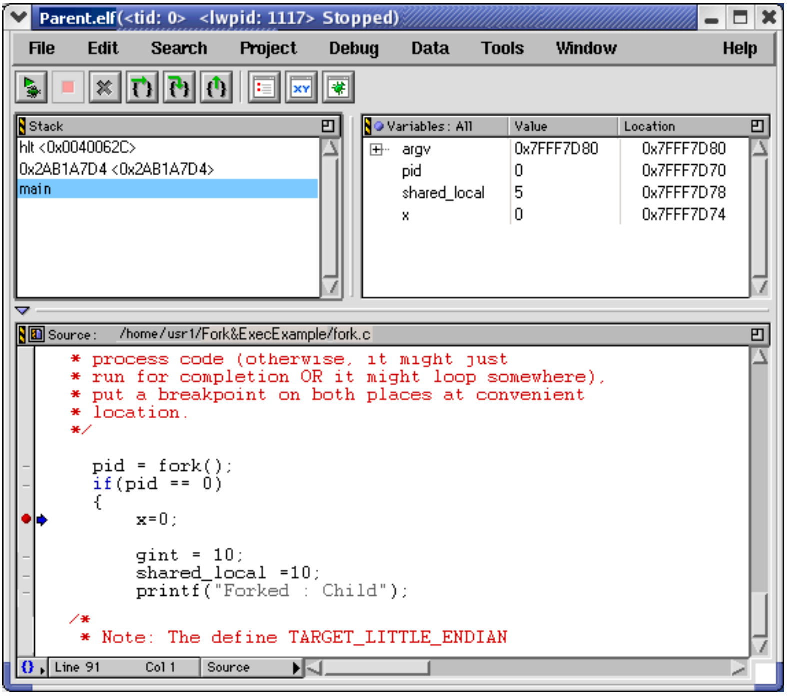
```
int fn2(int i);


/////////// 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 3.35) appears. You can now perform normal debugging operations in this window.

Figure 3.35 Debugger Window for Child Process



Stack:

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

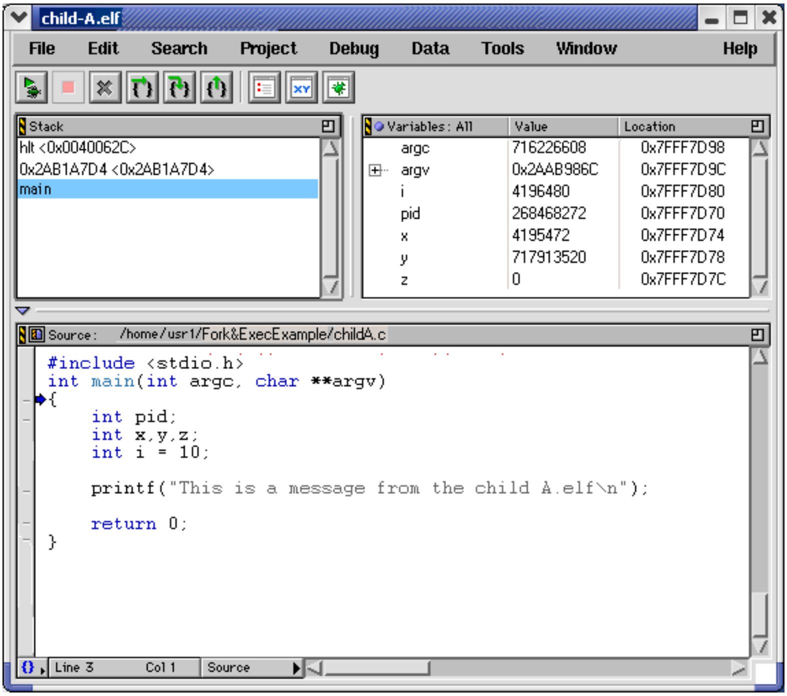
Variables : All	Value	Location
argv	0x7FFF7D80	0x7FFF7D80
pid	0	0x7FFF7D70
shared_local	5	0x7FFF7D78
x	0	0x7FFF7D74

Source : /home/usr1/Fork&ExecExample/fork.c

```
 * process code (otherwise, it might just
 * run for completion OR it might loop somewhere),
 * put a breakpoint 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 3.36) appears. This window displays the code of the `Child-A.elf` executable file. You can now perform normal debugging operations in this window.

Figure 3.36 Debugger Window for File Executed by Child Process



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

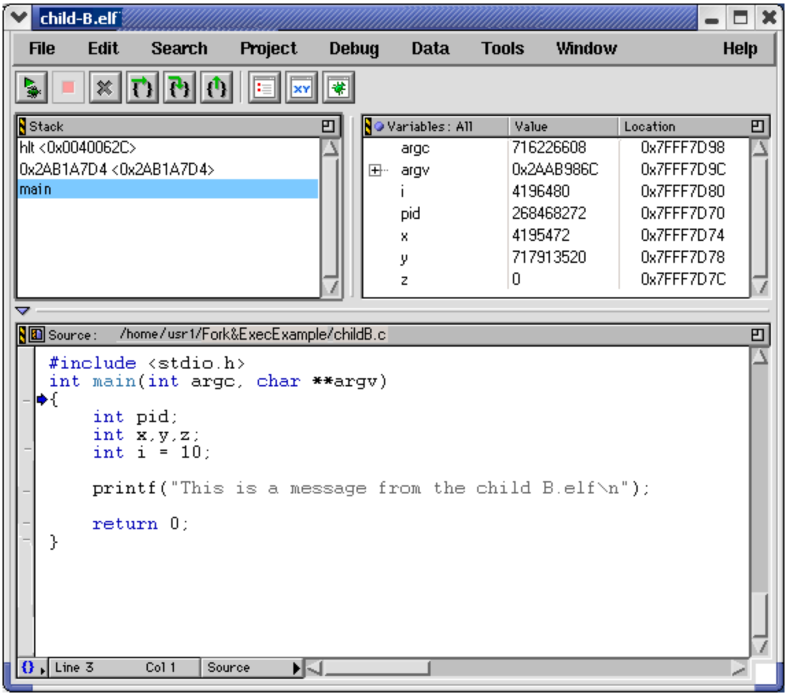
Variables: All	Value	Location
argc	716226608	0x7FFF7D98
⊞ argv	0x2AAB986C	0x7FFF7D9C
i	4196480	0x7FFF7D80
pid	268468272	0x7FFF7D70
x	4195472	0x7FFF7D74
y	717913520	0x7FFF7D78
z	0	0x7FFF7D7C

Source: /home/usr1/Fork&ExecExample/childA.c

```
#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 3.37). You can now perform normal debugging operations in this window.

Figure 3.37 Debugger Window for File Executed by Parent Process



Stack

```
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

Source: /home/usr1/Fork&ExecExample/childB.c

```
#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;
}
```

Line 3 Col 1 Source

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

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 3.38) contains commands that enable you to view and refresh the processes running on the target platform.

Figure 3.38 Linux Info Menu

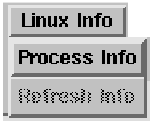


Table 3.13 describes the menu commands provided by the **Linux Info** menu.

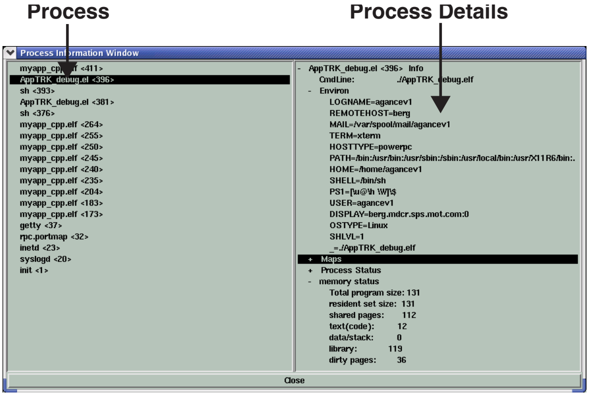
Table 3.13 Linux Info Menu - Description of Commands

Commands	Descriptions
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

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 3.39) appears.

Figure 3.39 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 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.c

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

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.

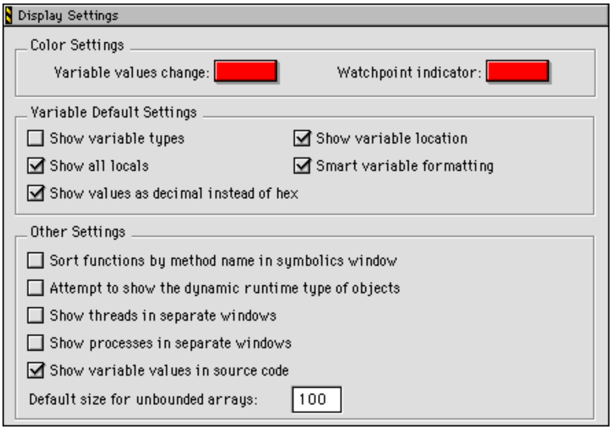
For example, let us take the example of the `ForkAndExec.mcp` project you created and debugged in the previous section. The parent process; `Parent.elf` forks a new child process; `Child-A.elf`. The child process appears in a separate thread window. You can show the parent process and the child process in a single thread window using the option made available in the **IDE Preferences** panel.

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 are cleared in the **Display Settings** panel (Figure 3.40).

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 3.40 Display Settings Panel



Display Settings

Color Settings

Variable values change: 🟥 Watchpoint indicator: 🟥

Variable Default Settings

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

Other Settings

□ 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

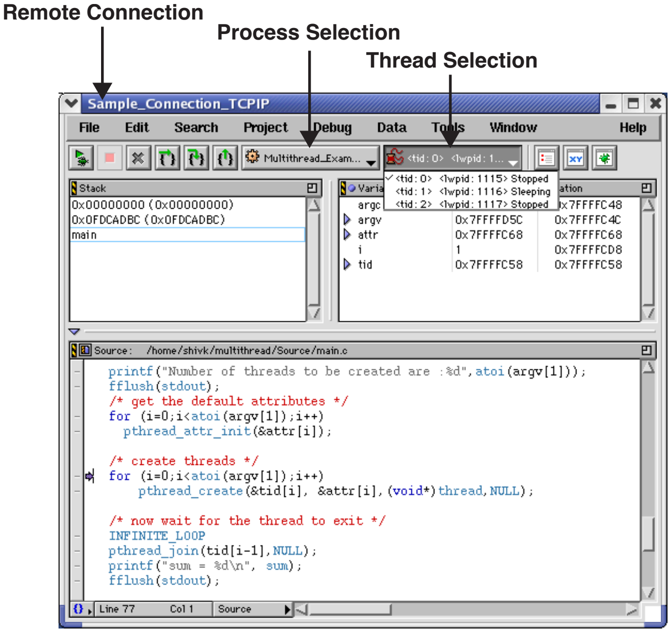
b. Click **Save** to save the settings.

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

3. Start the debugger.

a. Select **Project > Debug**. The thread window (Figure 3.41) for `Multithread_Example.elf` appears.

Figure 3.41 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                           x7FFFFC48
  argv        0x7FFFFD5C         0x7FFFFC4C
  attr        0x7FFFFC68         0x7FFFFC68
  i           1                  0x7FFFFCD8
  tid         0x7FFFFC58         0x7FFFFC58
```

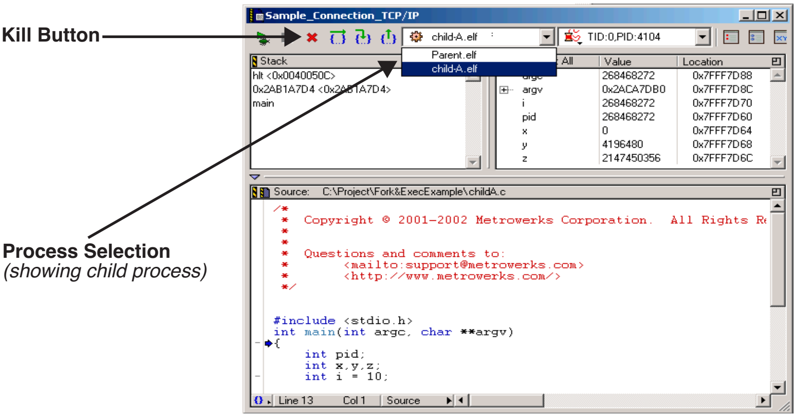
Source: /home/shivk/multithread/Source/main.c

```
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. Step over the code in the debugger window until the `exec()` function call in the child process code executes, the `Child-A.elf` thread window (Figure 3.42) appears in the same thread window. This window displays the code of the `Child-A.elf` executable file.

