CodeWarrior Development Studio for Microcontrollers Version10.x

Profiling and Analysis Tools

User Guide

Revised: January 12, 2011

freescale semiconductor

How to Contact Us

Corporate Headquarters	Freescale Semiconductor, Inc. 6501 William Cannon Drive West Austin, TX 78735 U.S.A.
World Wide Web	`http://www.freescale.com/codewarrior`
Technical Support	`http://www.freescale.com/support`

Table of Contents

1

Introduction

The CodeWarrior Profiling and Analysis tool is designed to help you make your code more efficient so that you can enhance the speed and performance of your applications. The CodeWarrior Profiling and Analysis tool helps you get hard and reliable data using which you can analyze the time spent by your code in performing various tasks. This user guide explains how to use the CodeWarrior Profiling and Analysis tool.

In this chapter, refer to the following topics:

Release Notes

Before using the CodeWarrior IDE, read the developer notes. These notes contain important information about last-minute changes, bug fixes, incompatible elements, or other topics that may not be included in this user guide.

NOTE The release notes for specific components of the CodeWarrior IDE are located in the `Release_Notes` folder in the CodeWarrior installation directory.

If you are new to the CodeWarrior IDE, read this chapter and the Getting Started chapter. This chapter provides references to resources of interest to new users; the Getting Started chapter helps you familiarize with the software features.

Related Documentation

This topic provides information about documentation related to the CodeWarrior IDE and Freescale Microcontrollers development.

CodeWarrior Information

- To view the online help for the CodeWarrior tools, select **Help > Help Contents** from the IDE menu bar. Select **Profiling and Analysis Users Guide** from the **Contents** list.

- For late-breaking information about new features, bug fixes, known problems, and incompatibilities, read the release notes in this folder:

  `CWInstallDir\MCU\Release_Notes`

  where `CWInstallDir` is the directory that CodeWarrior is installed into.
- For general information about the CodeWarrior IDE and debugger, see the *Freescale Eclipse Extension Guide* at the following location:

  `CWInstallDir\MCU\Help\PDF`

NOTE The *Freescale Eclipse Extension Guide* is a general guide that is also a part of other CodeWarrior Eclipse-based products. Therefore, it describes various features that are not available in Microcontrollers v10.x, such as Cache, Memory Management Unit (MMU) Configurator, and Multicores.
Also, it shows screenshots that are not just specific to Microcontrollers v10.x. Sometimes the screenshots are of other CodeWarrior products, such as StarCore v10.x or Power Architecture v10.x.

Table 1.1 lists the additional documents you can refer to for more information about CodeWarrior for Microcontrollers 10.x. These documents are categorized according to the four different documentation types as Getting Started, User Guides, Application Notes, and Supporting Information.

Table 1.1 Related Documentation

Documentation Type	Document	Description	PDF Location
Getting Started	Microcontrollers V10.x Getting Started Guide	Contains information to get you started using the CodeWarrior Development Studio to develop software that targets the HCS08/ RS08, ColdFire and Power architectures	*<CWInstallDir>*\MCU \ Getting Started Guide for Microcontrollers.pdf

Table 1.1 Related Documentation (*continued*)

Documentation Type	Document	Description	PDF Location
Getting Started	Microcontrollers V10.x Quick Start	Explains the steps to install Microcontrollers V10.x, and create and debug a project.	*<CWInstallDir>*\MCU \ Quick Start for Microcontrollers.pdf
	CodeWarrior Project Importer Quick Start	Explains the steps to convert a classic CodeWarrior project into an Eclipse IDE project.	*<CWInstallDir>*\MCU \ CodeWarrior Project Importer Quick Start.pdf
	Eclipse Quick Reference Card	Introduces you to the interface of CodeWarrior for Microcontrollers V10.x Eclipse-based IDE and provides a quick reference to the key bindings.	*<CWInstallDir>*\MCU \ Eclipse Quick Reference Card.pdf
	HCS08 Profiling and Analysis for Microcontrollers V10.x Quick Start	Explains how to collect trace and critical code data after creating, building, and running a project on the HCS08 MC9S08QE128 target in the CodeWarrior for Microcontrollers version 10.x debugger.	*<CWInstallDir>*\MCU \ HCS08 Profiling and Analysis Quick Start for Microcontrollers.pdf
	ColdFire Profiling and Analysis for Microcontrollers V10.x Quick Start	Explains how to collect trace and critical code data after creating, building, and running a project on the ColdFire V1 MCF51JM128 target in the CodeWarrior for Microcontrollers version 10.x debugger.	*<CWInstallDir>*\MCU \ ColdFire V1 Profiling and Analysis Quick Start for Microcontrollers.pdf
	Ethernet TAP Quick Start	Explains how to set up the Ethernet TAP probe for Freescale microcontrollers and processors.	*<CWInstallDir>*\MCU \ Ethernet TAP Quick Start for Microcontrollers.pdf

Table 1.1 Related Documentation (*continued*)

Documentation Type	Document	Description	PDF Location
User Guide	Freescale Eclipse Extensions Guide	Explains extensions to the CodeWarrior Eclipse IDE across all CodeWarrior products.	*<CWInstallDir>*\MCU\ Help\PDF\Freescale Eclipse Extensions Guide.pdf
	Microcontrollers V10.x Targeting Manual	Explains how to use CodeWarrior Development Studio for Microcontrollers V10.x	*<CWInstallDir>*\MCU\ Help\PDF\Targeting Microcontrollers.pdf
	Microcontrollers V10.x HC08 Build Tools Reference Manual	Describes the compiler used for the Freescale 8-bit Microcontroller Unit (MCU) chip series.	*<CWInstallDir>*\MCU\ Help\PDF\MCU_HCS08_Compiler.pdf
	Microcontrollers V10.x RS08 Build Tools Reference Manual	Describes the ANSI-C/C++ Compiler used for the Freescale 8-bit Microcontroller Unit (MCU) chip series.	*<CWInstallDir>*\MCU\ Help\PDF\MCU_RS08_Compiler.pdf
	Microcontrollers V10.x ColdFire Build Tools Reference Manual	Describes the compiler used for the Freescale 8-bit Microcontroller Unit (MCU) chip series	*<CWInstallDir>*\MCU\ Help\PDF\MCU_ColdFire_Compiler.pdf
	Microcontrollers V10.x Power Architectures Processors Build Tools Reference Manual	Describes the compiler used for the Power Architectures Processors	*<CWInstallDir>*\MCU\ Help\PDF\MCU_Power-Architecture_Compiler.pdf

Table 1.1 Related Documentation (*continued*)

Documentation Type	Document	Description	PDF Location
User Guide	Microcontrollers V10.x Kinetis Build Tools Reference Manual	Describes the compiler used for the Freescale 32-bit Microcontroller Unit (MCU) chip series.	*<CWInstallDir>*\MCU\Help\PDF\MCU_Kinetis_Compiler.pdf
	Microcontrollers V10.x MISRA-C:2004 Compliance Exceptions for the HC(S)08, RS08, ColdFire, Kinetis and Power Architecture Libraries Reference Manual	Describes the MISRA-C:2004 compliance exceptions for the HC(S)08, RS08, ColdFire, Kinetis and Power Architecture libraries.	*<CWInstallDir>*\MCU\ Help\PDF\MISRA_C_2004 Compliance Exceptions.pdf
	CodeWarrior Development Tools EWL C Reference	Describes the contents of the Embedded Warrior Library for C. This document is available only in ColdFire Architecture.	*<CWInstallDir>*\MCU\ Help\PDF\EWL C Reference.pdf
	CodeWarrior Development Tools EWL C++ Reference	Describes the contents of the Embedded Warrior Library for C++. This document is available only in ColdFire Architecture.	*<CWInstallDir>*\MCU\ Help\PDF\EWL C++ Reference.pdf
	Microcontrollers V10.x HC(S)08/ RS08 Assembler Reference Manual	Explains how to use the HC(S)08/RS08 Macro Assembler	*<CWInstallDir>*\MCU\ Help\PDF\HCS08-RS08_Assembler_MCU_Eclipse.pdf

Table 1.1 Related Documentation (*continued*)

Documentation Type	Document	Description	PDF Location
User Guide	Microcontrollers V10.x ColdFire Assembler Reference Manual	Explains the assembly-language syntax and IDE settings for the ColdFire assemblers	*<CWInstallDir>*\MCU\ Help\PDF\ColdFire_Assembler_MCU_Eclipse.pdf
	Microcontrollers V10.x Kinetis Assembler Manual	Explains the corresponding assembly-language syntax and IDE settings for these assemblers.	*<CWInstallDir>*\MCU\Help\PDF\Kinetis_Assembler_MCU_Eclipse.pdf
	Microcontrollers V10.x HC(S)08/ RS08 Build Tools Utilities Manual	Describes the following five CodeWarrior IDE utilities: SmartLinker, Burner, Libmaker, Decoder, and Maker.	*<CWInstallDir>**MCU*\ Help\PDF\Build_Tools_Utilities.pdf
	USB TAP Users Guide	Explains the steps to develop and debug a number of processors and microcontroller using CodeWarrior USB TAP probe.	*<CWInstallDir>*\MCU\ Help\PDF\USB TAP Users Guide.pdf
	Ethernet TAP Users Guide	Explains the steps to develop and debug a number of processors and microcontroller using CodeWarrior Ethernet TAP probe.	*<CWInstallDir>*\MCU\ Help\PDF\Ethernet TAP Users Guide.pdf
	Open Source BDM-JM60 Users Guide	Describes an Open Source programming and debugging development tool designed to work with Freescale HCS08, RS08, Coldfire V1,V2, V3 and V4, and DSC56800E microcontrollers.	*<CWInstallDir>*\MCU\ Help\PDF\OSBDM-JM60_Users_Guide.pdf

Table 1.1 Related Documentation (*continued*)

Documentation Type	Document	Description	PDF Location
	Processor Expert Users Manual	Provides information about Processor Expert plug-in, which generates code from the Embedded Beans.	*<CWInstallDir>*\MCU\ Help\PDF\ProcessorExpertHelp.pdf
	Device Initialization Users Manual	Provides information about the user interface, creating a simple design, configuring a device, generating initialization code, and using it in your application.	*<CWInstallDir>*\MCU\ Help\PDF\DeviceInitHelp.pdf
	Signal Processing Engine Auxiliary Processing Unit Programming Interface Manual	Helps programmers provide software that is compatible across the family of Power Architecture processors that use the signal processing engine auxiliary processing unit.	*<CWInstallDir>*\MCU\ Help\PDF\SPE Programming Interface Manual.pdf
Application Note	AN3859 - Adding Device(s) to the CodeWarrior Flash Programmer for Microcontrollers V10.x	Explains how to use the Flash Tool Kit to support additional flash devices on the Flash Programmer for CodeWarrior Development Studio for Microcontrollers V10.x.	*<CWInstallDir>*\MCU\ Help\PDF\AN3859.pdf
	AN3967 - How to Write Flash Programming Applets	Provides information on creating Flash configuration files for the Flash Programming interface.	*<CWInstallDir>*\MCU\ Help\PDF\AN3967.pdf
	AN4095 - CodeWarrior Build Tools Options for Optimal Performance on the Power Architecture e200 Core	Provides information on CodeWarrior build tools options for optimal performance on the Power ARchitecture e200 Core	*<CWInstallDir>*\MCU\ Help\PDF\AN4095.pdf

Table 1.1 Related Documentation (*continued*)

Documentation Type	Document	Description	PDF Location
Application Note	AN4104 - Converting Classic ColdFire Projects to Microcontrollers V10.x	Explains how to convert a ColdFire project created in CodeWarrior Development Studio for Microcontrollers V6.2 or CodeWarrior Development Studio for ColdFire Architectures V7.1 to CodeWarrior Development Studio for Microcontrollers V10.x	`<CWInstallDir>\MCU\ Help\PDF\AN4104.pdf`
	AN4188 - RS08 Upper Memory Access	Provides the RS08 programmer with information about the RS08 Upper Memory Access for Microcontrollers V10.x.	`<CWInstallDir>\MCU\ PDF\AN4188.pdf`
Supporting Information	Microcontrollers V10.x FAQ Guide	Lists most frequently asked or anticipated questions and answers to CodeWarrior Development Studio for Microcontrollers V10.x.	`<CWInstallDir>\MCU\ Help\PDF\Microcontrollers_FAQ_Guide.pdf`

2

Getting Started

The CodeWarrior Profiling and Analysis tool lets you collect data of an application. You can analyze this data to identify the bottlenecks, such as slow execution of routines or heavily-used routines within the application. This chapter explains features of the CW Profiling and Analysis tool, the CW interface that this tool uses, and the type of data that is collected using the tool.

Refer to the following topics:

Profiling and Analysis Tools

CodeWarrior Profiling and Analysis tools provide visibility into an application as it runs on the hardware. This visibility can help you understand how your application runs, as well as identify operational problems. The tools make it easy to collect the data.

CodeWarrior Profiling and Analysis tools are supported on both Windows and Linux environment.

Following are the basic features of the tools.

- Basic setup can be done using the **Trace and Profile** tab in the **Debug Configurations** dialog box
- Data files can be shared between teams
- Support for the HCS08, Coldfire V1–V4, ColdFire V4e, and Kinetis targets
- Trace is collected by setting triggers and using various trigger conditions — Applicable for HCS08, ColdFire V1 and Kinetis targets
- Trace is collected even when no triggers are set — Applicable only for the HCS08 target
- Profiling information is collected — Applicable for Coldfire V1–V4 and ColdFire V4e targets

The tools also provide user-friendly data viewing features and enables you to:

- step through trace data and the corresponding source code of that trace data simultaneously,

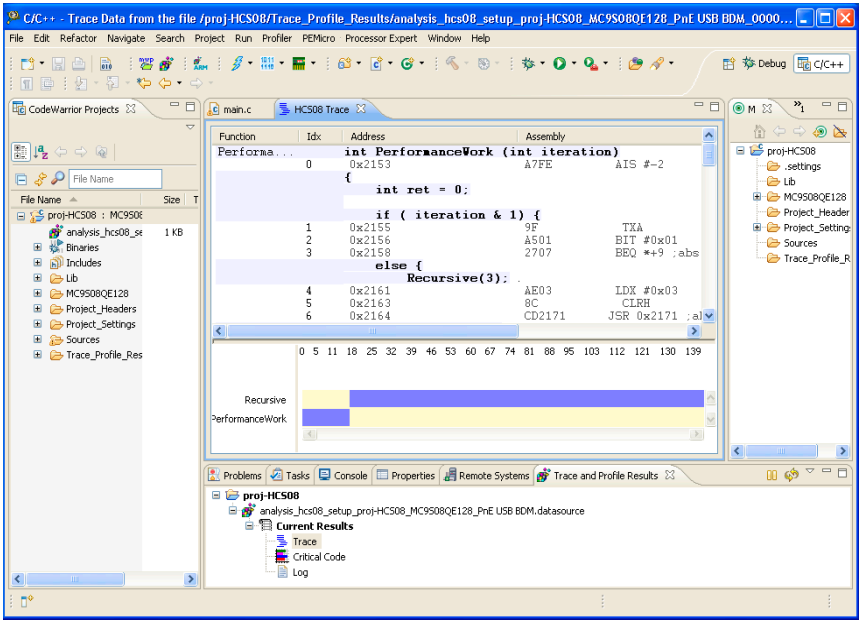
- display results in an intuitive and user friendly manner in the **Trace Data**, **Critical Code Data**, **Flat Profile**, and **Simple Profiler Data** viewers,
- export trace data, critical code data, flat profile data, profiling information into an Excel file, and
- copy and paste a line of the trace in a text file.

NOTE Profiling is not supported on optimized code. Only the O0 optimization level is supported.

CodeWarrior Interface

The CodeWarrior Development Studio provides a common interface for developing, debugging, and analyzing applications. The project-oriented **Workbench** window of CodeWarrior IDE provides numerous perspectives containing views, editors, and controls that appear in menus and toolbars.

Figure 2.1 Workbench Window



Each perspective is a collection of views, which provides a set of functionality aimed at accomplishing a specific type of task.

Following are the some of the views that are used in the profiling and analysis tools:

- **CodeWarrior Projects** view provides a hierarchical view of the project resources in the **Workbench** window. Context menus provide file-management controls.
- **Console** view shows the process output including actions, messages, and errors. It displays messages indicating when data collection is enabled, is in process, and is complete.
- **Trace and Profile Results** view provides a hierarchical view of the data sources, data files, and data sets of the project.
- **Profile Results** view provides hyperlinks to the **Trace** and **Flat Profile** results.
- **Simple Profiler Data Viewer** provides flat, tree, and class-based view of the profiler output.
- **Profiler Setup-Results** view provides the location of the target image file and the source files of the hardware.

After creating a project in the CodeWarrior IDE, build your application, define a launch configuration, and wait for data collection and data display.

NOTE In case, you have installed Microcontrollers CodeWarrior in `C` drive using an administrator account and created a stationery project for profiling, the error messages are displayed on the console when you try to launch the CodeWarrior from a guest account.

Data Collection

You can collect the following types of data for an application when it runs on the target hardware.

- Trace Data
- Critical Code Data
- Flat Profile Data
- Timeline Data
- Profiling Data — This does not require a target hardware, it uses the profiling system

Trace Data

The **Trace Data** viewer displays the trace data collected by the target hardware.

The features available for the **Trace Data** viewer include:

- stepping through trace data that is synchronized with the source code of the selected address,

- exporting trace data to an Excel file,
- allowing column reordering, and
- copying and pasting a line of the trace data.

Critical Code Data

The critical code data is generated based on the trace data. The **Critical Code Data** viewer displays name, start address, number of times each instruction is executed, and code size of each function in the program. The **Critical Code Data** viewer displays the detailed information of every instruction traced in the data.

The features available for the **Critical Code Data** viewer include:

- statistics at function level,
- statistics at instruction level,
- code view in source editor, and
- column ordering and sorting at function level.

Flat Profile Data

The flat profile data displays the summarized data of a function for the Kinetis target. It displays the name and start address of the program functions along with the number of bytes required by the functions, number of lines executed in the functions, and total number of clock cycles that a function takes.

Timeline Data

The timeline data displays a graphical view of the functions that are executed in the application and the number of cycles each function takes when the application is run.

Profiling Data

The profiling data is collected for the ColdFire V2–V4 and ColdFire V4e targets, which do not have the hardware capability to collect trace data. The profiling data displays the the summarized, detailed, and class-based information of each function profiled.

3

Collecting Data

The basic process of collecting data when an application runs on the HCS08, ColdFire V1, Kinetis, and ColdFire V4e target hardware includes:

- creating and configuring a project for the target hardware,
- setting up the debugger launch configuration to collect the analysis data from the target hardware, and
- running the application on the target hardware to collect data.

This process of collecting data is divided into the following topics:

Creating New Project

The CodeWarrior IDE is a project-oriented interface. You can use the **New Project** wizard to create new Microcontrollers projects for hardware profiling.

NOTE You must create a new project or open an existing project before using the Profiling and Analysis tools.

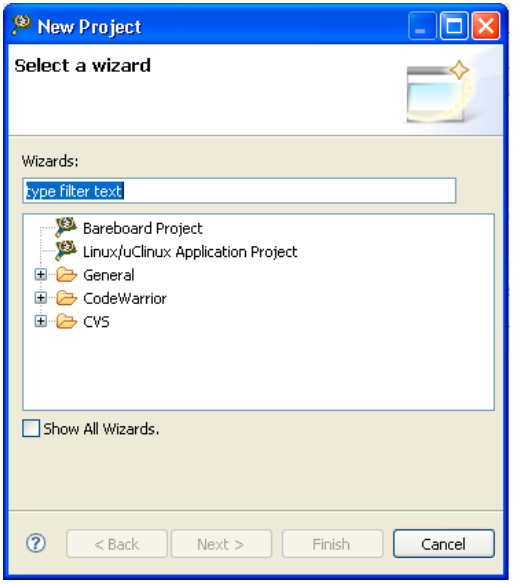
Using HCS08 Target

To create a new Microcontrollers project using the HCS08 target:

1. Select **File > New > Project**.

   The **New Project** dialog box appears, as shown in Figure 3.1.

Figure 3.1 New Project Wizard

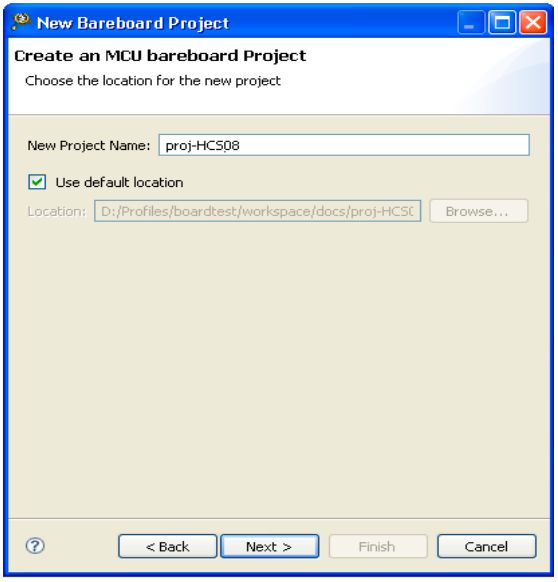


2. Select **Bareboard Project** and click **Next**.

   The **Create an MCU Bareboard Project** page appears.

3. Enter the name of your project in the **New Project Name** text box, as shown in Figure 3.2.

Figure 3.2 Create an MCU Bareboard Project Page



NOTE You can also open the **Create an MCU Bareboard Project** page directly by selecting **File > New > Bareboard Project**.

4. Clear the **Use default location** checkbox, and click **Browse** to specify a different location for the new project. The default setting of the **Use default location** checkbox is checked. Table 3.1 describes the Microcontrollers bareboard project settings.

Table 3.1 Create an MCU Bareboard Project Page Settings

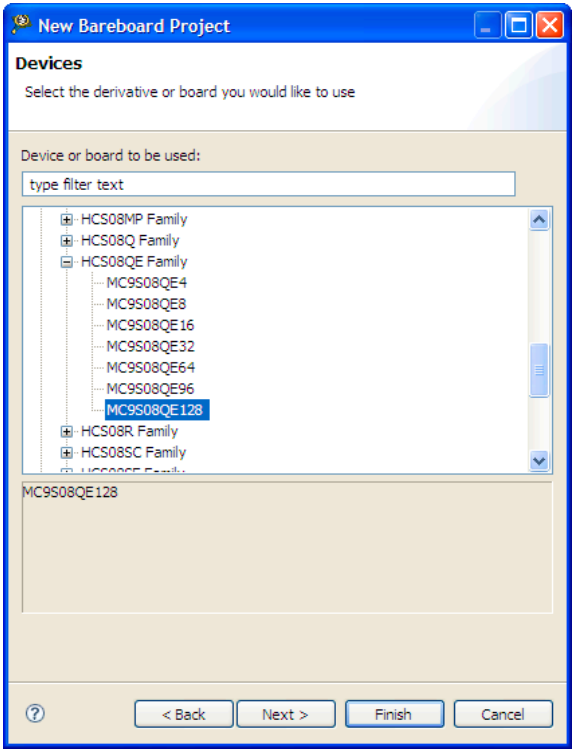
Option	Description
New Project Name	Enter the name of the project in this text box.
Use default location	If checked, the project stores the files required to build the program in the Workbench's current workspace directory. If cleared, the project files are located in the directory you specify. Use the **Location** text box to select the directory.
Location	Specifies the directory that contains the project files. Click **Browse** to navigate to the desired directory. This option is only available when **Use default location** is clear.

5. Click **Next**.

   The **Devices** page appears.

6. Select the target device or board for your project from the HCS08 family. For example, select **HCS08 > HCS08QE Family > MC9SO8QE128**, as shown in Figure 3.3.

Figure 3.3 Devices Page

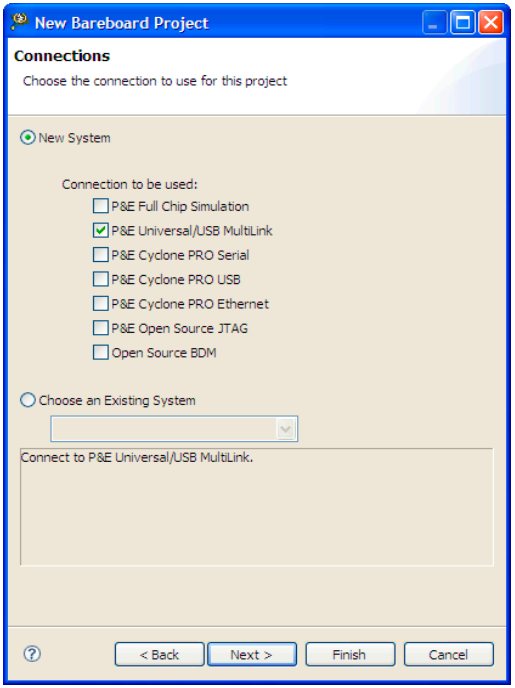


7. Click **Next**.

   The **Connections** page appears.

8. Select the available connection, as shown in Figure 3.4.

Figure 3.4 Connections Page

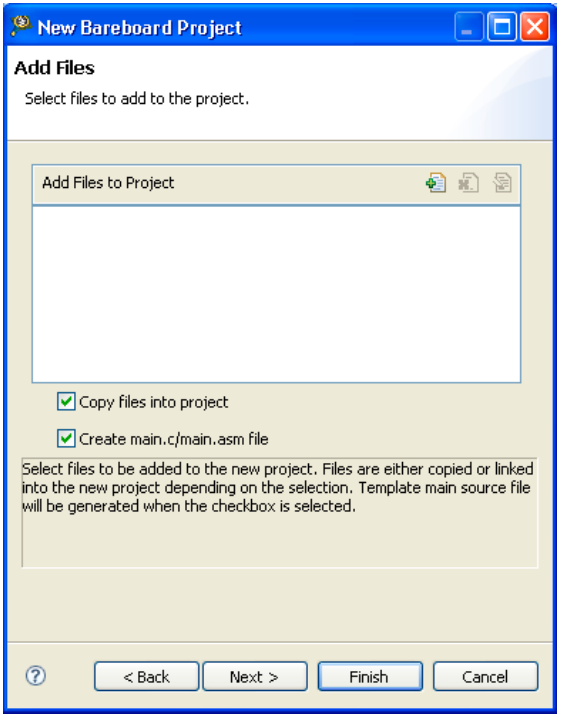


NOTE The Profiling supports all the connections except the simulator.

9. Click **Next**.

   The **Add Files** page appears.

Figure 3.5 Add Files Page



10. Click **Next**.

    The **Languages** page appears, as shown in Figure 3.6.

Figure 3.6 Languages Page

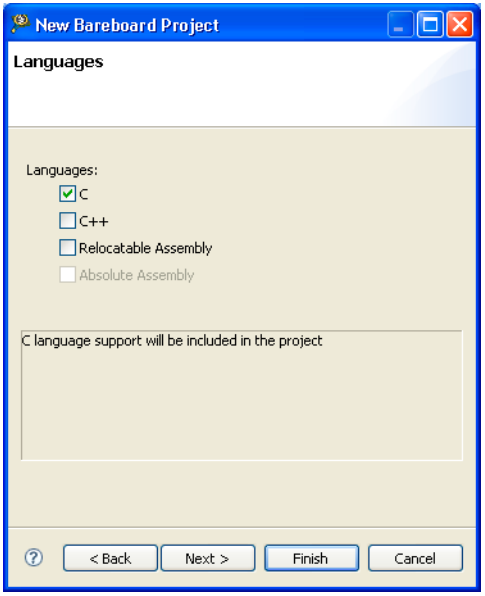


Table 3.2 describes the various options of the **Languages** page. You can use this page to select the programming language that you want to use when writing the program's source code. You can make multiple selections, creating the code in multiple formats.

Table 3.2 Languages Options

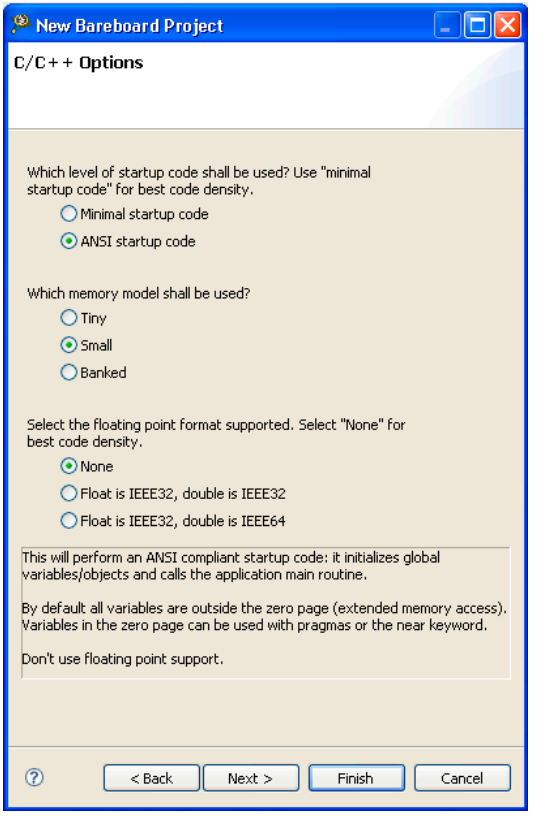
Option	Description
C	Checked — C language support will be included in the project.
C++	Checked — C++ language support will be included in the project.
Relocatable Assembly	Checked — Enables you to split up the application into multiple assembly source files. The source files are linked together using the linker.
Absolute Assembly	Checked — Enables you to use only one single assembly source file with absolute assembly. There is no support for relocatable assembly or linker.

11. Do not change the default settings on the **Languages** page.

12. Click **Next**.

    The **C/C++ Options** page appears, as shown in Figure 3.7.

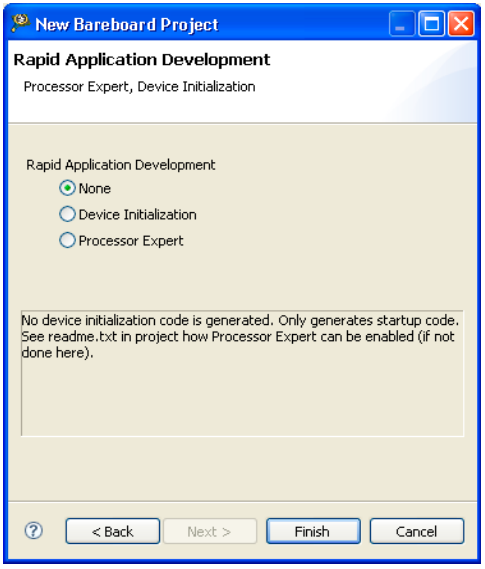
Figure 3.7 C/C++ Options Page



13. Do not change the default settings on the **C/C++ Options** page.
14. Click **Next**.

    The **Rapid Application Development** page appears.

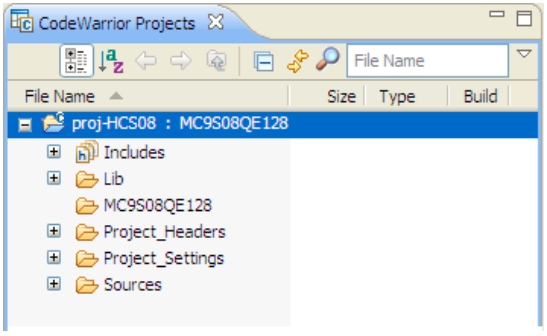
Figure 3.8 Rapid Application Development Page



15. Accept the default settings and click **Finish**.

    The project *proj-hcs08* is created and appears in the **CodeWarrior Projects** view, as shown in Figure 3.9.

Figure 3.9 CodeWarrior Projects View



16. Select the project in the **CodeWarrior Projects** view.
17. Select **Project > Build Project** to build the project.

Using ColdFire V1 Target

To create a new Microcontrollers project using the ColdFire V1 target:

1. Select **File > New > Project**.

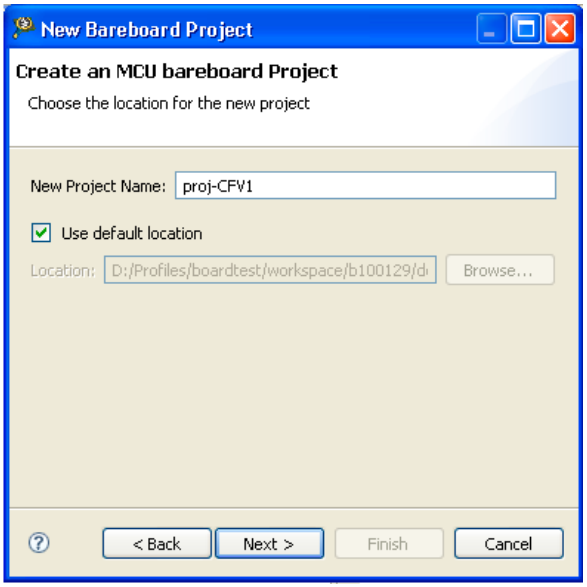
   The **New Project** dialog box appears, as shown in Figure 3.1.

2. Select **Bareboard Project** and click **Next**.

   The **Create an MCU Bareboard Project** page appears, as shown in Figure 3.10.

3. Enter the name of your project in the **New Project Name** text box, and specify the location of the project if you do not want to use the default location.