Figure 3.42 Thread Window Showing Child Process in the Same Thread Window



```
hlt <0x0040050C>
0x2AB1A7D4 <0x2AB1A7D4>
main
```

	Value	Location
argc	268468272	0x7FFF7D88
argv	0x2ACA7DB0	0x7FFF7D8C
i	268468272	0x7FFF7D70
pid	268468272	0x7FFF7D60
x	0	0x7FFF7D64
y	4196480	0x7FFF7D68
z	2147450356	0x7FFF7D6C

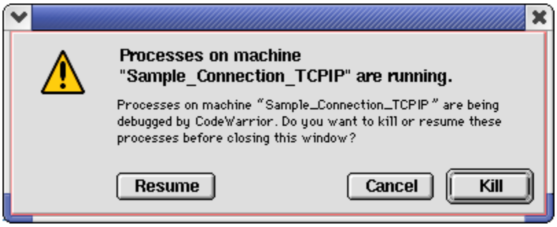
Source: C:\Project\Fork&ExecExample\childA.c

```
/*
 *  Copyright © 2001-2002 Metrowerks Corporation.  All Rights Re
 *
 *
 *  Questions and comments to:
 *       <mailto:support@metrowerks.com>
 *       <http://www.metrowerks.com/>
 */


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

c. You can now toggle between both the processes (`Parent.elf` and `child-A.elf`) and debug both of them, alternatively.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 3.43) appears.

Figure 3.43 CodeWarrior Message Box



Processes on machine "Sample_Connection_TCPIP" are running.

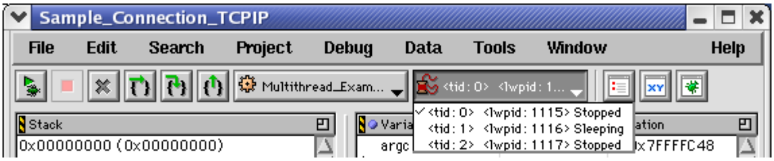
Processes on machine "Sample_Connection_TCPIP" are being debugged by CodeWarrior. Do you want to kill or resume these processes before closing this window?

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 3.44).

Figure 3.44 Multi threaded Application - Multiple Threads in Same Thread Window



Attaching to Processes

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

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

1. Open a CodeWarrior project.
2. Ensure that a linker is selected in the **Target Settings** panel.
3. Ensure that a TCP/IP remote connection is selected in the **Remote Debugging** target settings panel.
4. Check the **Show in processes list** checkbox in the remote connection settings.
5. Build the CodeWarrior project to generate a valid executable file.
6. Select **View > System > *Connection*** from the CodeWarrior menu bar (where ***Connection*** is the name of the selected remote connection).

   The **System Browser** window appears, displaying a list of the processes running on the target board.
7. In the System Browser window, select the process to which you want to attach, then click the Attach To Process button ().

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

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

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

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

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

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

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 CodeWarrior TRK on the remote target

1. Create a new project using the Linux Stationery Wizard with the following settings:

Table 3.14 Attach to Process Project Settings

Project Name:	ProcessAttach
Location of the Project:	/home/usr1/ProcessAttach
Languages:	C

Table 3.14 Attach to Process Project Settings (*continued*)

Output Type:	Application
Output File Name:	AttachToProcess.flt
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 3.11 into the editor window of `main.c` file.

Listing 3.11 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.flt` is generated and is placed in the specified location in the project folder.

4. Run the code.

   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.

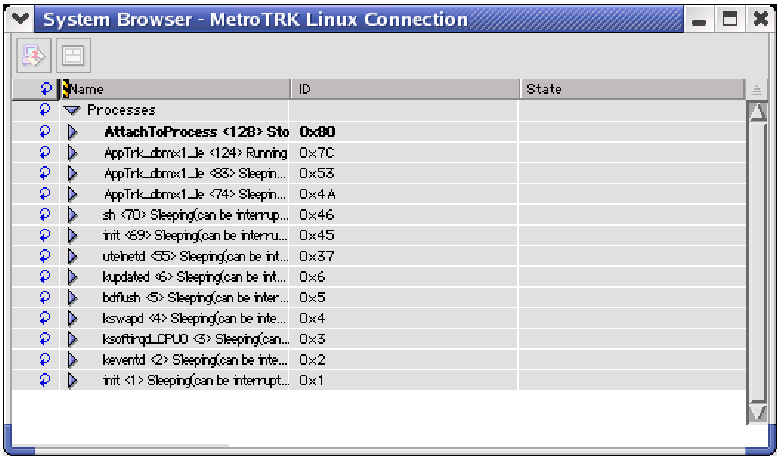
NOTE The **Connect** command is available only if a project is open. The CodeWarrior IDE uses the current connection selected in the **Remote Debugging** panel, to make a connection to the target system.

   b. Select **Window > System Windows**.

      The **System Browser** window (Figure 3.45) 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 3.45 Processes Window



System Browser - MetroTRK Linux Connection

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...	0x4 A	
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	

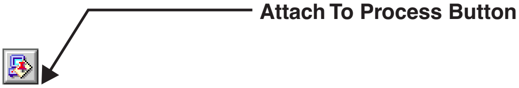
The **Processes** list in the left pane of the **System Browser** window displays the names of the processes running on the selected target system. Clicking a process name in the **Processes** list displays the threads associated with the process.

TIP You can also view the list of processes on another target system by selecting the corresponding connection name from the list box at the upper-left corner of the

System Browser window. 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 3.46).

Figure 3.46 Processes Window - Attach to Process Button



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.flt.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 binary 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 In the **Choose Executable** dialog box, be sure to select the correct executable file to which you want your process to attach; otherwise, the debugger may associate incorrect symbolic information 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. Select **Debug > Kill** to close the debugger session.

Stripping Debug Information From 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*) 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®, ColdFire™ , 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

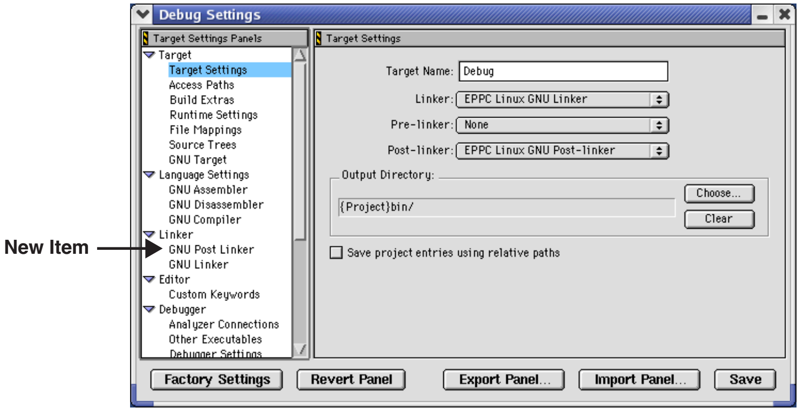
When the project is linked, the post linker adaptor calls the command line utility (*strip.exe*) specified in the **GNU Tools** settings panel and passes the pathname of the binary to be stripped of debug information. After a stripped version of the binary is created, the binary can be downloaded on 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 3.47 shows the **Post-linker** option.

Figure 3.47 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

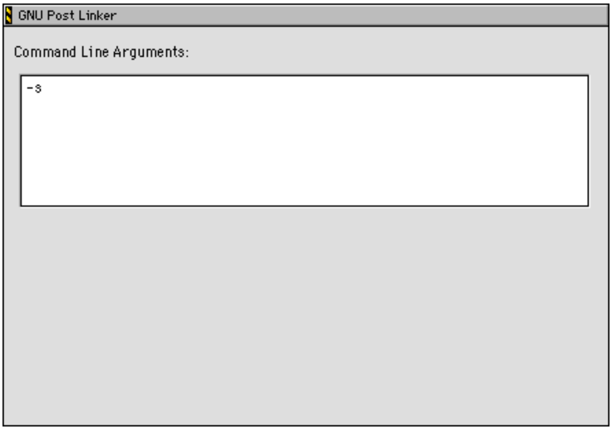
Output Directory: {Project}bin/

☐ 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 3.47).

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 3.48).

Figure 3.48 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.exe` 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) 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.

4

Debugging Boot Loaders, Kernels, Modules, and Threads

This chapter explains how to use the CodeWarrior tools to debug boot loaders, embedded Linux® kernels and loadable modules on ColdFire® hardware.

This chapter contains these sections:

- Debugging Boot Loaders
- Debugging Kernels
- Debugging Kernel Modules
- Viewing Loaded Kernel Modules
- Debugging Kernel Threads

Debugging Boot Loaders

The CodeWarrior IDE allows you to debug or develop your own boot loader (like the coolio boot loader). This section describes the steps to debug the coolio boot loader.

Before you can debug a boot loader on your target platform, you must install the board support package (BSP) for the target platform. You must also recompile the boot loader with `-g` and `-o1` flags so that debugger symbolic information is included in the boot loader.

To debug the boot loader using the CodeWarrior IDE:

1. From the CodeWarrior menu bar, select **File > Open** to open the boot loader binary (for example, `colilo_mcf5485.elf`).

   The CodeWarrior IDE creates a dummy project with the name of the boot loader binary file (for example, **colilo_mcf5485.elf.mcp**). A progress bar appears showing the status of creation of the project. When the import process is complete, the boot loader source files appear in the CodeWarrior project window.

2. Select **Edit > *TargetName* Settings** (where ***TargetName*** is the name of the current build target).

   The **Target Settings** window appears.

3. From the **Target Settings Panels** list, select **Remote Debugging**.

   The **Remote Debugging** settings panel appears.

4. From the **Connection** list box, select the correct remote connection you want to use.
5. Click the **Edit Connection** button.

   The **Edit Connection** dialog box appears.

6. Configure the remote connection, if needed.
7. Click **OK**.

   The **Edit Connection** dialog box disappears.

8. From the **Target Settings Panels** list, select **CF Debugger Settings**.

   The **CF Debugger Settings** settings panel appears.

9. From the **Target Processor** list box, select the ColdFire processor architecture that you are targeting.
10. From the **Target OS** list box, select **Bareboard**.
11. From the **Target Settings Panels** list, select **Debugger Settings**.

    The **Debugger Settings** panel appears.

12. Check the **Stop on application launch** checkbox.
13. Select the **Program entry point** option button.
14. Click **Save**.

    The IDE saves your changes.

15. From the CodeWarrior menu bar, select **Project > Debug**.

    The debugger connects to the target platform and displays a thread window.

You can now debug the boot loader application.

Debugging Kernels

The CodeWarrior IDE allows you to debug the Linux kernel on your host computer running Linux OS.

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.

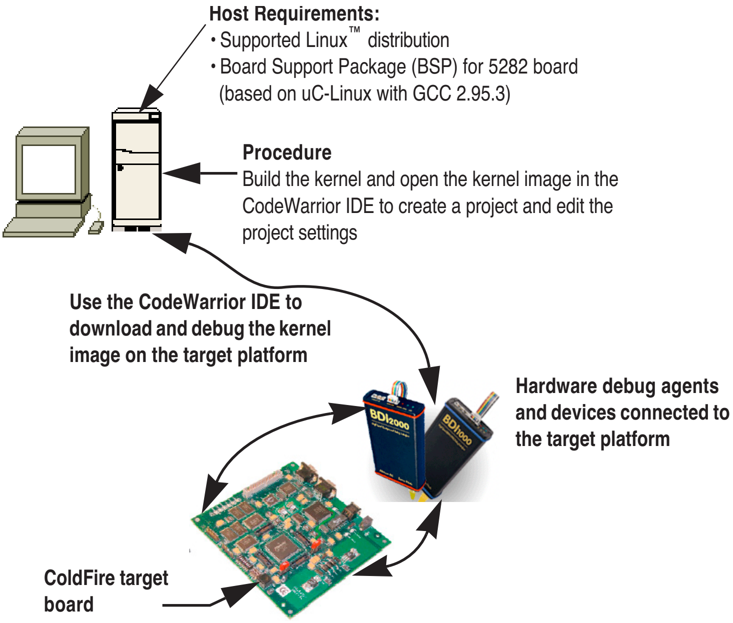
Prerequisites

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 you want to use to connect to the target platform. Also, the hardware debug agent must be properly configured before you can use it with the CodeWarrior debugger.

TIP You can use a target initialization file (such as `*CWInstall*/CodeWarriorIDE/CodeWarrior/ThirdPartyTools/MCF5208/Abatron/BDI-2000/Sample_BDI_Files/MCF5208_stop.bdi`) to initialize the Abatron BDI.

Figure 4.1 graphically illustrates the setup environment used by this product to debug the kernel on a ColdFire-based target platform.

Figure 4.1 Setup for Kernel Debugging Using the CodeWarrior IDE


Host Requirements:
• Supported Linux™ distribution
• Board Support Package (BSP) for 5282 board (based on uC-Linux with GCC 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
Use the CodeWarrior IDE to download and debug the kernel image on the target platform
Hardware debug agents and devices connected to the target platform
BDI2000
BDI1000
ColdFire target board

Kernel Debugging Methods

There are three methods for debugging the Linux kernel:

1. 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 boot loader to be present in the flash at the reset address. This manual describes this method.
2. Using boot loader initialization - Download and start the kernel using CodeWarrior IDE based on the initialization done by boot loader on the target platform.
3. Attaching to the running kernel - Start the kernel using any of the above methods and attach to the running kernel.

To debug the kernel, 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—an image without romfs (`linux`) and an image with romfs (`image.elf`) You can find the kernel image with romfs in the following folder of the base directory on your computer:

LinuxInstall`/uClinux-dist/images/`

You can find the compressed kernel image at the following location on your computer:

LinuxInstall`/uClinux-dist/Linux-2.4.x/`

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:

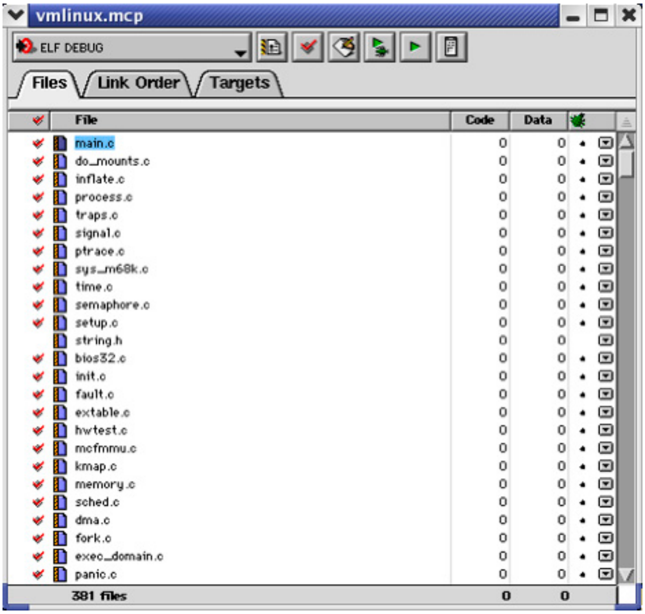
1. Select **File > Open** to open the kernel image without romfs file (*vmlinux*) built with full debug information in the CodeWarrior IDE window.
2. If the **Choose Debugger** dialog box appears, select the remote connection you want to use for downloading the kernel image, then click **OK**.

NOTE The **Choose Debugger** dialog box appears only when you have multiple remote connections defined in the **Remote Connections** settings panel.

NOTE If you do not want to specify any remote connections at this point, click **Cancel**.

The CodeWarrior IDE creates a dummy project with the name of the image file (such as `vmlinux.mcp`). 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 the project window (Figure 4.2).

Figure 4.2 Linux Kernel Project Window



✓	File	Code	Data	
✓	main.c	0	0	●
✓	do_mounts.c	0	0	●
✓	inflate.c	0	0	●
✓	process.c	0	0	●
✓	traps.c	0	0	●
✓	signal.c	0	0	●
✓	ptrace.c	0	0	●
✓	sys_m68k.c	0	0	●
✓	time.c	0	0	●
✓	semaphore.c	0	0	●
✓	setup.c	0	0	●
	string.h	0	0	
✓	bios32.c	0	0	●
✓	init.c	0	0	●
✓	fault.c	0	0	●
✓	extable.c	0	0	●
✓	hwtest.c	0	0	●
✓	mcfmmu.c	0	0	●
✓	kmap.c	0	0	●
✓	memory.c	0	0	●
✓	sched.c	0	0	●
✓	dma.c	0	0	●
✓	fork.c	0	0	●
✓	exec_domain.c	0	0	●
✓	panic.c	0	0	●
	381 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 new kernel project is created 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 steps on how to setup your kernel project for debugging on a specific ColdFire target platform, refer to the quick start guide located in the CodeWarrior installation directory.

1. Set a program entry point in the kernel code.
   a. Select **Edit > *TargetName* Settings** (where ***TargetName*** is the name of the current build target displayed in the project window).

      The **Target Settings** window appears.

   b. From the **Target Settings Panels** list, select **Debugger Settings**.

      The **Debugger Settings** panel appears.

   c. Check the **Stop on application launch** checkbox.
   d. If you want the debugger to stop program execution upon entering the program, select the **Program entry point** option button.

NOTE You can specify an alternate location for the debugger to halt execution. For example, to instruct the debugger to halt execution when the debugger enters the `start_kernel()` function, click the **User specified** option button, then enter `start_kernel` 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 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**.

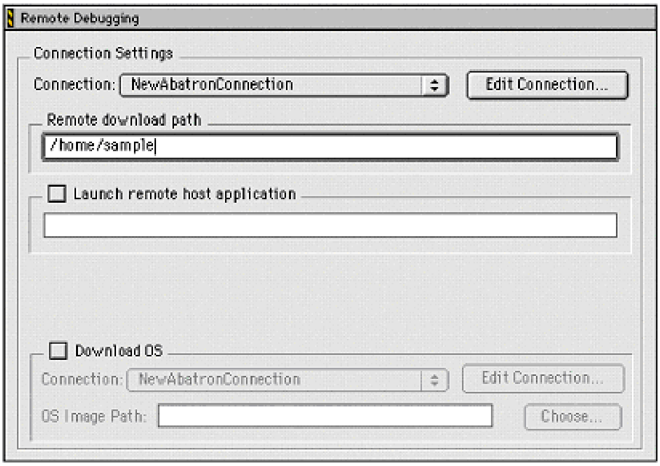
NOTE You do not have to specify the remote target path for downloading the kernel, because the kernel is downloaded to target platform RAM.

   c. Click **Download OS** checkbox to activate the kernel image with romfs (`image.elf`) download options.
   d. Select the correct remote connection for downloading the kernel image with romfs (`image.elf`) 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 4.3).

Figure 4.3 Specifying Kernel Download Options



Remote Debugging

Connection Settings

Connection: NewAbatronConnection

Remote download path: /home/sample

□ Launch remote host application

□ Download OS

Connection: NewAbatronConnection

OS Image Path:

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

3. Specify a target processor, operating system, and an initialization file for the debugger.

a. Click **CF Debugger Settings** in the **Target Settings Panels** list.

The **CF Debugger Settings** panel appears.

NOTE An .xml file with pre-configured settings for this panel is provided for the target platforms board support packages supported by this product. 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 directory (where *TargetName* is the name of a supported target board, such as `MCF5329`):
`*CWInstall*/CodeWarriorIDE/CodeWarrior/E68K_Support/KernelDebug_Settings/`*TargetBoard*.

b. Select the ColdFire 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 **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 **CF Debugger Settings** panel, see "CF Debugger Settings" on page 128.

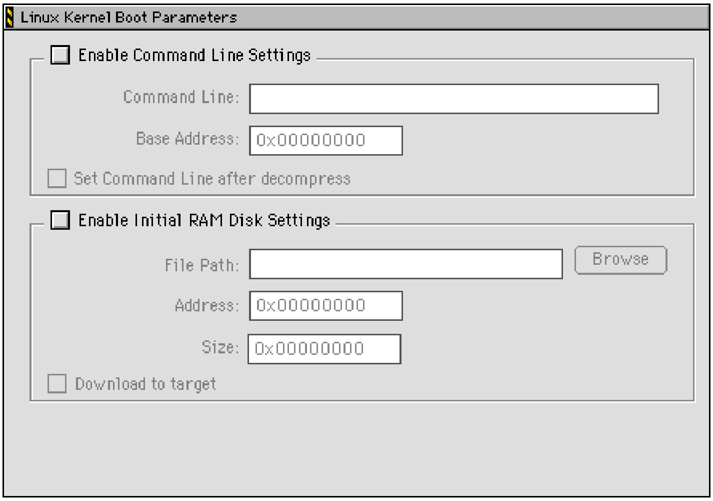
e. Click **Save** to save the settings.

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 4.4).

NOTE The **Linux Kernel Boot Parameters** settings panel is displayed in the **Target Settings Panels** list only when you select 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 4.4 Linux Kernel Boot Parameters - Command Line and initrd Settings



Linux Kernel Boot Parameters

☐ Enable Command Line Settings

Command Line:

Base Address: 0x00000000

☐ Set Command Line after decompress

☐ Enable Initial RAM Disk Settings

File Path:

Address: 0x00000000

Size: 0x00000000

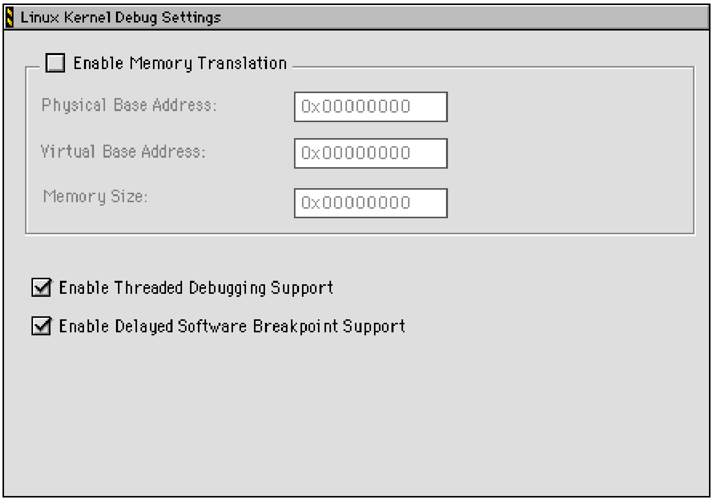
☐ Download to target

   b. Check the **Enable Command Line Setting** checkbox.
   c. 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.
   d. Check the **Enable Initial RAM Disk Settings** checkbox.
   e. Click **Browse** to specify the location on the host computer of the initial RAM disk (initrd) file.
   f. Enter the size of the RAM Disk file in the **Size** text box.
   g. Check the **Download to target** checkbox to download the RAM Disk file to the target platform.
   h. Click **Save** to save the settings.

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 4.5) appears.

NOTE The **Linux Kernel Debug Settings** panel is displayed in the **Target Settings Panels** list only when you select a 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 4.5 Linux Kernel Debug Settings



Linux Kernel Debug Settings

☐ Enable Memory Translation

Physical Base Address: 0x00000000

Virtual Base Address: 0x00000000

Memory Size: 0x00000000

☑ 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. In the **Virtual Base Address** text box, enter the virtual base address for memory translation.

d. Enter the memory (RAM) size on your target board/platform in the **Memory Size** text box. For board memory size, refer your board documentation.

e. Check the **Enable Threaded Debugging Support** checkbox 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 "Debugging Kernel Threads" on page 114.

f. Check the **Enable Delayed Software Breakpoint Support** checkbox if you want to delay the setting of software breakpoints till the MMU is enabled.

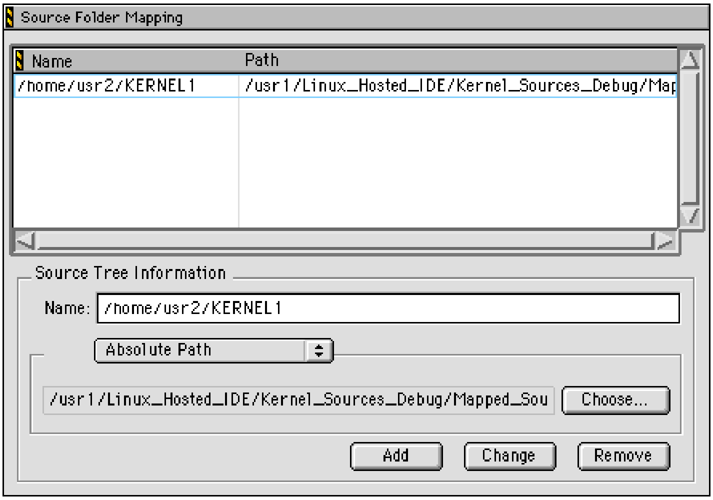
g. Click **Save** to save the settings.

6. 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 4.6) 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 4.6 Source Folder Mapping



Source Folder Mapping

Name	Path
/home/usr2/KERNEL1	/usr1/Linux_Hosted_IDE/Kernel_Sources_Debug/Map

Source Tree Information

Name: /home/usr2/KERNEL1

Absolute Path

/usr1/Linux_Hosted_IDE/Kernel_Sources_Debug/Mapped_Sou

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

c. Close the ***Target* Settings** window.

7. 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.

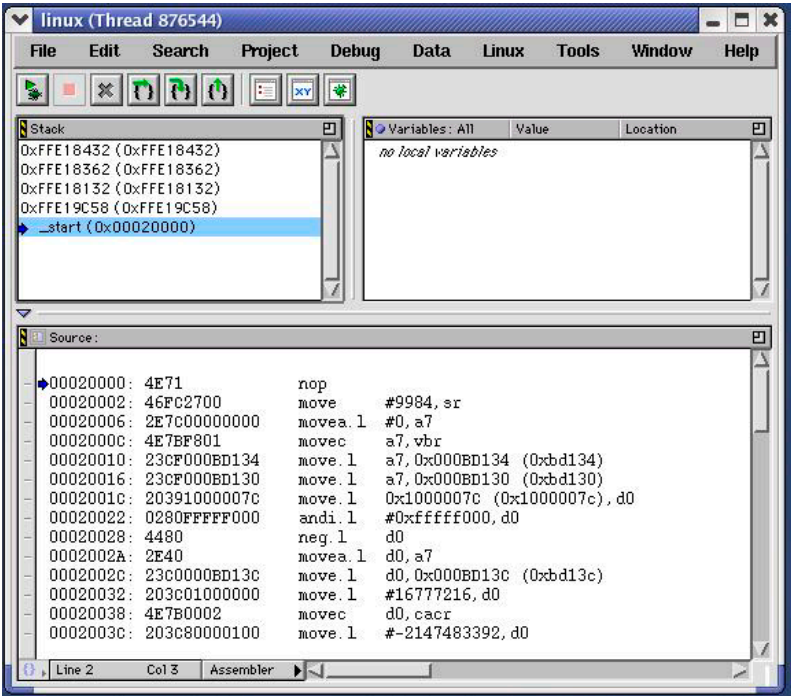
NOTE Before you download the kernel image to the target platform, make sure that you *switch off* and then *switch on* the target platform. Otherwise, the kernel image does not get downloaded to the target platform.

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 4.7).

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, romfs download to the target platform, and the name of the initialization file that you specified in the **CF Debugger Settings** panel.

Figure 4.7 Kernel Debugger Window