Figure 3.10 Create an MCU Bareboard Project Page



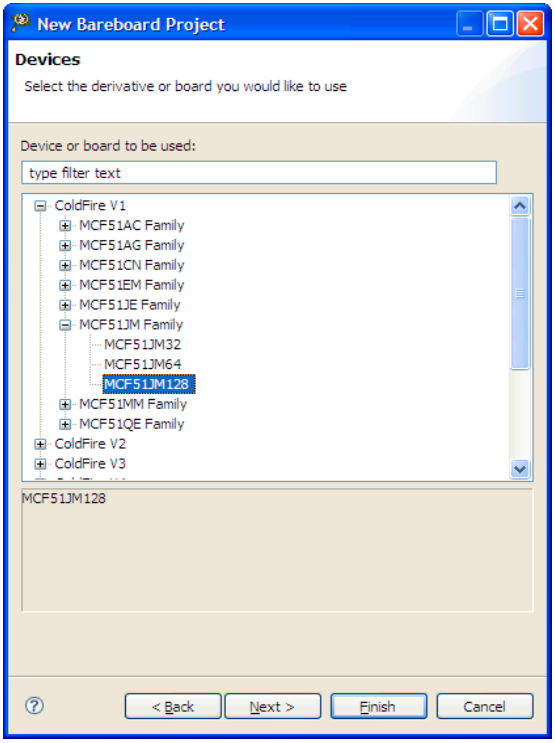
NOTE You can also open the **Create an MCU Bareboard Project** page directly by selecting **File > New > Bareboard Project**.

4. Click **Next**.

   The **Devices** page appears.

5. Select the target device or board for your project from the ColdFire V1 family. For example, select **ColdFire V1 > MCF51JM Family > MCF51JM128**, as shown in Figure 3.11.

Figure 3.11 Devices Page

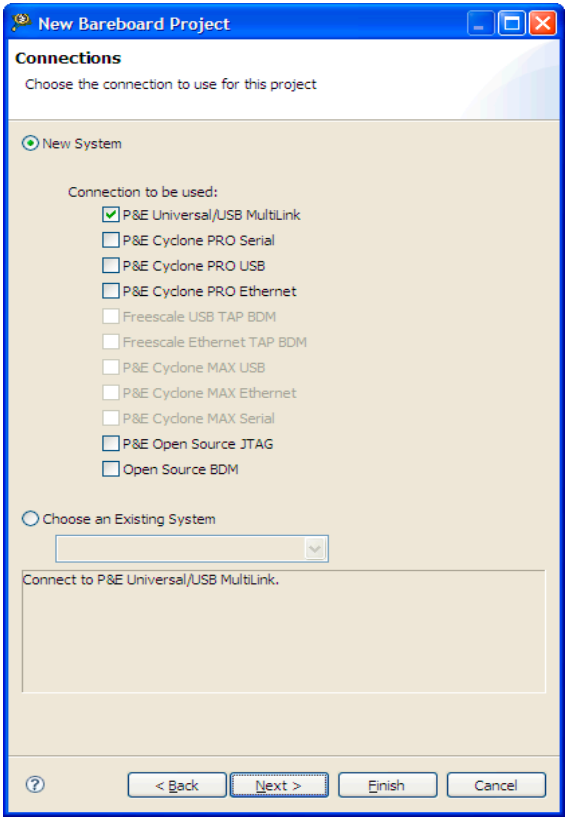


6. Click **Next**.

   The **Connections** page appears.

7. Select the available connection, as shown in Figure 3.12.

Figure 3.12 Connections Page

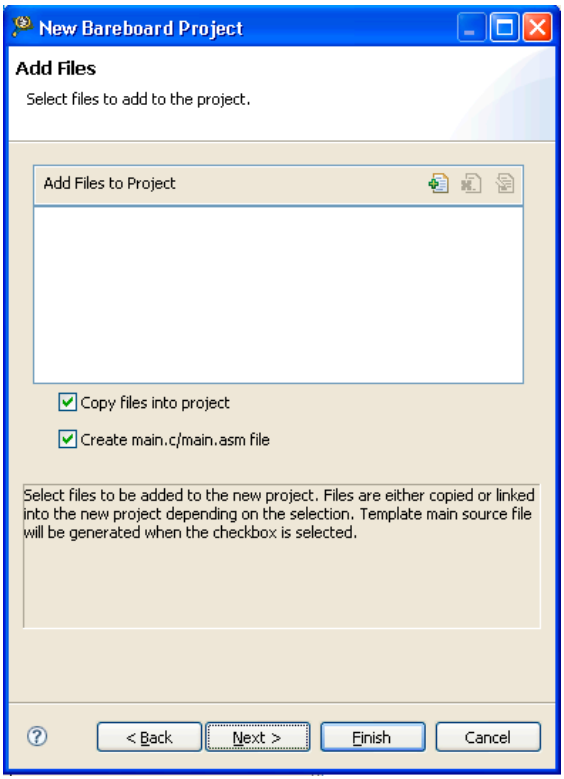


NOTE The Profiling supports all the connections except the simulator.

8. Click **Next**.

   The **Add Files** page appears.

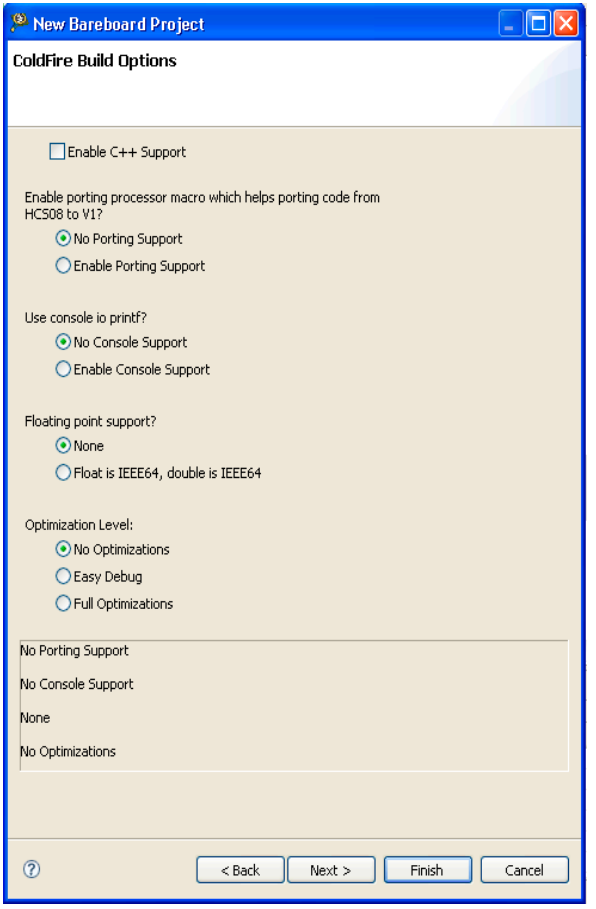
Figure 3.13 Add Files Page



9. Click **Next**.

   The **ColdFire Build Options** page appears, as shown in Figure 3.14.

Figure 3.14 ColdFire Build Options Page



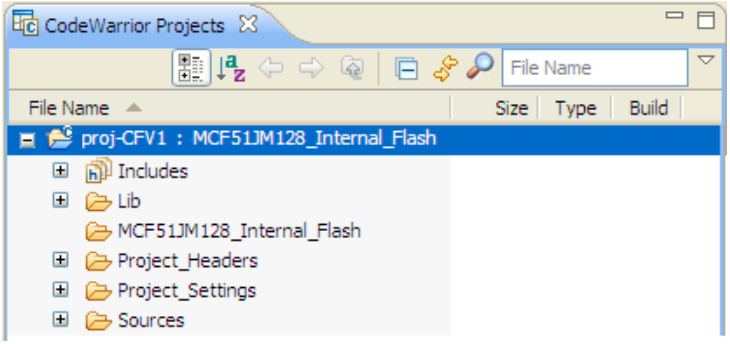
10. Do not change the default settings on the **ColdFire Build Options** page.
11. Click **Next**.

    The **Rapid Application Development** page appears.

Figure 3.15 Rapid Application Development Page

12. Accept the default settings and click **Finish**.
13. The project *proj-CFV1* is created and appears in the **CodeWarrior Projects** view, as shown in Figure 3.16.

Figure 3.16 CodeWarrior Projects View



14. Select the project in the **CodeWarrior Projects** view.
15. Select **Project > Build Project** to build your project.

Using Kinetis Target

To create a new Microcontrollers project using the Kinetis target:

1. Select **File > New > Project**.

   The **New Project** dialog box appears, as shown in Figure 3.1.

2. Select **Bareboard Project** and click **Next**.

   The **Create an MCU Bareboard Project** page appears, as shown in Figure 3.2.

3. Enter the name of your project in the **New Project Name** text box, for example, *TraceProject*, and specify the location of the project if you do not want to use the default location.

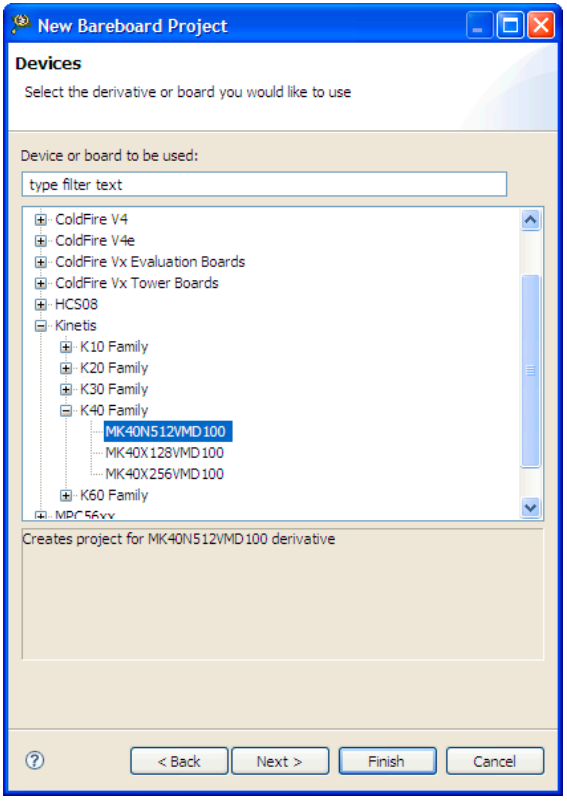
NOTE You can also open the **Create an MCU Bareboard Project** page directly by selecting **File > New > Bareboard Project**.

4. Click **Next**.

   The **Devices** page appears.

5. Select the target device or board for your project from the Kinetis family. For example, select **Kinetis > K40 Family > MK40N512VMD100**, as shown in Figure 3.17.

Figure 3.17 Device and Connection Page

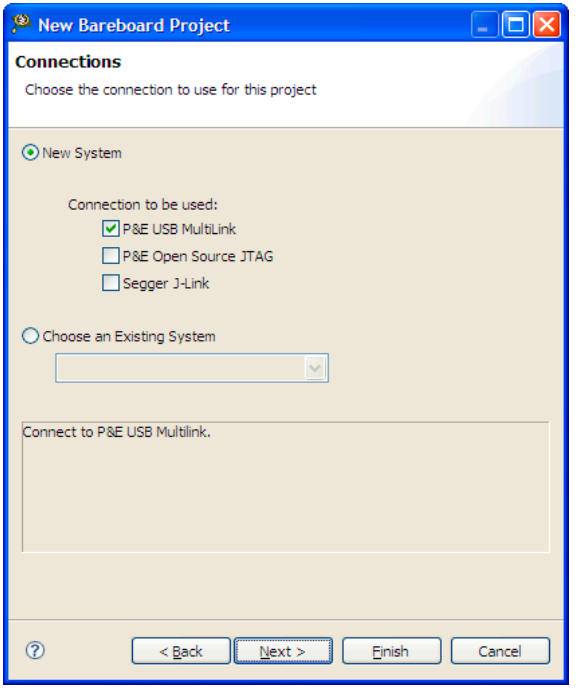


6. Click **Next**.

   The **Connections** page appears.

7. Select the available connection, as shown in Figure 3.18.

Figure 3.18 Connections Page



NOTE The Profiling supports all the connections except the simulator.

8. Click **Next**.

   The **Add Files** page appears.

9. Click **Next**.

   The **Languages** page appears.

10. Do not change the default settings on the **Languages** page.

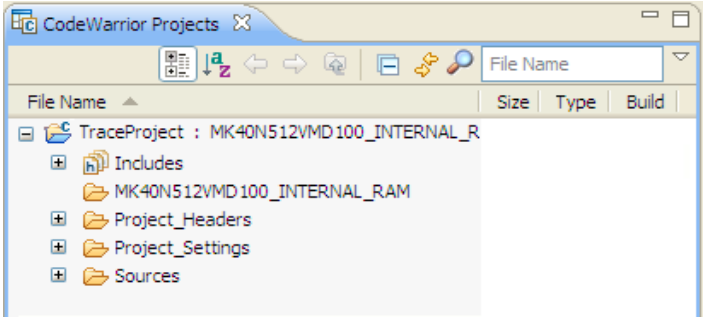
11. Click **Next**.

    The **Rapid Application Development** page appears.

12. Click **Finish**.

13. The project *TraceProject* is created and appears in the **CodeWarrior Projects** view, as shown in Figure 3.19.

Figure 3.19 CodeWarrior Projects View



14. Select the project in the **CodeWarrior Projects** view.
15. Select **Project > Build Project** to build your project.

Using the ColdFire V4e

To create a new Microcontrollers project using the ColdFire V4e target:

1. Select **File > New > Project**.

   The **New Project** dialog box appears.

2. Select **Bareboard Project** and click **Next**.

   The **Create an MCU Bareboard Project** page appears.

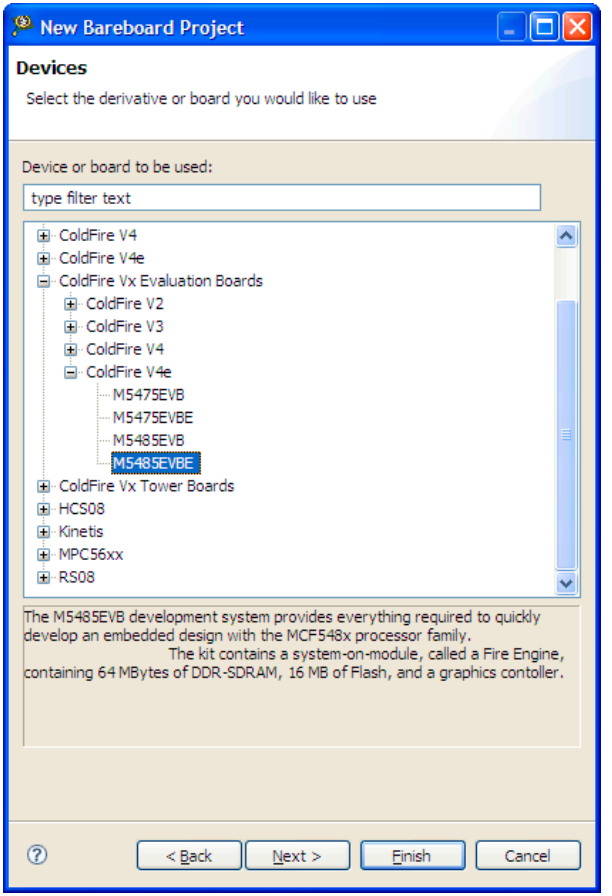
3. Enter the name of your project in the **New Project Name** text box, and specify the location of the project if you do not want to use the default location.

4. Click **Next**.

   The **Devices** page appears.

5. Select the target device or board for your project from the ColdFire V4e family. For example, select **ColdFire Evaluation Boards > ColdFire V4e > M5485EVBE**, as shown in Figure 3.20.

Figure 3.20 Devices Page

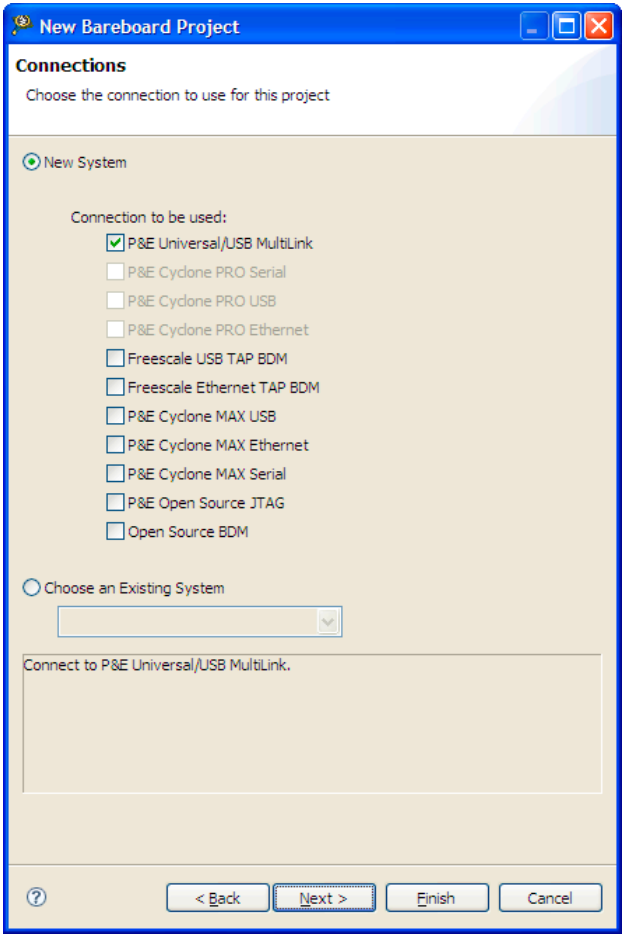


6. Click **Next**.

   The **Connections** page appears.

7. Select the available connection, as shown in Figure 3.21.

Figure 3.21 Connections Page



8. Click **Next**.

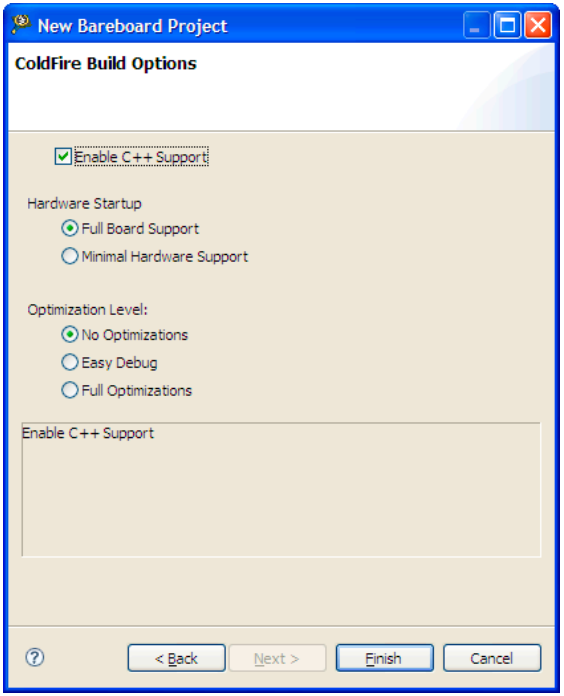
   The **Add Files** page appears.

9. Click **Next**.

   The **ColdFire Build Options** page appears.

10. Check the **Enable C++ Support** checkbox, as shown in Figure 3.22.

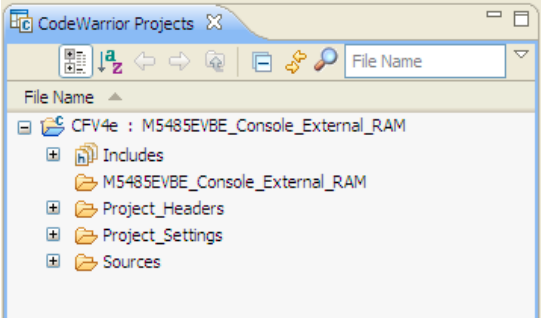
Figure 3.22 ColdFire Build Options Page



11. Click **Finish**.

    The project is created and appears in the **CodeWarrior Projects** view, as shown in Figure 3.23.

Figure 3.23 CodeWarrior Projects View



12. Collect profiling information using the steps described in the chapter, Profiling on ColdFire V4e Target.

Configuring Launcher

Before debugging an application, you need to configure the debug launcher for profiling.

For HCS08 Target

To configure the launch configuration for the HSC08 target:

1. In the **CodeWarrior Projects** view, right-click on the project and select **Debug As > Debug Configurations** from the context menu.

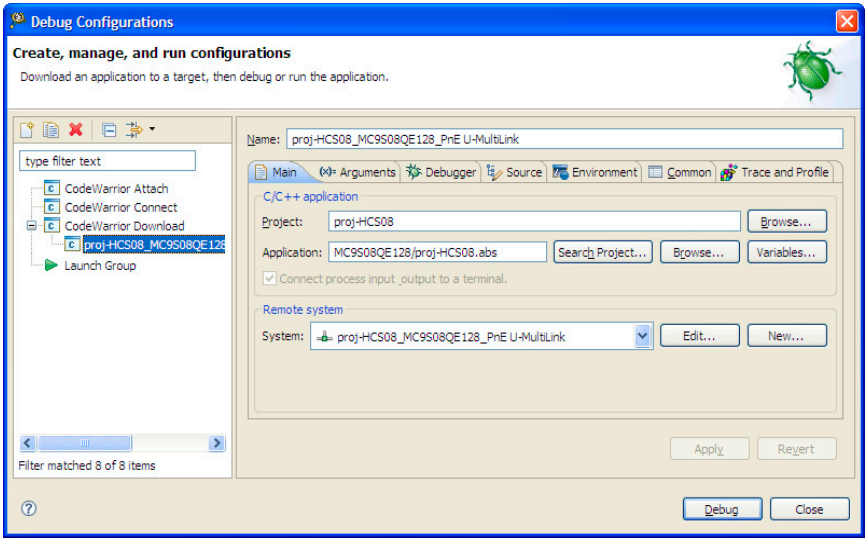
   The **Debug Configurations** dialog box appears.

2. In the **Debug Configurations** dialog box, expand the **CodeWarrior Download** configuration in the tree structure on the left, and select the launch configuration corresponding to the project you are using. For example, select *proj-HCS08_MC9S08QE128_PnE USB BDM*.

3. On the **Main** tab page (Figure 3.24), verify that *proj-HCS08* is displayed in the **Project** field. If it does not appear, click **Browse** and locate the project.

4. If the application is not displayed in the **Application** field, click **Search Project** to select the application image.

Figure 3.24 Debug Configurations — Main Page



To configure the launch configuration for the measurement of data:

1. Click the **Trace and Profile** tab.
2. Check the **Enable Trace and Profile** checkbox to enable the **Trace Mode Options** and **Trace Start/Stop Conditions** groups.

Figure 3.25 Debug Configurations — Trace and Profile Page

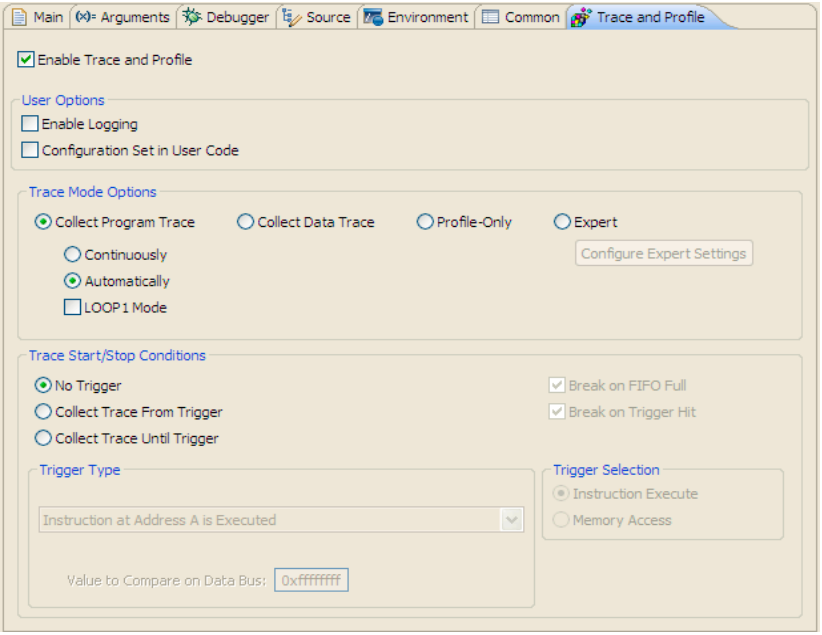


Table 3.3 describes the various **Trace and Profile** options.

Table 3.3 Trace and Profile Options for HCS08

Group	Options	Descriptions
User Options	Enable Logging	When checked, creates a log file that keeps details of the actions that took place in the application. For example, when the debug session terminated, when the target execution resumed or stopped.
	Configuration Set in User Code	When checked, lets you configure trace registers from the application without using the **Trace and Profile** page. In this scenario, you can write the appropriate registers in the source code to configure the trace mode and triggers. To understand how to configure trace registers in the application for the HCS08 target, refer the *HCS08* topic in Appendix B, Configuring Trace Registers in Source Code.

Table 3.3 Trace and Profile Options for HCS08 (*continued*)

Group	Options	Descriptions
Trace Mode Options	Collect Program Trace	Consists of these options: • **Continuously** — When selected, collects the trace data continuously. The trace buffer is read, processed, and emptied periodically, so that the **Trace Data** viewer can collect all the trace records generated by the application. In this mode, the trace data is not lost. It is a bit intrusive as it stops the target repeatedly in the background for collecting the trace buffers. • **Automatically** — When selected, the entries in the buffer start overwriting without interruption when the data reaches at the end of the buffer. If there is more trace data than the size of the buffer, the old entries will be overwritten. • **LOOP1 Mode** — Lets you collect the trace data without any consecutive identical addresses. If the next address to be stored in FIFO is the same as the one stored last time, it is ignored. This mode is particularly useful with short busy-wait type loops, which are repeated a large number of times or recursive calls, and is recommended when you want to view the coverage of that code, but not necessarily the number of times the code executed. For more information on Loop1 mode, refer to the *MC9S08QE128 Reference Manual*. **Note**: The **LOOP1 Mode** option is visible only for the debug version 3 (DbgVer 3) targets, that is HCS08 target with three comparators. For any other targets with two comparators, this option is not visible.
	Collect Data Trace	Collects the trace data of the values of a variable, which is located at the address where trigger B is set, for all the accesses (Read/ Write/Both).
	Profile-Only	When selected, collects trace by sampling the program counter (PC) from time to time.
	Expert	When selected, enables the **Configure Expert Settings** button and gives you access to most of the on-chip DBG module registers. To configure expert settings, download the processor specific manual from the site: http://www.freescale.com/

Table 3.3 Trace and Profile Options for HCS08 (*continued*)

Group	Options	Descriptions
Trace Start/ Stop Conditions	No Trigger	Specifies that no triggers are set for collecting trace. When no triggers are set and trace is collected, the trace data starts collecting from the beginning of the application. **Note**: For more information on tracepoints, see Setting Tracepoints (HCS08).
	Collect Trace From Trigger	Starts collecting trace when the triggers generate, that is when the condition for A and B is met. **Note**: For more information on tracepoints, see Setting Tracepoints (HCS08).
	Keep Last Buffer Before Trigger	When checked, overwrites the trace buffer during trace collection before the trigger is hit. When trigger is hit, trace starts collecting, gets appended to the existing buffer, and only the last part of the buffer is displayed in the **Trace Data** viewer. It gets enabled in the **Continuously** mode when the **Collect Trace From Trigger** option is selected.
	Break on FIFO Full	While debugging, suspends the application automatically when buffer gets full. The checkbox gets enabled in the **Automatically** mode when the **Collect Trace From Trigger** option is selected.
	Collect Trace Until Trigger	Starts collecting trace and stops when the condition for triggers, A and B is met. This option is not enabled in the **Continuously** mode. **Note**: For more information on tracepoints, see Setting Tracepoints (HCS08).
	Break on Trigger Hit	While debugging, suspends the application automatically when the trigger is hit, that is when the trigger condition is met. The checkbox gets enabled when the **Collect Trace Until Trigger** option is selected.

Table 3.3 Trace and Profile Options for HCS08 (*continued*)

Group	Options	Descriptions
Trigger Type		Contains various conditions of triggers, A and B for starting/ stopping trace collection. **Note**: For more information on tracepoints, see Setting Tracepoints (HCS08).
	Instruction at Address A is Executed	Starts trace from the address or source line corresponding to trigger A. For more information, see Setting Triggers in Automatically Mode.
	Instruction at Address A or Address B is Executed	Starts trace from the address or source line corresponding to trigger A or trigger B whichever occurs first. For more information, see Instruction at Address A or Address B is Executed.
	Instruction Inside Range from Address A to Address B is Executed	Starts trace when any instruction in the range between trigger address A and trigger address B is executed. That is, when *[address at trigger A] <= [current address] <= [address at trigger B]* For more information, see Instruction Inside Range from Address A to Address B is Executed.
	Instruction Outside Range from Address A to Address B is Executed	Starts trace when any instruction outside the range between trigger address A and trigger address B is executed. That is, when *[current address] < [address at trigger A or address at trigger B] < [current address].* For more information, see Instruction Outside Range from Address A to Address B is Executed.

Table 3.3 Trace and Profile Options for HCS08 (*continued*)

Group	Options	Descriptions
	Instruction at Address A, Then Instruction at Address B are Executed	Starts trace from trigger B only if trigger A occurred before. For more information, see Instruction at Address A, Then Instruction at Address B are Executed.
	Instruction at Address A is Executed, and Value on Data Bus Match	Collects the trace data from the instruction where trigger A is set when the value specified in the **Value to Compare on Data Bus** text box matches with the opcode read from trigger A address, that is the value in memory at trigger A address. For more information, see Instruction at Address A is Executed, and Value on Data Bus Match. **Note**: Because the hardware has a small delay in enabling the triggers, trace won't be collected as expected if data match is done for the instruction immediately following the line where trigger is set.
	Instruction at Address A is Executed, and Value on Data Bus Mismatch	Collects the trace data from the instruction, where trigger A is set, on data mismatch. That is, trace is triggered at address A when the value specified in the **Value to Compare on Data Bus** text box does not match with the opcode read from trigger A address. For more information, see Instruction at Address A is Executed, and Value on Data Bus Mismatch.
	Value to Compare on Data Bus	Contains the value that you specify to be matched or not matched with the opcode read from trigger A address.
	Capture Read/Write Values at Address B	Captures accesses to the variable address, where trigger B is set, after you press resume. Appears only when the **Collect Data Trace** mode is selected. For more information, see Capture Read/Write Values at Address B.
	Capture Read/Write Values at Address B, After Access at Address A	Waits for the program to execute the instruction at the address where trigger A is set, monitors the variable address where trigger B is set, and collects trace from there. Appears only when the **Collect Data Trace** mode is selected. For more information, see Capture Read/Write Values at Address B, After Access at Address A.

Table 3.3 Trace and Profile Options for HCS08 (*continued*)

Group	Options	Descriptions
Trigger Selection	Instruction Execute	This option is related to how the hardware executes triggering. An address is triggered only when the opcode is actually executed, but this circuitry has a delay which sometimes makes the very next instruction in memory not caught in the trace when you press resume. In this mode, the output of the comparator must propagate through an opcode tracking circuit before triggering FIFO actions.
	Memory Access	When selected, allows memory access to both variables and instructions. For more information, see Memory Access Triggers.

NOTE The **Trigger Selection** group is disabled if the **No Trigger** option is selected in the **Trace Start/Stop Conditions** group.

For ColdFire V1 Target

To configure the launch configuration for the ColdFire V1 target:

1. In the **CodeWarrior Projects** view, right-click on the project and select **Debug As > Debug Configurations** from the context menu.

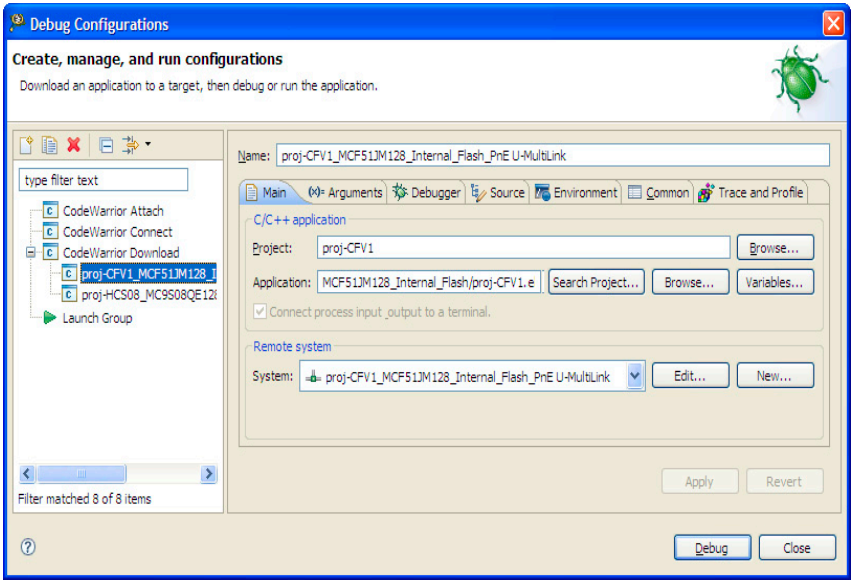
   The **Debug Configurations** dialog box appears.

2. In the **Debug Configurations** dialog box, expand the **CodeWarrior Download** configuration in the tree structure on the left, and select the launch configuration corresponding to the project you are using. For example, select *proj-CFV1_MCF51JM128_Internal_Flash_PnE USB BDM*.

3. On the **Main** tab page (Figure 3.26), verify that *proj-CFV1* is displayed in the **Project** field. If it does not appear, click **Browse** and locate the project.

4. If the application is not displayed in the **Application** field, click **Search Project** to select the application image.

Figure 3.26 Debug Configurations — Main Page



To configure the launch configuration for measurement of data:

1. Click the **Trace and Profile** tab.
2. Check the **Enable Trace and Profile** checkbox to enable the disabled options on the page.