```
0xFFE18432 (0xFFE18432)
0xFFE18362 (0xFFE18362)
0xFFE18132 (0xFFE18132)
0xFFE19C58 (0xFFE19C58)
_start (0x00020000)
```

no local variables

```
00020000: 4E71            nop
00020002: 46FC2700        move      #9984,sr
00020006: 2E7C00000000    movea.l   #0,a7
0002000C: 4E7BF801        movec     a7,vbr
00020010: 23CF000BD134    move.l    a7,0x000BD134  (0xbd134)
00020016: 23CF000BD130    move.l    a7,0x000BD130  (0xbd130)
0002001C: 20391000007C    move.l    0x1000007C  (0x1000007c),d0
00020022: 0280FFFFF000    andi.l    #0xfffff000,d0
00020028: 4480            neg.l     d0
0002002A: 2E40            movea.l   d0,a7
0002002C: 23C0000BD13C    move.l    d0,0x000BD13C  (0xbd13c)
00020032: 203C01000000    move.l    #16777216,d0
00020038: 4E7B0002        movec     d0,cacr
0002003C: 203C80000100    move.l    #-2147483392,d0
```

b. Select **Project > Run**.

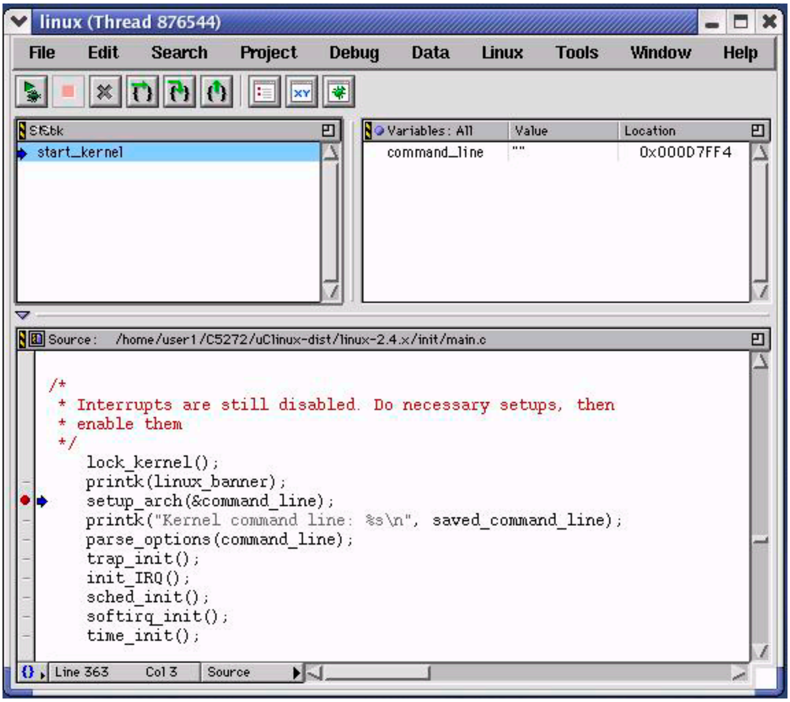
The debugger stops execution of the program at `start_kernel()` only if a breakpoint is set in this function (Figure 4.8).

If you checked the **Enable Delayed Software Breakpoint Support** checkbox, the debugger sets a hardware breakpoint (Resolver Eventpoint) at this line:

```
printk (linux_banner);
```

When the debugger encounters this Resolver Eventpoint, all the subsequent software breakpoints you have set are enabled. For more information about Resolver Eventpoint, see *CodeWarrior IDE User's Guide*.

Figure 4.8 Program Entry Point in Kernel Code



Variables: All	Value	Location
command_line	""	0x000D7FF4

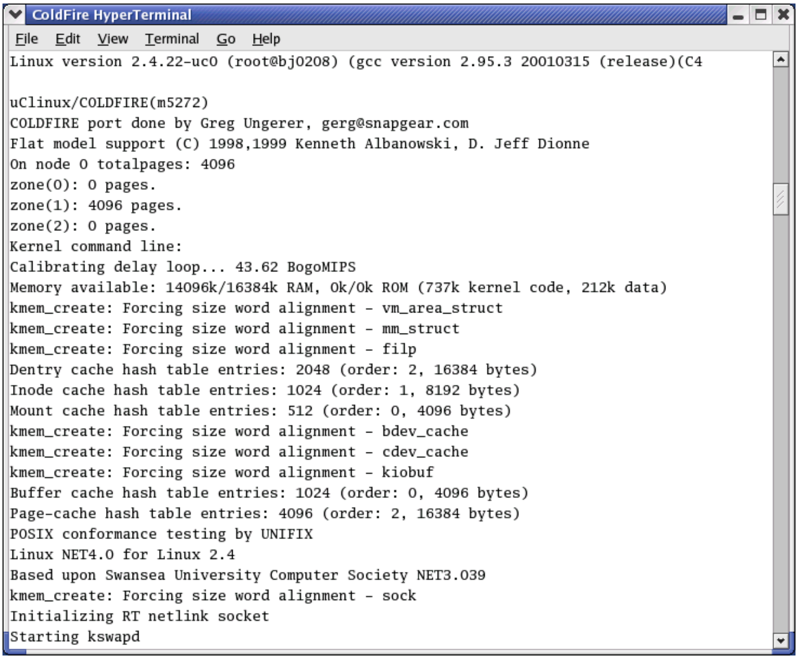
Source: /home/user1/C5272/uClinux-dist/linux-2.4.x/init/main.c

```
/*
 * 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();
	init_IRQ();
	sched_init();
	softirq_init();
	time_init();
```

Line 363   Col 3   Source

c. Run through the rest of the code until the kernel starts booting. When the kernel boots up you can see the boot messages in your Terminal window (Figure 4.9).

Figure 4.9 Terminal Window Showing the Kernel Boot Messages



```
Linux version 2.4.22-uc0 (root@bj0208) (gcc version 2.95.3 20010315 (release)(C4

uClinux/COLDFIRE(m5272)
COLDFIRE port done by Greg Ungerer, gerg@snapgear.com
Flat model support (C) 1998,1999 Kenneth Albanowski, D. Jeff Dionne
On node 0 totalpages: 4096
zone(0): 0 pages.
zone(1): 4096 pages.
zone(2): 0 pages.
Kernel command line:
Calibrating delay loop... 43.62 BogoMIPS
Memory available: 14096k/16384k RAM, 0k/0k ROM (737k kernel code, 212k data)
kmem_create: Forcing size word alignment - vm_area_struct
kmem_create: Forcing size word alignment - mm_struct
kmem_create: Forcing size word alignment - filp
Dentry cache hash table entries: 2048 (order: 2, 16384 bytes)
Inode cache hash table entries: 1024 (order: 1, 8192 bytes)
Mount cache hash table entries: 512 (order: 0, 4096 bytes)
kmem_create: Forcing size word alignment - bdev_cache
kmem_create: Forcing size word alignment - cdev_cache
kmem_create: Forcing size word alignment - kiobuf
Buffer cache hash table entries: 1024 (order: 0, 4096 bytes)
Page-cache hash table entries: 4096 (order: 2, 16384 bytes)
POSIX conformance testing by UNIFIX
Linux NET4.0 for Linux 2.4
Based upon Swansea University Computer Society NET3.039
kmem_create: Forcing size word alignment - sock
Initializing RT netlink socket
Starting kswapd
```

After the kernel is booted on the target platform, you can now install, load, and debug the kernel modules.

Debugging Kernel Modules

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 symbolic information for the kernel module you want to debug

The first step is to create a kernel module project using the Linux Stationery Wizard.

1. Create a project using the Linux Stationery Wizard with the following settings:

Table 4.1 Kernel Module Project Settings

Project Name:	MyKernel_Module.mcp
Project Location:	/home/usr1/KernelModule
Languages:	C
Build Target (Debug)	
- Output Type:	Loadable Module
- Output File:	hello.o
- Output File Location:	/home/usr1/KernelModule/Output

NOTE You must select the *Loadable Module* item in the Linux Stationery 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

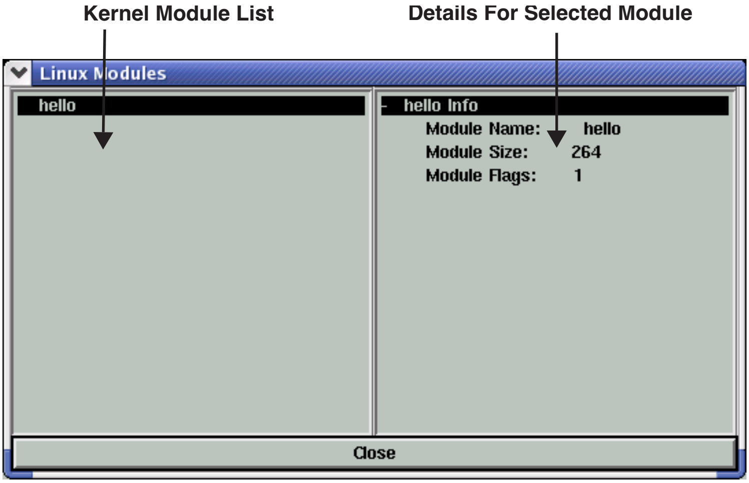
You can view a list of kernel modules that are currently installed into the kernel by using the CodeWarrior IDE. The CodeWarrior IDE provides the **Linux** menu that allows you display the kernel modules that are currently installed.

WARNING! 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 a list of kernel modules currently installed into the kernel:

1. Select **Linux > Display Modules**. The **Linux Modules** window appears (Figure 4.10), which displays all the kernel modules (in the left pane) that are currently installed in the kernel.

Figure 4.10 Linux Modules Window - List of Kernel Modules Installed



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 Module’s Symbolic Information

After you select the kernel module that you want to debug, the next step is to load the symbolic information for the selected kernel module.

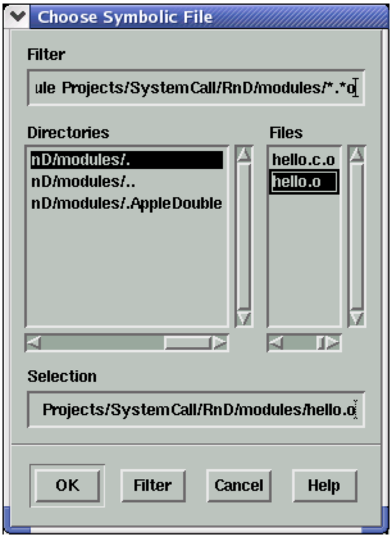
To load the symbolic information for a kernel module (`hello.o`):

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 4.11).

Figure 4.11 Choose File Dialog Box



Filter
ule Projects/SystemCall/RnD/modules/*.*o

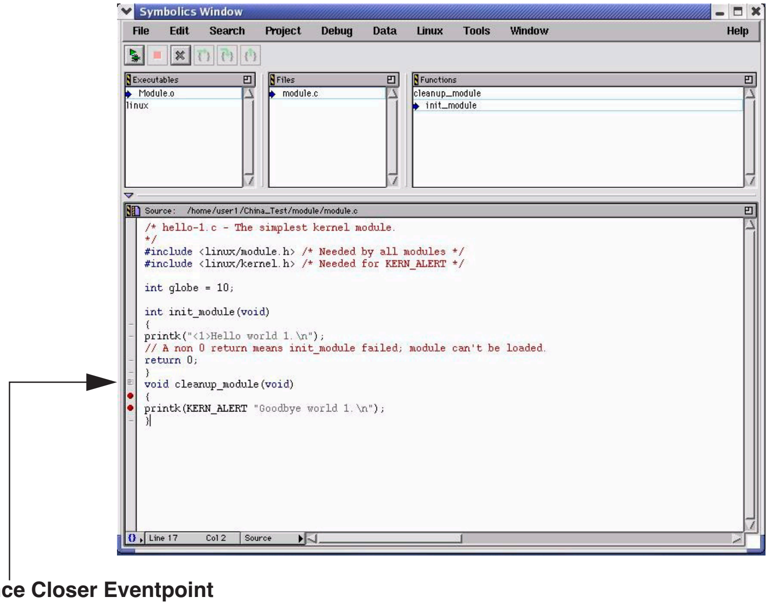
Directories
nD/modules/.
nD/modules/..
nD/modules/.AppleDouble

Files
hello.c.o
hello.o

Selection
Projects/SystemCall/RnD/modules/hello.o

3. Click **OK** in the **Choose File** dialog box. The symbolics for the selected kernel module are displayed in the **Symbolics Window** (Figure 4.12).

Figure 4.12 Symbolics Window - Symbolics for the Loaded Kernel Module



```
/* hello-1.c - The simplest kernel module.
*/
#include <linux/module.h> /* Needed by all modules */
#include <linux/kernel.h> /* Needed for KERN_ALERT */

int globe = 10;

int init_module(void)
{
printk("<1>Hello world 1.\n");
// A non 0 return means init_module failed; module can't be loaded.
return 0;
}
void cleanup_module(void)
{
printk(KERN_ALERT "Goodbye world 1.\n");
}
```

NOTE For detailed information about **Symbolics Window**, see the *CodeWarrior 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.

A sample kernel module project is available in your CodeWarrior installation directory. For more information about sample projects, see “Sample Projects” on page 20.

Viewing Loaded Kernel Modules

The CodeWarrior IDE **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 4.13)heading displays a pull-down menu containing the menu commands. The menu commands are also available as buttons on the toolbar.

Figure 4.13 Linux Menu

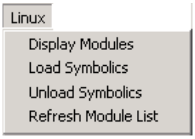


Table 4.2 describes the menu commands provided by the **Linux** menu.

Table 4.2 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

Debugging Kernel Threads

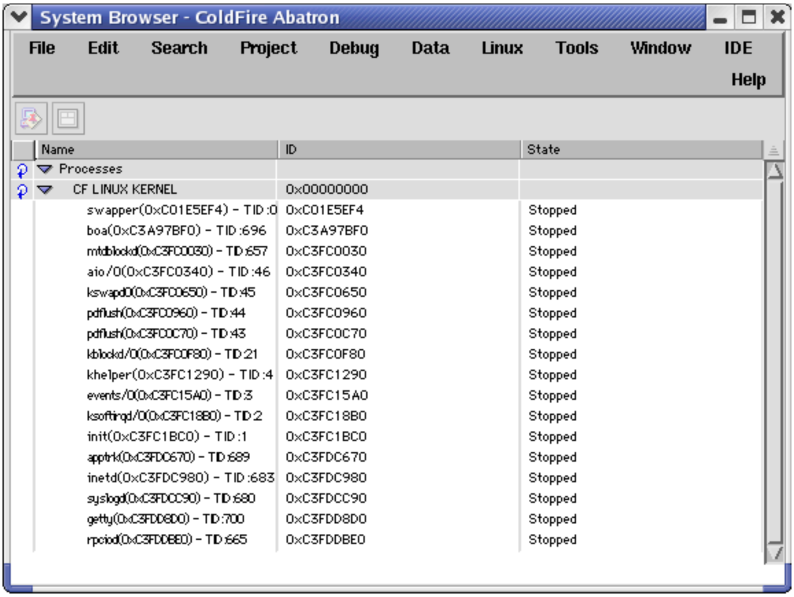
The CodeWarrior debugger enables you to view and debug kernel threads in separate thread windows. Each kernel thread debug window displays its own stack crawl pane, source pane, and variables pane.

The steps to open multiple kernel thread windows for debugging are:

1. From the CodeWarrior menu bar, select **Window > System Windows > ColdFire Abatron**. The **System Browser** window (Figure 4.14) appears. This window displays

the currently running process and the tasks for a particular remote connection. You may select the required remote connection from the upper-left corner of the window.

Figure 4.14 System Browser Window



Name	ID	State
Processes		
CF LINUX KERNEL	0x00000000	
swapper(0xC01E5EF4) - TID:0	0xC01E5EF4	Stopped
boa(0xC3A97BF0) - TID:696	0xC3A97BF0	Stopped
mtdblockd(0xC3FC0030) - TID:657	0xC3FC0030	Stopped
aio/0(0xC3FC0340) - TID:46	0xC3FC0340	Stopped
kswapd0(0xC3FC0650) - TID:45	0xC3FC0650	Stopped
pdflush(0xC3FC0960) - TID:44	0xC3FC0960	Stopped
pdflush(0xC3FC0C70) - TID:43	0xC3FC0C70	Stopped
kblockd/0(0xC3FC0F80) - TID:21	0xC3FC0F80	Stopped
khelper(0xC3FC1290) - TID:4	0xC3FC1290	Stopped
events/0(0xC3FC15A0) - TID:3	0xC3FC15A0	Stopped
ksoftirqd/0(0xC3FC18B0) - TID:2	0xC3FC18B0	Stopped
init(0xC3FC1BC0) - TID:1	0xC3FC1BC0	Stopped
apptrk(0xC3FDC670) - TID:689	0xC3FDC670	Stopped
inetd(0xC3FDC980) - TID:683	0xC3FDC980	Stopped
syslogd(0xC3FDCC90) - TID:680	0xC3FDCC90	Stopped
getty(0xC3FDD8D0) - TID:700	0xC3FDD8D0	Stopped
rpciod(0xC3FDDBE0) - TID:665	0xC3FDDBE0	Stopped

2. Select any of the task that you want to debug for a particular process from the **Task** list.

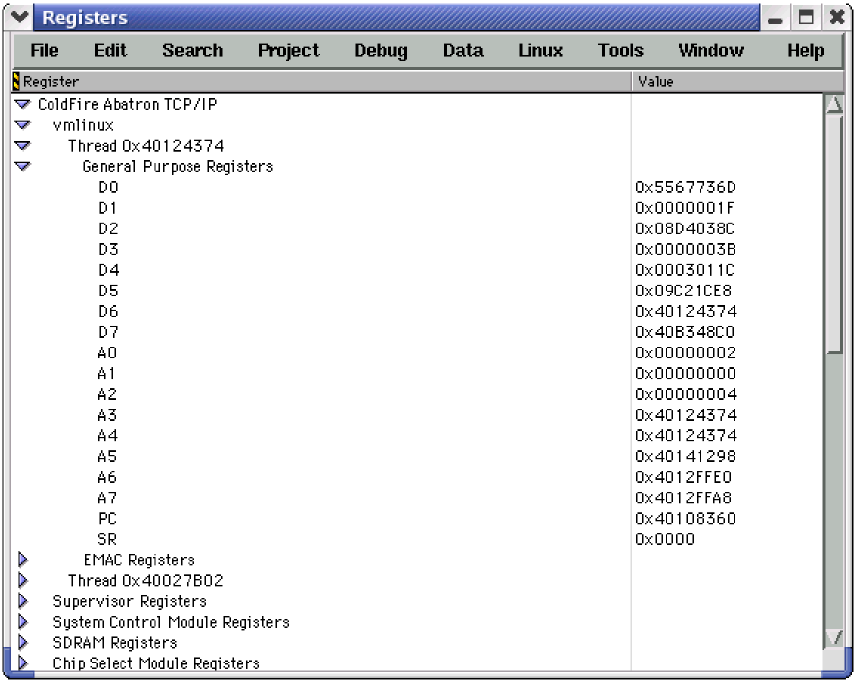
3. Double-click the selected task. A new thread window appears displaying the symbolics for the selected task.

   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 81.
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 4.15 displays the register values for thread windows.

Figure 4.15 Registers Window Displaying the Currently Open Thread Windows



Register	Value
ColdFire Abatron TCP/IP	
vmlinux	
Thread 0x40124374	
General Purpose Registers	
D0	0x5567736D
D1	0x0000001F
D2	0x08D4038C
D3	0x0000003B
D4	0x0003011C
D5	0x09C21CE8
D6	0x40124374
D7	0x40B348C0
A0	0x00000002
A1	0x00000000
A2	0x00000004
A3	0x40124374
A4	0x40124374
A5	0x40141298
A6	0x4012FFE0
A7	0x4012FFA8
PC	0x40108360
SR	0x0000
EMAC Registers	
Thread 0x40027B02	
Supervisor Registers	
System Control Module Registers	
SDRAM Registers	
Chip Select Module Registers	

5

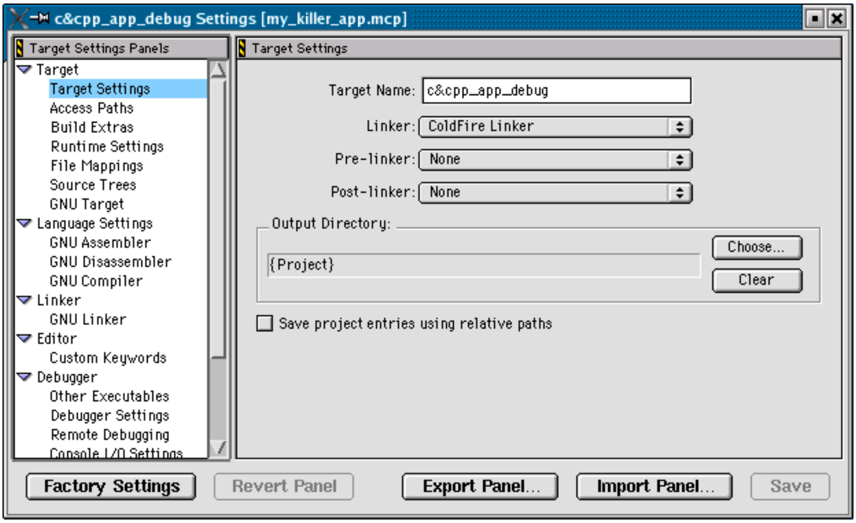
Target Settings Reference

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 for ColdFire® hardware. See the *CodeWarrior IDE User's Guide* for information about other settings panels.

Target Settings Overview

All settings for a CodeWarrior build target are organized into panels you can display in the **Target Settings** window (Figure 5.1).

Figure 5.1 Target Settings Window



c&cpp_app_debug Settings [my_killer_app.mcp]

Target Settings

Target Name: c&cpp_app_debug

Linker: ColdFire Linker

Pre-linker: None

Post-linker: None

Output Directory: {Project}

☐ Save project entries using relative paths

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

This manual documents only those settings panels of specific interest to developers of embedded Linux software for ColdFire hardware. Table 5.1 lists the settings panels that are described in this chapter.

Table 5.1 EPPC Target Settings Panels Covered In This Manual

Target Settings	GNU Linker
GNU Target	CF Debugger Settings
GNU Assembler	Source Folder Mapping
GNU Disassembler	Console I/O Settings
GNU Compiler	GNU Environment
GNU Post Linker	GNU Tools

Other Settings Panels Documentation

A large number of settings panels in the Target Settings window control settings not specific to embedded Linux development. These settings panels are described in the *CodeWarrior IDE User's Guide*. Table 5.2 lists the settings panels not documented in this manual.

Table 5.2 Settings Panels Covered In Other Manuals

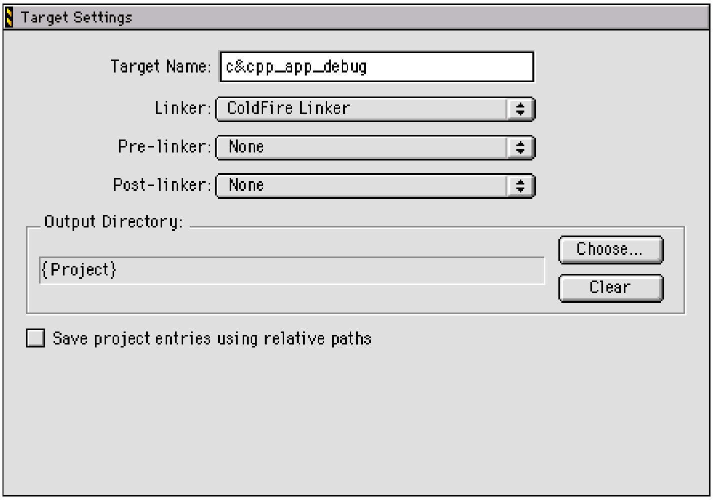
Settings Panel	Manual
Target Settings Access Paths Build Extras Runtime Settings File Mappings Source Trees External Build Custom Keywords Other Executables Debugger Settings Remote Debugging	*CodeWarrior IDE User's Guide*

Target Settings

The **Target Settings** panel (Figure 5.2) is the most important settings panel in the Target Settings window, because when you select a linker in this panel, you specify the target operating system and processor for the build target.

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

Figure 5.2 Target Settings Panel



Target Settings

Target Name: c&cpp_app_debug

Linker: ColdFire Linker

Pre-linker: None

Post-linker: None

Output Directory:

{Project}

□ Save project entries using relative paths

Table 5.3 describes the items in this settings panel.

Table 5.3 Target Settings Panel Items

Item	Description
Target Name	This text box contains the name of the build target. This is the name by which you identify the target. The IDE displays this name in several places: • the **Targets** page of the project window • the **Edit** menu • the build target list box un the project window
Linker	Select an item from this list box to set the linker and corresponding compiler the IDE uses to build the target. The IDE changes the settings panels list in the Target Settings window to match your selection. Table 5.4 on page 121 describes the available linker choices.
Pre-Linker	Select an item in this list box to set the prelinker the IDE uses to build the target. Some CodeWarrior products have prelinkers that perform work on object code before it is linked. There are currently no prelinkers available in this product.
Post-Linker	Select an item in this list box to set the postlinker the IDE uses to build the target. Some CodeWarrior products have prelinkers that perform work on the final output file (such as object code format conversion). Table 5.5 on page 122 describes the available post-linker choices.

Table 5.3 Target Settings Panel Items (*continued*)

Item	Description
Output Directory	This is the folder where the IDE places the final output file when the IDE builds the target. Click **Choose** to specify another location for the final output file. Click **Clear** to clear the current directory. The default location is `{Project}`, the folder that contains the CodeWarrior project file.
Save Project Entries Using Relative Paths	Check this checkbox when you want to add two or more files with the same name to a project. When this box is checked, the IDE stores information about project files using relative paths. When searching for project files, the IDE combines Access Paths settings with the stored path information to find the files. When this box is cleared, each project file must have a unique name and the IDE only records information about project files using only the file names. When searching for project files, the IDE only uses only the Access Paths settings to locate the files.

Table 5.4 Linker Items

Item	Description
External Build Linker	The External Build Wizard configures build targets to use this linker to import Makefile-based projects into CodeWarrior IDE projects. To learn more about the External Build Wizard, read the *CodeWarrior IDE User's Guide*.
ColdFire Linker	Select to have the IDE use the ColdFire compiler and linkers to compile and link code for the ColdFire platform.

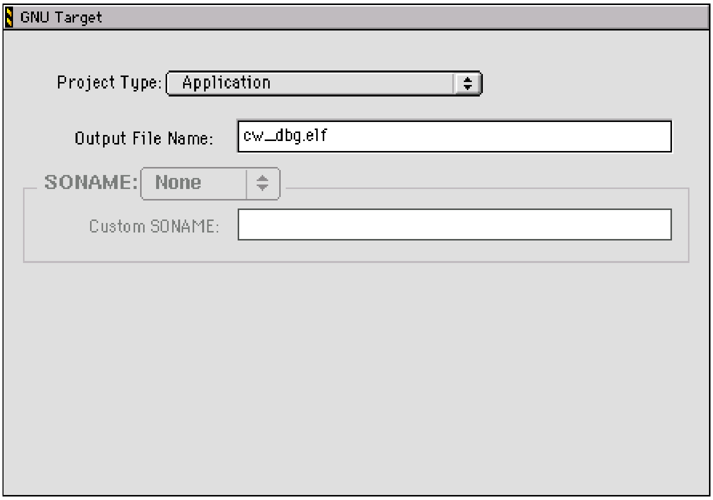
Table 5.5 Post-Linker Items

Post-Linker	Description
ColdFire Post-Linker	Select to enable the GNU Post Linker. For more information about this post-linker read "GNU Post Linker" on page 127.
Shell Tool Post-Linker	Select to have the IDE automatically run shell scripts included in the project at the post-link stage of the build. For details about this post-linker, read "Shell Tool Post-Linker" on page 143.

GNU Target

Use the **GNU Target** settings panel (Figure 5.3) to specify the name and configuration of the final output file. Select **ColdFire Linker** from the **Linker** list box in the **Target Settings** panel to view this settings panel.

Figure 5.3 GNU Target Settings Panel



GNU Target

Project Type: Application

Output File Name: cw_dbg.elf

SONAME: None

Custom SONAME:

Table 5.6 describes the items in this settings panel.

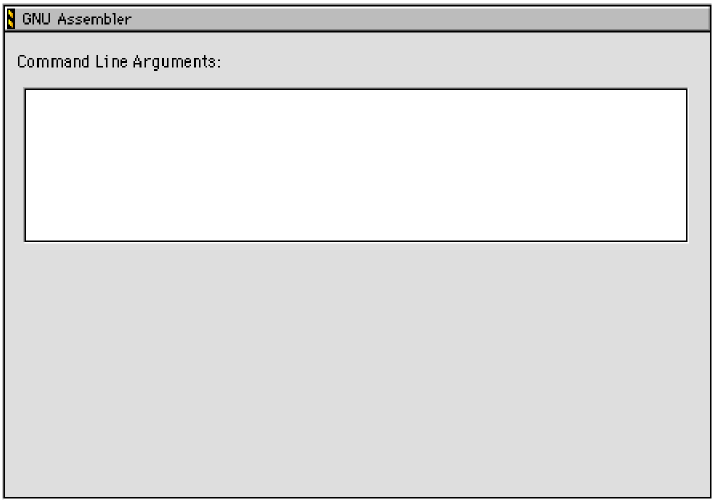
Table 5.6 GNU Target settings panel items

Item	Description
Project Type	Select the type of final output file you want the IDE to generate when you build the target. Available selections are: • Application • Shared Library • Library • Loadable Module The IDE shows and hides some items in this panel based on their relevancy to this selection.
Output File Name	Enter the name of the output file the IDE generates. By convention, application names end with `.elf`, shared library names end with `.so`, library names end with `.a`, and loadable module names end with `.o`.
SONAME	(only available when Project Type is set to Shared Library) Select an item from this list box to set the shared object name for the shared library.

GNU Assembler

Use the **GNU Assembler** settings panel (Figure 5.4) to specify command-line arguments the IDE passes to the GCC assembler. Select **ColdFire Linker** from the **Linker** list box in the **Target Settings** panel to view this settings panel.

Figure 5.4 GNU Assembler Settings Panel

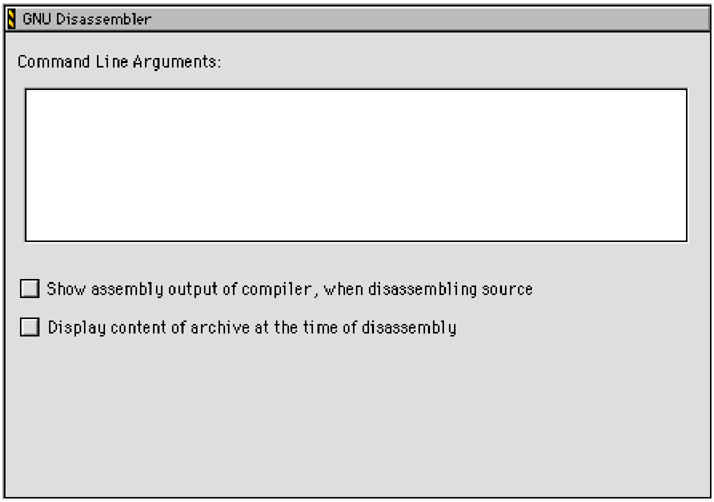


You can enter the command line arguments for the GCC assembler in the **Command Line Arguments** text box. The contents of this text box are passed as command-line switches in the `gcc` command line for each file in your project as they are assembled.

GNU Disassembler

Use the **GNU Disassembler** settings panel (Figure 5.5) to specify command-line arguments the IDE passes to the GCC disassembler. The panel also allows you to control whether the IDE displays disassembly output during disassembly. Select **ColdFire Linker** from the **Linker** list box in the **Target Settings** panel to view this settings panel.

Figure 5.5 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

Table 5.7 describes the items in this settings panel.

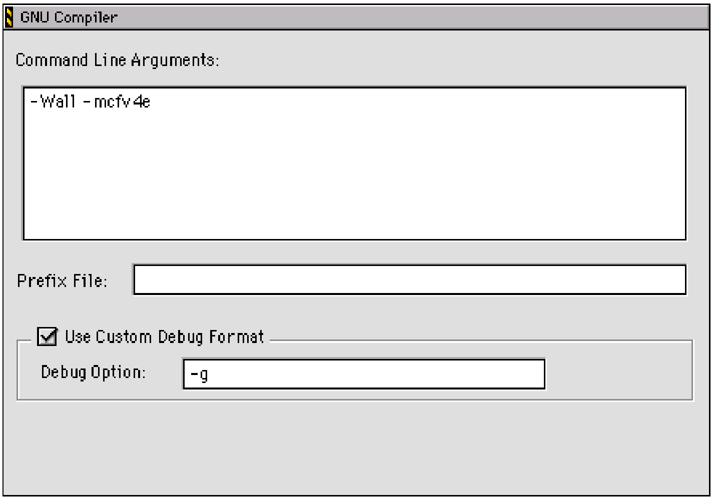
Table 5.7 GNU Disassembler settings panel items

Item	Description
Command Line Arguments	Enter command-line switches you want the IDE to include in the GCC disassembler command line.
Show assembly output of compiler, when disassembling source	Check to have the IDE display the assembly output of the compiler before displaying output from the disassembler.
Display content of archive at the time of disassembly	Check to have the IDE display the list of objects archived in the library while the disassembler is processing it.

GNU Compiler

Use the **GNU Compiler** settings panel (Figure 5.6) to specify command-line arguments for the GCC compiler, prefix file settings, and the format for generating debugging information. Select **ColdFire Linker** from the **Linker** list box in the **Target Settings** panel to view this settings panel.

Figure 5.6 GNU Compiler Settings Panel



Command Line Arguments:

-Wall -mcfv4e

Prefix File:

☑ Use Custom Debug Format

Debug Option: -g

Table 5.8 describes the items in this settings panel.

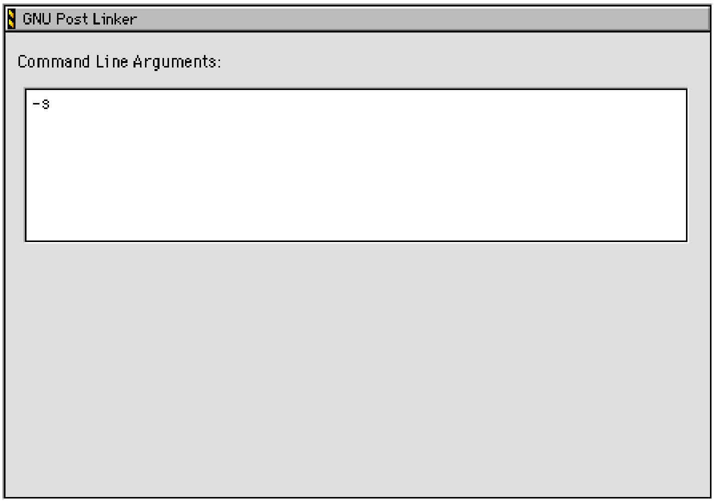
Table 5.8 GNU Compiler settings panel items

Item	Description
Command Line Arguments	Enter command-line switches you want the IDE to include in the GCC compiler command line.
Prefix File	Enter the name of a prefix file you want the IDE to include before each source file in the project during a build. This option corresponds with the `-include` argument of the GCC command-line compiler.
Use Custom Debug Format	Check to have the compiler generate debugging information with a specified custom format. Enter the switch corresponding to the debug format you want the IDE to pass to the GCC cross compiler tools. The CodeWarrior debugger uses the `-gstabs` or `-DWARF2` custom debug format switches. Clear to have the compiler generate debugging information in the default `-g` format switch.

GNU Post Linker

Use the **GNU Post Linker** settings panel (Figure 5.7) to specify command-line arguments the IDE should pass to the post linker adaptor, for example, to strip debugging information from the generated binary executable file. Select **ColdFire Linker** from the **Linker** list box in the **Target Settings** panel to view this settings panel.

Figure 5.7 GNU Post Linker Settings Panel



Command Line Arguments:

```
-s
```

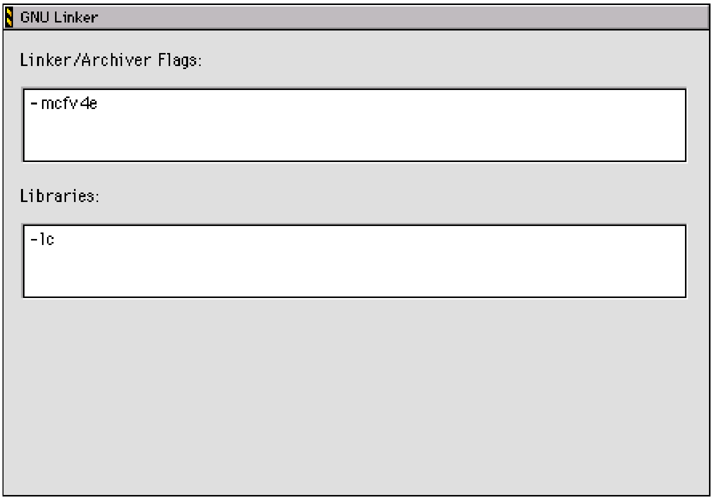
In the **Command Line Arguments** text box, enter the command-line switches the IDE should include in the post linker command line.

WARNING! Do not specify command-line arguments that remove the *ELF Symbol Table*, or you may not be able to debug the stripped file on the target platform. The *ELF Symbol Table* data is required by CodeWarrior TRK for debugging purposes.

GNU Linker

Use the **GNU Linker** settings panel (Figure 5.8) to specify command-line arguments you would like the IDE to pass to the GCC linker during a build. Select **ColdFire Linker** from the **Linker** list box in the **Target Settings** panel to view this settings panel.

Figure 5.8 GNU Linker Settings Panel



GNU Linker

Linker/Archiver Flags:

-mcfv4e

Libraries:

-lc

Table 5.9 describes the items in this settings panel.

Table 5.9 GNU Linker settings panel items

Item	Description
Linker/Archiver Flags	Enter linker command-line switches the IDE should include in the GCC command line for each file in the project.
Libraries	Enter linker command-line switches for any libraries the IDE should include in the GCC command line for each file in the project.

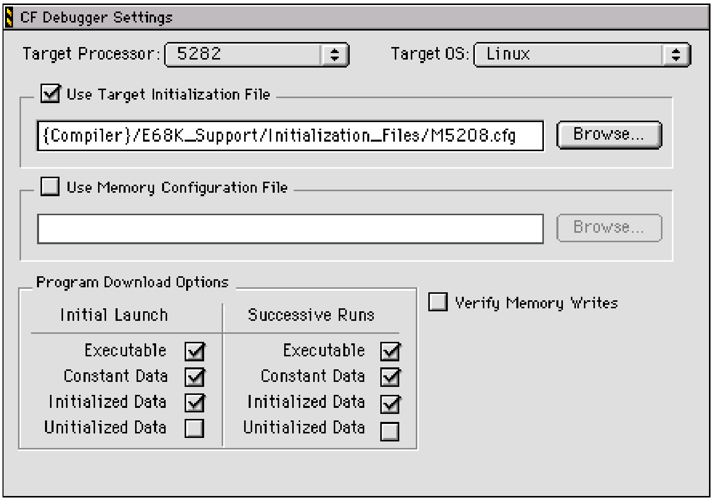
CF Debugger Settings

Use the **CF Debugger Settings** panel (Figure 5.9) to specify the processor, operating system, initialization file, and memory configuration file for the target platform, and other debugger-related options. Select any connection other than a CodeWarrior TRK connection from the **Connection** list box in the **Remote Debugging** panel to view this settings panel.

NOTE We have provided XML files for the platforms and board support packages (BSPs) supported by this product with pre-configured settings for this pane.

The XML files are located in this directory: *`CWInstall`*`/CodeWarriorIDE/CodeWarrior/E68K_Support/KernelDebug_Settings/`.

Figure 5.9 CF Debugger Settings Panel



CF Debugger Settings

Target Processor: 5282   Target OS: Linux

☑ Use Target Initialization File

{Compiler}/E68K_Support/Initialization_Files/M5208.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

Table 5.9 describes the items in this settings panel.

Table 5.10 GNU Linker settings panel items

Item	Description
Target Processor	Select the processor architecture of the target system from this list box. This setting controls the register views the debugger displays.
Target OS	Select the operating system running on the target system from this list box. To do kernel-level debugging, select **Linux**. To do application-level debugging, select **Bareboard**.

Table 5.10 GNU Linker settings panel items (*continued*)

Item	Description
Use Target Initialization File	Check to have the debugger use a target initialization file to initialize the target board for debugging. Click **Browse** to locate and select a target initialization file. Default target initialization files are automatically selected for supported boards. Sample target initialization files are in this directory: *CWInstall*`/CodeWarriorIDE/CodeWarrior/E68K_Support/Initialization_Files/` For more detailed information, see “Debug Initialization Files” on page 153.
Use Memory Configuration File	Check to have the debugger use a memory configuration file to define the valid accessible areas of memory of the target board for debugging.Click **Browse** to locate and select a memory configuration file. If you are using a memory configuration file, and you try to read from an invalid address, the debugger fills the memory buffer with a reserved character (defined in the memory configuration file). If you try to write to an invalid address, the write command is ignored and fails. For more details, see “Memory Configuration Files” on page 161.

Table 5.10 GNU Linker settings panel items (*continued*)

Item	Description
Program Download Options	Check or clear these checkboxes to specify which sections of a program the debugger should download to the target board on initial or successive launches of the program. There are four types of sections listed: • 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 runtime code Check one of these boxes to have the debugger download that section when you debug the program **Note:** You do not need to download uninitialized data if you are using CodeWarrior runtime code.
Verify Memory Writes	Check this checkbox to verify that sections of the program are successfully downloaded to the target board, and that code has not unintentionally modified the sections. For example, after the debugger downloads a text section you might not need to download it again; but you may want to verify that it still exists.

Source Folder Mapping

Use the **Source Folder Mapping** settings panel (Figure 5.10) to configure source and destination folders for executable files you want to debug. These mappings allow the CodeWarrior debugger to find and display source code files even though they are not in the locations specified in the executable file’s debug information.

NOTE When you create a CodeWarrior project by opening an ELF file with the IDE, the IDE automatically creates entries in this settings panel. These entries consist of the ELF file parent folder the existing folder information in the ELF's debug information.

Figure 5.10 Source Folder Mapping Settings Panel

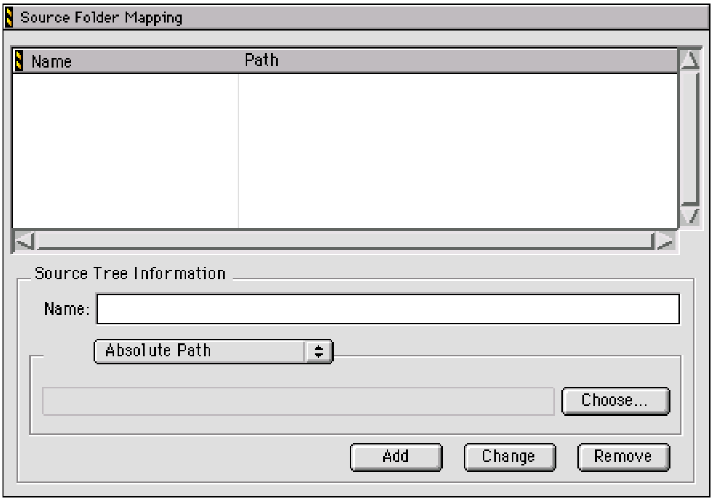


Table 5.11 describes the items in this settings panel.

Table 5.11 Source Folder Mapping Settings Panel Items

Item	Description
Source Folder Mapping (list)	The IDE displays all of the source folder mappings in the current build target in this list. You can use the **Add**, **Change**, and **Remove** buttons to add, change, and remove items in this list.
Build Folder	Enter the path to the folder that *used to* contain the executable's source files when this executable was originally built, or click **Browse** to select the folder.
Current Folder	Enter the path to the folder that *currently* contains the executable's source files, or click **Browse** to select the folder. See "Current Folder" on page 133 for details about this setting.

Table 5.11 Source Folder Mapping Settings Panel Items (*continued*)

Item	Description
Add	Enter paths in the **Build Folder** and **Current Folder** text boxes, then click this button to add a source folder mapping to the list.
Change	Select an existing source folder mapping, edit the paths in the **Build Folder** and **Current Folder** text boxes, then click this button to change the existing source folder mapping in the list.
Remove	Select an existing source folder mapping, then click this button to delete the source folder mapping from the list.

Current Folder

Enter the path to the folder that *currently* contains the executable's source files, or click **Browse** to select the folder.

The supplied path can be the root of a source code tree. For example, if you have the source code files for an executable in these directories:

`/home/me/my_source/headers`

`/home/me/my_source/source`

You might create a source folder mapping with these settings:

- **Build Folder:** `vob_1`
- **Current Folder:** `/home/me/my_source`

With this source folder mapping, if the debugger cannot find a file referenced in the executable's debug information, the debugger replaces the `vob_1` portion of the missing file's path name with `/home/me/my_source` and tries again. The debugger repeats this process for each source folder mapping until it finds the missing file or no more folder pairs remain.

Console I/O Settings

The **Console I/O Settings** panel (Figure 5.11) lets you specify where the CodeWarrior IDE should redirect standard input, standard output, and error messages while debugging an application.

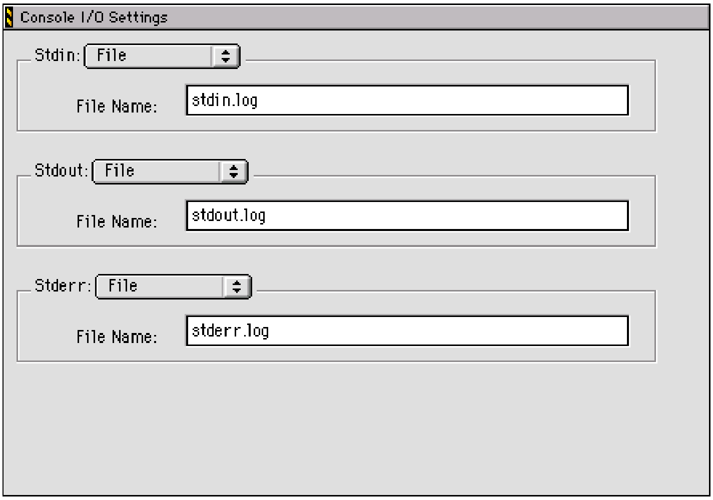
You can redirect standard input, standard output, and error messages to:

- a file on the target system

- the debugger console window
- the console window from where you launched CodeWarrior TRK.

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

Figure 5.11 Console I/O Settings Panel


Console I/O Settings

Stdin: File

File Name: stdin.log

Stdout: File

File Name: stdout.log

Stderr: File

File Name: stderr.log

Table 5.12 describes the items in this settings panel.

Table 5.12 Console I/O Settings Panel Items

Item	Description
Stdin	Select an item from this list box to specify where the CodeWarrior debugger reads standard input during a debug session. For more information, see "Console I/O Redirection Options" on page 135

Table 5.12 Console I/O Settings Panel Items (*continued*)

Item	Description
Stdout	Select an item from this list box to specify where the CodeWarrior debugger writes standard output during a debug session. For more information, see “Console I/O Redirection Options” on page 135
Stderr	Select an item from this list box to specify where the CodeWarrior debugger writes standard error messages during a debug session. For more information, see “Console I/O Redirection Options” on page 135

Console I/O Redirection Options

Each of the list boxes in this settings panel have the menu options listed in Table 5.13.

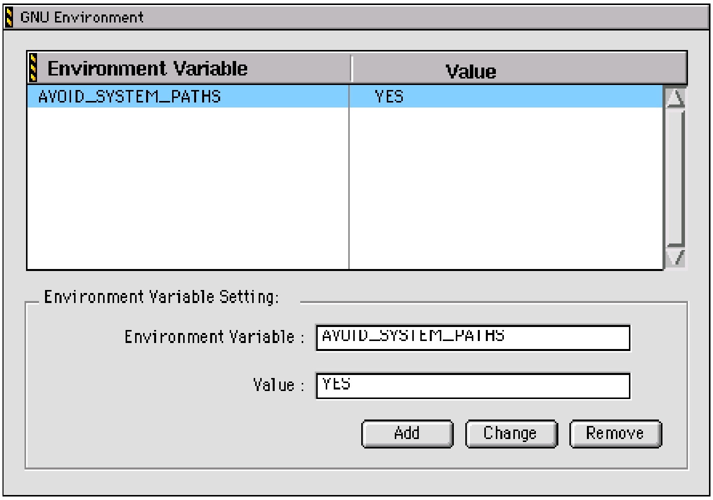
Table 5.13 Console I/O Redirection Options

Menu Item	Description
File	The debugger reads/writes messages to/from the specified file. **Note:** If the file resides on the target system and is not in the same directory as the CodeWarrior TRK binary file on the target system, you must specify the full path on the target system.
Debugger	The debugger reads/writes messages to/from a CodeWarrior debugger console window during the debug session.
Console I/O	The debugger reads/writes messages from the same console window you used to launch CodeWarrior TRK.

GNU Environment