Figure 3.27 Debug Configurations — Trace and Profile Page

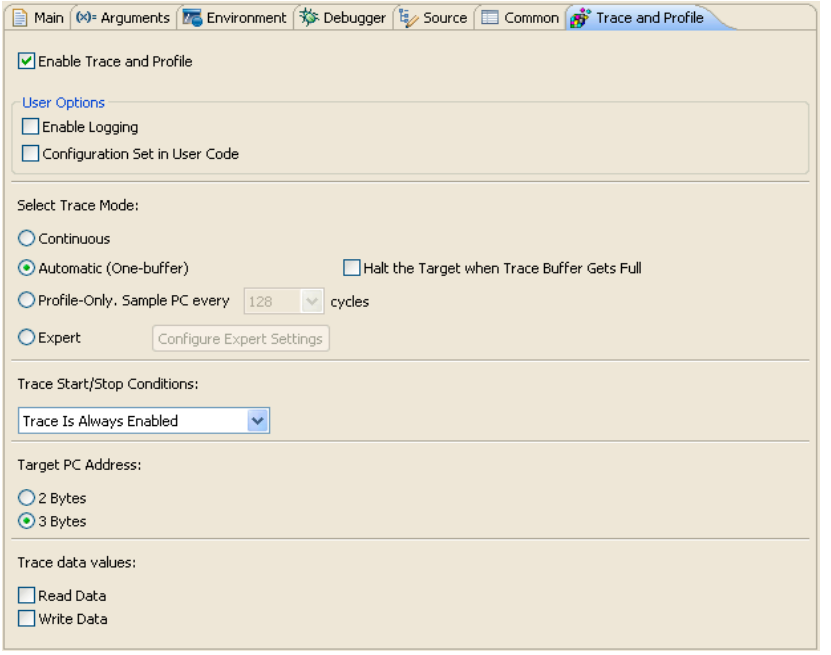


Table 3.4 describes the various **Trace and Profile** options.

Table 3.4 Trace and Profile Options for ColdFire V1

Group	Options	Descriptions
User Options	Enable Logging	Creates a log file that keeps details of the actions that took place in the application. For example, when the debug session terminated, when the target execution resumed or stopped.
	Configuration Set in User Code	When checked, lets you configure trace registers from the application without using the **Trace and Profile** page. In this scenario, you can write the appropriate registers in the source code to configure the trace mode and triggers. To understand how to configure trace registers in the application for the ColdFire V1 target, refer the *ColdFire V1* topic in Appendix B, Configuring Trace Registers in Source Code.

Table 3.4 Trace and Profile Options for ColdFire V1 (*continued*)

Group	Options	Descriptions
Select Trace Mode	Continuous	When selected, collects the trace data continuously. It produces best possible trace and profile results because it captures all executed instructions. However, it is slow and intrusive as it stops the target in the background every about `500` cycles.
	Automatic (One-buffer)	When selected, captures only the last instructions executed before the target gets suspended. It is totally unintrusive.
	Halt the Target when Trace Buffer Gets Full	Appears only when the **Automatic** (**One-Buffer)** option is selected. It acts as a breakpoint for stopping the application. If selected, stops the application automatically when trace buffer gets full.
	Profile-Only. Sample PC every cycles	When selected, captures the PC address every `N` cycles, where `N` is `128/256/512.....16384`. Trace is mostly irrelevant in this mode, but Profile Statistics will be fairly accurate for a long cyclic run. This method is a bit intrusive because it stops the target in the background every about `8*N` cycles.
	Expert	When selected, enables the **Configure Expert Settings** button and lets you configure the ColdFire V1 trace and debug registers directly. To configure expert settings, download the processor specific manual from http://www.freescale.com/
Trace Start/ Stop Conditions		Includes various conditions of triggers, A, B, and C, for starting and stopping trace. For details, refer to Conditions for Starting/ Stopping Triggers

Table 3.4 Trace and Profile Options for ColdFire V1 (*continued*)

Group	Options	Descriptions
Target PC Address	2 Bytes	Select this option to save 5–30% trace-buffer space if your PC addresses never exceed 16-bits (64K). This results in less intrusiveness, or more instructions traced, depending on the trace mode you use. This feature is not supported for the **Expert** trace mode.
	3 Bytes	Select this default and recommended option if the PC address length in your program exceeds 16-bits (64K). This feature is not supported for the **Expert** trace mode.
Trace data values	Read Data	Traces the values of data operands being read from the memory. This feature is not supported for the **Profile-Only** and **Expert** trace modes.
	Write Data	Traces the values of data operands being written to the memory. This feature is not supported for the **Profile-Only** and **Expert** trace modes.

For Kinetis Target

To configure the launch configuration for the Kinetis target:

1. In the **CodeWarrior Projects** view, right-click on the project and select **Debug As > Debug Configurations** from the context menu.

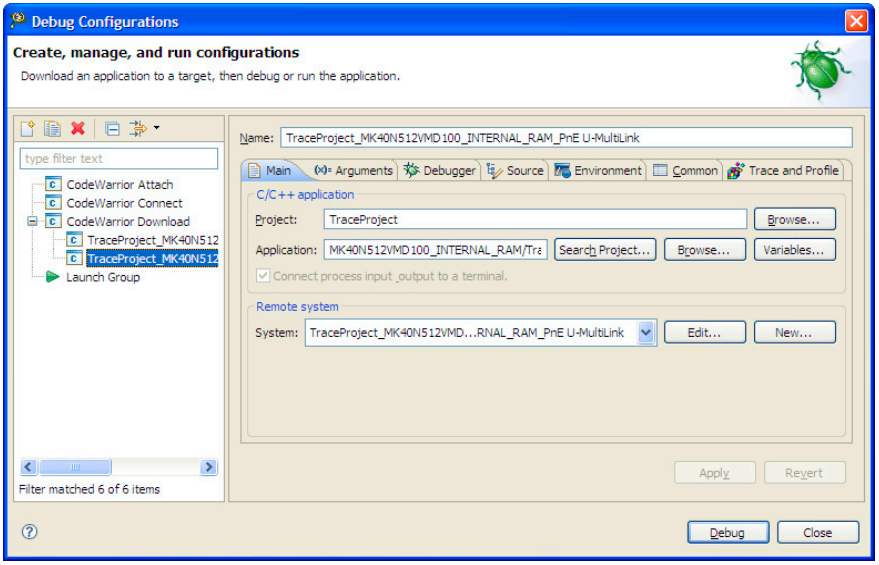
   The **Debug Configurations** dialog box appears.

2. In the **Debug Configurations** dialog box, expand the **CodeWarrior Download** configuration in the tree structure on the left, and select the launch configuration corresponding to the project you are using. For example, select *TraceProject_MK40N512VMD100_INTERNAL_RAM_PnE U-MultiLink*.

3. On the **Main** tab page (Figure 3.28), verify that name of the project, for example, *TraceProject* is displayed in the **Project** field. If it does not appear, click **Browse** and locate the project.

4. If the application is not displayed in the **Application** field, click **Search Project** to select the application image.

Figure 3.28 Debug Configurations — Main Page



To configure the launch configuration for the measurement of data:

1. Click the **Trace and Profile** tab.
2. Check the **Enable Trace and Profile** checkbox to enable the disabled options.

Figure 3.29 Debug Configurations — Trace and Profile Page

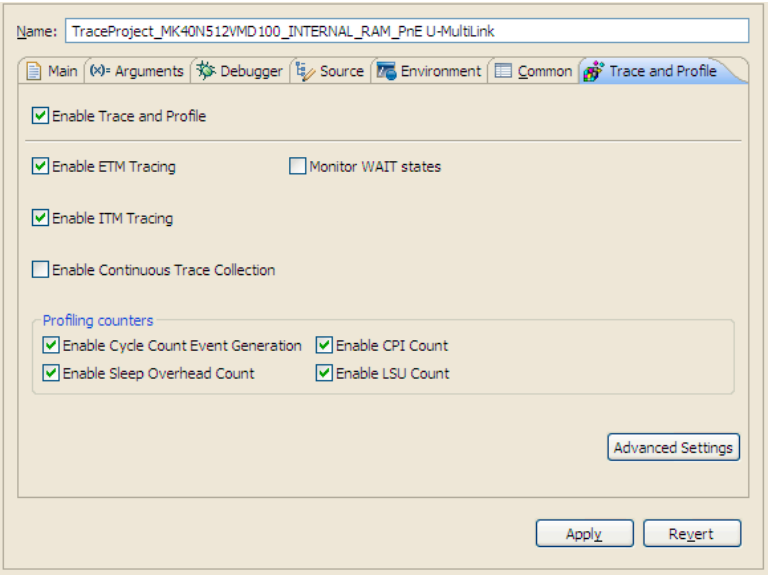


Table 3.5 describes the various **Trace and Profile** options.

Table 3.5 Trace and Profile Options for Kinetis

Option	Description
Enable ETM Tracing	Enables/disables trace output from the Embedded Trace Macrocell (ETM) block. It controls the ETM port selection bit from ETM's control register. For details, refer Embedded Trace Macrocell (ETM).
Monitor WAIT states	Allows monitoring of low power Wait states. This state lets peripherals to function, while allowing CPU to go to sleep reducing power. For more details on low power WAIT state, refer Low Power WAIT Mode.
Enable ITM Tracing	Enables/disables trace output from the Instrumentation Trace Macrocell (ITM) block. For details, refer Instrumentation Trace Macrocell (ITM).
Enable Continuous Trace Collection	Allows you to collect continuous trace data when checked. That is, it stops the target in the background to read the trace every time the FIFO is almost full.

Table 3.5 Trace and Profile Options for Kinetis

Option	Description
Profiling Counters	
Enable Cycle Count Event Generation	Enables cycle count, allowing it to increment and generate synchronization and count events.
Enable CPI Count	Enables the Clocks Per Instruction (CPI) count event.
Enable Sleep Overhead Count	Enables the sleep count event.
Enable LSU Count	Enables the Load Store Unit (LSU) count event.

Tracing and profiling on the Kinetis target is divided into three components:

- Embedded Trace Macrocell (ETM)
- Instrumentation Trace Macrocell (ITM)
- Embedded Trace Buffer (ETB)

Embedded Trace Macrocell (ETM)

An ETM is a debug component that enables reconstruction of program execution. ETM is a high-speed, low-power debug tool that only supports instruction trace. Therefore, ETM helps in minimizing area and reducing gate count.

The main features of an ETM are:

- Trace generation

  Trace generation outputs information that helps understand the operation of the processor. The trace protocol provides a real-time trace capability for processor cores that are deeply embedded in much larger ASIC designs.

- Triggering and filtering

  You can control tracing by specifying the exact set of triggering and filtering resources required for a particular application. Resources include address comparators, data value comparators, counters, and sequencers.

ETM compresses the trace information and writes it directly to an on-chip ETB. An external Trace Port Analyzer (TPA) captures the trace information. The trace is read out at low speed using the JTAG interface when the trace capture is complete.

When the trace has been captured, the profiling and analysis tool extracts the information from ETB and decompresses it to provide a full disassembly, with symbols, of the code that was executed.

Triggering Trace

You can use a trigger signal to specify when a trace run is to occur. You determine the trigger condition by using the event logic (AND/OR) to configure the event resources, such as address comparators and data value comparators.

The trigger event specifies the conditions that must be met to generate a trigger signal on the trace port. When the trigger event occurs, the trigger is output as soon as possible, and therefore might not be aligned with the rest of the trace. The trigger is output over the trace port using a code that can be readily understood by the Trace Capture Device (TCD).

The TCD uses the trigger in the following ways:

- Trace after

  This is often called a start trigger that can indicate to the TCD that the trace information must be collected from the trigger point onwards. A start trigger finds out what happens after a particular event, for example, what happens after entering an interrupt service routine. In addition, a small amount of trace data is collected before the trigger condition. This enables the decompression software to synchronize with the trace, ensuring that it can successfully decompress the code around the trigger point.

- Trace before

  This is often called a stop trigger which is used to stop collection of the trace. In this case, the TCD acts like a large FIFO so that it always contains the most recent trace information and the older information overflows out of the trace memory. The trigger indicates that the FIFO must stop, so the memory contains all the trace information before the trigger event. A stop trigger finds out what caused a certain event, for example, to see what sequence of code was executed before entering an error handler routine. In addition, a small amount of trace data is collected after the trigger condition.

- Trace about

  This is often called a center trigger which you can set between the start point and the stop point. This allows trace memory to contain a defined number of events before the trigger point and a defined number of events after the trigger point.

NOTE The generation of a trigger does not affect the tracing in any way. In any trace run, only a single trigger can be generated by ETM.

Instrumentation Trace Macrocell (ITM)

ITM provides a memory-mapped register interface to allow applications to write logging/event words to the optional external Trace Port Interface Unit (TPIU). ITM also supports control and generation of timestamp information packets. The event words and timestamp

information are formed into packets and multiplexed with hardware event packets from DWT.

NOTE In order to collect ITM trace, DWT must be enabled. This is because sync packet is not sent when ITM is enabled, the DWT cycle counter function must be used to generate the sync packet before writing any stimulus registers. This mechanism does not effect ETM trace in any way.

ITM supports Timestamps and Synchronization.

Timestamps

Timestamps provide information on the timing of event generation with respect to their visibility at a trace output port. A timestamp packet can be generated and appended to a single event packet, or a stream of back-to-back packets where multiple events generate a packet stream with no idle time. The timestamp status information is merged with the timestamp packets to indicate if the timestamp packet transfer is delayed by the FIFO, or if there is a delay in the associated event packet transfer to the output FIFO. The timestamp count continues until it can be sampled and delivered in a packet to the FIFO.

The ARMv7 processor can implement either or both of the following types of timestamps:

- Local timestamps

  Local timestamps provide delta timestamp values, which means that each local timestamp indicates the elapsed time since generating the previous local timestamp. The ITM generates local timestamps from timestamp clock in the ITM block. Each time ITM generates a local timestamp packet, it resets this clock to provide the delta functionality.

- Global timestamps

  Global timestamps provide absolute timestamp values based on a system global timestamp clock. They provide synchronization between different trace sources in the system.

Synchronization

Synchronization packets are independent of timestamp packets. They are used to recover bit to byte alignment information. The packets are required on synchronous TPIU ports that are dedicated to an ARMv7-M core or in complex systems where multiple trace streams are formatted into a single output. When enabled, synchronization packets are emitted on a regular basis and can be used as a system heartbeat.

Embedded Trace Buffer (ETB)

ETB stores data that ETM produces. ETB provides on-chip storage of trace data using a configurable sized RAM. This reduces the clock rate and removes the requirement of

high-speed for collecting trace data. The debugging tools access the buffered data using a JTAG interface.

ETB contains a trace formatter, an internal input block that embeds the trace source ID within the data to create a single trace stream. The trace formatter uses a protocol that allows trace from several sources to be merged into a single stream and later separated. This protocol outputs data in 16-byte frames.

Configuring Advanced Settings on Kinetis

To configure advanced settings for tracing and profiling on Kinetis:

1. Click **Advanced Settings** on the **Trace and Profile** tab of the **Debug Configurations** dialog box.

   The **Preferences** dialog box appears. The default page is **ETM Settings**.

NOTE Use the **Advanced Settings** button if you want to configure advanced settings for collecting trace.

2. Ensure that the **Enable ETM Tracing** checkbox is checked.

Figure 3.30 Preferences Dialog Box

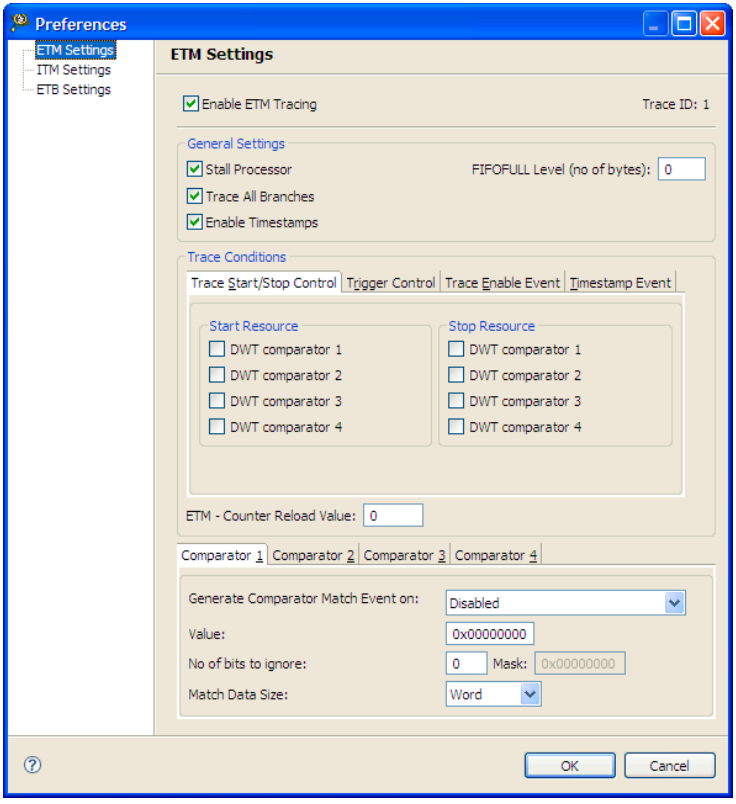


Table 3.6 describes the various options available on the **ETM Settings** page.

Table 3.6 Options Available in ETM Settings Page

Group	Option	Description
General Settings	Stall Processor	If checked, enables the FIFOFULL output that causes the processor to stall when the FIFO is close to overflow. When cleared, the FIFOFULL output remains low and the FIFO overflows if there is a large number of trace packets.
	Trace All Branches	If checked, all branch addresses are collected. Otherwise, only indirect branches are collected. It enables reconstruction of the program flow with the information from the binary executable.
	Enable Timestamps	Enables timestamping when checked.
	FIFOFULL Level (no. of bytes)	Specifies the number of bytes left in the FIFO, below which it is considered full. When the space left in the FIFO gets lower than the specified value, the FIFOFULL or SupressData signal is asserted. For example, setting the value to 15 causes data trace suppression or processor stalling, if enabled, when there are less than 15 free bytes in the FIFO.
Trace Conditions		
Trace Start/Stop Control		Specifies the watchpoint comparator inputs that are used as trace start and stop resources.
	Start Resource	Selects the corresponding DWT comparator to control trace start. **Note**: Data Watchpoint and Trace (DWT) is an optional debug unit that provides watchpoints, data tracing, and system profiling for the processor.
	Stop Resource	Selects the corresponding DWT comparator to control trace stop.

Table 3.6 Options Available in ETM Settings Page

Group	Option	Description
Trigger Control		Defines an event that generates on meeting a condition and appears in the trace data. You can define resource **A** and resource **B** and a function between A and B. When the function occurs, the event is generated and is seen in the **Trace Data** viewer.
	Function	Specifies the condition that must be met to generate a trigger signal on the trace port.
	A	Is the ETM event resource used for triggering. It can take four values: • DWT comparator — Any of the four comparators • ETM - Counter at zero — 16-bit counter reload value • Start/stop — Start or stop tracepoint set in the application • Always true — Default option that triggers all the time
	B	Same as **A**.
	Index	Specifies the value of the comparator if you select **DWT comparator** in the **A** or **B** fields.
	Collect Trace after Trigger	When selected, changes the value of **Trigger Counter** to `480` words in the **Trigger Settings** section of the **ETB Settings** page. This means that `480` words of trace will be collected after trigger hit and `32` words will be collected before trigger hit.

Table 3.6 Options Available in ETM Settings Page

Group	Option	Description
	Collect Trace before Trigger	When selected, changes the value of **Trigger Counter** to `32` words in the **Trigger Settings** section of the **ETB Settings** page. This means that `32` words of trace will be collected after trigger hit and `480` words will be collected before trigger hit.
	Collect Trace about Trigger	When selected, changes the value of **Trigger Counter** to `256` words in the **Trigger Settings** section of the **ETB Settings** page. This means that `256` words of trace will be collected after trigger hit and `256` words will be collected before trigger hit.
Trace Enable Event		Enables the trace when an event occurs.
Note: It is not guaranteed that trace will be generated exactly when the event occurs. It may have a few cycles delay.	Function	Specifies the condition that must be met to enable trace collection. For example, if you set **A** as **DWT comparator 1**, and set DWT comparator 1 to fire at instruction at address 0x800 with mask 0xF, and specify **Function** as **A** then the Trace enable will be active continuously between addresses 0x800 and 0x80F.
	A	Is the ETM event resource used for collecting trace. It can take four values: • DWT comparator — Any of the four comparators • ETM - Counter at zero — 16-bit counter reload value • Start/stop — Start or stop tracepoint set in the application • Always true — Default option that enables the trace
	B	Same as **A**.
	Index	Specifies the value of the comparator if you select **DWT comparator** in the **A** or **B** fields.

Table 3.6 Options Available in ETM Settings Page

Group	Option	Description
Timestamp Event		Generates a timestamp in trace when the event gets activated.
	Function	Specifies the condition that must be met to generate the timestamp event.
	A	Is the ETM event resource used for triggering. It can take four values: • DWT comparator — Any of the four comparators • ETM - Counter at zero — 16-bit counter reload value • Start/stop — Start or stop tracepoint set in the application • Always true — Default option that triggers all the time
	B	Same as **A**.
	Index	Specifies the value of the comparator if you select **DWT comparator** in the **A** or **B** fields.
ETM - Counter Reload Value		Is the value with which the counter is automatically loaded when the register is programmed and when the ETM Programming bit is set. This is a 16- bit field that should be specified in hexadecimal form. The ETM counter decrements at each ETM cycles. Once it reaches 0, it generates an **ETM - Counter at zero** event.

Table 3.6 Options Available in ETM Settings Page

Group	Option	Description
Comparator Settings	Comparator 1, Comparator 2, Comparator 3, Comparator 4	Allows choosing one of the four DWT comparators to configure trace conditions.
	Generate Comparator Match Event On	Allows selecting the event that generates a comparator match.
	Value	Indicates the reference value against which comparison is done.
	No. of bits to ignore	Indicates the size of the ignore mask (0 - 31 bits) applied to the matching address range.
	Match Data Size	Defines the size of the data in the associated comparator register for value matching.

NOTE Timestamps provide information on the timing of event generation with respect to their visibility at a trace output port.

3. Click **ITM Settings** on left to display its corresponding options on right.

Figure 3.31 ITM Settings Page

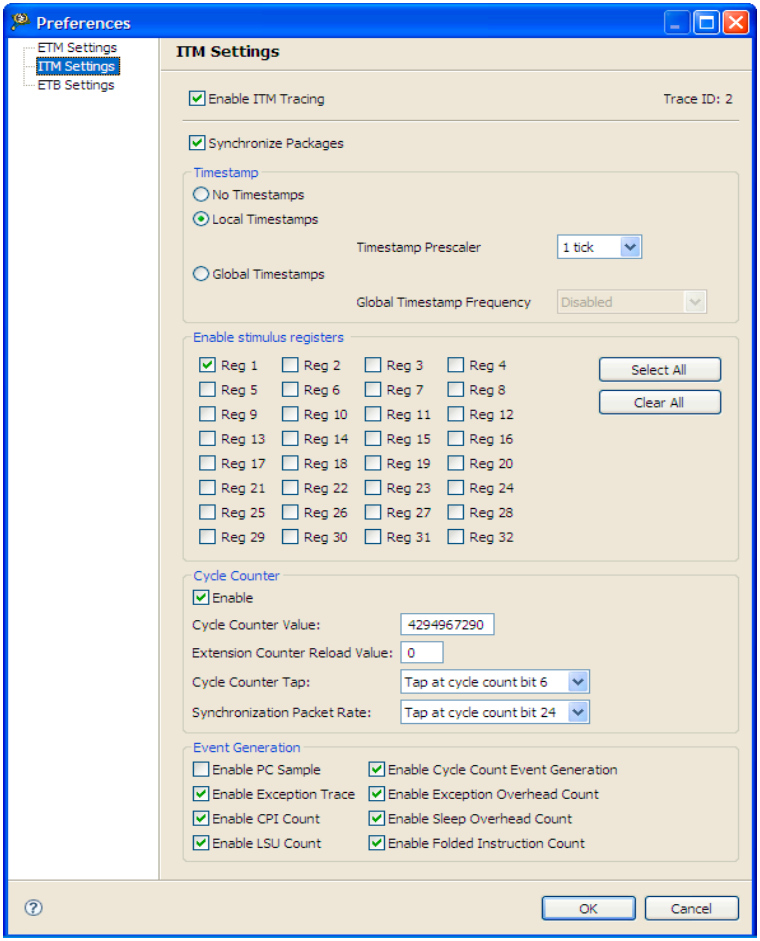


Table 3.7 describes the various options available on the **ITM Settings** page.

Table 3.7 Options Available in ITM Settings Page

Group	Option	Description
Enable ITM Tracing		Enables ITM tracing
Synchronize Packages		Enables packages synchronization when checked.

Table 3.7 Options Available in ITM Settings Page

Group	Option	Description
Timestamp		Enables ITM timestamping (delta).
	No Timestamps	Disables timestamping.
	Local Timestamps	Defines the number of ITM clock ticks on which the timestamp counts. ITM clock is a clock on 20 bits. **Timestamp Prescaler**: Modifies the scaling of the timestamps clock. For example, if set to `16`, the timestamp counts once every `16` clock ticks.
	Global Timestamps	Defines the number of ATCLK (or Advanced Trace Clock) ticks on which the timestamp counts. ATCLK is a clock on 48 bits, and it is common to ETM and ITM. **Global Timestamp Frequency**: Decides when the global cycle count is written in the trace buffer, that is every `128`/`8192` cycles or at every packet.
	Enable Stimulus Registers	Each bit location corresponds to a virtual stimulus register. When a bit is set, a write to the appropriate stimulus location results in a packet being generated, except when the FIFO is full. Use **Select All** or **Clear All** to check or clear all the checkboxes.

Table 3.7 Options Available in ITM Settings Page

Group	Option	Description
Cycle Counter	Enable	Enables cycle counter which counts the number of core cycles. The counting is suspended when the core halts in debug state.
	Extension Counter Reload Value (hex)	Defines the cycle count event combining with Cycle Counter Tap and Synchronization Packet Rate.
	Cycle Counter Tap	Selects a tap on the cycle counter register — Tap at cycle count bit 6 or Tap at cycle count bit 10. This means that the Cycle Counter Tap event fires at every change of either bit `6` or bit `10` of the cycle counter. **Note**: Cycle count is a 32-bit, incrementing (up) cycle counter.
	Synchronization Packet Rate	Selects a synchronization packet rate. CYCCNTENA and ITM_TCR.SYNCENA must also be enabled for this feature. Synchronization packets (if enabled) are generated on tap transitions (0 to1 or 1 to 0).

Table 3.7 Options Available in ITM Settings Page

Group	Option	Description
Event Generation	Enable PC Sample	Controls PC sample event generation.
	Enable Cycle Count Event Generation	Enables cycle count, allowing it to increment and generate synchronization and count events.
	Enable Exception Trace	Enables exception trace that traces exception entry, exit and return to a pre-empted handler or thread.
	Enable Exception Overhead Count	Enables exception overhead event. **Note**: The exception overhead counter counts the total cycles spent in exception processing. For example, entry stacking, return unstacking, or pre-emption. An event is emitted on counter overflow which occurs after every 256 cycles.
	Enable CPI Count	Enables the CPI count event.
	Enable Sleep Overhead Count	Enables sleep overhead count event.
	Enable LSU Count	Enables the Load Store Unit (LSU) count event. **Note**: The LSU counter increments on the additional cycles required to execute all load and store instructions.
	Enable Folded Instruction Count	Enables the folded instruction count event. **Note**: The folded instruction counter increments on any instruction that executes in zero cycles.

4. Click **ETB Settings** on left to display its corresponding options on right.

Figure 3.32 ETB Settings Page

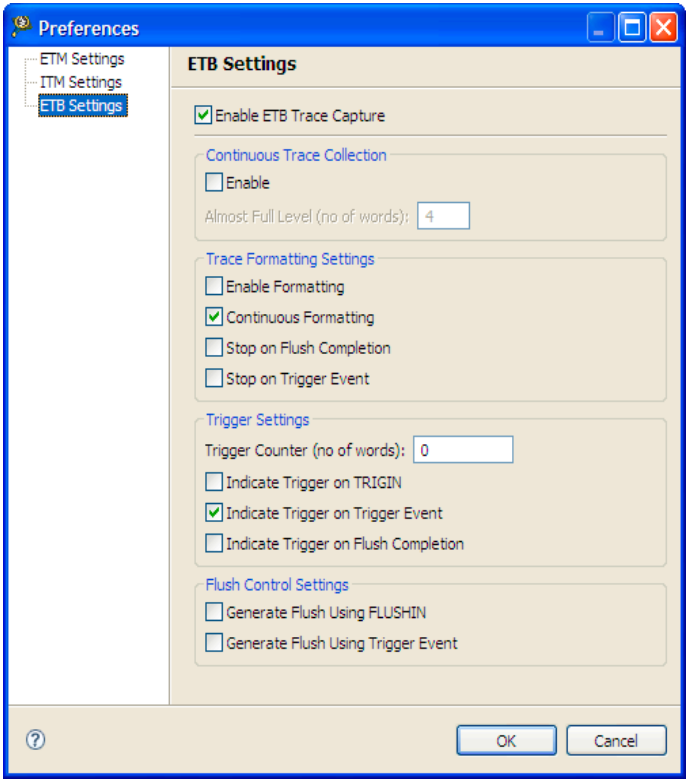


Table 3.8 describes the various options available on the **ETM Settings** page.

Table 3.8 Options Available in ETB Settings Page

Group	Option	Description
Enable ETB Trace Capture		Enables ETB tracing.

Table 3.8 Options Available in ETB Settings Page

Group	Option	Description
Continuous Trace Collection		Allows you to collect continuous trace data.
	Enable	Enables the continuous trace collection when checked. It also enables the **Almost Full Level** checkbox.
	Almost Full Level (no. of words)	Indicates the number of words (of 4 bytes) for which the trace buffer is considered almost full, and trace must be read from the hardware memory. This mode offers the possibility to collect the trace data when the buffer gets full to prevent data loss. When the Embedded Trace Buffer (ETB) becomes almost full, a signal is asserted to cause an interrupt on the core or to cause the core to halt. For now, the profiling tools offer support only for halting the core. **Note**: The maximum and minimum value for the **Almost Full Level** field is application-dependant. However, the safe maximum and minimum values are *450* and *50*. The trace and profile results might not be collected for any value above *450* and below *50*.

Table 3.8 Options Available in ETB Settings Page

Group	Option	Description
Trace Formatting Settings		Trace formatting inserts the source ID signal into a special format data packet stream. Trace formatting is done to enable trace data to be re-associated with a trace source after data is read back out of the ETB.
	Enable Formatting	When checked, prevents triggers from being embedded into the formatted stream.
	Continuous Formatting	Enables the continuous mode. In the ETB, this mode corresponds to the normal mode with the embedding of triggers.
	Stop on Flush Completion	Stops trace formatting when a flush is completed. This forces the FIFO to drain off any partially completed packets.
	Stop on Trigger Event	Stops trace formatting when a trigger event is observed. A trigger event occurs when the trigger counter reaches zero (where fitted) or the trigger counter is zero (or not fitted) when the TRIGIN signal is HIGH.

Table 3.8 Options Available in ETB Settings Page

Group	Option	Description
Trigger Settings		A trigger event occurs when the trigger counter reaches zero, or when the trigger counter is zero, when TRIGIN is HIGH. The trigger counter register controls how many words are written into the trace RAM after a trigger event. After the formatter is flushed in the normal or continuous mode, a complete empty frame is generated. This is a data overhead of seven extra words in the worst case. If the formatter is in bypass mode, a maximum of two additional words are stored for the trace capture postamble.
	Trigger Counter (no. of words)	Defines the number of 32-bit words remaining to be stored in the ETB Trace RAM. **Note**: The hardware buffer can hold upto `512` words of trace. **Trigger Counter** indicates how many words of trace will be collected in the buffer before and after the trigger gets activated. The value in **Trigger Counter** is the number of words of trace that will be collected after trigger hit, the rest will be collected before trigger hit. For example, if the value is `32` then `32` words of trace will be collected after trigger hit and `480` will be collected before trigger hit. The value of **Trigger Counter** depends on the option selected from **ETM Settings** page in the **Trigger Control** tab.
	Indicate Trigger on TRIGN	Indicates a trigger on TRIGIN being asserted.
	Indicate Trigger on Trigger Event	Indicates a trigger on a trigger event.
	Indicate Trigger on Flush Completion	Indicates a trigger on flush completion.

Table 3.8 Options Available in ETB Settings Page

Group	Option	Description
Flush Control Settings		There are three flush generating conditions that can be enabled together. If more flush events are generated while a flush is in progress, the current flush is serviced before the next flush is started. Only one request for each source of flush can be pended. If a subsequent flush request signal is deasserted while the flush is still being serviced or pended, a second flush is not generated. Flush from FLUSHIN takes priority over Flush from Trigger, which in turn is completed before a manual flush is activated.
	Generate Flush Using FLUSHIN	Generates flush using the FLUSHIN interface.
	Generate Flush Using Trigger Event	Generates flush using the trigger event.

Collecting Data

After setting the debugger launch configuration, you need to run the application on the target hardware to collect data.