Use the **GNU Environment** settings panel (Figure 5.12) to specify environment variables that you want the ISE to pass to the external compiler, linker, assembler, and other build tool processes when the IDE invokes them. Select **ColdFire Linker** from the **Linker** list box in the **Target Settings** panel to view this settings panel.

Figure 5.12 GNU Environment Settings Panel



Environment Variable	Value
AVOID_SYSTEM_PATHS	YES

Environment Variable : AVOID_SYSTEM_PATHS

Value : YES

Table 5.14 describes the items in this settings panel.

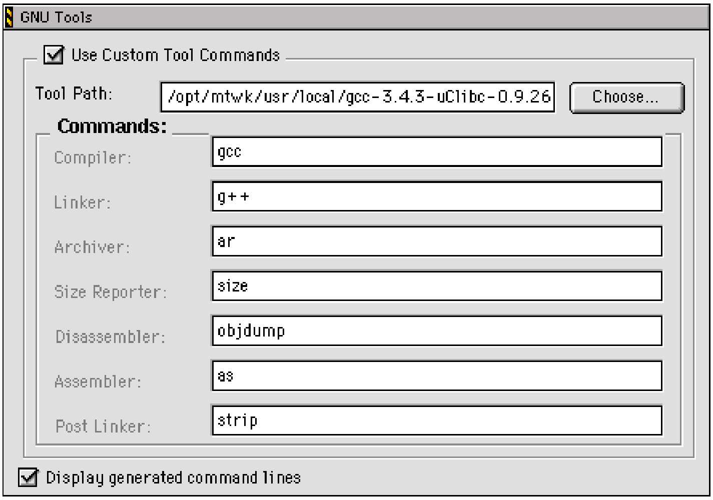
Table 5.14 GNU Environment settings panel items

Item	Description
Environment Variable (list)	This list displays all of the currently-defined GNU environment variables.
Environment Variable (text box)	Enter the name of the environment variable.
Value	Enter the value the environment variable should have.
Add	Click to add the environment variable specified in the **Environment Variable** and **Value** text boxes to the list.
Change	Select an environment variable in the list, edit the **Environment Variable** and **Value** text boxes, and click this button to change the values of the environment variable in the list.
Remove	Select an environment variable in the list and click this button to delete the environment variable from the list.

GNU Tools

Use the **GNU Tools** settings panel (Figure 5.13) to set the path and names of the various command-line tools the IDE should invoke to compile, assemble, link, post-link, disassemble, archive projects, and report the code and data size of project files. Select **ColdFire Linker** from the **Linker** list box in the **Target Settings** panel to view this settings panel.

Figure 5.13 GNU Tools Settings Panel



GNU Tools

☑ Use Custom Tool Commands

Tool Path: /opt/mtwk/usr/local/gcc-3.4.3-uClibc-0.9.26

Commands:

Compiler: gcc

Linker: g++

Archiver: ar

Size Reporter: size

Disassembler: objdump

Assembler: as

Post Linker: strip

☑ Display generated command lines

Table 5.15 describes the items in this settings panel.

Table 5.15 GNU Tools settings panel items

Item	Description
Use Custom Tool Commands	Check to have the IDE use the GNU tools specified in this settings panel, rather than the default tools.
Tool Path	Enter the full path to the folder containing the command-line tools you want to use, or click the **Choose** button to use a standard open folder dialog box to locate and select the folder.
Compiler	Specify only the file name of the compiler tool.
Linker	Specify only the file name of the linker tool.

Table 5.15 GNU Tools settings panel items (*continued*)

Item	Description
Archiver	Specify only the file name of the archiver tool.
Size Reporter	Specify only the file name of the tool that reports code and data size of files after they are compiled. The IDE displays this code and data size information in the project window while building the project.
Disassembler	Specify only the file name of the disassembler tool.
Assembler	Specify only the file name of the assembler tool.
Post Linker	Specify the name of the tool you use to process files after the link stage of a build (for example, to strip debugging information from files).
Display generated command lines	Check to have the IDE display generated command-line input and output during the build process.

6

Working With Hardware Tools

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

This chapter contains these sections:

- Flash Programmer
- Hardware Diagnostics

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 flash, memory/register, and save memory contents to a file) are provided by the CodeWarrior debugger. Therefore, these features are not a part of the CodeWarrior flash programmer.

This product includes a default flash configuration file for supported target boards. These files have the `.xml` filename extension, and are located in this directory:

`CWInstall``/CodeWarriorIDE/CodeWarrior/CodeWarrior_Plugins/Support/Flash_Programmer/ColdFire`

To load a flash configuration file:

1. Select **Tools > Flash Programmer**

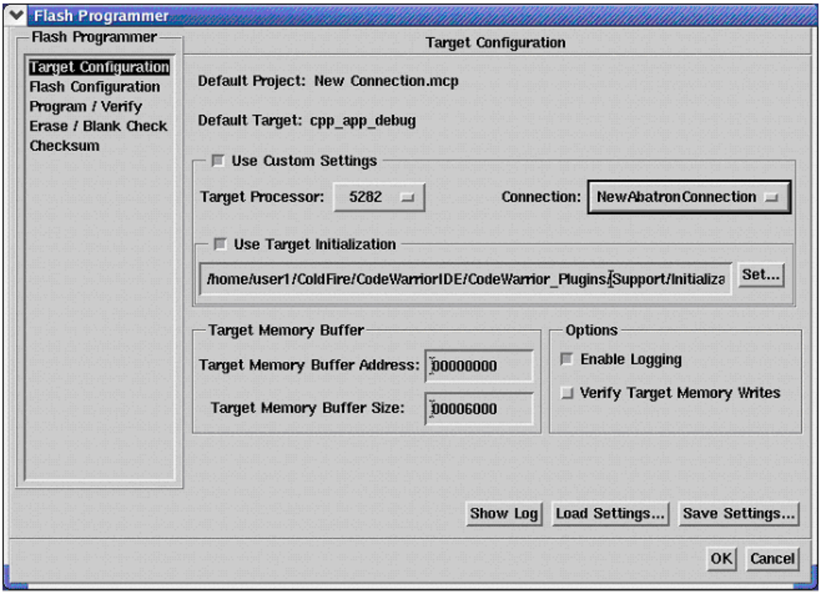
   The **Flash Programmer** window appears.

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

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

   The Target Configuration preference panel appears on the right side of the **Flash Programmer** window. (See Figure 6.1).

Figure 6.1 Flash Programmer window



Default Project: New Connection.mcp

Default Target: cpp_app_debug

☐ Use Custom Settings

Target Processor: 5282

Connection: NewAbatronConnection

☐ Use Target Initialization

/home/user1/ColdFire/CodeWarriorIDE/CodeWarrior_Plugins/Support/Initializa

Target Memory Buffer

Target Memory Buffer Address: 00000000

Target Memory Buffer Size: 00006000

Options

☐ Enable Logging

☐ Verify Target Memory Writes

3. Click **Load Settings**

   A standard open file dialog box appears.

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

5. Click **Open**

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

6. Click **OK**

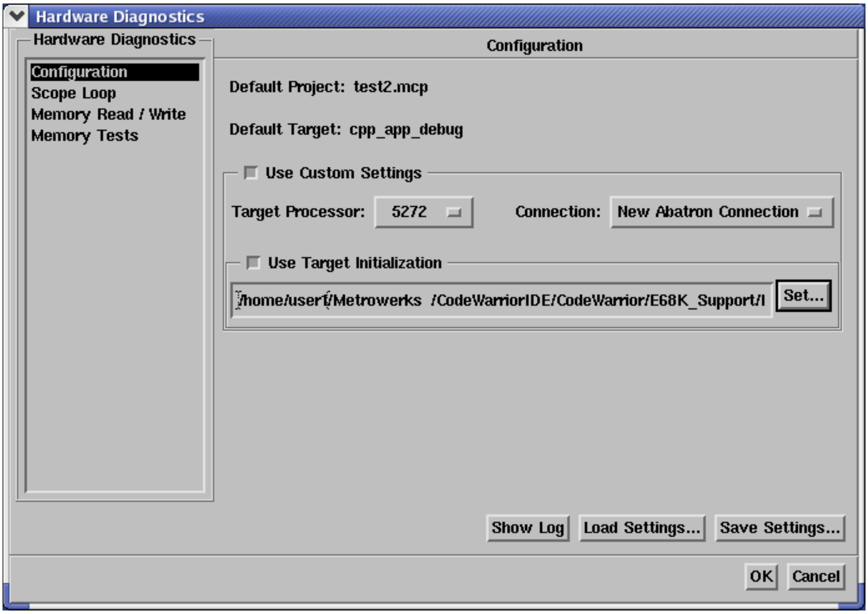
   The **Flash Programmer** window saves your selections and closes.

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

Hardware Diagnostics

The **Hardware Diagnostics** window (Figure 6.2) lists global options for the hardware diagnostic tools. These preferences apply to every open project file. Select **Tools > Hardware Diagnostics** to display the **Hardware Diagnostics** window.

Figure 6.2 Hardware Diagnostics window



Default Project: test2.mcp

Default Target: cpp_app_debug

□ Use Custom Settings

Target Processor: 5272

Connection: New Abatron Connection

□ Use Target Initialization

/home/user1/Metrowerks /CodeWarriorIDE/CodeWarrior/E68K_Support/I

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

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

A

Shell Tool Post-Linker

This appendix describes the CodeWarrior Shell Tool post linker and explains how to use it with your CodeWarrior projects.

You can use the Shell Tool post-linker to automatically run shell scripts as part of the IDE's build process, either during the precompile stage or during the post-link stage of a build. One of its most common and useful purposes occurs during post-compilation to copy additional files or resources into the application package created by the IDE.

This appendix contains these sections:

- Shell Tool Setup
- Environment Variables
- Shell Tool Example

Shell Tool Setup

To use the Shell Tool post-linker in a build target:

1. Select **Edit >** ***TargetName*** **Settings** from the CodeWarrior menu bar (where ***TargetName*** is the name of the build target you want to use the Shell Tool).

   The IDE displays the Target Settings window.

2. Select **Target Settings** from the **Target Settings Panels** list on the left side of the window.

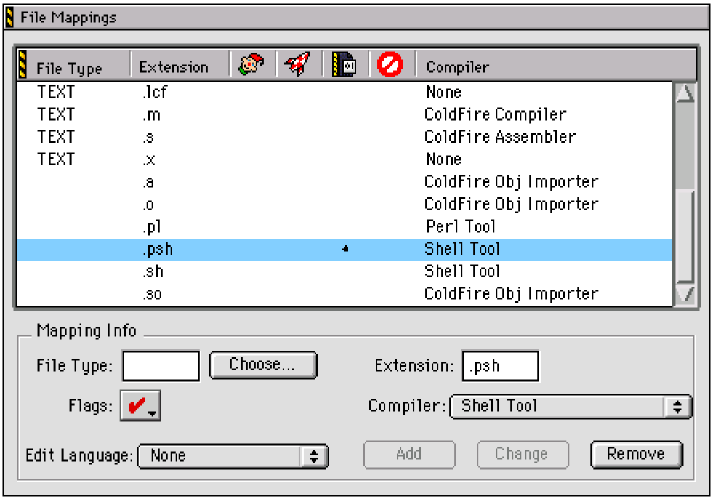
   The IDE displays the Target Settings panel.

3. Select **Shell Tool Post Linker** from the **Post-linker** menu.

4. Select **File Mappings** from the **Target Settings Panels** list on the left side of the window.

   The IDE displays the File Mappings settings panel (Figure A.1).

Figure A.1 File Mappings settings panel showing Shell Tool mappings



File Type	Extension					Compiler
TEXT	.lcf					None
TEXT	.m					ColdFire Compiler
TEXT	.s					ColdFire Assembler
TEXT	.x					None
	.a					ColdFire Obj Importer
	.o					ColdFire Obj Importer
	.pl					Perl Tool
	.psh			•		Shell Tool
	.sh					Shell Tool
	.so					ColdFire Obj Importer

Mapping Info

File Type:

Extension: .psh

Flags: ✓

Compiler: Shell Tool

Edit Language: None

5. If they do not exist, add new entries with the following information:

   For scripts that are to run *after* the target is linked (post-link stage):

   - **File Type:** TEXT
   - **Extension:** **`.sh`** (or whatever you use for shell scripts)
   - **Compiler:** Shell Tool
   - **Flags:** None (no checkmarks in the **Flags** menu)

   For scripts that are to run *before* compiling begins (pre-compile stage):

   - **File Type:** TEXT
   - **Extension:** **`.psh`** (or whatever you use for shell scripts)
   - **Compiler:** Shell Tool
   - **Flags:** Precompiled

NOTE If you wish to run shell scripts during both post-link and pre-compile stages, the filename extensions must be different for each file mapping, as shown above.

6. Add your shell scripts to the build target.

NOTE The Shell Tool only parses source files with UNIX style line endings. Make sure the line endings in your files are correct.

7. Build the target.

TIP If there are multiple shell scripts in a stage, the IDE executes the scripts in the order they appear in the **Link Order** page of the project window.

Environment Variables

The IDE passes the environment variables in Table A.1 to the shell script. The working directory for shell scripts the IDE invokes is the **Output Directory** specified in the **Target Settings** settings panel. All output the script directs to stdout appears in a new CodeWarrior text window after the script completes.

Table A.1 Shell Tool environmental variables

Variable	Description
`$MW_CURRENT_TARGET`	the name of the current build target from the **Target Settings** settings panel in the IDE
`$MW_PROJECT_DIRECTORY`	the directory containing the IDE project that is running the script
`$MW_PROJECT_FILE`	the full path to the CodeWarrior project file
`$MW_PROJECT_NAME`	the file name of the project file
`$MW_OUTPUT_DIRECTORY`	the **Output Directory** from the **Target Settings** settings panel in the IDE
`$MW_OUTPUT_FILE`	the full path to the output file
`$MW_OUTPUT_NAME`	the name of the build target output file

NOTE Be careful with items placed into environment variables. Any item containing non-alphanumeric values may cause the Shell Tool not to operate as you expect, or not work at all.

Shell Tool Example

Listing A.1 shows one way of writing a shell script that the IDE can use to verify the Shell Tool's environment variables.

Listing A.1 Shell Tool example script