- For HCS08 Target
- For ColdFire V1 Target
- For Kinetis Target

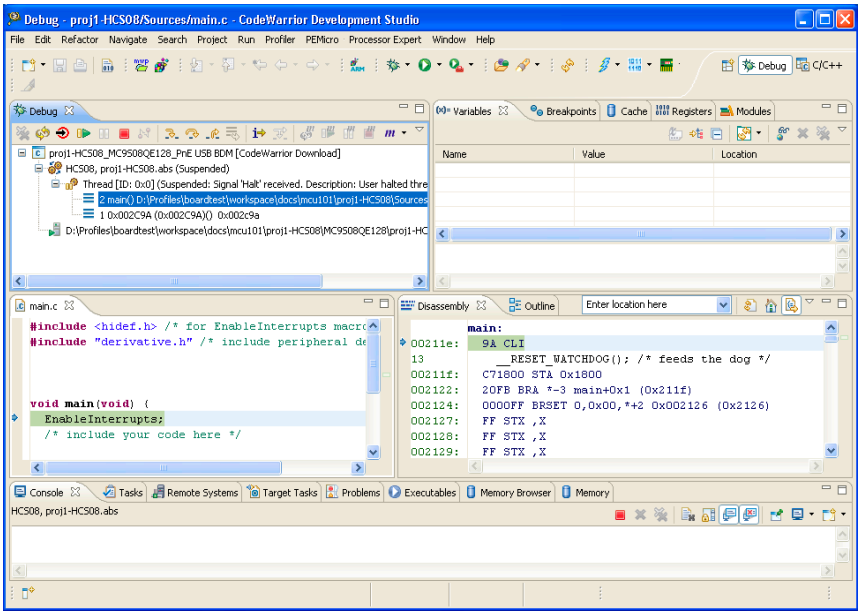
For HCS08 Target

To collect data for the HCS08 target:

1. In the **Debug Configurations** dialog box, click **Debug** to launch the project.

   The application halts at the beginning of `main()`.

Figure 3.33 Debug Perspective Page — HCS08



2. Click the **Resume** icon to resume the execution and begin measurement. Let the application run for several seconds before performing the next step. If you want to stop the execution of the application while it is running, click the **Suspend** icon. Click the **Terminate** icon to stop the measurement.

NOTE To switch to a different perspective, click the **Show List** icon on the upper right corner of the window.

For ColdFire V1 Target

To collect data for the ColdFire V1 target, follow the steps explained in For HCS08 Target.

For Kinetis Target

To collect data for the Kinetis target:

1. In the **Debug Configurations** dialog box, click **Debug** to launch the project. The application halts at the beginning of `main()`.
2. Click **Resume** to resume the execution.

3. Click **Suspend** after some time.
4. Wait till **Resume** gets enabled again.

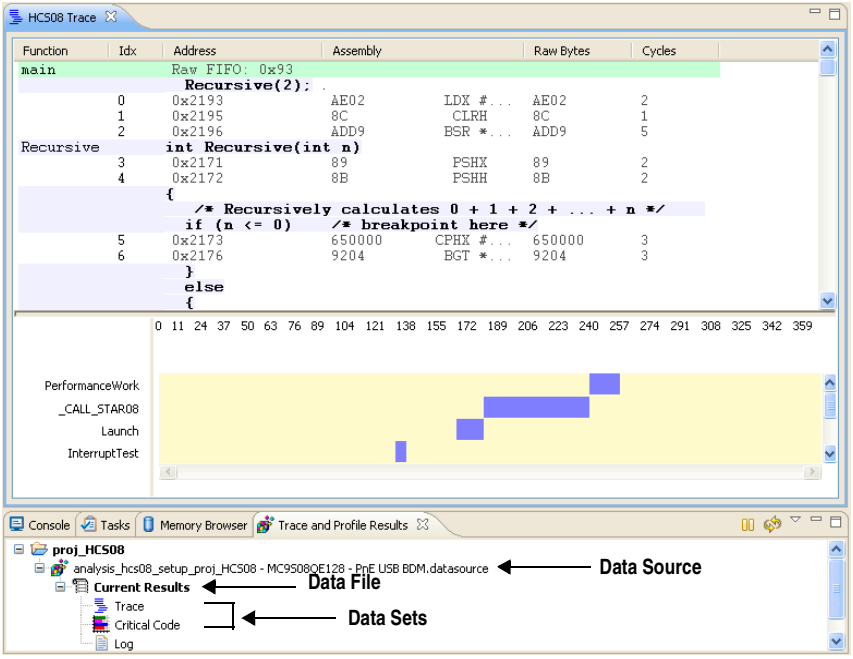
Viewing Data

For HCS08 Target

To view the data file created while running the application on the HCS08 target:

1. From the menu bar, select **Profiler > All Results** to open the **Trace and Profile Results** view (see Figure 3.34).
2. The data source is listed under the project name.
3. Expand the data source and note the addition of the data file, **Current Results**. Expand the data file to view the **Trace** and **Critical Code** data sets for the collected data.

Figure 3.34 Trace and Profile Results View



In the **Trace and Profile Results** view, you can see a **Start** /**Stop Trace** toggle button and a **Reset Trace** button on the toolbar. The state of these buttons depend upon the debug session.

- If there is no debug session, the **Start/Stop Trace** button is enabled and the **Reset Trace** button is disabled.
- If the debug session is running, both **Start/Stop Trace** and the **Reset Trace** buttons are disabled.
- If the debug session is stopped, both the buttons become enabled.

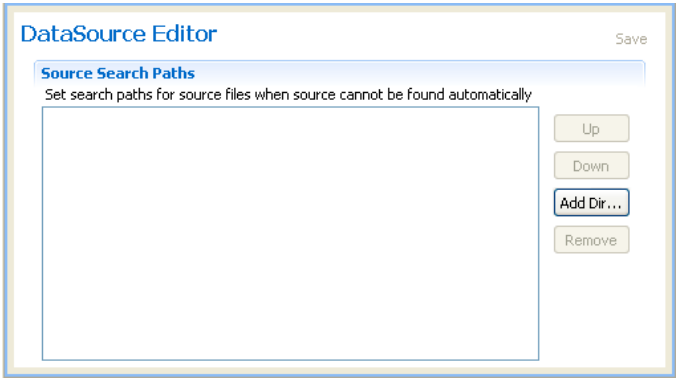
The trace collection is enabled by default. The **Stop Trace** button is visible when the trace collection is enabled. When you click this button, the trace collection is disabled and the **Start Trace** button becomes visible. You can enable the trace collection by clicking the **Start Trace** button. The trace data will not be collected for the time interval when trace collection was stopped. The **Reset Trace** button deletes the collected trace.

You can also modify the settings of the collected data through **Trace and Profile Results** view. Refer to the topic, Modifying Data Collection Settings.

Modifying Data Collection Settings

If you want to review or modify the data collection settings, double-click the data source. The **DataSource Editor** page appears (see Figure 3.35). This page lets you specify the source search paths when source file, `main.c` is not found automatically. This may happen if you change the default location of the source file. Therefore, you need to specify the new location of the source file for trace collection.

Figure 3.35 Data Collection Settings Page — HCS08



To specify the modified source search path, click **Add Dir** and browse to the path where the `main.c` file is located.

For ColdFire V1 Target

To view the data file created while running the application on the ColdFire V1 target, follow the steps explained in For HCS08 Target.

You can also modify the settings of the collected data through **Trace and Profile Results** view. Refer to the topic, Modifying Data Collection Settings.

Modifying Data Collection Settings

If you want to review or modify the data collection settings, double-click the data source. The **Data Collection Settings** page appears. This page lets you specify the source search paths when source file, `main.c` is not found automatically. This may happen if you change the default location of the source file. Therefore, you need to specify the new location of the source file for trace collection.

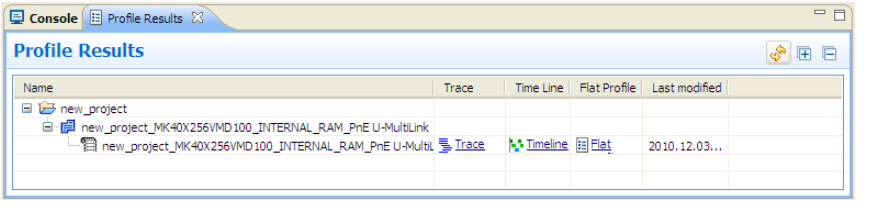
For Kinetis Target

To view the data file created while running the application on the Kinetis target:

1. From the menu bar, select **Profiler > ARM - Trace and Profile Results** to open the **Profile Results** view (see Figure 3.36).
2. Expand the project name.

   The data source is listed under the project name along with the hyperlinks to the **Trace**, **Timeline**, and **Flat Profile** results.

Figure 3.36 Profile Results View



3. Click on the **Trace**, **Timeline**, and **Flat Profile** hyperlinks and view the trace, timeline and flat profile data results in the **ARM Pioneer Trace Data** and **Flat Profile** pages.

You can click  to refresh the displayed data. Click  to expand all the nodes and  to collapse all the nodes in the **Profile Results** view.

Alternatively, you can perform these actions by right-clicking on the data source and selecting the appropriate option from the context menu.

To save the trace, timeline, and profile results, select the **Save Results** option from the context menu. To delete the results, select the **Delete Results** option.

To copy the cell that is currently selected, select the **Copy Cell** option from the context menu. The name of the data source is copied to the clipboard. To copy the complete line of the data source, select the **Copy Line** option.

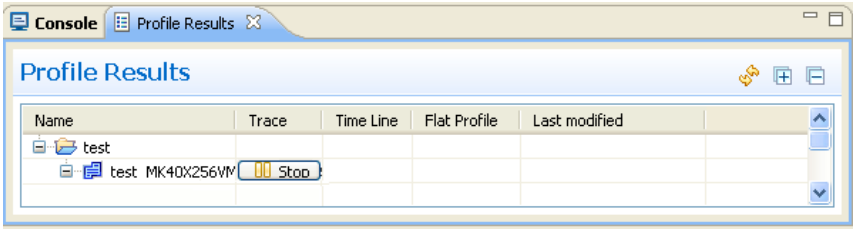
You can control generation of trace using the **Profile Results** view. Refer Controlling Trace Generation Using Start/Stop Toggle Button for more information.

Controlling Trace Generation Using Start/Stop Toggle Button

Every project that has the Trace and Profile enabled will be listed in the **Profile Results** view. The **Trace**, **Timeline**, and **Flat Profile** hyperlinks appear when trace is collected for that project. You can control the generation of trace from the **Profile Results** view using the **Start/Stop** toggle button. The same button is used to start or stop the trace. This toggle button appears on launching the debug session of an application. The default status of trace collection is ON. Therefore, when the application is debugged, the **Stop** toggle button appears next to the data source as shown in Figure 3.37. When clicked, the button toggles to 

.

Figure 3.37 Stop Toggle Button



To stop trace collection:

1. Debug the application.

   The **Stop** toggle button appears next to the data source name in the **Profile Results** view.

2. When the application halts at the program entry point, `main()`, click the **Stop** toggle button.

   The button toggles to Start.

3. Click **Resume**.
4. Click **Suspend** after some time.
5. Open the **Profile Results** view.

   The trace is not collected.

To start trace collection:

1. Click Start.

   The button toggles to Stop.

2. Click **Resume**.
3. Click **Suspend** after some time.
4. Open the **Profile Results** view.

   The trace is collected.

The toggle button disappears when you click **Resume** or terminate the debug session. After clicking **Suspend**, it is visible again with the last selected status.

4

Viewing Data

To understand different types of data collected by the target hardware and how you can view that data, read the following topics:

- Viewing Data for HCS08 Target
- Viewing Data for ColdFire V1 Target
- Viewing Data for Kinetis Target

Viewing Data for HCS08 Target

You can view trace and critical code data for the HCS08 target from the data file generated after running the application.

- Trace Data
- Critical Code Data

Trace Data

To view the data file:

1. Select **Profiler > All Results**. Alternatively, select **Window > Show View > Other > Analysis > Trace and Profile Results**.

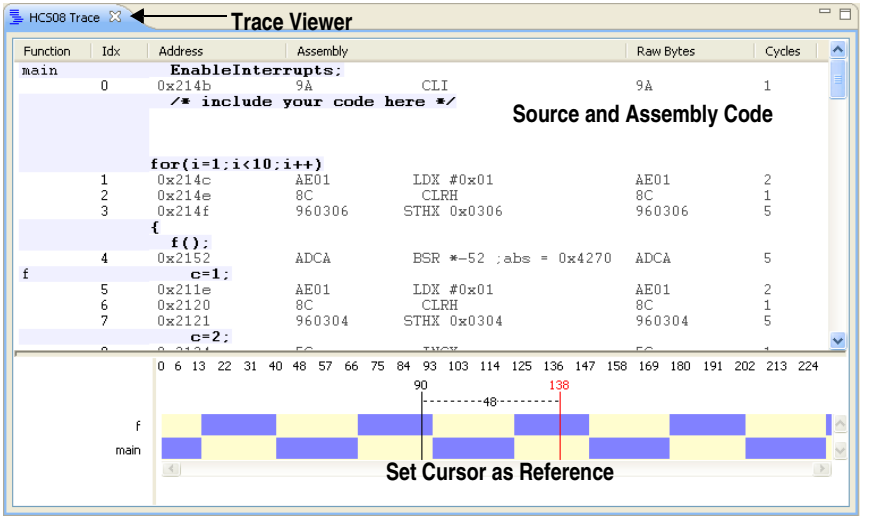
   The **Trace and Profile Results** view appears in the **Debug** perspective of the CodeWarrior IDE.

2. To view the trace data, expand the **Current Results** data file.

   A list of data sets appears.

3. Double-click on the **Trace** data set to view the trace data, as shown in Figure 4.1.

Figure 4.1 Trace Data Page — HCS08



The **Trace Data** viewer displays the trace data collected by the target hardware. The critical code data is also generated based on the trace data. Both the source and assembly code are displayed in the trace table but only the source code is highlighted in bold. The **Trace Data** viewer also displays the **Raw Fifo** information, which shows the actual contents of the hardware trace buffer. When you decode these contents, you obtain the regular trace. To view the **Raw Fifo** information, right-click on the **Trace Data** viewer, and select the **Show Raw Fifo** option from the context menu. Figure 4.2 shows the **Trace Data** viewer with the **Raw Fifo** information.

You can hide the **Raw Fifo** information by selecting the **Hide Raw Fifo** option from the context menu.

Figure 4.2 Trace Data Viewer Displaying Raw Fifo Information — HCS08

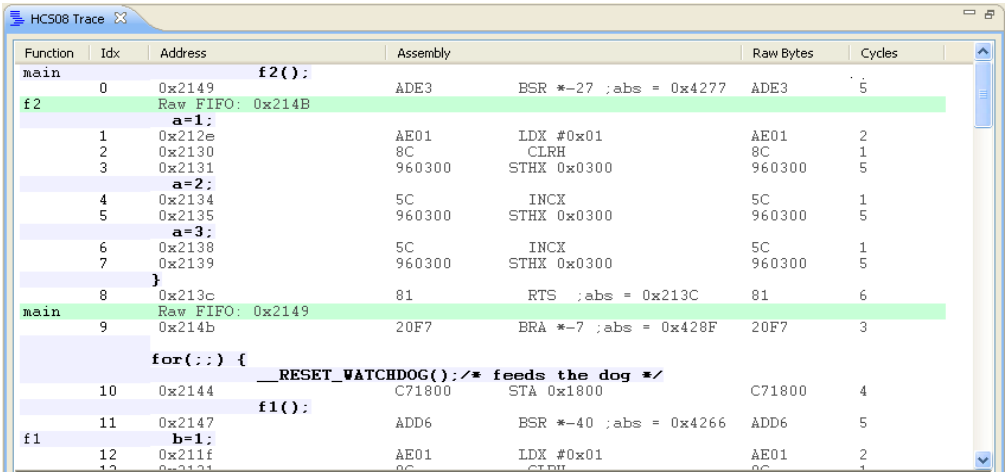


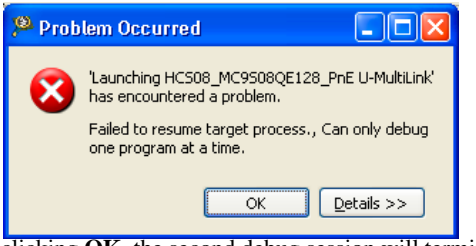
Table 4.1 describes the fields of the trace data.

Table 4.1 Trace Data — Description of Fields

Field	Description
Function	Displays the name of the function.
Idx	Displays the order number of the instructions in assembly code.
Address	Displays the address of the function or the variable. This field also displays the source code of the functions in the program along with their program counter.
Assembly	Displays the disassembly of the instruction.
Raw Bytes	Displays the machine code of the instruction.
Cycles	Displays the number of clock cycles that the instruction takes to execute.

NOTE While a debug session is running, if you relaunch the same debug session by clicking the Debug icon or pressing the `F11` key, the trace data is not collected anymore for the initial debug session. An error message will appear, as shown in Figure 4.3.

Figure 4.3 Error Message



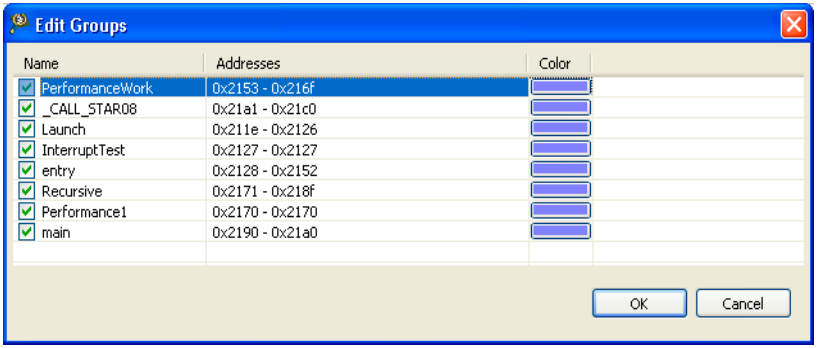
On clicking **OK**, the second debug session will terminate and trace will not collect. The *No trace collected yet* message will appear on double-clicking the Trace node in the **Trace and Profile Results** view. To collect trace, terminate the initial debug session and start it again.

You can also copy the contents of a line or cell to any text file. To copy a line or cell, right-click on the trace table and select **Copy Line** or **Copy Cell** from the context menu. If you are in the middle of the data and want to navigate to the start or end of the trace data, right-click on the trace table and select **Go to Start** or **Go To End** from the context menu.

The graph in the **Trace Data** viewer window shows the time (in cycles) that the application takes in executing the instructions. Each function is shown on the y-axis and time is shown on the x-axis. The length of the function bars depends on the number of cycles. You can zoom in the graph by moving the mouse wheel to view a range of addresses and the actual number of cycles taken by the functions. You can perform the following actions on the graph:

- **Set Reference** — Lets you set the cursor for reference by pressing `ALT+S` in the graph and compute the difference between the cycles. When you place the cursor on the graph, the GUI searches for a nearby edge to move the cursor there. Alternatively, you can right-click on the graph and select the **Set Reference** option from the context menu. You can delete a reference by selecting the **Delete Reference** option from the context menu or pressing `ALT+D` in the graph.
- **Sync with Trace** — Lets you synchronize data shown in the graph with the trace data present in the **Trace Data** viewer. Right-click on the graph and select the **Sync with Trace** option. Alternatively, click on the graph at a place which you want to sync with trace in the **Trace Data** viewer and press `ALT+T`. The trace data corresponding to the data in the graph will be highlighted in the **Trace Data** viewer.
- **Edit Groups** — Lets you customize the timeline according to your requirements. For example, you can change the default color of the line bars representing the functions to differentiate between them. You can add/remove a function to/from the timeline. To perform these functions, right-click on the graph and select **Edit Groups**. Alternatively, press `ALT+G`. The **Edit Groups** dialog box appears.

Figure 4.4 Edit Groups Dialog Box



You can perform the following operations in the **Edit Groups** dialog box:

- Add/Remove Function
- Edit Address Range of Function
- Change Color
- Add/Remove Group
- Merge Groups/Functions

Add/Remove Function

Right-click on the function name in the **Name** column, and select **Insert Function** or **Delete Selected** from the context menu. You can also remove a function from the graph by clearing the corresponding checkbox in the **Name** column. Check it again to include it in the graph.

Edit Address Range of Function

1. Select the function of which you want to change the address range.
2. Double-click on the cell of the **Addresses** column of the selected function.

   The cell becomes editable.
3. Type an address range for the group/function in the cell.

NOTE You can specify multiple address ranges to a function. The multiple address ranges are separated by a comma.

Change Color

You can change the color of a function displayed as a horizontal bar in the timeline graph. Click on the **Color** column of the corresponding function, and select the color of your choice from the **Color** window that appears.

Add/Remove Group

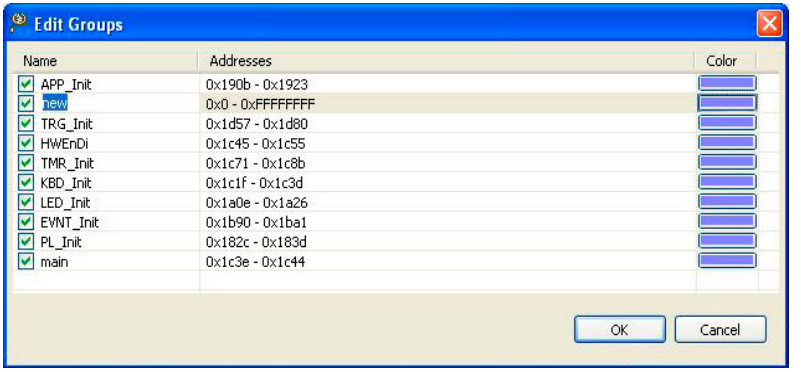
A group is a range of addresses. In case, you want to view trace of a part of a function only, for example, `for` loop, you can find the addresses of the loop and create a group for those addresses.

To add a group:

1. Right-click on the **Edit Groups** dialog box, and select **Insert Group** from the context menu.

   A row is added to the table with *new* as function name.

Figure 4.5 Adding Group

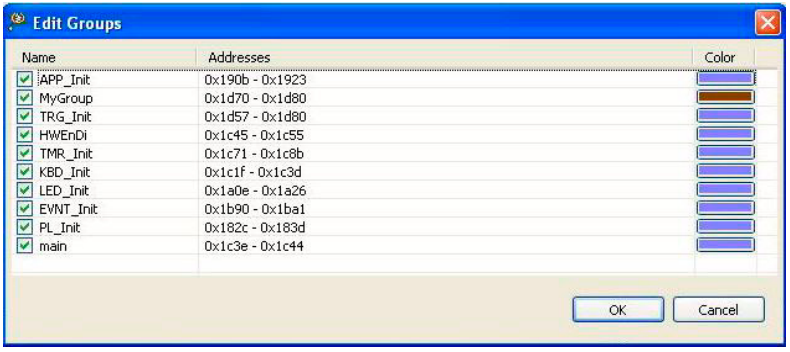


2. Double-click on the *new* group cell.

   The cell becomes editable.

3. Type a name for the group, for example, *MyGroup*.
4. Double-click on the cell of the corresponding **Addresses** column, and edit the address range according to requirements.
5. Change the color of the group.

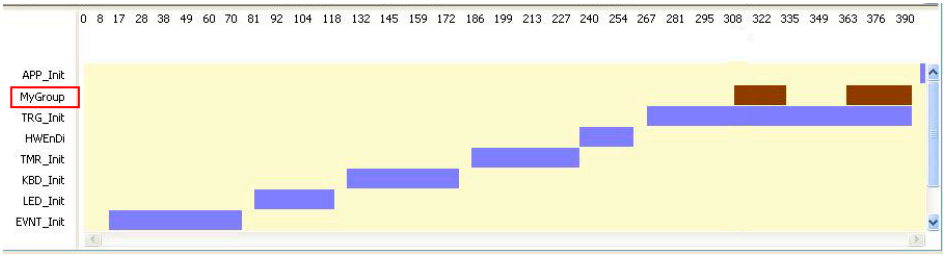
Figure 4.6 Edit Groups Dialog Box After Editing Address Range and Color of Group



6. Click **OK**.

   The *MyGroup* group is added to the timeline.

Figure 4.7 Timeline After Adding Group



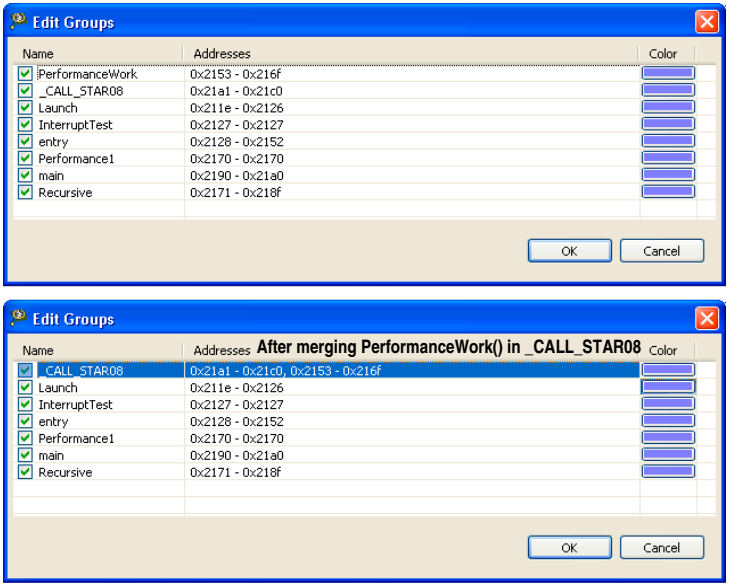
To delete a group, select it, right-click on the **Edit Groups** dialog box, and select the **Delete Selected** option from the context menu. You can also remove a group from the graph by clearing the corresponding checkbox in the **Name** column. Check it again to include it in the graph.

Merge Groups/Functions

1. In the **Edit Groups** dialog box, select the function/group to be merged.
2. Drag and drop it in the function/group with which you want it to get merged with.

   Both the functions/groups merge into a single function/group that covers both address ranges, as shown in Figure 4.8, where the function, `PerformanceWork()` is merged into another function, `_CALL_STAR08`.

Figure 4.8 Merging Functions/Groups



3. Click **OK**.

NOTE Merging is useful in case there are many functions and you do not want to view trace of each and every function.
You cannot undo this operation, that is you cannot separate the merged functions/groups. To view the original trace data, reopen the **Trace Data** viewer.

Critical Code Data

To view a critical code data file:

1. Select **Profiler > All Results**. Alternatively, select **Window > Show View > Other > Analysis > Trace and Profile Results**.

   The **Trace and Profile Results** view appears.

2. Expand the **Current Results** data file.

   A list of data sets appears.

3. Double-click on the **Critical Code** data set to view the critical code data, as shown in Figure 4.9.

The **Critical Code Data** viewer displays the summarized data of a function. The information in the table can be sorted by column in ascending or descending order. You can sort the summarized data by clicking a column header.

Figure 4.9 Critical Code Data Page — HCS08

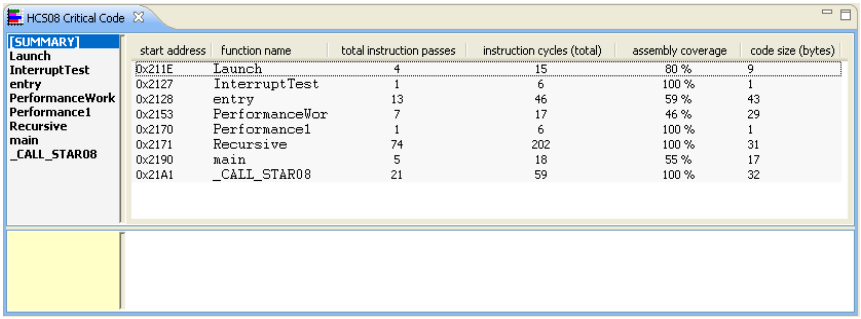


Table 4.2 describes the fields of the critical code data.

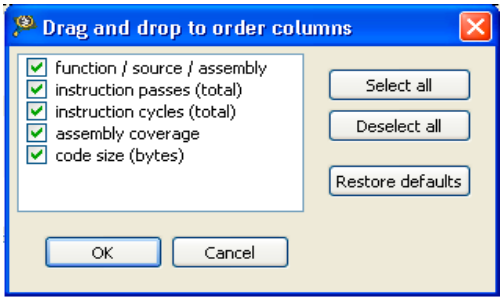
Table 4.2 Critical Code Data — Description of Fields

Field	Description
Start Address	Displays the start address of the function.
Function Name	Displays the name of function that has executed.
Total instruction passes	Displays the total number of times instruction executed in the function.
Instruction cycles (total)	Displays the total number of cycles that the instruction takes.
Assembly coverage	Displays the number of instructions executed from the total number of instructions in a function.
Code size (bytes)	Displays the number of bytes required by each function.

When you click on `[SUMMARY]`, the detailed information of all the functions appears in a table on the right side. Click on the function name below `[SUMMARY]` to view the details of each instruction. The `[* Processing]` tag appears for a short duration and starts computing the information for that function. After computing, the table shows the source or assembly code and statistics for all the instructions in that function. Once the statistics are ready, the `[* Processing]` tag disappears. The frequent access of the same function takes less time to compute the related statistics.

By default, all the columns appear in the summary table. To show or hide a column, right-click on the table and select **Display All Columns** option from the context menu. The **Drag and drop to order columns** dialog box appears, as shown in Figure 4.10. Check or clear the checkboxes according to your requirements and click **OK**.

Figure 4.10 Drag and Drop to Order Columns Dialog Box



You can sort the summary table but not the function table as it is already in order. The progress bar appears only for the **passes** field for each instruction in the function table, as shown in Figure 4.11.

Figure 4.11 Statistics of Critical Code Data Page — HCS08

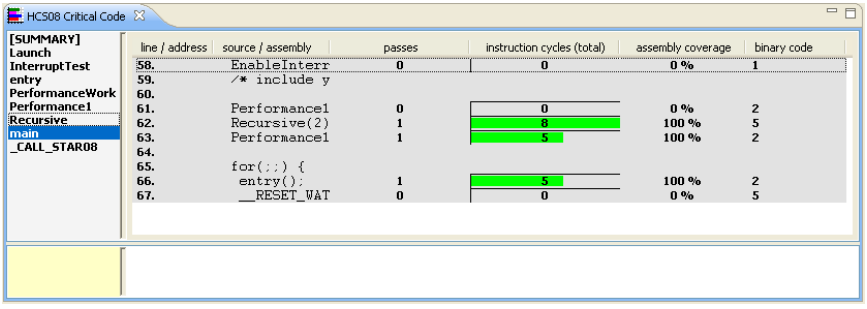


Table 4.3 describes the fields of statistics of the critical code data.

Table 4.3 Statistics of Critical Code Data

Field	Description
Line/Address	Displays either the line number for each instruction in the source code or the address for the assembly code.
Source/Assembly	Displays either the source code or the assembly code of a function.

Table 4.3 Statistics of Critical Code Data

Field	Description
Passes	Displays the number of times each instruction is executed.
Instruction cycles (total)	Displays the total number of cycles that the instruction takes.
Assembly coverage	Displays the number of instructions executed from the total number of instructions in a function.
Binary code	Displays the binary code in the hexadecimal format for the assembly code and displays the number of bytes for the source code.

To copy an instruction to the clipboard, select the instruction in the critical code data table, right-click, and select the **Copy Line** option from the context menu. To go to the `main()` function definition in the source editor area, right-click on the critical code data table and select the **Sync with function definition** option from the context menu.

To view the source, assembly or mixed code of a particular function, right-click on the function in the critical code data table and select **Show Code > Source** or **Assembly** or **Mixed** from the context menu. If you want to view the progress bar for either source or assembly for the **passes** field, right-click and select **Graphics on** from the context menu. The **Source** and **Assembly** options appear. You can select either of the option to view the progress bar for source or assembly code.

NOTE In the HCS08 target, critical code data is not available in the **Automatically**, **Collect Data Trace**, and **Expert** modes.

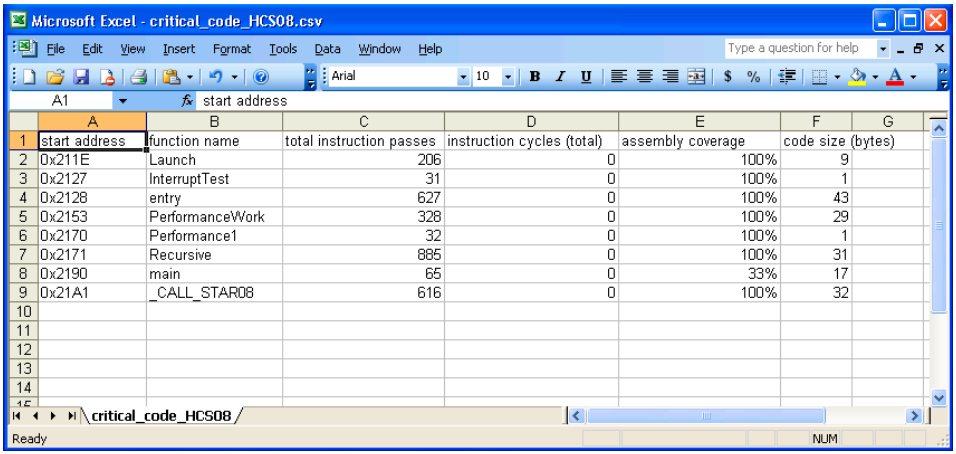
You can export the trace and critical code data collected on the HCS08 target. Refer to the topic, Exporting Data.

Exporting Data

You can export the contents of the trace data and the critical code data tables to a CSV file. To export the trace or critical code data, right-click on the trace or critical code data table and select **Export** from the context menu. The **Save** dialog box appears.

You can specify the name of the file in which you want to export the trace or critical code data. The data will be exported to a file, as shown in Figure 4.12.

Figure 4.12 CSV File Page — HCS08 Critical Code Data



	A	B	C	D	E	F	G
1	start address	function name	total instruction passes	instruction cycles (total)	assembly coverage	code size (bytes)	
2	0x211E	Launch	206	0	100%	9	
3	0x2127	InterruptTest	31	0	100%	1	
4	0x2128	entry	627	0	100%	43	
5	0x2153	PerformanceWork	328	0	100%	29	
6	0x2170	Performance1	32	0	100%	1	
7	0x2171	Recursive	885	0	100%	31	
8	0x2190	main	65	0	33%	17	
9	0x21A1	_CALL_STAR08	616	0	100%	32	
10							
11							
12							
13							
14							

Viewing Data for ColdFire V1 Target

You can view trace and critical code data for the ColdFire V1 target from the data file generated after running the application.

- Trace Data
- Critical Code Data

Trace Data

To view the data file:

1. Select **Profiler > All Results**. Alternatively, select **Window > Show View > Other > Analysis > Trace and Profile Results**.

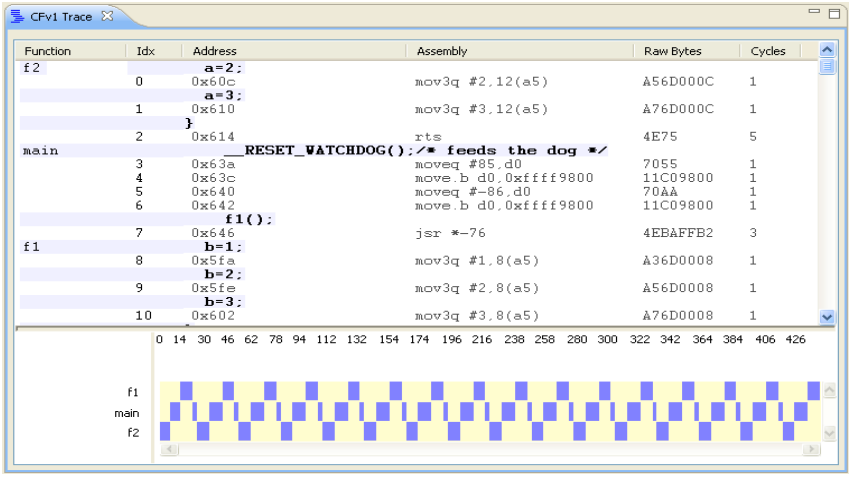
   The **Trace and Profile Results** view appears in the **Debug** perspective of the CodeWarrior IDE.

2. To view the trace data, expand the **Current Results** data file.

   A list of data sets appears.

3. Double-click on the **Trace** data set to view the trace data, as shown in Figure 4.13.

Figure 4.13 Trace Data Page — ColdFire V1



The **Trace Data** viewer displays the trace data collected by the target hardware. The critical code data is also generated based on the trace data. Both source and assembly codes are displayed in the trace table but only the source code is highlighted in bold. The **Trace Data** viewer also displays the **Raw Fifo** information, which shows the actual contents of the hardware trace buffer. When you decode these contents, you obtain the regular trace. To view the **Raw Fifo** information, right-click on the **Trace Data** viewer, and select the **Show Raw Fifo** option from the context menu. Figure 4.14 shows the **Trace Data** viewer with the **Raw Fifo** information.

You can hide the **Raw Fifo** information by selecting the **Hide Raw Fifo** option from the context menu.

Figure 4.14 Trace Data Viewer Displaying Raw Fifo Information — ColdFire V1

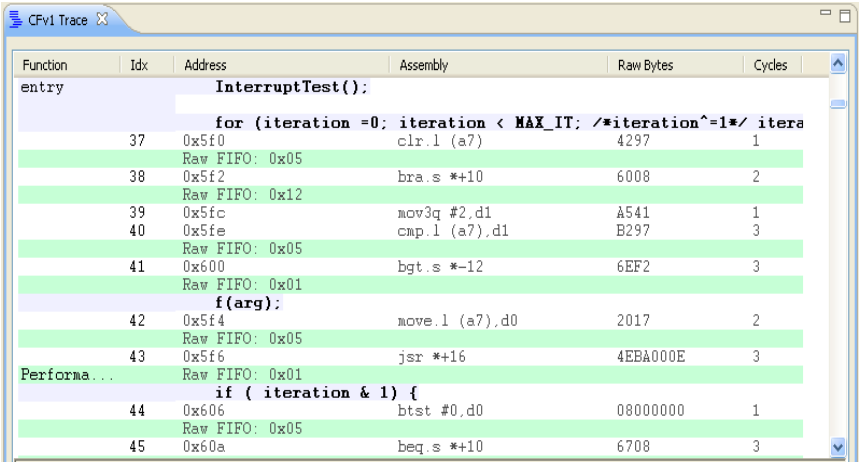


Table 4.4 lists the fields of the trace data table.

Table 4.4 Trace Data — Description of Fields

Field	Description
Function	Displays the name of the function.
Idx	Displays the order number of the instructions in assembly code.
Address	Displays the address of the function or the variable. This field also displays the source code of the functions in the program along with their program counter.
Assembly	Displays the disassembly of the instruction.
Raw Bytes	Displays the machine code of the instruction.
Cycles	Displays the number of clock cycles that the instruction takes to execute.

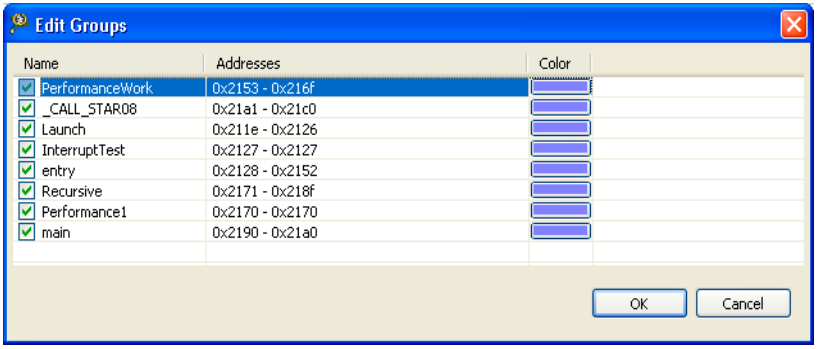
You can also copy the contents of a line or cell to any text file. To copy a line or cell, right-click on the trace table and select **Copy Line** or **Copy Cell** from the context menu. If you are in the middle of the data and want to navigate to the start or end of the trace data, right-click on the table and select **Go to Start** or **Go To End** from the context menu.

The graph in the **Trace Data** viewer window shows the time (in cycles) that the application takes in executing the instructions. Each function is shown on the y-axis and

time is shown on the x-axis. You can zoom in the graph to view a range of addresses and the actual number of cycles taken by the functions. You can perform the following actions on the graph:

- **Set Reference** — Lets you set the cursor for reference by pressing `ALT+S` in the graph and compute the difference between the cycles. When you place the cursor on the graph, the GUI searches for a nearby edge to move the cursor there. Alternatively, you can right-click on the graph and select the **Set Reference** option from the context menu. You can delete a reference by selecting the **Delete Reference** option from the context menu or pressing `ALT+D` in the graph.
- **Sync with Trace** — Lets you synchronize data shown in the graph with the trace data present in the **Trace Data** viewer. Right-click on the graph and select the **Sync with Trace** option. Alternatively, click on the graph at a place which you want to sync with trace in the **Trace Data** viewer and press `ALT+T`. The trace data corresponding to the data in the graph will be highlighted in the **Trace Data** viewer.
- **Edit Groups** — Lets you customize the timeline according to your requirements. For example, you can change the default color of the line bars representing the functions to differentiate between them. You can add/remove a function to/from the timeline. To perform these functions, right-click on the graph and select **Edit Groups**. Alternatively, press `ALT+G`. The **Edit Groups** dialog box appears.

Figure 4.15 Edit Groups Dialog Box



You can perform the following operations in the **Edit Groups** dialog box:

- Add/Remove Function
- Edit Address Range of Function
- Change Color
- Add/Remove Group
- Merge Groups/Functions

Critical Code Data

To view a critical code data file for the ColdFire V1 target:

1. Select **Profiler > All Results**. Alternatively, select **Window > Show View > Other > Analysis > Trace and Profile Results**.

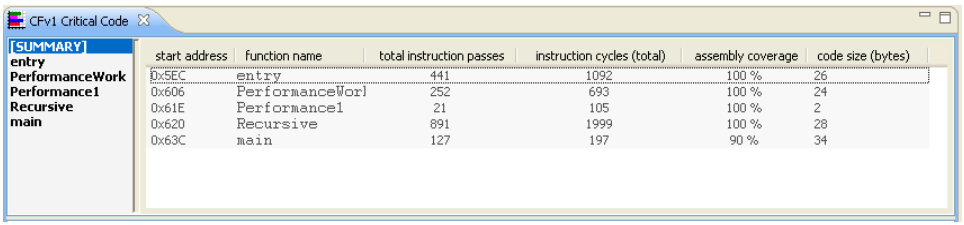
   The **Trace and Profile Results** view is displayed.

2. Expand the **Current Results** data file.

   A list of data sets appears.

3. Double-click on the **Critical Code** data set to view the critical code data, as shown in Figure 4.16.

Figure 4.16 Critical Code Data Page — ColdFire V1



The critical code data displays the summarized data of a function. The information in the table can be sorted by column in ascending or descending order. You can sort the summarized data by clicking a column header.

Table 4.5 describes the fields of the critical code data.

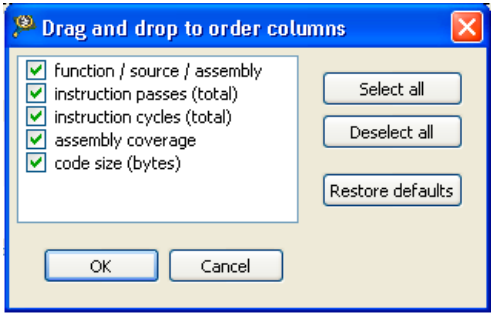
Table 4.5 Critical Code Data — Description of Fields

Field	Description
Start Address	Displays the start address of the function.
Function Name	Displays the name of function that has executed.
Total instruction passes	Displays the total number of times instruction executed in the function.
Instruction cycles (total)	Displays the total number of cycles that the instruction takes.
Assembly coverage	Displays the number of instructions executed from the total number of instructions in a function.
Code size (bytes)	Displays the number of bytes required by each function.

When you click on `[SUMMARY]`, the detailed information of all the functions appears in a table on the right side. Click on the function name below `[SUMMARY]` to view the details of each instruction. The `[* Processing]` tag appears for a short duration and starts computing the information for that function. After computing, the table shows the source or assembly code and statistics for all the instructions in that function. Once the statistics are ready, the `[* Processing]` tag disappears. The frequent access of the same function takes less time to compute the related statistics.

By default, all the columns appear in the summary table. To show or hide a column, right-click on the table and select **Display All Columns** option from the context menu. The **Drag and drop to order columns** dialog box appears, as shown in Figure 4.17. Check or clear the checkboxes according to your requirements and click **OK**.

Figure 4.17 Drag and Drop to Order Columns Dialog Box



You can sort the summary table but not the function table as it is already in order. The progress bar is shown only for the **passes** field for each instruction in the function table, as shown in Figure 4.18.

Figure 4.18 Statistics of Critical Code Data Page — ColdFire V1

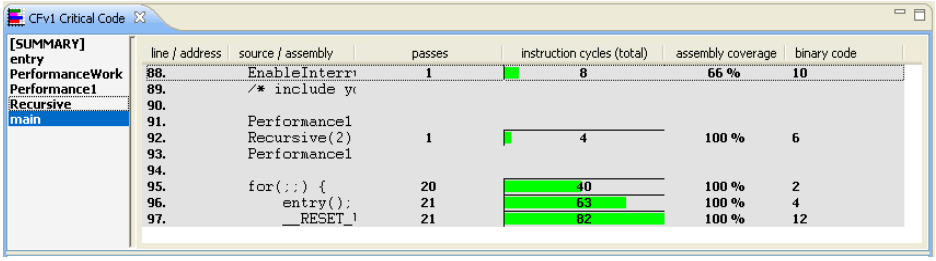


Table 4.6 describes the fields of statistics of the critical code data.

Table 4.6 Statistics of Critical Code Data

Field	Description
Line/Address	Displays either the line number for each instruction in the source code or the address for the assembly code.
Source/Assembly	Displays either the source code or the assembly code of a function.
Passes	Displays the number of times each instruction is executed.
Instruction cycles (total)	Displays the total number of cycles that the instruction takes.
Assembly coverage	Displays the number of instructions executed from the total number of instructions in a function.
Binary code	Displays the binary code in the hexadecimal format for the assembly code and displays the number of bytes for the source code.

To copy an instruction to the clipboard, select the instruction in the critical code data table, right-click, and select the **Copy Line** option from the context menu. To go to the `main()` function definition in the source editor area, right-click on the critical code data table and select the **Sync with function definition** option from the context menu.

To view the source, assembly or mixed code of a particular function, right-click on the function in the table and select **Show Code > Source** or **Assembly** or **Mixed** from the context menu. If you want to view the progress bar for either source or assembly for the **passes** field, right-click on the table and select **Graphics on** from the context menu. The **Source** and **Assembly** options appear. You can select either of the option to view the progress bar for source or assembly code.

NOTE In the ColdFire V1 target, critical code data is not available in the **Automatic (One-buffer)** mode.

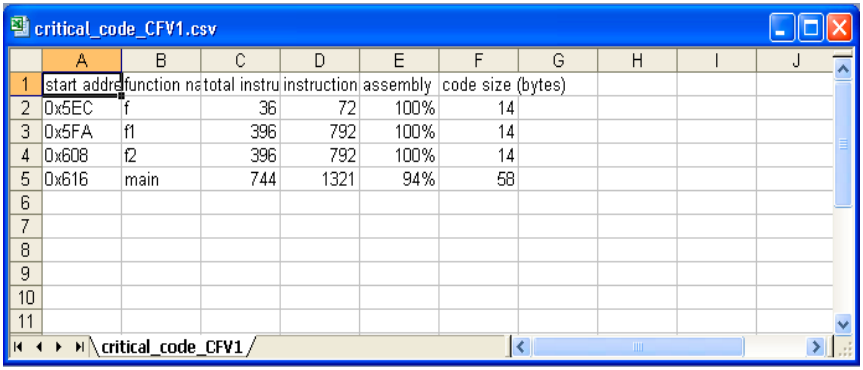
You can export the trace and critical code data collected on the ColdFire V1 target. Refer to the topic, Exporting Data.

Exporting Data

You can export the contents of the trace data and the critical code data tables to a CSV file. To export the trace or critical code data, right-click on the trace or critical code data table and select **Export** from the context menu. The **Save** dialog box appears.

You can specify the name of the file in which you want to export the trace or critical code data. The data will be exported to a file, as shown in Figure 4.19.

Figure 4.19 Critical Code Data CSV File Page — ColdFire V1



Viewing Data for Kinetis Target

You can view trace and flat profile data for the Kinetis target from the data file generated after running the application.

- Trace Data
- Timeline
- Flat Profile Data

Trace Data

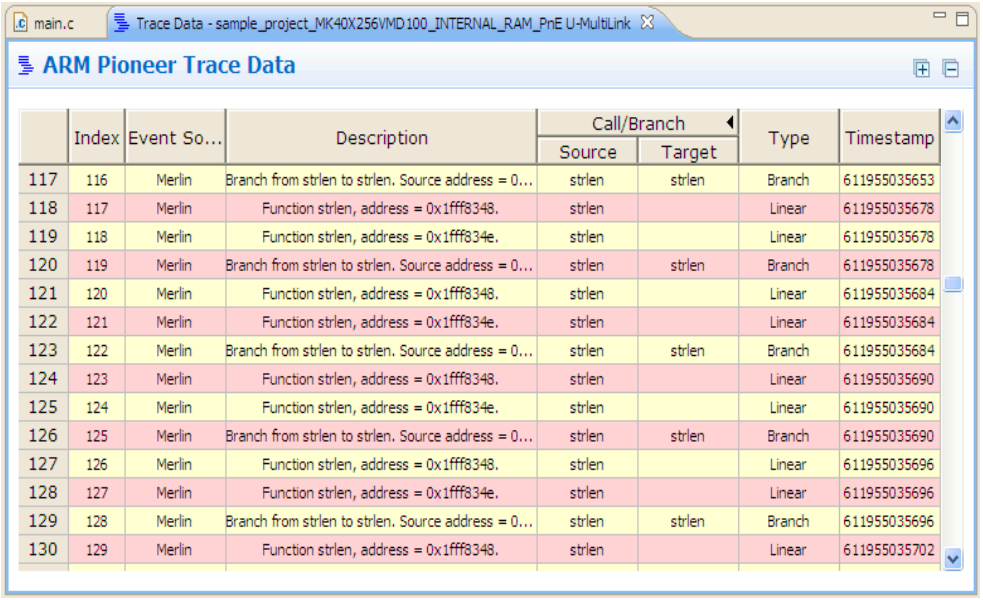
To view trace data:

1. From the menu bar, select **Profiler > ARM - Trace and Profile Results** to open the **Profile Results** view.
2. Expand the project name.

   The data source is listed under the project name.
3. Click on the **Trace** hyperlink.

   The **ARM Pioneer Trace Data** page appears, as shown in Figure 4.20.

Figure 4.20 ARM Pioneer Trace Data Page



	Index	Event So...	Description	Call/Branch Source	Call/Branch Target	Type	Timestamp
117	116	Merlin	Branch from strlen to strlen. Source address = 0...	strlen	strlen	Branch	611955035653
118	117	Merlin	Function strlen, address = 0x1fff8348.	strlen		Linear	611955035678
119	118	Merlin	Function strlen, address = 0x1fff834e.	strlen		Linear	611955035678
120	119	Merlin	Branch from strlen to strlen. Source address = 0...	strlen	strlen	Branch	611955035678
121	120	Merlin	Function strlen, address = 0x1fff8348.	strlen		Linear	611955035684
122	121	Merlin	Function strlen, address = 0x1fff834e.	strlen		Linear	611955035684
123	122	Merlin	Branch from strlen to strlen. Source address = 0...	strlen	strlen	Branch	611955035684
124	123	Merlin	Function strlen, address = 0x1fff8348.	strlen		Linear	611955035690
125	124	Merlin	Function strlen, address = 0x1fff834e.	strlen		Linear	611955035690
126	125	Merlin	Branch from strlen to strlen. Source address = 0...	strlen	strlen	Branch	611955035690
127	126	Merlin	Function strlen, address = 0x1fff8348.	strlen		Linear	611955035696
128	127	Merlin	Function strlen, address = 0x1fff834e.	strlen		Linear	611955035696
129	128	Merlin	Branch from strlen to strlen. Source address = 0...	strlen	strlen	Branch	611955035696
130	129	Merlin	Function strlen, address = 0x1fff8348.	strlen		Linear	611955035702

The **ARM Pioneer Trace Data** page displays the trace data collected by the Kinetis target in a tabular form.

NOTE Depending on the application and runtime conditions, the last few instructions are missed in the trace results.

Table 4.7 describes the fields of the **ARM Pioneer Trace Data** page.

Table 4.7 ARM Pioneer Trace Data

Name	Description
Index	Displays the order number of the instructions.
Event Source	Displays program trace messages from ETM (Merlin) or special messages from ITM.
Description	Displays detailed information about the trace line.
Source	Displays the source function of the trace line if it is a call or a branch.

Table 4.7 ARM Pioneer Trace Data

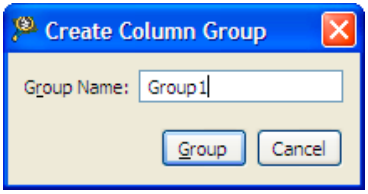
Name	Description
Target	Displays the target function of the trace line if it is a call or a branch.
Type	Displays the type of the trace line, which can either be a linear instruction, a branch, a call, or a custom message.
Timestamp	Displays the absolute clock cycles that the instruction takes to execute.

Click the **Expand All** button,  to display the source as well as assembly code of the instructions in the trace viewer. To display only the assembly code, click the **Collapse All** button,  .

You can modify the appearance or display of the trace results on the **ARM Pioneer Trace Data** page. Right-click on a column to open a context menu that lets you perform the following actions:

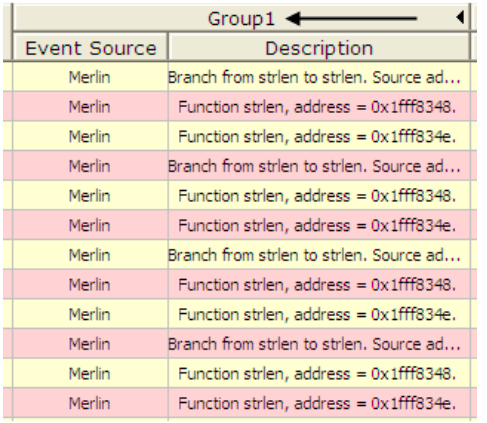
- Hide column — Allows you to hide column(s). To hide a column on the **ARM Pioneer Trace Data** page, select that column, right-click and select the **Hide column** option from the context menu. To display it back, select the **Show all columns** option from the context menu. To hide multiple columns, select the columns with `Ctrl` key pressed, right-click and select the **Hide column** option from the context menu.
- Create column group — Allows you to group multiple columns into one. To group columns, select the columns with `Ctrl` key pressed. Right-click and select the **Create column** group option. The **Create Column Group** dialog box appears.

Figure 4.21 Create Column Group Dialog Box



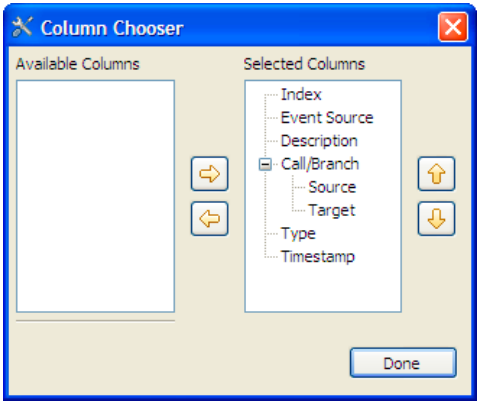
Type a name for the group in the **Group Name** text box and click **Group**. Figure 4.22 shows the **Event Source** and **Description** columns grouped together.

Figure 4.22 Columns Grouping



- Ungroup columns — To ungroup the columns, select the grouped columns, right-click and select the **Ungroup columns** option from the context menu.
- Choose columns — You can set the columns that you want to display on the **ARM Pioneer Trace Data** page. Right-click on any column and select the **Choose columns** option. The **Column Chooser** dialog box appears.

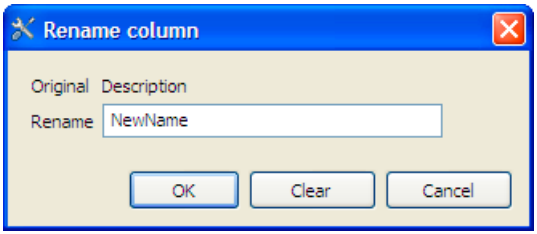
Figure 4.23 Column Chooser Dialog Box



Select the column in the **Selected Columns** section, and use the up and down arrow buttons to position the column up or down according to your choice. Use the left and right arrow buttons to move the columns from **Available Columns** to **Selected Columns** and vice-versa. The columns moved to the **Available Columns** section are not shown on the **ARM Pioneer Trace Data** page. Click **Done** to save the settings.

- Auto resize column — Select the columns, right-click and select the **Auto resize columns** option to resize them according to their width,
- Rename column — Select the column, right-click and select the **Rename column** option. The **Rename Column** dialog box appears. Type a new name for the column in the **Rename** text box and click **OK** (see Figure 4.24).

Figure 4.24 Rename Column Dialog Box



Timeline

The timeline data displays the functions that are executed in the application and the number of cycles each function takes when the application is run.

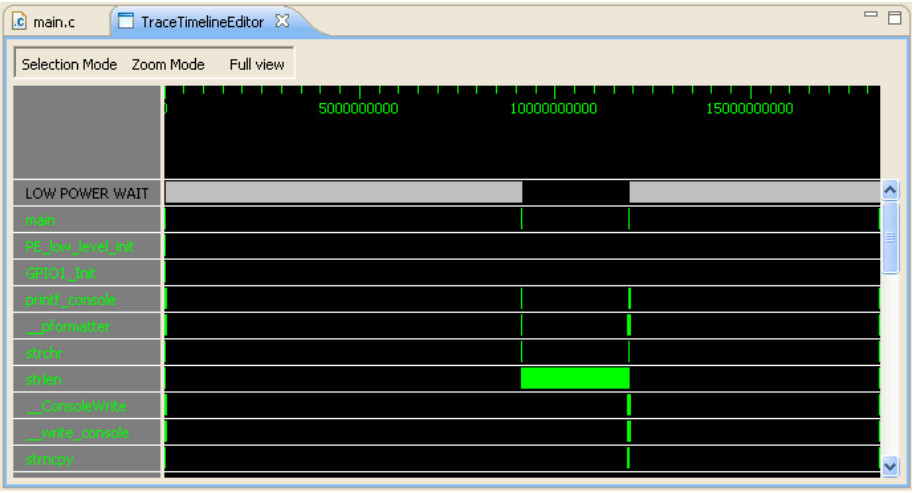
To view timeline data:

1. From the menu bar, select **Profiler > ARM - Trace and Profile Results** to open the **Profile Results** view.
2. Expand the project name.

   The data source is listed under the project name.
3. Click on the **Timeline** hyperlink.

The **TraceTimelineEditor** viewer appears, as shown in Figure 4.25.

Figure 4.25 TraceTimelineEditor Viewer Displaying Timeline Data



The **TraceTimelineEditor** viewer shows a timeline graph in which the functions appear on y-axis and the number of cycles appear on x-axis. The green-colored bars show the time and cycles that the function takes. The gray-colored bar represents the Low Power WAIT Mode mode of the application.

The **TraceTimelineEditor** viewer also have the following buttons:

- **Selection Mode** — Allows you to mark points in the function bars to measure the difference of cycles between those points. To mark a point in the bar:

  a. Click **Selection Mode**.

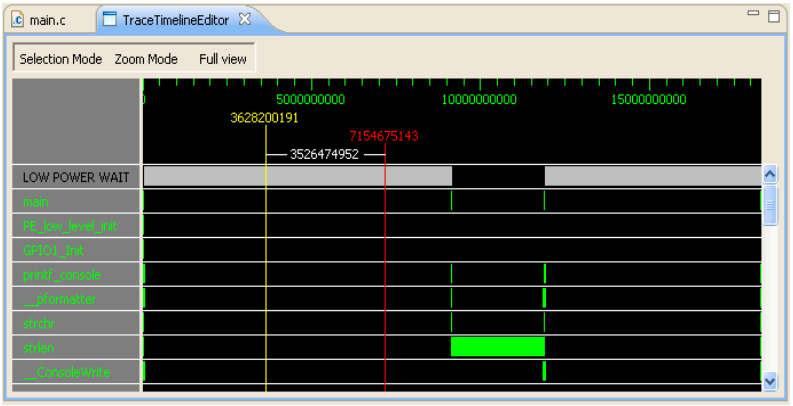
  b. Click on the bar where you want to mark the point.

     A yellow vertical line appears displaying the number of cycles at that point.

  c. Right-click on another point in the bar.

     A red vertical line appears displaying the number of cycles at that point along with the difference of cycles between two marked points (see Figure 4.26).

Figure 4.26 Selection Mode to Measure Difference of Cycles Between Functions



- **Zoom Mode** — Allows you to zoom-in and zoom-out in the timeline graph. Click **Zoom Mode** and then click on the timeline graph to zoom-in. To zoom-out, right-click on the timeline graph. You can also move the mouse wheel up and down to zoom-in and zoom-out.
- **Full View** — Allows you to get back to the original view if you selected the zoom mode.

NOTE The **Selection Mode** is the default mode of the timeline view.

Flat Profile Data

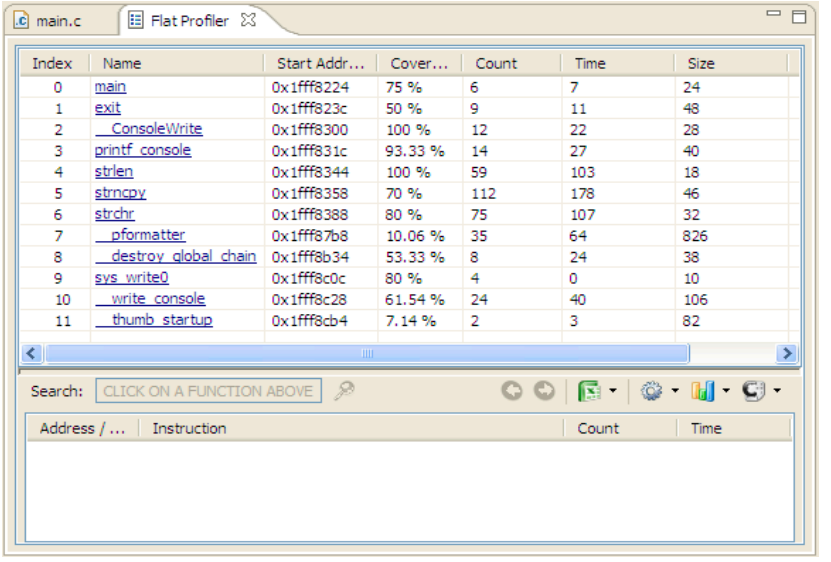
To view flat profile data:

1. From the menu bar, select **Profiler > ARM - Trace and Profile Results** to open the **Profile Results** view.
2. Expand the project name.

   The data source is listed under the project name.
3. Click on the **Flat Profile** hyperlink.

The **Flat Profile** viewer appears, as shown in Figure 4.27.

Figure 4.27 Flat Profile Viewer



The flat profile data displays the summarized data of a function in a tabular form. Table 4.8 describes the fields of the flat profile data.

Table 4.8 Flat Profile Data

Name	Description
Index	Displays the order number.
Name	Displays the name of the function that has executed.
Start Address	Displays the start address of the function.
Coverage	Displays the percentage of number of instructions executed from the total number of instructions in a function.
Count	Displays the number of lines executed in the function.
Time	Displays the total number of clock cycles that the function takes.
Size	Displays the number of bytes required by each function.

The **Flat Profile** viewer divides the flat profile data in two views; the top view displays the summary of the functions, and the bottom view displays the statistics for all the instructions executed in a particular function. Click on a hyperlinked function in the top

view of the **Flat Profile** viewer to view the corresponding statistics for the instructions executed in that function. For example, the statistics of the `main()` function are shown in Figure 4.28.

Figure 4.28 Statistics of Flat Profile Data

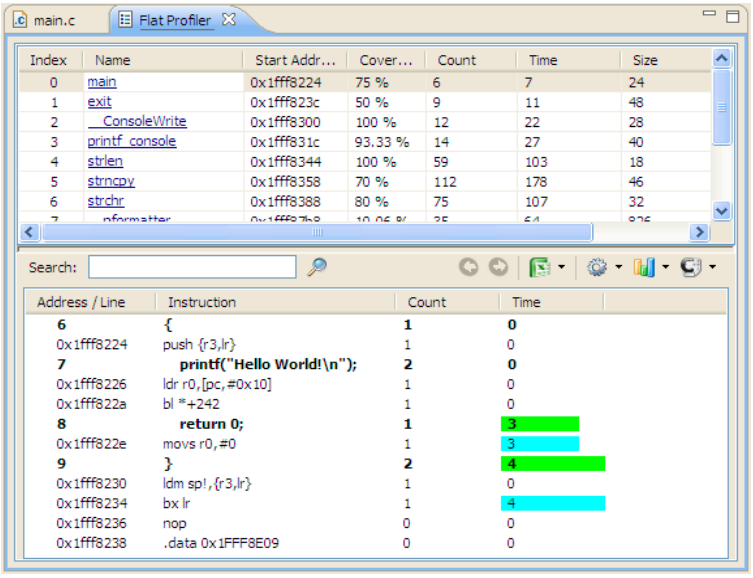


Table 4.9 describes the fields of the statistics of the flat profile data.

Table 4.9 Description of Statistics of Flat Profile Data

Name	Description
Address/Line	Displays either the line number for each instruction in the source code or the address for the assembly code.
Instruction	Displays all the instructions executed in the selected function.
Count	Displays the number of times each instruction is executed.
Time	Displays the total number of clock cycles that each instruction in the function takes.

Click on the column header to sort the flat profile data by that column. However, you can only sort the flat profile data available on the top view. The icons available in the statistics view of the **Flat Profile** viewer allow you to perform the following actions:

- Search — Lets you search for a particular text in the statistics view. In the **Search** text box, type the data that you want to search and click  . The first instance of the data is selected in the statistics view. Click  again or press the `Enter` key to view the next instances of the data.
- Previous function — Lets you view the details of the previous function that was displayed in the bottom view. Click  to view the details of the previous function.
- Next function — Lets you view the details of the next function that was displayed in the bottom view. Click  to view the details of the previous function.
- Export — Lets you export the flat profile data of both top and bottom views in a CSV file. Click  and select the **Export the statistics above** option to export the details of the top view or the **Export the statistics below** option to export the details of the bottom view respectively.
- Configure table — Lets you show and hide column(s) of the flat profile data. Click  and select the **Configure the table above** option to show/hide columns of the top view or the **Configure the table below** option to show/hide columns of the bottom view. The **Drag and drop to order columns** dialog box appears in which you can check/uncheck the checkboxes corresponding to the available columns to show/hide them in the **Flat Profile** viewer.
- Graphics — Lets you display the histograms in two colors for the **Cycle** and **Time** columns in the bottom view of the flat profile data. Click  and select the **Cycle** option to display histograms in the **Cycle** column or the **Time** option to display histograms in the **Time** column. The colors in these columns differentiate source code with the assembly code.
- Show code — Lets you display the assembly or mixed code in the statistics of the flat profile data. Click  and select the **Assembly** option to display the assembly code or the **Mixed** option to display the mixed code in the statistical details.