```
#!/bin/sh

cd ${HOME}
FILEWANTED=.tcshrc
OBJECTS=”`ls -al ./ | grep ${FILEWANTED}`”

echo “This is a simple test of the shell tool plugin...”
echo
echo
echo “=====================================================”
echo “Check if passed the IDE ENV variables correctly...”
echo “=====================================================”
echo  “The name of the current target from the “Target Settings”
echo  “pref panel in the IDE is:”
echo  $MW_CURRENT_TARGET
echo
echo  “The directory containing the IDE project that is running the”
echo  “script is:”
echo  $MW_PROJECT_DIRECTORY
echo
echo  “The full path to the project file is:”
echo  $MW_PROJECT_FILE
echo
echo  “The basename of the project file is:”
echo  $MW_PROJECT_NAME
echo
echo  “The output directory from the “Target Settings” pref panel”
echo  “in the IDE is:”
echo  $MW_OUTPUT_DIRECTORY
echo
echo  “The full path to the output file, if any, is:”
echo  $MW_OUTPUT_FILE
echo
echo  “The basename of the output file, if any, is:”
echo  $MW_OUTPUT_NAME
echo
echo


echo “=====================================================”
echo “Now list the \”.tcshrc\” file found in your HOME directory...”
echo “=====================================================”
echo

if [ ! -x ${OBJECTS} ]
then
```

```
   echo “found: ${OBJECTS}”
   cat ${FILEWANTED}
else
   echo “shell tool could not find ${FILEWANTED} in ${PWD}”
fi
```

B

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.

The CodeWarrior IDE packages the cross compiler tools for all the supported target platforms. Table B.1 lists the location where you can find the target platform-specific cross compiler tools.

Table B.1 Cross Compiler Tools Locations

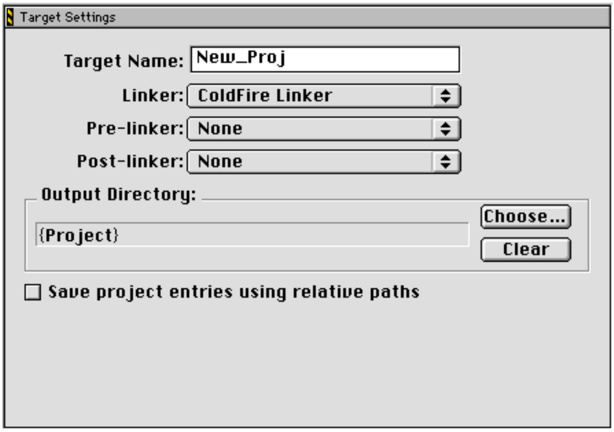
Platform	Cross Compiler Tools Location
MCF5208	`/opt/mtwk/usr/local/gcc-3.4.0-uClibc-20050919/m68k-uclinux/m68k-uclinux/bin/`
MCF5282 and MCF5272	`/usr/local/bin/`
MCF5475 and MCF5485	`/opt/Embedix/usr/local/m68k-linux/gcc-3.4.0-glibc-2.3.2-v4e/bin/`

NOTE This procedure 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 the **Target Settings** option from the panel tree. The **Target Settings** panel appears.
3. Select the appropriate linker from **Linker** list box (Figure B.1).

Figure B.1 Linker Settings



Target Name: New_Proj

Linker: ColdFire Linker

Pre-linker: None

Post-linker: None

Output Directory: {Project}

□ Save project entries using relative paths

4. Click **Save** to save the settings.
5. Specify the path where the third party cross compiler tools are installed/copied.
   a. Select the **GNU Tools** option from the panel tree.
   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/coldfire,` then the Tool Path will be at `/usr/local/coldfire/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.
6. Change the access path settings for kernel and gcc-lib include files in the Access Paths settings panel.
   a. Select the **Access Paths** options from the panel tree. 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. 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.

C

Debug Initialization Files

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

This appendix contains these sections:

- Using Debug Initialization Files
- Debug Initialization File Commands

Using Debug Initialization Files

A debug initialization file is a command file processed and executed each time the debugger is invoked. It is usually necessary to include an initialization file if debugging via BDM or JTAG to ensure that the target memory is initialized correctly and that any register values that need to be set for debugging purposes are set correctly. You specify whether or not to use an initialization file and which file to use in the **CF Debugger Settings** panel.

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

We provide samples of initialization files for supported evaluation boards. The sample files are located here:

CWInstall/CodeWarriorIDE/CodeWarrior/E68K_Support/Initialization_Files/

Debug Initialization File Commands

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

- white spaces and tabs are ignored
- character case is ignored
- unless otherwise notes, values may be specified in hexadecimal, octal, or decimal:
  - hexadecimal values are preceded by 0x (for example, 0xDEADBEEF)
  - octal values are preceded by 0 (for example, 01234567)

  - decimal values start with a non-zero numeric character (for example, 1234)
- comments start with a semicolon (`;`) or pound sign (#), and continue to the end of the line

ANDmem.l

Reads four bytes starting a the specified address, performs a bit AND operation of this value with the supplied 32-bit mask, and writes the result back to the same address. No read/write verify is performed.

Syntax

`ANDmem.l` *`address mask`*

Arguments

`address`

the address at which the command should start reading four bytes of data

`mask`

the 32-bit mask

Example

To perform a bit AND operation with the four-byte value at memory location `0xC30A0004` and the 32-bit mask `0xFFFFFFFF`, and store the resulting value back into memory location `0xC30A0004`:

```
ANDmem.l 0xC30A0004 0xFFFFFEFF
```

ORmem.l

Reads four bytes starting a the specified address, performs a bit OR operation of this value with the supplied 32-bit mask, and writes the result back to the same address. No read/write verify is performed.

Syntax

`ORmem.l` *`address mask`*

Arguments

`address`

the address at which the command should start reading four bytes of data

`mask`

the 32-bit mask

Example

To perform a bit OR operation between the four-byte value at memory location `0xC30A0008` with the 32-bit mask `0x01000800`, and store the resulting value back into memory location `0xC30A0004`:

```
ORmem.l 0xC30A0008 0x01000800
```

hreset

Performs a hard reset of the target system.

Syntax

```
hreset
```

sreset

Performs a soft reset of the target system.

Syntax

```
sreset
```

run

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

Syntax

```
run
```

sleep

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

Syntax

```
sleep milliseconds
```

Arguments

`milliseconds`

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

Example

To pause execution for 10 milliseconds:

```
sleep 10
```

stop

Stops program execution and halts the target processor.

Syntax

```
stop
```

physicalbase

Sets the physical base address on the target system for the kernel.

Syntax

```
physicalbase address
```

Arguments

`address`

the physical base memory address (in hexadecimal, octal, or decimal)

Example

To set the physical base address of the kernel to memory location `0x00010000`:

```
physicalbase 0x00010000
```

virtualbase

Sets the virtual base address on the target system for the kernel.

Syntax

`virtualbase` *`address`*

Arguments

`address`

the virtual base memory address (in hexadecimal, octal, or decimal)

Example

To set the virtual base address of the kernel to memory location `0x00000C00`:

```
virtualbase 0x00000C00
```

semihosting

Enables or disables semihosting while using the ColdFire RDI protocol.

Syntax

`semihosting` *`value`*

Arguments

`value`

a boolean value indicating whether semihosting should be enabled. Specify 1 to enable semihosting. Specify 0 to disable semihosting.

Example

To enable semihosting while using the ColdFire RDI protocol:

```
semihosting 1
```

writemem.b

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

Syntax

`writemem.b` *`address value`*

Arguments

`address`

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

`value`

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

Example

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

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

writemem.w

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

Syntax

`writemem.w` *`address value`*

Arguments

`address`

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

`value`

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

Example

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

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

writemem.l

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

Syntax

`writemem.l` *`address value`*

Arguments

`address`

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

`value`

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

Example

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

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

writereg

Writes the specified data to a register.

Syntax

`writereg` *`regName value`*

Parameters

`regName`

the name of the register

`value`

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

Example

To write the value `0x00001002` to the MSR register:

```
writereg MSR 0x00001002
```

D

Memory Configuration Files

A memory configuration file contains commands that define the accessible areas of memory for your specific board.

This appendix consists of these topics:

- Command Syntax
- Memory Configuration File Commands

Command Syntax

In general, the syntax of memory configuration file commands follows these rules:

- white spaces and tabs are ignored
- character case is ignored
- unless otherwise notes, values may be specified in hexadecimal, octal, or decimal:
  - hexadecimal values are preceded by 0x (for example, 0xDEADBEEF)
  - octal values are preceded by 0 (for example, 01234567)
  - decimal values start with a non-zero numeric character (for example, 1234)
- comments start with standard C and C++ comment characters, and continue to the end of the line

Memory Configuration File Commands

This section lists the command name, its usage, a brief explanation of the command, examples of how the command may appear in configuration files, and any important notes about the command.

range

This command sets debugger access to a block of memory.

Syntax

`range` *`loAddress hiAddress size access`*

Arguments

`loAddress`

the starting address of the memory range

`hiAddress`

the ending address of the memory range

`size`

the size, in bytes, the debug monitor or emulator uses for memory accesses

`access`

controls what type of access the debugger has to the memory block — supply one of: `Read`, `Write`, or `ReadWrite`

Examples

To set memory locations `0xFF000000` through `0xFF0000FF` to read-only with a size of 4 bytes:

```
range 0xFF000000 0xFF0000FF 4 Read
```

To set memory locations `0xFF0001000` through `0xFF0001FF` to write-only with a size of 2 bytes:

```
range 0xFF000100 0xFF0001FF 2 Write
```

To set memory locations `0xFF0002000` through `0xFFFFFFFF` to read and write with a size of 1 byte:

```
range 0xFF000200 0xFFFFFFFF 1 ReadWrite
```

reserved

This command allows you to specify a reserved range of memory. If the debugger attempts to read reserved memory, the resulting buffer is filled with the reserved character. If the debugger attempts to write to reserved memory, no write takes place.

NOTE Refer to "reservedchar" on page 163 for information showing how to set the reserved character.

Syntax

```
reserved loAddress hiAddress
```

Arguments

`loAddress`

the starting address of the memory range

`hiAddress`

the ending address of the memory range

Examples

To reserve memory starting at `0xFF000024` and ending at `0xFF00002F`:

```
reserved 0xFF000024 0xFF00002F
```

reservedchar

This command sets the reserved character for the memory configuration file. When the debugger attempts to read a reserved or invalid memory location, it fills the buffer with this character.

Syntax

```
reservedchar rChar
```

Arguments

`rChar`

the one-byte character the debugger uses when it accesses reserved or invalid memory

Example

To set the reserved character to “∞“:

```
reservedchar 0xB0
```

E

Frequently Asked Questions

This appendix discusses the frequently asked questions about CodeWarrior Development Studio for ColdFire targets.

This appendix has these topics:

- Settings
- Debugging
- CodeWarrior IDE

Settings

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.

Debugging

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: I am unable to launch an executable using exec() system call from a thread program. The debugger displays the 'CodeWarrior TRKProtocolPlugin: 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.

CodeWarrior IDE

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.

Index