5

Managing Data File

A data file is a file that is created after collecting data on a target hardware. This data file contains the data that is analyzed for enhancing the performance of an application. This chapter explains how to manage a data file.

Managing a data file consists of the following topics:

- Configuring Data File
- Saving Data

Configuring Data File

When a data file is created, the data source configuration settings are applied to that data file. You can later move the data file or source files to a different location. You can specify a new location of the data file using the **DataFile Editor**.

To open the DataFile Editor for both the targets, HCS08 and ColdFire V1, double-click on the data file, for example, **Current Results**. Figure 5.1 and Figure 5.2 show the HCS08 and ColdFire V1 **DataFile Editor** pages respectively.

Figure 5.1 HCS08 DataFile Editor Page

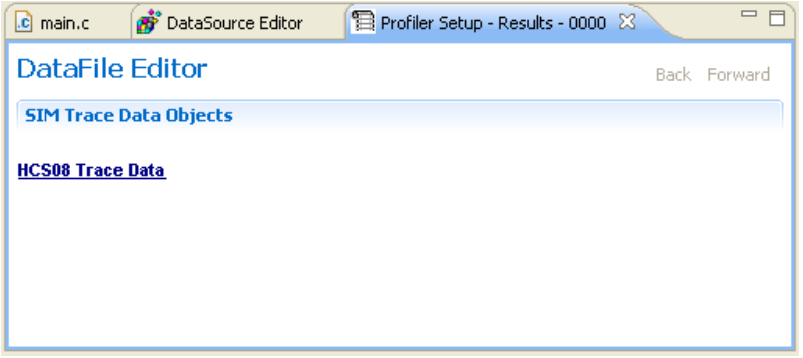
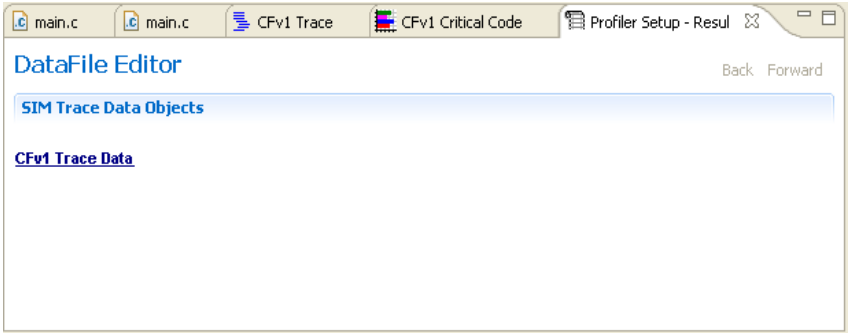


Figure 5.2 ColdFire V1 DataFile Editor Page



To view or modify the settings of the data files, click **HCS08 Trace Data** (for HCS08) or **CFv1 Trace Data** (for ColdFire V1). Figure 5.3 and Figure 5.4 show the **HCS08 Trace Data Object Settings** and **CFv1 Trace Data Object Settings** pages respectively.

Figure 5.3 HCS08 Trace Data Object Settings Page

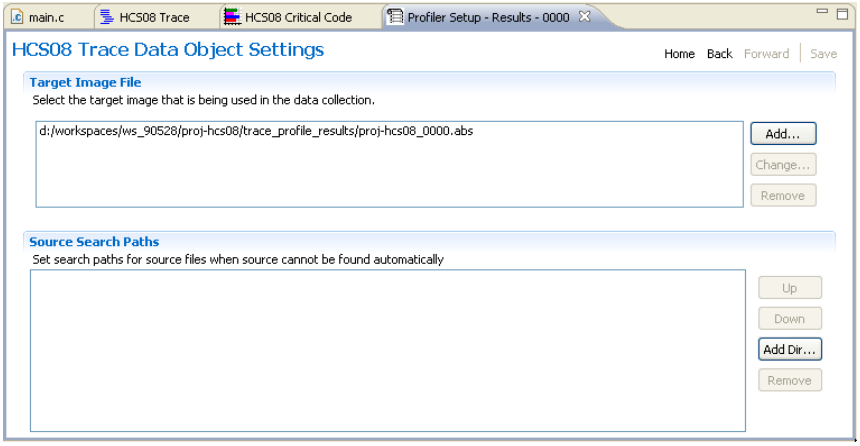
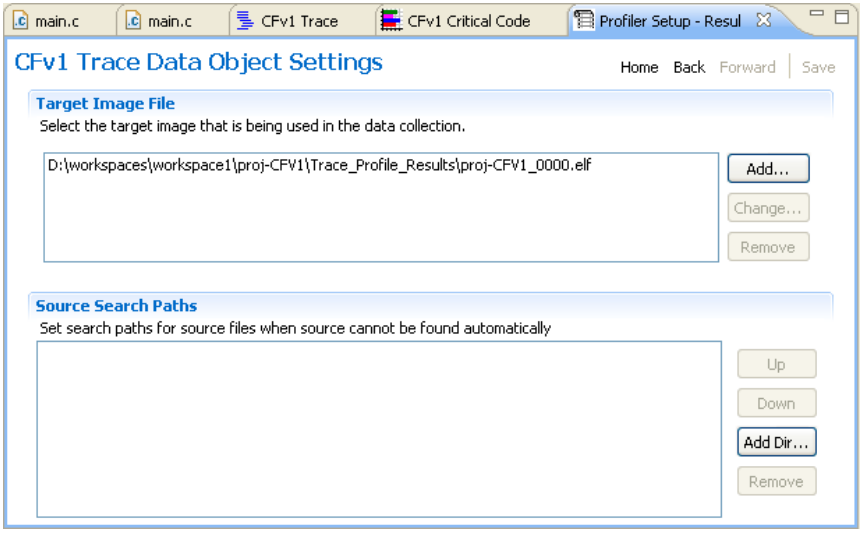


Figure 5.4 CFv1 Trace Data Object Settings Page



To navigate within the **DataFile Editor**, use the following controls:

- Click **Home** to return to the home page.
- Click **Back** to go backward in the display history.
- Click **Forward** to go forward in the display history.
- Click **Save** to save the configuration. The **Save** button gets enabled when you change the location of the target image file or the source search path.

The default location of the target image files is specified in the **Trace Data Object Settings** page. However, if you have imported data files or moved an image after the image was built, you can specify the new location of the target image file.

To specify the target image file, click **Add** in the **Target Image File** section and browse to the application image.

- Click **Change** to change the path to the selected file.
- Click **Remove** to remove the selected file.

NOTE The target image file is the executable file that contains the disassembly information, the source information, and the type of the instruction, such as jump, branch conditional, sequential operation, call, or return.

To specify the source search paths, click **Add Dir** in the **Source Search Paths** section and browse to the path to the source files

- Click **Remove** to remove the selected path

- Click **Up** to move the selected path up in the order
- Click **Down** to move the selected path down in the order

You can also set the source search paths from the **Critical Code Data** viewer. That is, if you have moved the source file from its default location to another location, you can use the **Critical Code Data** viewer to set the source search path to its modified location. These paths will automatically be added to the data source and the **Current Results** data file. To set the source search paths from the **Critical Code Data** viewer:

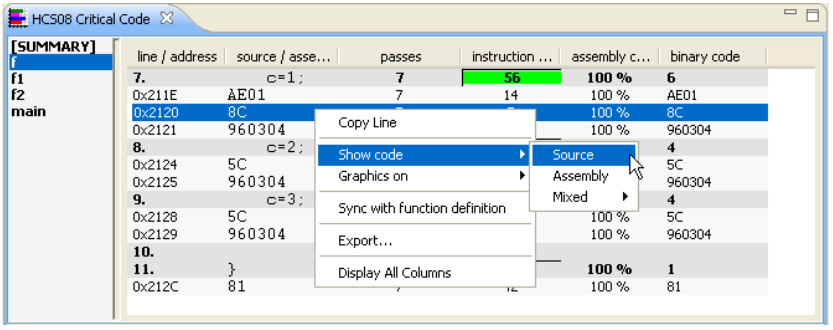
1. Open the **Critical Code Data** viewer.

NOTE To open the **Critical Code Data** viewer, refer Critical Code Data.

2. Double-click on a function.
3. Right-click on the source code and select **Show code > Source** from the context menu (see Figure 5.5).

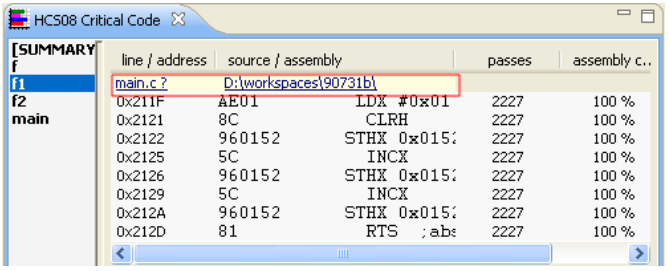
   A hyperlink appears in the function.

Figure 5.5 Setting Source Search Paths from Critical Code Data



4. Click on the hyperlink that appears in the function (see Figure 5.6).

Figure 5.6 Hyperlink in Critical Code Data View



5. Browse to the modified location of the source file in the **Search file** dialog box.

6. Click **Open**. The specified source file location automatically adds to the data source and the **Current Results** data file.
7. Double-click **Current Results** and data source in the **Trace and Profile Results** view to see the results.

Saving Data

The project, data source, data files, and data sets are automatically saved in the Workspace folder. When a new trace is generated, it overwrites the previous trace in the **Current Results** data file. If you want the trace to be saved and not overwritten, you can save the data of the data file. Right-click on the data file and select **Save Current Results**. It will save the data with the date and time stamp, as shown in Figure 5.7 (for HCS08) and Figure 5.8 (for ColdFire V1).

Figure 5.7 HCS08 Trace and Profile Results View — Save Current Results

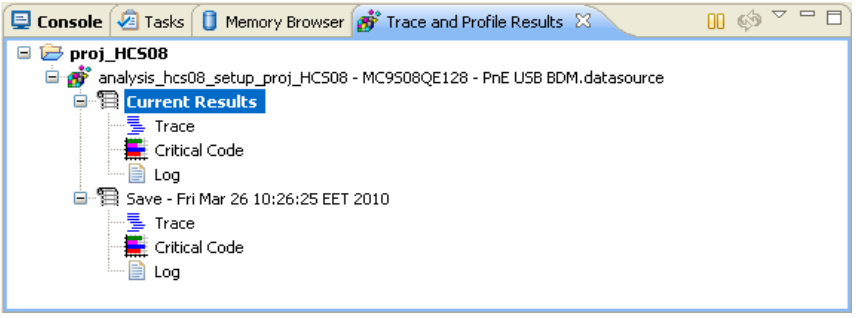
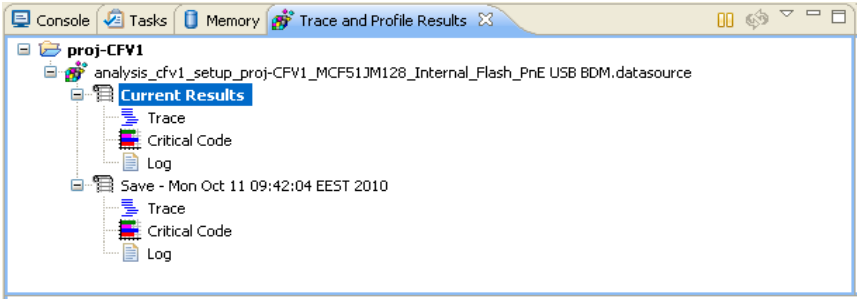


Figure 5.8 ColdFire V1 Trace and Profile Results View — Save Current Results



To delete a data file, right-click on the data file and select **Delete Results**.

6

Setting Tracepoints (HCS08)

Tracepoint is a point in the target program where start or stop triggers are set at a line of the source code or the assembly code or on the memory address. The start and stop tracepoints are triggers for enabling and disabling the trace output.

The advantage of setting start and stop tracepoints is that the trace data can be captured from the specific part of the program. This solves the problem of tracing a large application because a full trace is sometimes extremely difficult to follow. Tracepoints reduce intrusiveness and help collecting the trace data closer to the point of interest.

The trace data displays the trace result based on the tracepoints. This chapter explains how to set start and stop tracepoints and how to enable and disable a tracepoint.

The tracepoints for the HCS08 target can be set on:

- addresses and source files

  The tracepoints on addresses and source files can be set in the editor area and the **Disassembly** view. The source line tracepoint is set in the editor area, and the address tracepoint is set in the **Disassembly** view.

- data and memory

  The tracepoints on data and memory can be set from the **Variables** and **Memory** views.

This chapter consists of the following topics:

Conditions for Starting/Stopping Triggers

In the HCS08 target, A and B are two address comparators referred as triggers that make one big trigger. The trace collection starts or ends depending on the *From* or *Until* trigger selected along with a combination of actions involving A and B.

Both the triggers, A and B, perform the same action depending on the option selected from the **Trace Start/Stop Conditions** group. If you select the **Collect Trace From Trigger** option, both A and B are used for starting the trace collection. If you select the **Collect Trace Until Trigger** option, both A and B are used for stopping the trace collection.

If *From* trigger is selected, the application activates the trace when conditions A and B are met and starts collecting the trace data.

- In the **Automatically** mode, if you set the **Break on FIFO Full** option, the application runs until the trace buffer fills and then stops automatically. If you do not set the **Break on FIFO Full** option, the application keeps running until you suspend it manually, but does not collect trace anymore. So either way, the trace is collected only till the trace buffer gets filled for the first time.
- In the **Continuously** mode, if you do not check the **Keep Last Buffer Before Trigger** checkbox, the application runs until the trace buffer fills. While the application is being debugged, it stops temporarily in background, resumes execution, and then collects a new buffer of trace without using triggers anymore. This new buffer of trace is collected till you manually suspend the application.

  If you set the **Keep Last Buffer Before Trigger** option, the trace is collected and trace buffer is overwritten until the trigger is hit, that is until trigger conditions are met. The application stops temporarily, resumes, and collects more trace without using triggers anymore till you manually suspend the application.

NOTE In the **Continuously** mode, the application always stops at the address where you have suspended the application manually. The application does not stop on the trigger address. The **Keep Last Buffer Before Trigger** option only control the data that is traced around the trigger. After the necessary trigger conditions are met, the trace data is collected normally till you suspend the application manually.

- In the **Collect Data Trace** mode, the data involved in a read and/or write access to the addresses specified by triggers, A and B, such as the address of a particular control register or program variable, is captured. If the **Break on FIFO Full** option is set, the application stops automatically, else the application stops when you suspend it manually.

If *Until* trigger is selected, the application starts collecting trace and stops when conditions for A and B are met.

- In the **Automatically** mode, when you set the **Break on Trigger Hit** option, the trace gets collected until the trigger is hit, that is until trigger conditions are met. The trace buffer is overwritten during trace collection; therefore, when conditions for A and B are met, only last part of the buffer can be read. Whether the application stops automatically or manually, the trace data that is collected is only the last part of the buffer before the trigger.

NOTE In the **Continuously** and **Automatically** modes, if the **Instruction Execute** option is selected, triggers are set on the program instruction execution. If the **Memory Access** option is selected, the triggers are set on memory locations and variables.

For the following trigger modes, trace starts from the triggered address rather than from the first address in the trace buffer.

- Instruction at Address A is Executed
- Instruction at Address A or at Address B is Executed
- Instruction at Address A, Then Instruction at Address B are Executed
- Instruction at Address A is Executed, and Value on Data Bus Match
- Instruction at Address A is Executed, and Value on Data Bus Mismatch

For the following trigger modes, the trigger address cannot be determined. Therefore, trace starts from the first entry in the trace buffer.

- Instruction Inside Range from Address A to Address B is Executed
- Instruction Outside Range from Address A to Address B is Executed

NOTE To know more about trigger modes, see Trace and Profile Options for HCS08.

Trace Modes

The triggers can be set in the following trace modes:

- Collect Program Trace
  - Continuously — Setting Triggers in Continuously Mode

    This topic covers the following trigger types: *Instruction at Address A, then Instruction at Address B are executed*, *Instruction at Address A or Address B is executed*, *Instruction Inside Range from Address A to Address B is Executed*, *Instruction at Address A is Executed, and Data Match/Mismatch on Data Bus*.
  - Automatically — Setting Triggers in Automatically Mode

    This topic covers the following trigger types, *Instruction at Address A is executed* and *Instruction Outside Range from Address A to Address B is Executed*.
- Collect Data Trace — Setting Triggers in Collect Data Trace Mode

  This topic covers the following trigger types: *Capture Read/Write Values at Address B* and *Capture Read/Write Values at Address B, After Access at Address A*.
- Profile-Only — Setting Triggers in Profile-Only Mode

- Expert — Setting Triggers in Expert Mode

Setting Triggers in Continuously Mode

The **Continuously** mode collects trace continuously till you suspend the target application. This topic explains how to set *From* trigger in the **Continuously** mode in the editor area for the following trigger types:

- Instruction at Address A, Then Instruction at Address B are Executed
- Instruction at Address A or Address B is Executed
- Instruction Inside Range from Address A to Address B is Executed
- Instruction at Address A is Executed, and Value on Data Bus Match
- Instruction at Address A is Executed, and Value on Data Bus Mismatch

This topic also explains how to set Memory Access Triggers in the **Continuously** mode.

Before setting the triggers, see Collecting Data chapter for information about how to collect the trace and profiling data.

Instruction at Address A, Then Instruction at Address B are Executed

To set the tracepoints in the editor area for the HCS08 target:

1. In the **CodeWarrior Projects** view, expand the **Sources** folder of your project.
2. Double-click on the source file, for example, `main.c` to display its contents in the editor area. Replace the source code in the `main.c` file with the source code shown in Listing 6.1.

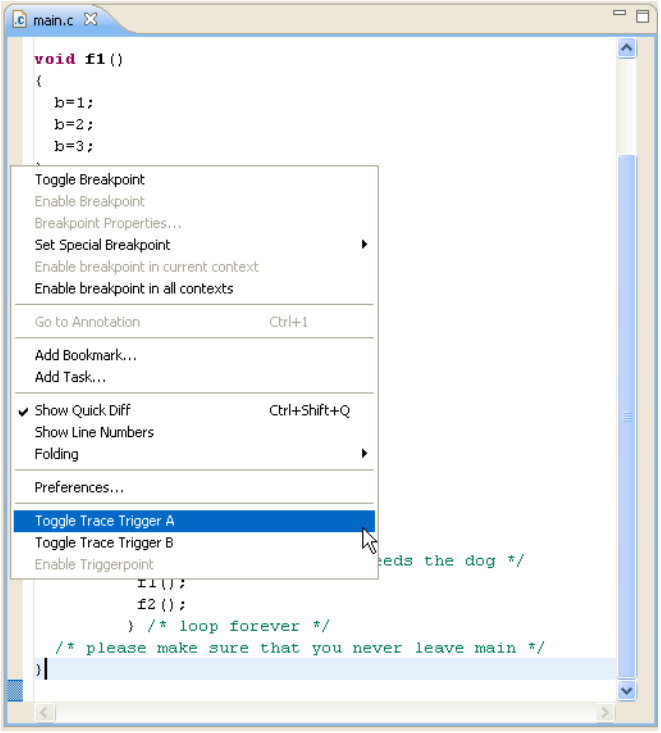
Listing 6.1 Source code 1

```
#include <hidef.h> /* for EnableInterrupts macro */
#include "derivative.h" /* include peripheral declarations */

volatile int a, b;
void f() {}

void f1()
{
  b=1;
  b=2;
  b=3;
}

void f2()
```

```
{
  a=1;
  a=2;
  a=3;
}

void main(void) {
  EnableInterrupts;
  /* include your code here */

f();
f();
f();

for(;;) {
          __RESET_WATCHDOG();/* feeds the dog */
          f1();
          f2();
        } /* loop forever */
  /* please make sure that you never leave main */
}
```

3. Save and build the project.
4. Open the **Debug Configurations** dialog box, and select your project in the tree structure.
5. Click the **Trace and Profile** tab, and check the **Enable Trace and Profile** checkbox.
6. Click **Apply** and close the **Debug Configurations** dialog box.
7. In the editor area, right-click on the marker bar corresponding to the statement, `f1();`.
8. Select **Trace Triggers > Toggle Trace Trigger A** from the context menu (see Figure 6.1). The same option is also used to remove trigger A from the marker bar.

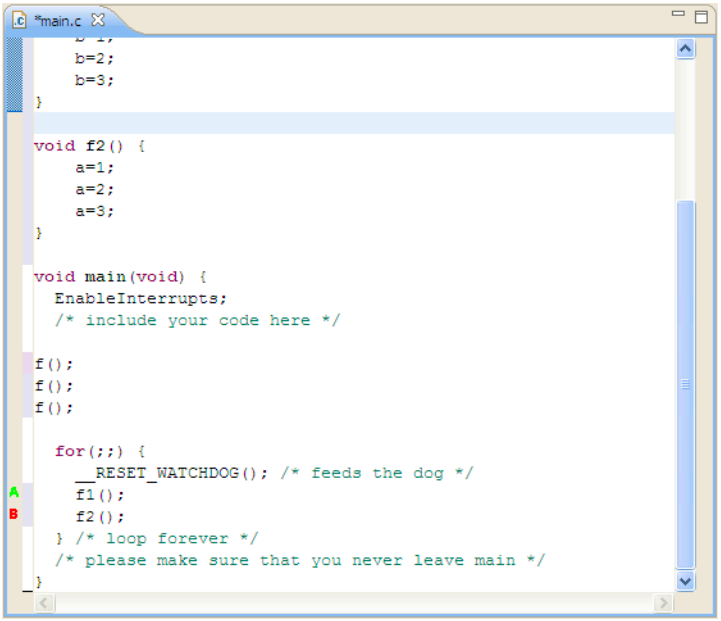
Figure 6.1 Setting Trigger A



9. Right-click on the marker bar corresponding to the statement, `f2();`.
10. Select the **Trace Triggers > Toggle Trace Trigger B** option from the context menu. The same option is also used to remove trigger B from the marker bar.

NOTE It is recommended to set both triggers in the same function so that the trace data that is collected is meaningful.

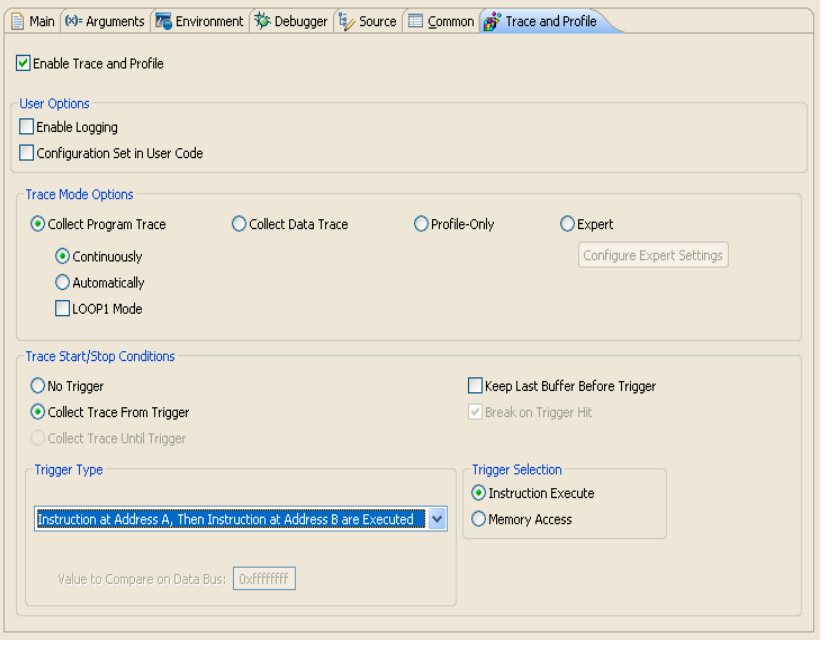
Figure 6.2 Trigger A and Trigger B Set in the Editor Area



NOTE Do not set tracepoints on the statements containing only comments, brackets, and variable declaration with no value. If tracepoints are set on invalid lines, they are automatically disabled when the application is debugged.

11. Open the **Debug Configurations** dialog box, and select your project in the tree structure.
12. Click the **Trace and Profile** tab.
13. Select the **Collect Program Trace** option in the **Trace Mode Options** group.
14. Select the **Continuously** option.
15. Ensure that the **Instruction Execute** option is selected in the **Trigger Selection** group.
16. Select the **Collect Trace From Trigger** option in the **Trace Start/Stop Conditions** group.
17. Clear the **Keep Last Buffer Before Trigger** checkbox.
18. Select the **Instruction at Address A, Then Instruction at Address B are Executed** option from the **Trigger Type** drop-down list (see Figure 6.3).

Figure 6.3 Setting Trigger Conditions



19. Click **Apply** to save the settings.
20. Click **Debug** to debug the application.
21. Collect the trace data following the steps explained in the topic, Collecting Data.
22. Open the **Trace Data** viewer following the steps explained in the topic, Viewing Data for HCS08 Target to view the collected data.

Figure 6.4 shows the data files that are generated by the application in which the data has been collected after setting the tracepoints in the source code.

In Figure 6.4, the **Function** field of the trace data shows that the `main()` function calls the `f2()` function. Because you selected the **Instruction at Address A, Then Instruction at Address B are Executed** mode, both A and B trigger trace. Also, trace starts collecting from trigger B after trigger A has occurred.

Figure 6.4 Trace Data After Setting *From* Trigger in Continuously Mode

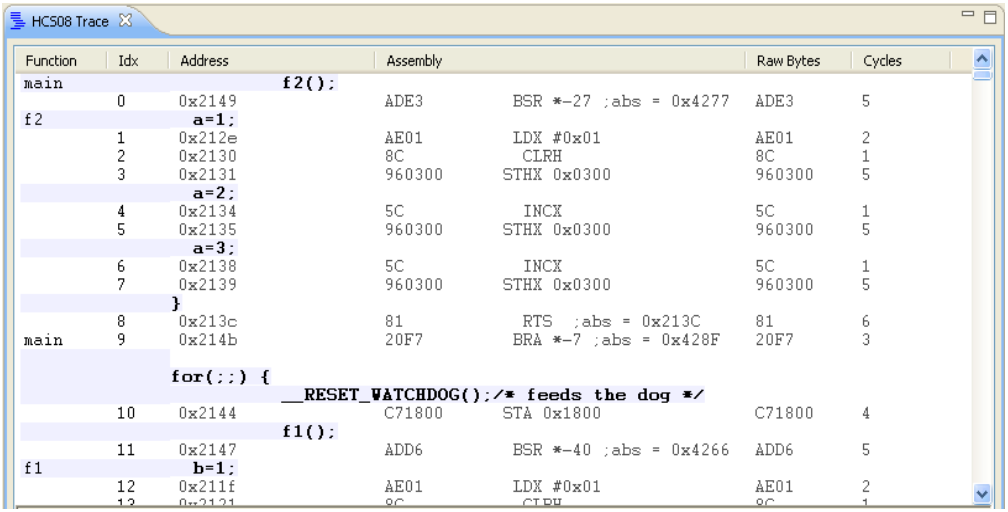
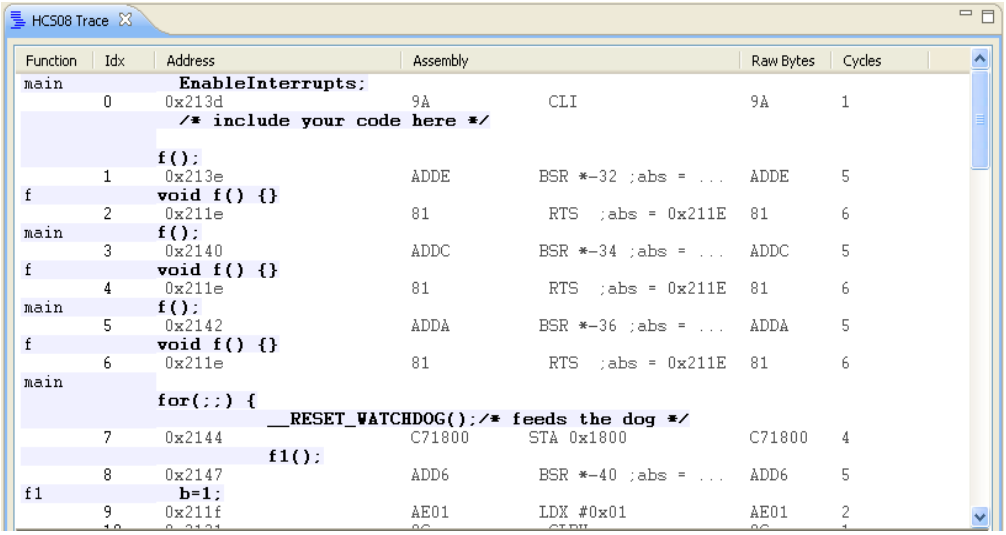


Figure 6.5 shows the data file generated by the application in which the data has been collected before setting the tracepoints in the source code. In this data file, the **Function** field shows that the `main()` function is called and it further calls the `f()`, `f1()`, and `f2()` functions. The `f1()` and `f2()` functions are called in a loop.

Note that the data file generated with tracepoints (Figure 6.4) is different from the data file generated without the tracepoints (Figure 6.5).

NOTE The trace data only contains destinations that cannot be obtained from disassembly. For example, return addresses, which are kept on stack, or conditional destinations, such as jump address and next address.

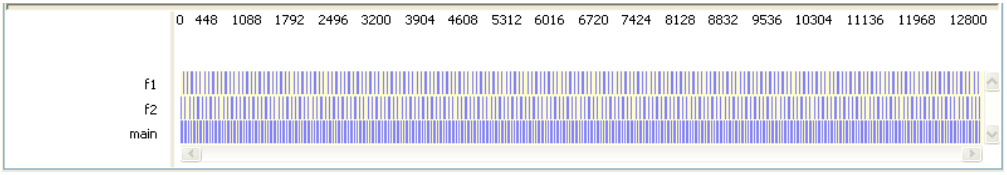
Figure 6.5 Trace Data Before Setting Tracepoints



NOTE Hover mouse pointer over a tracepoint icon in the marker bar to view the attributes of the tracepoint on the corresponding line of code.

The graph in Figure 6.6 shows the timeline of the trace data which is collected after setting the tracepoints. In this graph, you can see that the `main()` function calls the `f1()` and `f2()` functions and not the `f()` function. To have a clearer view of the graph, you can zoom-in or zoom-out in the graph by scrolling the mouse wheel up or down.

Figure 6.6 Graph Displaying Timeline of Trace Data



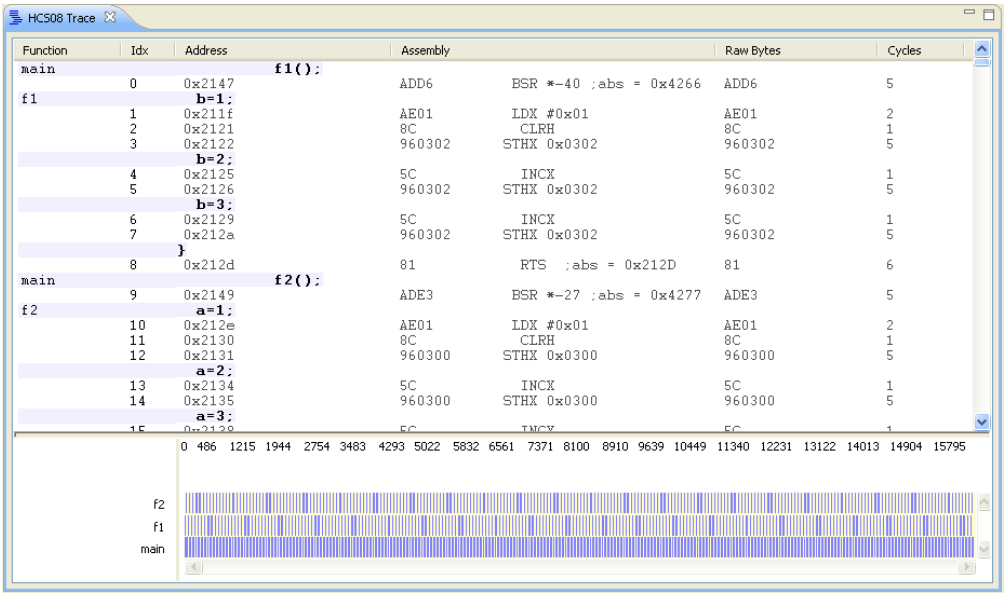
Similarly, you can collect the trace data with the **Keep Last Buffer Before Trigger** checkbox checked in the **Continuously** mode. The trace collection remains same except that the trace buffer is overwritten until the triggers are hit and only the last part of the buffer trace is visible in the **Trace Data** viewer.

Instruction at Address A or Address B is Executed

You can set triggers A and B and collect trace using the *Instruction at Address A or Address B is Executed* trigger type by following the steps explained in the topic, Instruction at Address A, Then Instruction at Address B are Executed. The only difference is that you need to select the **Instruction at Address A or Address B is Executed** option from the **Trigger Type** drop-down list.

Figure 6.7 shows the trace data that is collected after setting this trigger condition. The collection of the trace data starts from the address corresponding to trigger A, that is the `f1()` function, which occurs first in execution.

Figure 6.7 Trace Data — Instruction at Address A or Address B is Executed



Instruction Inside Range from Address A to Address B is Executed

The **Instruction Inside Range from Address A to Address B is Executed** trigger type is used to trigger on a program instruction execution inside the range, Address A - Address B, where Address A is the address at which trigger A is set and Address B is the address at which trigger B is set.

To collect trace using the **Instruction Inside Range from Address A to Address B is Executed** trigger type:

1. In the **CodeWarrior Projects** view, select the **Sources** folder of your project.
2. Double-click on the source file, for example, `main.c` to display its contents in the editor area. Replace the source code in the `main.c` file with the source code shown in Listing 6.2.

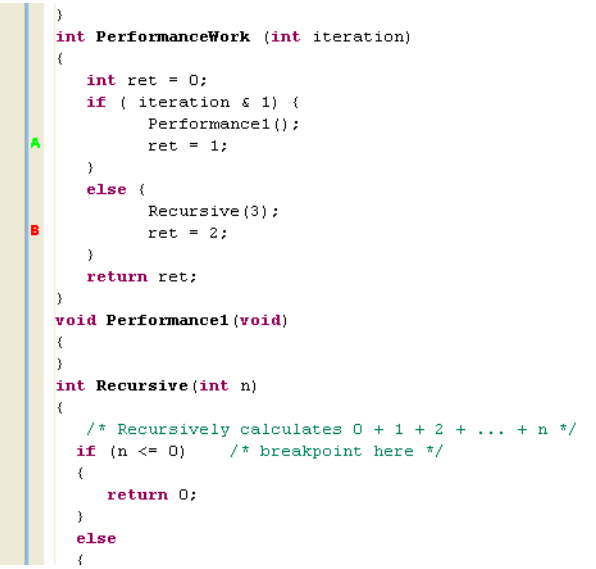
Listing 6.2 Source code 2

```
#include <hidef.h> /* for EnableInterrupts macro */
#include "derivative.h" /* include peripheral declarations */

#define MAX_IT 2
#define SIMPLE 1

typedef int(*FUNC_TYPE)(int);
void entry();
void InterruptTest();
void ContextSwitch(FUNC_TYPE, int);
int PerformanceWork (int);
void Performance1(void);
int Recursive(int);

void Launch(FUNC_TYPE f, int arg)
{
    f(arg);
}

void InterruptTest()
{}
void entry()
{
  volatile int iteration =0;
  InterruptTest();
  for (iteration =0; iteration < MAX_IT; /*iteration^=1*/ iteration++)
  { Launch(PerformanceWork, iteration);}
}
int PerformanceWork (int iteration)
{
   int ret = 0;
   if ( iteration & 1) {
         Performance1();
         ret = 1;
   }
   else {
         Recursive(3);
         ret = 2;
```

```
   }
   return ret;
}
void Performance1(void)
{
}
int Recursive(int n)
{
   /* Recursively calculates 0 + 1 + 2 + ... + n */
  if (n <= 0)    /* breakpoint here */
  {
     return 0;
  }
  else
  {
     return (n + Recursive(n-1));
  }
}
void main(void) {
  EnableInterrupts; /* enable interrupts */
  /* include your code here */

  Performance1();
  Recursive(2);
  Performance1();

  for(;;) {
   entry();
   __RESET_WATCHDOG(); /* feeds the dog */
  } /* loop forever */
}
```

3. Save and build the project.
4. Open the **Debug Configurations** dialog box, and select your project in the tree structure.
5. Click the **Trace and Profile** tab, and check the **Enable Trace and Profile** checkbox.
6. Click **Apply** and close the **Debug Configurations** dialog box.
7. Set trigger A at `ret = 1;` and trigger B at `ret = 2;` in the `PerformanceWork()` function, as shown in Figure 6.8.

Figure 6.8 Setting Trigger A and Trigger B in Source Code — Instruction Inside Range

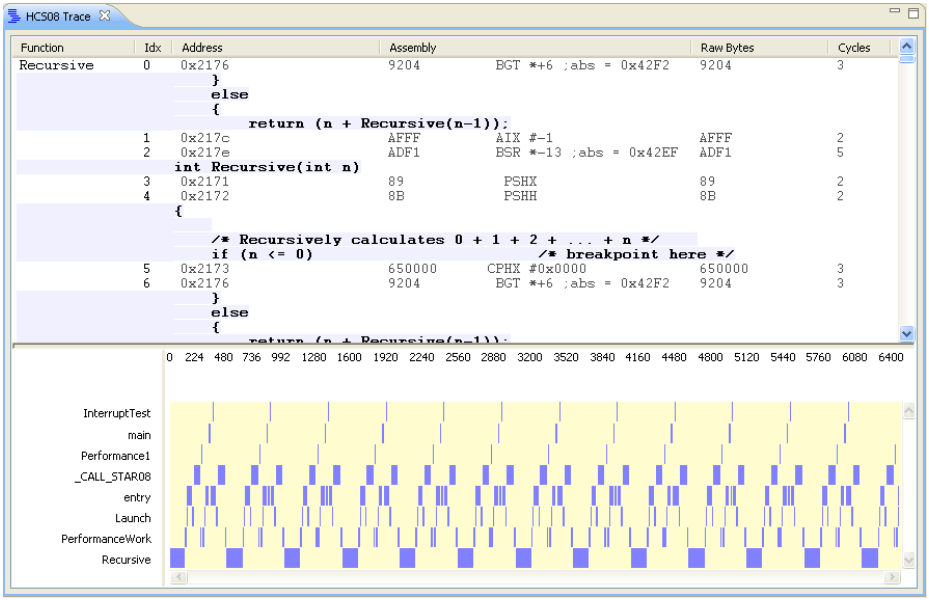


8. Open the **Debug Configurations** dialog box, and select your project in the tree structure.
9. Click the **Trace and Profile** tab.
10. Select the **Collect Program Trace** option in the **Trace Mode Options** group.
11. Select the **Continuously** option.
12. Select the **Collect Trace From Trigger** option in the **Trace Start/Stop Conditions** group.
13. Clear the **Keep Last Buffer Before Trigger** checkbox.
14. Ensure that the **Instruction Execute** option is selected in the **Trigger Selection** group.
15. Select the **Instruction Inside Range from Address A to Address B is Executed** option from the **Trigger Type** drop-down list.
16. Click **Apply** to save the settings.
17. Click **Debug** to debug the application.
18. Collect the trace data following the steps explained in the topic, Collecting Data.
19. Open the **Trace Data** viewer following the steps explained in the topic, Viewing Data for HCS08 Target to view the collected data.

    Figure 6.9 shows the data files that are generated by the application in which trace starts from `Recursive(3)` (in `PerformanceWork()`), which is the first address

that the application finds between trigger A (`ret = 1;`) and trigger B (`ret = 2;`) address range after hitting trigger A.

Figure 6.9 Trace Data — Instruction Inside Range



NOTE In the **Instruction Inside Range from Address A to Address B is Executed** trigger type, a time delay of one to four instructions, depending on the processor type, might occur when tracing starts.

Instruction at Address A is Executed, and Value on Data Bus Match

The **Instruction at Address A is Executed, and Value on Data Bus Match** trigger type is used to trigger on a program instruction execution at trigger A address when the opcode of that instruction matches a specific byte value. This trigger type is useful in detecting the self-modifying instructions or code in RAM so that you can trace what exactly executes when the instructions are modified.

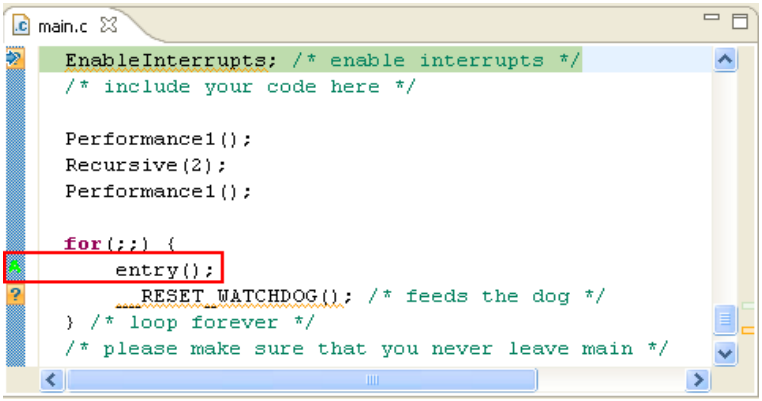
For example, if in a source code, a random pointer is changing the instruction executed at an address to a specific value (opcode), you can set a trigger at this instruction. Also, you need to specify that particular opcode value while configuring the debug launcher. Now,

when the application executes and reaches the instruction where trigger is set, the specified opcode value matches with the opcode of the instruction and the trigger is fired.

To set the **Instruction at Address A is Executed, and Value on Data Bus Match** trigger type:

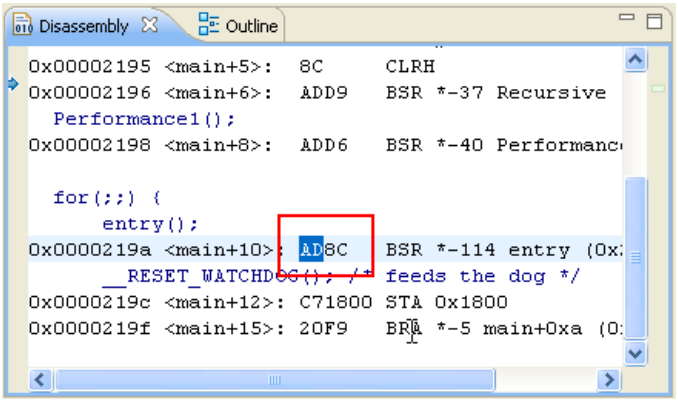
1. In the **CodeWarrior Projects** view, select the **Sources** folder of your project.
2. Double-click on the source file, for example, `main.c` to display its contents in the editor area. Replace the source code in the `main.c` file with the source code shown in Listing 6.2.
3. Save and build the project.
4. Open the **Debug Configurations** dialog box, and select your project in the tree structure.
5. Click the **Trace and Profile** tab, and check the **Enable Trace and Profile** checkbox.
6. Select the **Collect Program Trace** option in the **Trace Mode Options** group.
7. Select the **Continuously** option.
8. Ensure that the **Instruction Execute** option is selected in the **Trigger Selection** group.
9. Select the **Collect Trace From Trigger** option in the **Trace Start/Stop Conditions** group.
10. Check the **Keep Last Buffer Before Trigger** checkbox if not checked.
11. Select the **Instruction at Address A is Executed, and Value on Data Bus Match** option from the **Trigger Type** drop-down list.
12. Click **Apply** to save the settings.
13. Click **Debug** to debug the application and collect the trace data.
14. After the application is debugged, set trigger A at `entry();` in the `main()` function, as shown in Figure 6.10.

Figure 6.10 Setting Trigger A in the Source Code



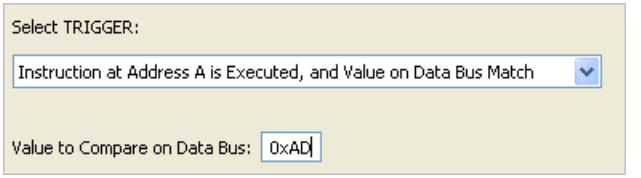
15. In the **Disassembly** view, copy first two hexadecimal digits at call to `entry()` from `main()`, as shown in Figure 6.11.

Figure 6.11 Copying Hexadecimal Letters in Disassembly View



16. Open the **Debug Configurations** dialog box, and click the **Trace and Profile** tab.
17. In the **Value to Compare on Data Bus** text box, keep `0x` and paste the two hexadecimal digits, as shown in Figure 6.12.

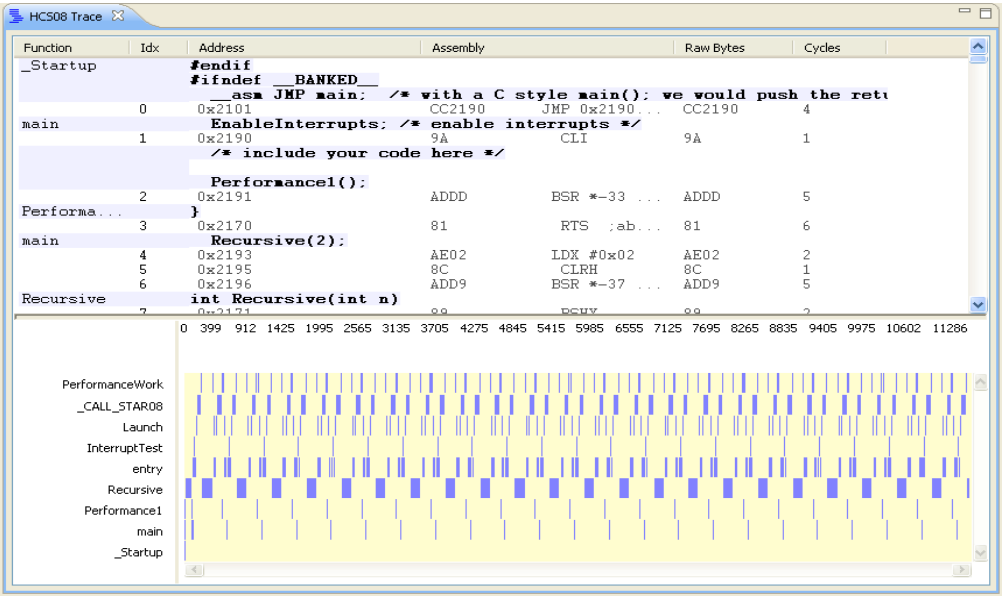
Figure 6.12 Value to Compare on Data Bus Text Box



18. Click **Apply** and close the **Debug Configurations** dialog box.
19. Click the **Terminate** icon in the **Debug** perspective to terminate the application.
20. Debug the application again.
21. Click **Resume** to resume the application and after a while click **Suspend** to suspend the application.
22. Open the **Trace Data** viewer following the steps explained in the topic, Viewing Data for HCS08 Target to view the collected data.

Figure 6.13 shows the trace data collected after setting the **Instruction at Address A is Executed, and Value on Data Bus Match** trigger type. When the opcode specified in the **Value to Compare on Data Bus** text box matches with the opcode of the instruction where trigger A is set, trigger is fired, and the collection of the trace data starts from there.

Figure 6.13 Trace Data — Instruction at Address A is Executed, and Value on Data Bus Match



Instruction at Address A is Executed, and Value on Data Bus Mismatch

Like **Instruction at Address A is Executed, and Value on Data Bus Match**, the **Instruction at Address A is Executed, and Value on Data Bus Mismatch** trigger type is used in detecting self-modifying code in the memory. For example, one of the instruction in your source code writes a new opcode at its address, *ADDR*, and you want to trace what executes after the original opcode at *ADDR* has been replaced. You can set trigger A at *ADDR,* select the **Instruction at Address A is Executed, and Value on Data Bus Mismatch** option, and specify the original opcode value in the debug configuration. When the application will execute the modified opcode starting with *ADDR*, a mismatch will occur, and the trigger will be fired.

Memory Access Triggers

The memory access triggers allow memory access to both variables and instructions. A memory access trigger if set on an instruction fires when the instruction is fetched from the memory. A memory access trigger if set on a variable fires when the variable is fetched from the memory or when the variable is written back to the memory. When a

memory access trigger is set on a variable, trace collection starts from the first access to that variable address.

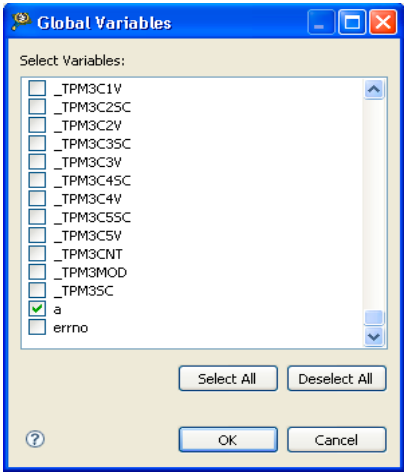
This topic explains how to collect trace using the **Memory at Address A is Accessed** trigger type.

To collect trace using **Memory at Address A is Accessed**:

1. In the **CodeWarrior Projects** view, expand the **Sources** folder of your project.
2. Double-click on the source file, for example, `main.c` to display its contents in the editor area. Replace the source code in the `main.c` file with the source code shown in Listing 6.2.
3. Press the **Enter** key after the statement, `void ContextSwitch(FUNC_TYPE, int);` in the source code.
4. Type the statement, `volatile int a;` at the cursor position.
5. Press the **Enter** key after the statement, `Performance1();` in the `main ()` function of the source code.
6. Type the statement, `a = 1;`.
7. Save and build the project.
8. Open the **Debug Configurations** dialog box, and select your project in the tree structure.
9. Click the **Trace and Profile** tab, and check the **Enable Trace and Profile** checkbox.
10. Select the **Collect Program Trace** option in the **Trace Mode Options** group.
11. Select the **Continuously** option.
12. Select the **Collect Trace From Trigger** option in the **Trace Start/Stop Conditions** group.
13. Clear the **Keep Last Buffer Before Trigger** checkbox.
14. Select the **Memory Access** option in the **Trigger Selection** group.
15. Select the **Memory at Address A is Accessed** option from the **Trigger Type** drop-down list.
16. Click **Apply** to save the settings.
17. Click **Debug** to debug the application.
18. After the application is debugged, right-click in the **Variables** view, and select the **Add Global Variables** option from the context menu.

    The **Global Variables** dialog box appears.
19. Scroll down and select the checkbox corresponding to the variable, `a`, as shown in Figure 6.14.

Figure 6.14 Adding Variable in Global Variables Dialog Box

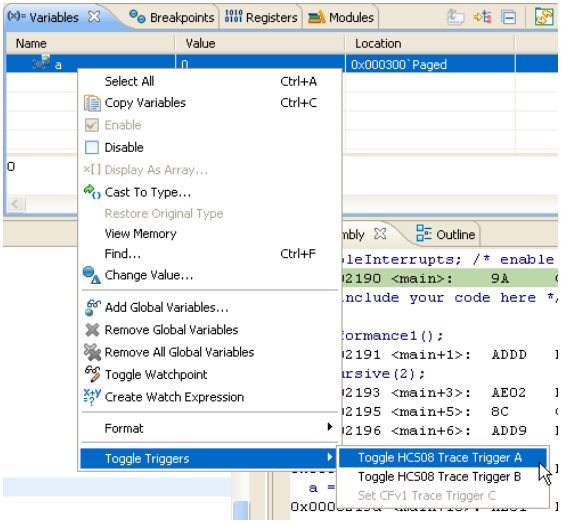


20. Click **OK**.

    The **Global Variables** dialog box closes and the variable, `a` is added in the **Variables** view.

21. Right-click on `a` and select **Toggle Triggers > Toggle HCS08 Trace Trigger A** from the context menu.

Figure 6.15 Setting Trigger in Variables View



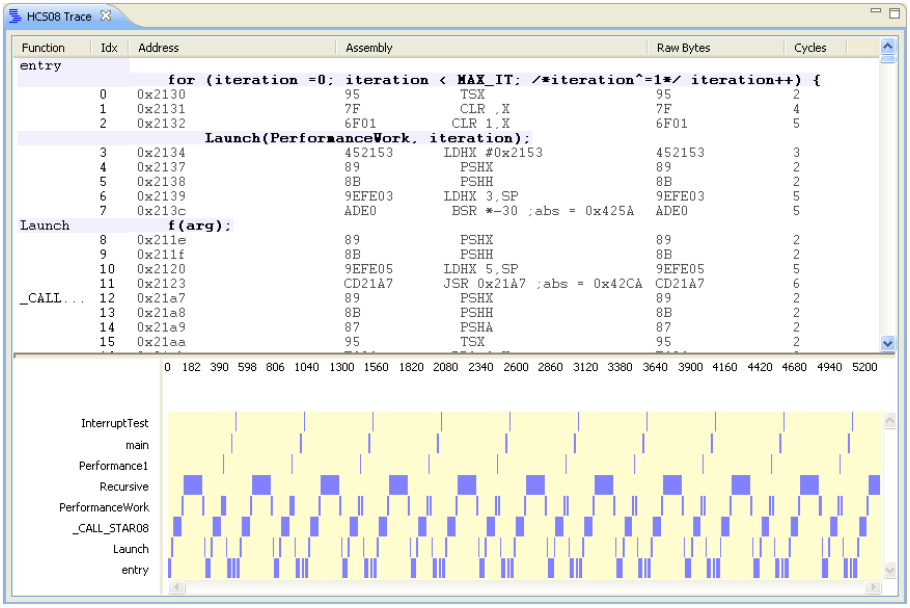
22. Click **Resume**.

    The tracing starts when the variable, `a` is accessed.

23. Click **Suspend** after a while.

24. Open the **Trace Data** viewer following the steps explained in the topic, Viewing Data for HCS08 Target to view the trace results. Figure 6.16 shows the data files generated by the application after setting memory access trigger A. The trace data starts collecting from trigger A, that is `a=1;` onwards.

Figure 6.16 Trace Data After Setting Memory Access Trigger



NOTE Similarly, you can set memory access triggers in the **Automatically** mode.

Setting Triggers in Automatically Mode

The **Automatically** mode collects trace till the trace buffer gets full. This topic explains how to collect trace using the **Instruction at Address A is Executed** trigger type in:

- Automatically mode From the Disassembly View — *From* trace condition
- Automatically mode On Data and Memory — *Until* trace condition
- LOOP1 Mode

The topic also explains how to collect trace using the Instruction Outside Range from Address A to Address B is Executed trigger type with *From* trace condition selected.

From the Disassembly View

To set a trigger in the **Disassembly** view:

1. In the **CodeWarrior Projects** view, expand the **Sources** folder of your project.
2. Double-click on the source file, for example, `main.c` to display its contents in the editor area. Replace the source code in the `main.c` file with the source code shown in Listing 6.3.

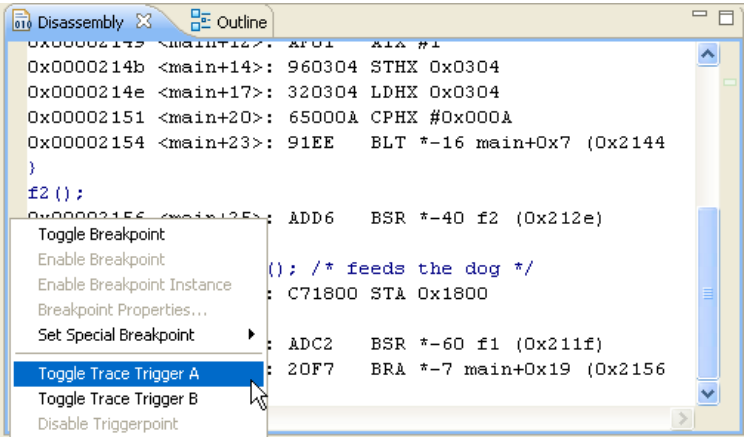
Listing 6.3 Source code 3

```
#include <hidef.h> /* for EnableInterrupts macro */
#include "derivative.h" /* include peripheral declarations */

volatile int a, b, i;

void f(){}

void f1() {
  b=1;
  b=2;
  b=3;
}

void f2() {
  a=1;
  a=2;
  a=3;
}

void main(void) {
  EnableInterrupts;
  /* include your code here */
for(i=1;i<10;i++)
{
  f();
}
f2();
  for(;;) {
    __RESET_WATCHDOG(); /* feeds the dog */
    f1();
    f2();
  } /* loop forever */
```

```
}
```

3. Save and build the project.
4. Open the **Debug Configurations** dialog box, and select your project in the tree structure.
5. Click the **Trace and Profile** tab, and check the **Enable Trace and Profile** checkbox.
6. Select the **Collect Program Trace** option in the **Trace Mode Options** group.
7. Select the **Automatically** option.
8. Select the **Collect Trace From Trigger** in the **Trace Start/Stop Conditions** group.
9. Check the **Break on FIFO Full** checkbox.
10. Select the **Instruction at Address A is Executed** option from the **Trigger Type** drop-down list.
11. Click **Apply** to save the settings.
12. Click **Debug** to debug the application.
13. In the **Disassembly** view, right-click on the marker bar corresponding to the address line of the function, `f2()` (see Figure 6.17).

Figure 6.17 Disassembly View



14. Select **Trace Triggers > Toggle Trace Trigger A** from the context menu.

    The trigger icon, A appears in green color on the marker bar, in the editor area and the **Disassembly** view.
15. Click **Resume** to continue collecting trace.

NOTE Because you selected the **Break on FIFO Full** check box in the Trace and Profile tab, the application will stop automatically after the trigger hit. You do not need to stop it manually by clicking **Suspend**.

16. Open the **Trace Data** viewer following the steps explained in the topic, Viewing Data for HCS08 Target to view the collected data.

Figure 6.18 shows the data files and the timeline graph that is generated by the application in which the data has been collected after setting trigger A. You can see Figure 6.5 to view the data files that are generated before setting tracepoints. The **Trace Data** viewer in Figure 6.18 shows that trace starts collecting from where you set the trigger, and when the trace buffer gets full, the application stops collecting trace. That is, application starts collecting trace from `f2()` as highlighted in the figure, and continues till buffer trace gets full.

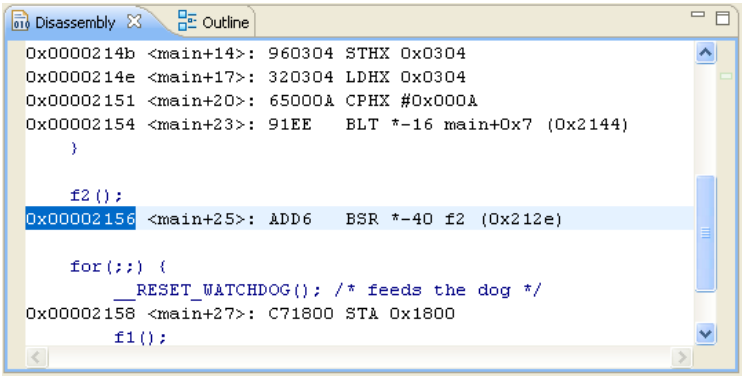
Figure 6.18 Trace Data After Setting *From* Trigger in Automatically Mode

On Data and Memory

To set triggers on data and memory:

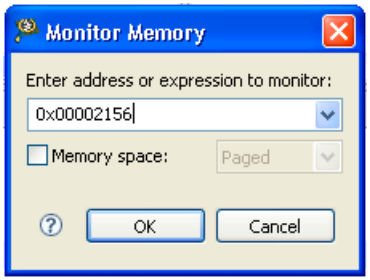
1. In the **CodeWarrior Projects** view, expand the **Sources** folder of your project.
2. Double-click on the source file, for example, `main.c` to display its contents in the editor area. Replace the source code in the `main.c` file with the source code shown in Listing 6.3.
3. Open the **Debug Configurations** dialog box, and select your project in the tree structure.
4. Click the **Trace and Profile** tab, and check the **Enable Trace and Profile** checkbox.
5. Select the **Automatically** option in the **Trace Mode Options** group.
6. Select **Collect Trace Until Trigger** in the **Trace Start/Stop Conditions** group.
7. Check the **Break on Trigger Hit** checkbox.
8. Select the **Instruction at Address A is Executed** option in the **Trigger Type** drop-down list.
9. Click **Apply** to save the settings.
10. Click **Debug** to debug the application.
11. In the **Disassembly** view, select the address corresponding to the call to `f2()` (see Figure 6.19)

Figure 6.19 Disassembly View — Selecting Function Address



12. Copy the memory address and select **Window > Show View > Other > Debug > Memory** to open the **Memory** view.
13. Click  in the **Memory** view to open the **Monitor Memory** dialog box.
14. Paste the memory address in the **Enter address or expression to monitor** text box (see Figure 6.20).

Figure 6.20 Monitor Memory Dialog Box

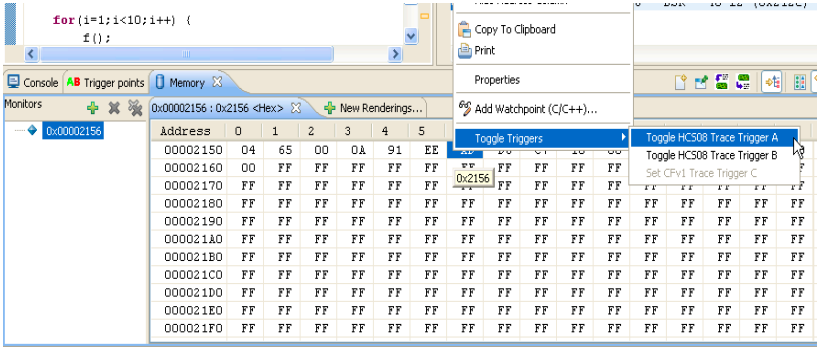


15. Click **OK**.

    The memory address appears in the **Memory** view (see Figure 6.21).

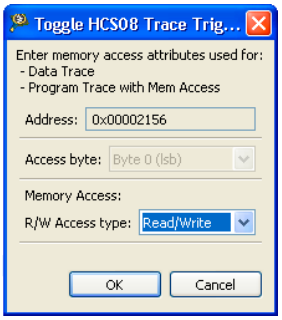
16. Right-click on the cell containing the memory address and select **Toggle Triggers > Toggle HCS08 Trace Trigger A** option from the context menu (see Figure 6.21).

Figure 6.21 Memory View



The **Toggle HCS08 Trace Trigger** dialog box appears.

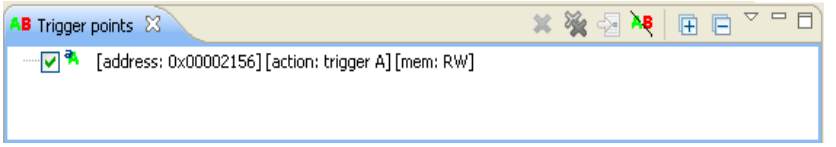
Figure 6.22 Toggle HCS08 Trace Trigger Dialog Box



17. Click **OK**.
18. Select **Window > Show View > Other > Analysis > Trigger points** to open the **Trigger points** view.

    The **Trigger points** view displays the trigger that you set on the memory address (see Figure 6.23).

Figure 6.23 Trigger Points View

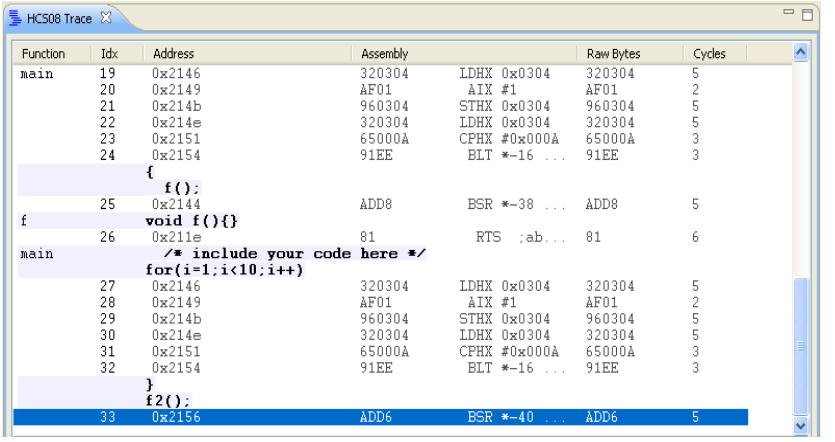


19. Click **Resume**.

    The application stops automatically and trace is collected.
20. Open the **Trace Data** viewer following the steps explained in the topic, Viewing Data for HCS08 Target to view the trace results.

Figure 6.24 shows the data files generated by the application after setting trigger A in the **Automatically** mode on memory.

Figure 6.24 Trace Data After Setting *Until* Trigger in Automatically Mode



Because you set the *Until* trigger at the memory address of `f2()`, the trace data is collected and the trace buffer is overwritten till the trigger is hit. Therefore, only the last part of the trace buffer is collected before the trigger, and the application stops at the memory address of `f2()`.

LOOP1 Mode

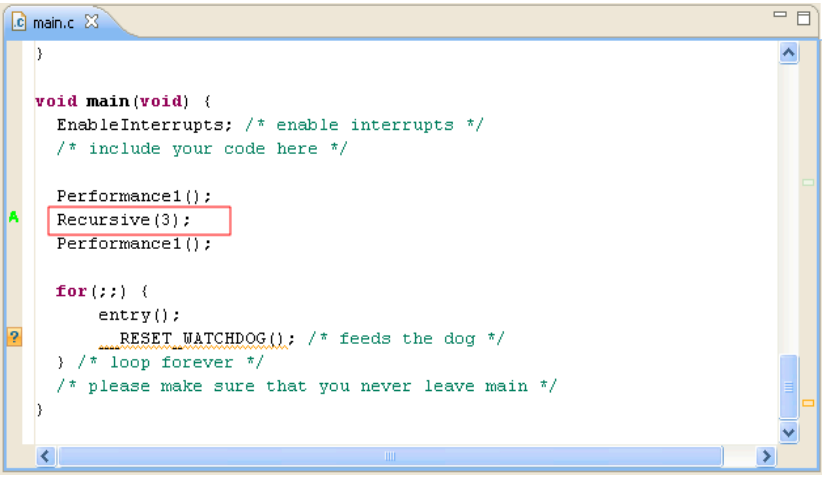
The LOOP1 Mode feature when selected writes a register to allow the hardware to use the C comparator and not store duplicate addresses in trace. In LOOP1 capture mode, the addresses for instructions executed repeatedly, for example, loops with no change of flow instructions and recursive calls, are stored and showed in trace only once.

The hardware uses comparator C available only on variants with DBGv3, which cannot be used for trace, to store the last FIFO address. Only comparators A and B can be used for trace on all variants. However, in LOOP1 capture mode, comparator C is not available for use as a normal hardware breakpoint, and is managed by logic in the DBG module to track the address of the most recent change-of-flow event that was captured into the FIFO buffer.

To set a trigger in Loop1 Mode:

1. In the **CodeWarrior Projects** view, select the **Sources** folder of your project.
2. Double-click on the source file, for example, `main.c` to display its contents in the editor area. Replace the source code in the `main.c` file with the source code shown in Listing 6.5
3. Save and build the project.
4. Open the **Debug Configurations** dialog box, and select your project in the tree structure.
5. Click the **Trace and Profile** tab, and check the **Enable Trace and Profile** checkbox.
6. Click **Apply** to save the settings, and close the **Debug Configurations** dialog box.
7. In the editor area, right-click on the marker bar corresponding to the statement, `Recursive(3);` and select **Trace Triggers > Toggle Trace Trigger A** from the context menu.

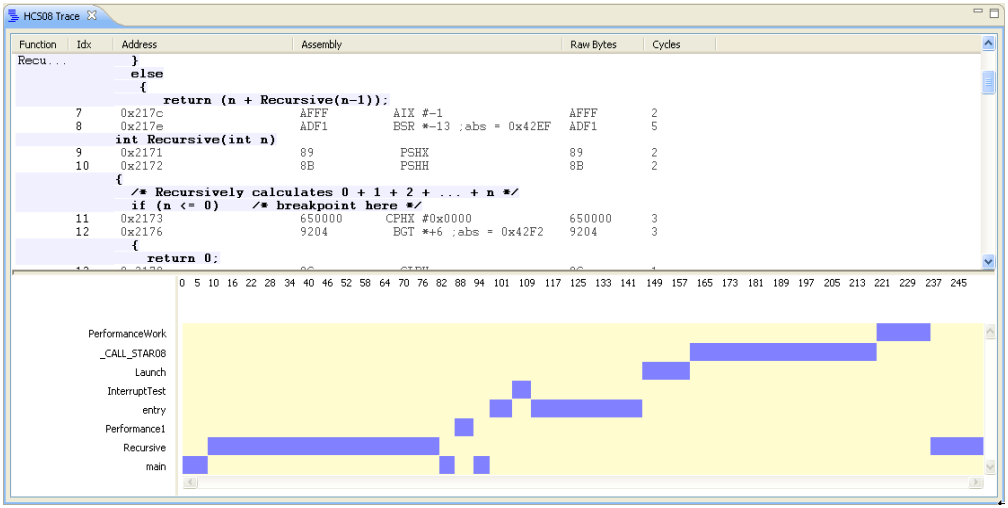
Figure 6.25 Setting Trigger A in LOOP1 Mode



8. Open the **Debug Configurations** dialog box, and click the **Trace and Profile** tab.
9. Select the **Collect Program Trace** option in the **Trace Mode Options** group.
10. Select the **Automatically** option.
11. Select the **LOOP1 Mode** check box.
12. Select **Collect Trace From Trigger** in the **Trace Start/Stop Conditions** group.
13. Check the **Break on FIFO Full** checkbox.
14. Select **Instruction at Address A is Executed** from the **Trigger Type** drop-down list.
15. Click **Apply** to save the settings.
16. Click **Debug** to debug the application.
17. Click **Resume**.

    The application stops automatically and collects data in data file.
18. Open the **Trace Data** viewer following the steps explained in the topic, Viewing Data for HCS08 Target to see the trace results. Figure 6.26 shows the data files generated by the application after setting trigger A in the LOOP1 Mode.

Figure 6.26 Trace Data After Setting Trigger A in LOOP1 Mode — Automatically



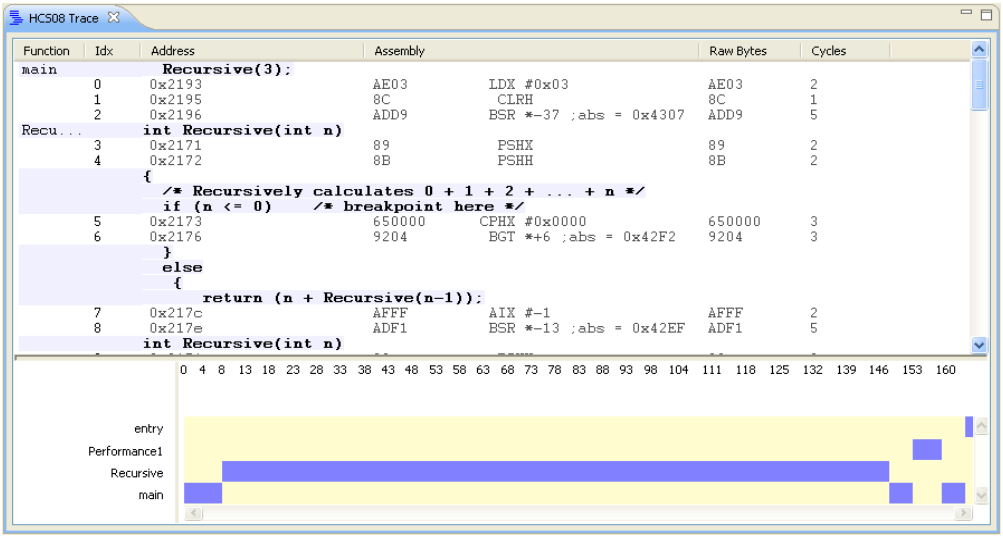
The trace data that is collected contains only two calls to `Recursive(int n)`. This is because the three identical PCs in trace for `if (n <= 0)` is ignored by the hardware and appears only once in LOOP1 mode.

In the normal mode, the hardware would have stored the identical addresses one after another. Figure 6.27 shows the trace data that is collected when LOOP1 Mode option is disabled. Make sure that you clear the **LOOP1 Mode** checkbox in the **Trace and Profile** tab, keep the remaining settings same and then collect trace.

In the normal mode, the trace data contains four calls to `Recursive(int n)` with arguments `3`, `2`, `1`, `0`, and three times branch for `if (n <= 0)`, that is three identical PCs in trace. Fourth address of the conditional assembly instruction for `if (n <= 0)` is not in the trace because that branch is not taken anymore and the last recursive call exits with `return 0`.

With the LOOP1 Mode option enabled, the hardware skips the identical addresses and keeps only one address instead of three; therefore, the trace data contains only two calls to `Recursive(int n)`.

Figure 6.27 Trace Data After Setting Trigger A in Normal Mode — Automatically



Instruction Outside Range from Address A to Address B is Executed

The **Instruction Outside Range from Address A to Address B is Executed** trigger type is used to trigger on a program instruction execution outside the range, Address A - Address B, where Address A is the address at which trigger A is set and Address B is the address at which trigger B is set.

If the triggers A and B set in this trigger type contain a function between them, the trace data will start collecting from the code inside that function because that code is outside the address range of trigger A and trigger B.

To collect trace using the **Instruction Outside Range from Address A to Address B is Executed** trigger type:

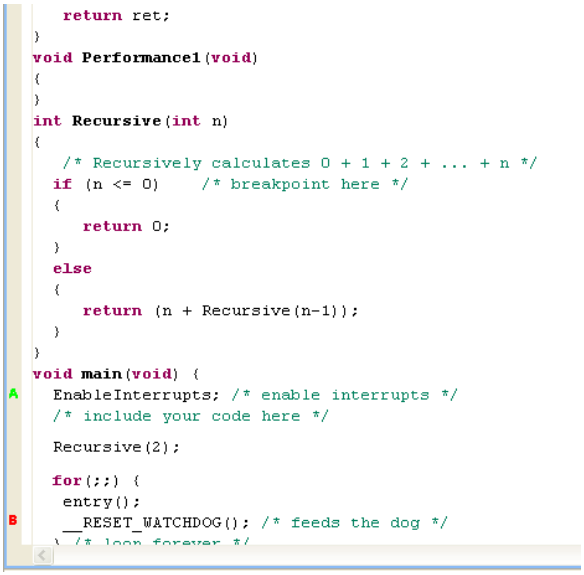
1. In the **CodeWarrior Projects** view, select the **Sources** folder of your project.
2. Double-click on the source file, for example, `main.c` to display its contents in the editor area. Replace the source code in the `main.c` file with the source code shown in Listing 6.2.
3. Remove the entries of `Performance1();` from the `main();` function. The `main()` function now should look as shown in Listing 6.4:

Listing 6.4 main() function of source code 2

```
void main(void) {
  EnableInterrupts; /* enable interrupts */
  /* include your code here */
   Recursive(2);
    for(;;) {
   entry();
   __RESET_WATCHDOG(); /* feeds the dog */
  } /* loop forever */
}
```

4. Save and build the project.
5. Open the **Debug Configurations** dialog box, and select your project in the tree structure.
6. Click the **Trace and Profile** tab, and check the **Enable Trace and Profile** checkbox.
7. Click **Apply** and close the **Debug Configurations** dialog box.
8. Set trigger A at `EnableInterrupts;` and trigger B at `__RESET_WATCHDOG();` in the `main()` function, as shown in Figure 6.28.

Figure 6.28 Setting Trigger A and Trigger B in Source Code — Instruction Outside Range



9. Open the **Debug Configurations** dialog box, and select your project in the tree structure.

10. Click the **Trace and Profile** tab.
11. Select the **Collect Program Trace** option in the **Trace Mode Options** group.
12. Select the **Automatically** option.
13. Select the **Collect Trace From Trigger** option in the **Trace Start/Stop Conditions** group.
14. Check the **Break on FIFO Full** checkbox.
15. Ensure that the **Instruction Execute** option is selected in the **Trigger Selection** group.
16. Select the **Instruction Outside Range from Address A to Address B is Executed** option from the **Trigger Type** drop-down list.
17. Click **Apply** to save the settings.
18. Click **Debug** to debug the application.
19. Click **Resume** to collect the trace data.

    The application stops automatically.
20. Open the **Trace Data** viewer following the steps explained in the topic, Viewing Data for HCS08 Target to view the collected data.

    Figure 6.29 shows the data files that are generated by the application in which trace starts at `Recursive(2)`, which is called from `main()`. The tracing starts at `Recursive(2)` because this function is placed between address range of trigger A and trigger B, therefore the code of this function is outside the address range of the two triggers. The application stops automatically when buffer gets full.

Figure 6.29 Trace Data — Instruction Outside Range

Setting Triggers in Collect Data Trace Mode

In the **Collect Data Trace** mode, triggers are set to capture specific values. This mode consists of the following two trigger types:

- Capture Read/Write Values at Address B
- Capture Read/Write Values at Address B, After Access at Address A

The trigger address is typically not a program code address (program counter), but rather a data/memory address.

Capture Read/Write Values at Address B

This option captures the data involved in a read and/or write access to the address specified by trigger B, such as the address of a particular control register or program variable. To set trigger B in the **Collect Data Trace** mode:

1. In the **CodeWarrior Projects** view, expand the **Sources** folder of your project.
2. Double-click on the source file, for example, `main.c` to display its contents in the editor area. Replace the source code in the `main.c` file with the source code shown in Listing 6.5.

Listing 6.5 Source code 4

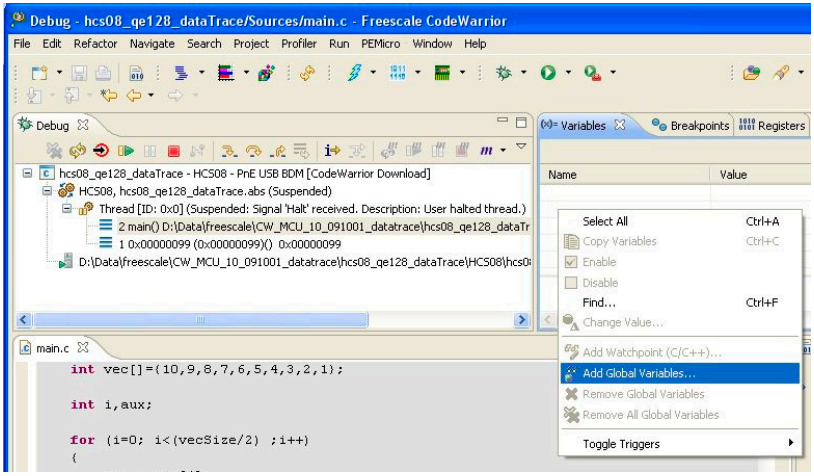
```
#include <hidef.h> /* for EnableInterrupts macro */
#include "derivative.h" /* include peripheral declarations */

#define MAX_IT 2

typedef int(*FUNC_TYPE)(int);
void entry();
void InterruptTest();
void ContextSwitch(FUNC_TYPE, int);

int PerformanceWork (int);
void Performance1(void);
int Recursive(int);

volatile char iteration = 0;

void Launch(FUNC_TYPE f, int arg)
{
   f(arg);
}
void InterruptTest()
{}

void entry()
{
   InterruptTest();
   Launch(PerformanceWork, iteration++);
   Launch(PerformanceWork, iteration++);
   if (iteration >= 254) iteration = 0;
}

int PerformanceWork (int iteration)
{
   int ret = 0;
   if (iteration & 1)
   {
        Performance1();
        ret = 1;
   }
   else
   {
        Recursive(3);
        ret = 2;
   }
   return ret;
}
```

```
void Performance1(void)
{
}
int Recursive(int n)
{
  /* Recursively calculates 0 + 1 + 2 + ... + n */
  if (n <= 0)    /* breakpoint here */
  {
    return 0;
  }
  else
   {
      return (n + Recursive(n-1));
   }
}
void main(void)
{
  EnableInterrupts; /* enable interrupts */

/* include your code here */
  Performance1();
  Recursive(3);
  Performance1();

  for(;;)
  {
    entry();
    __RESET_WATCHDOG(); /* feeds the dog */
  } /* loop forever */
  /* please make sure that you never leave main */
}
```

3. Save and build the project.
4. Open the **Debug Configurations** dialog box, and select your project in the tree structure.
5. Click the **Trace and Profile** tab, and check the **Enable Trace and Profile** checkbox.
6. Select the **Collect Data Trace** option in the **Trace Mode Options** group.
7. Select **Collect Trace From Trigger** in the **Trace Start/Stop Conditions** group.
8. Check the **Break on FIFO Full** checkbox.
9. Select **Capture Read/Write Values at Address B** from the **Trigger Type** drop-down list.
10. Click **Apply** to save the settings.
11. Click **Debug** to debug the application.

12. In the **Debug** window, right-click in the **Name** column of the **Variables** view (see Figure 6.30)

Figure 6.30 Variables View — Add Global Variables

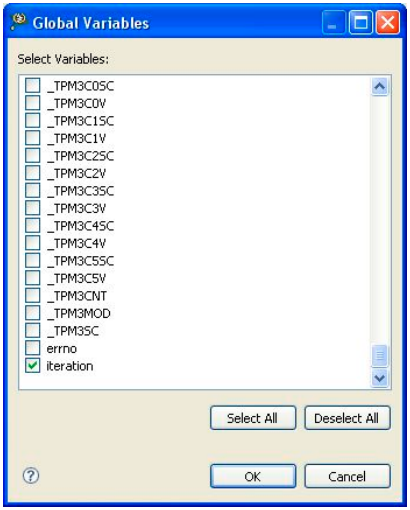


13. Select the **Add Global Variables** option from the context menu.

    The **Global Variables** dialog box appears.

14. Check `iteration` from the list of available checkboxes (see Figure 6.31).

Figure 6.31 Global Variables Dialog Box

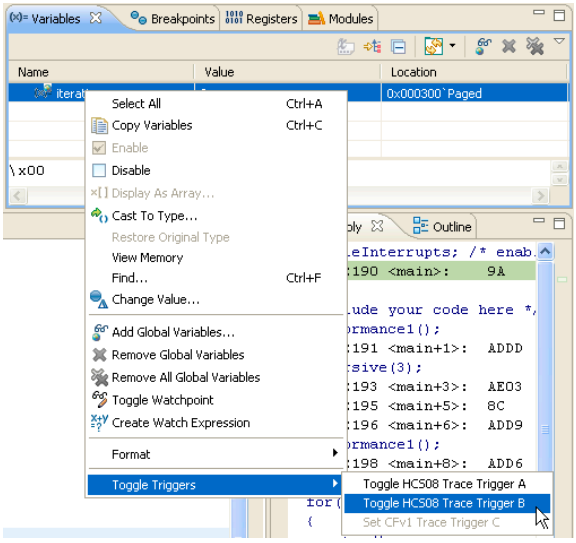


15. Click **OK**.

    The entry for the `iteration` variable gets added to the **Variables** view.

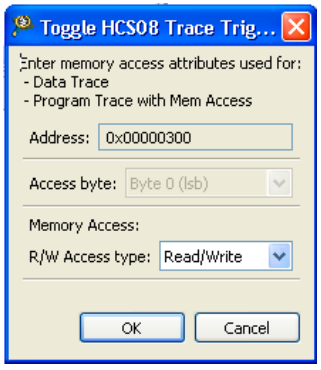
16. Right-click on `iteration` and select **Toggle Triggers > Toggle HCS08 Trace Trigger B** from the context menu.

Figure 6.32 Setting Trigger B



The **Toggle HCS08 Trace Trigger B** dialog box appears.

Figure 6.33 Toggle HCS08 Trace Trigger B Dialog Box

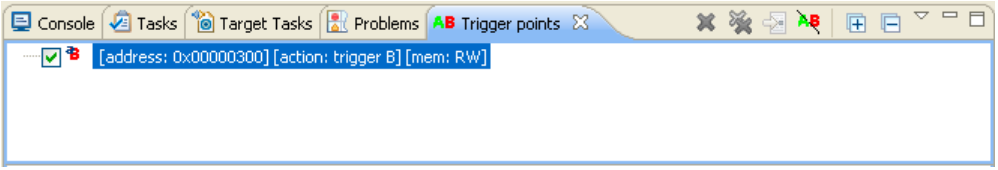


17. Click **OK**.

18. Select **Window > Show View > Other > Analysis > Trigger points** to open the **Trigger points** view.

    The **Trigger points** view displays trigger B that you set on the `iteration` variable.

Figure 6.34 Trigger Points View

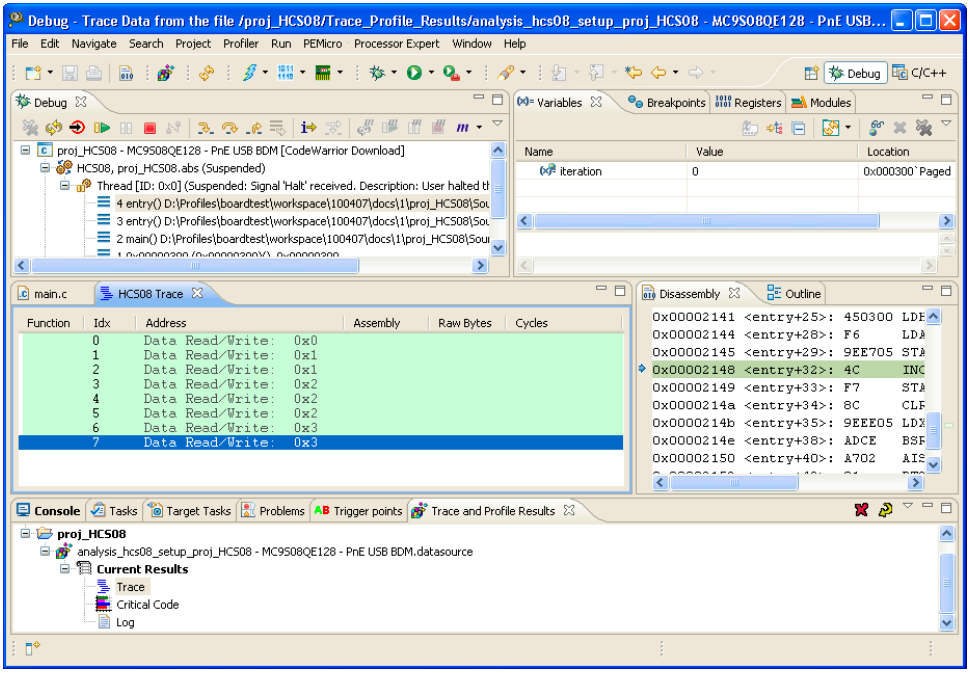


19. Click **Resume**.

    The application captures accesses to the variable (`iteration`) address on which you set trigger B, and stops automatically when buffer gets full.

20. Open the **Trace Data** viewer following the steps explained in the Viewing Data for HCS08 Target topic to view the trace results. Figure 6.35 shows the data files generated by the application after setting trigger B in the Collect Data Trace mode. The trace data that is collected contains values of `iteration` from `0` to `3`.

Figure 6.35 Trace Collected at Address B

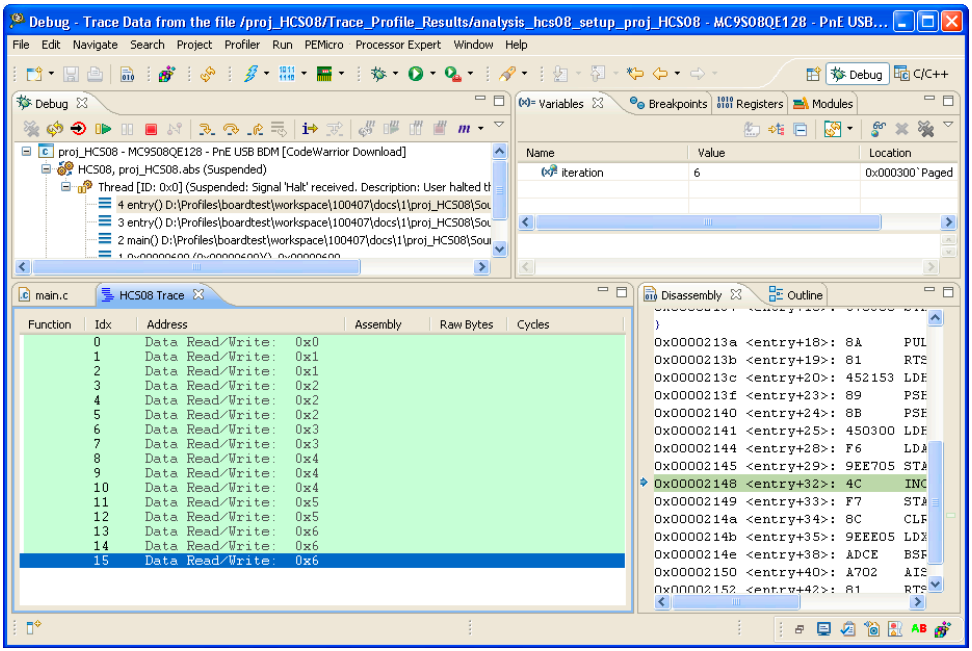


21. Click **Resume** again.

    The application captures more data and stops automatically.

22. Open the **Trace Data** viewer and see new data being appended to the old data (see Figure 6.36).

Figure 6.36 Trace Data Viewer — New Data Appended



NOTE In the **Collect Data Trace** mode, if you perform target stepping instead of full-run, the collected trace will contain mixed data and program trace. This happens because CodeWarrior changes the trigger mode and FIFO shifts storage condition when target stepping is performed. In target stepping, the processor executes one step each time you press the `F6` key (Step Over) and then returns to the suspended (halt) state.

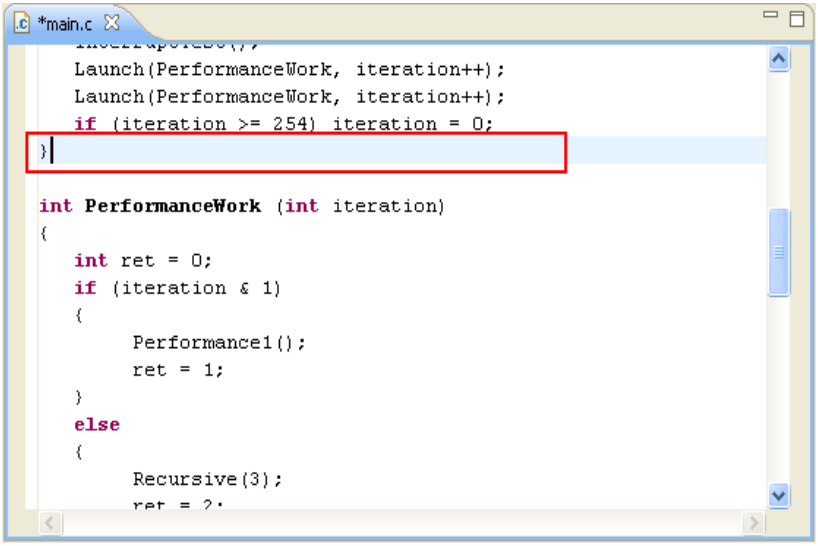
Capture Read/Write Values at Address B, After Access at Address A

This option captures the data involved in a read and/or write access to the addresses specified by trigger B after the instruction at trigger A address has been executed. To capture read/write values at trigger B after trigger A in the Collect Data Trace mode:

1. In the **CodeWarrior Projects** view, expand the **Sources** folder of your project.
2. Double-click on the source file, for example, `main.c` to display its contents in the editor area. Replace the source code in the `main.c` file with the source code shown in Listing 6.5.

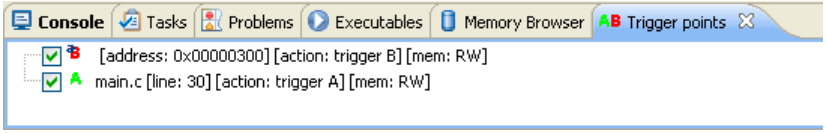
3. Save and build the project.
4. Open the **Debug Configurations** dialog box, and select your project in the tree structure.
5. Click the **Trace and Profile** tab, and check the **Enable Trace and Profile** checkbox.
6. Click **Apply** to save the settings, and close the **Debug Configurations** dialog box.
7. In the editor area, right-click on the marker bar corresponding to the statement as highlighted in Figure 6.37.

Figure 6.37 Setting Trigger A in Collect Data Trace Mode



8. Select **Trace Triggers > Toggle Trace Trigger A** from the context menu.
9. Open the **Debug Configurations** dialog box, and click the **Trace and Profile** tab.
10. Select the **Collect Data Trace** option in the **Trace Mode Options** group.
11. Select **Collect Trace From Trigger** in the **Trace Start/Stop Conditions** group.
12. Check the **Break on FIFO Full** checkbox.
13. Select **Capture Read/Write Values at Address B, After access at Address A** from the **Trigger Type** drop-down list.
14. Click **Apply** to save the settings.
15. Click **Debug** to debug the application.
16. Perform steps 12 – 17 of the topic, Capture Read/Write Values at Address B. After setting triggers, A and B, the **Trigger points** view contains the following entries.

Figure 6.38 Trigger Points View with Trigger A and Trigger B

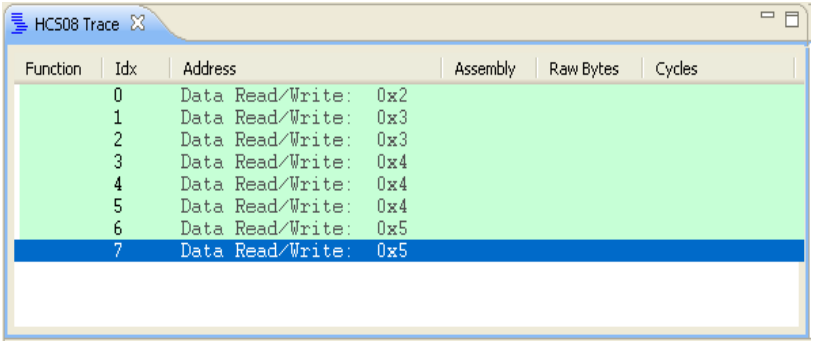


17. Click **Resume**.

    The application waits for the instruction at trigger A address to execute, monitors the address of `iteration`, collects the trace data till buffer gets full, and then stops automatically.

18. Open the **Trace Data** viewer following the steps explained in the topic, Viewing Data for HCS08 Target to view the trace results. Figure 6.39 shows the data files generated by the application after setting triggers A and B in the Collect Data Trace mode. The trace data that is collected contains values of `iteration` starting from `2`. This is because `iteration` has been incremented twice by the time application reaches the address where trigger A is set.

Figure 6.39 Trace Collected at Address B, After Access at Address A

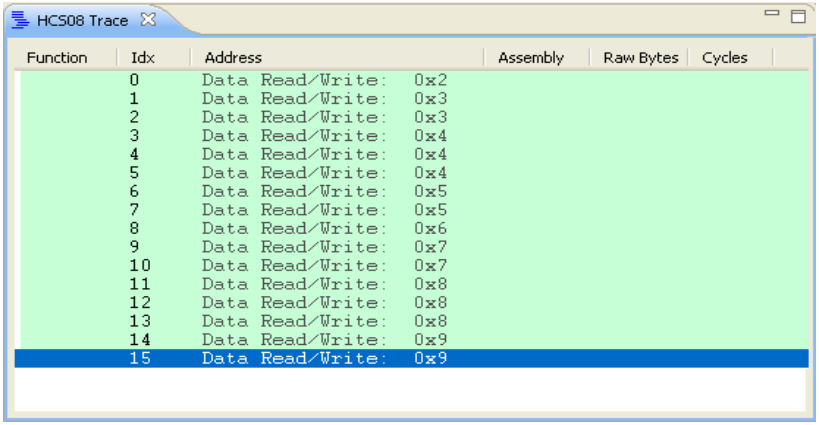


19. Click **Resume** again.

    The application captures more data and stops automatically.

20. Open the **Trace Data** viewer and see new data being appended to the old data (see Figure 6.40).

Figure 6.40 Trace Collected at Address B, After Access at Address A — New Data Appended



NOTE With a trigger condition selected, full trace is collected even when no triggers are set. That is, if you specify a trigger condition in the **Trace and Profile** tab of the **Debug Configurations** dialog box, but do not set the trigger in the application then full trace data will be collected from the beginning of the application.

Setting Triggers in Profile-Only Mode

The Profile-Only mode does not collect the trace data; it only profiles the data. Trace is empty in this mode; you can only see the profiling information in the **Critical Code Data** viewer. To set a trigger in Profile-Only mode:

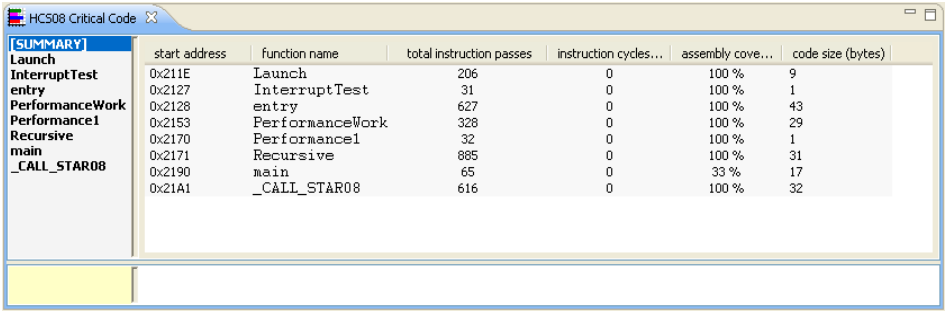
1. In the **CodeWarrior Projects** view, expand the **Sources** folder of your project.
2. Double-click on the source file, for example, `main.c` to display its contents in the editor area. Replace the source code in the `main.c` file with the source code shown in Listing 6.2.
3. Save and build the project.
4. Open the **Debug Configurations** dialog box, and select your project in the tree structure.
5. Click the **Trace and Profile** tab, and check the **Enable Trace and Profile** checkbox.
6. Select the **Profile-Only** option from the **Trace Mode Options** group.
7. Click **Apply** to save the settings.
8. Click **Debug** to debug the application.

9. Click **Resume** and after a short while, click **Suspend**.

   The application halts and data is collected.

10. Open the **Critical Code Data** viewer following the steps explained in the topic, Viewing Data for ColdFire V1 Target to view the critical code data results.

Figure 6.41 Critical Code Data — Profile-Only Mode of HCS08



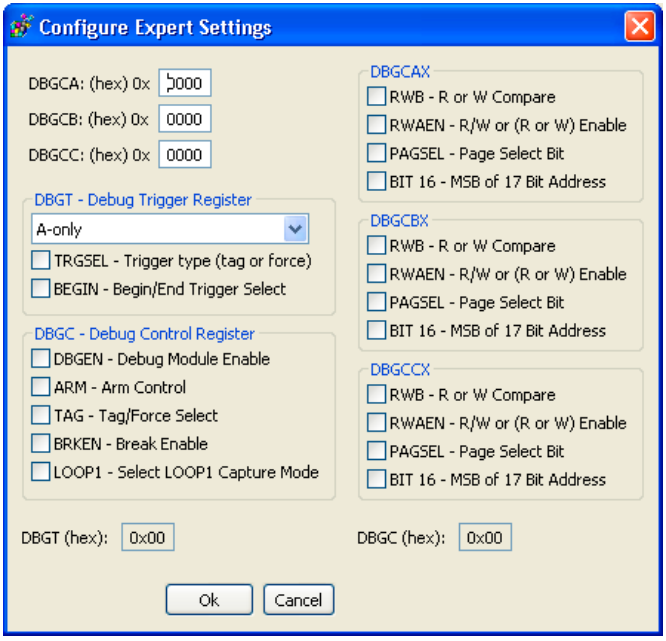
Setting Triggers in Expert Mode

The **Expert** mode gives you an access to most of the on-chip DBG module registers. This mode contains the newest comparator *C* controls and lets you set trigger types directly. To collect trace using the **Expert** mode:

1. Open the **Debug Configurations** dialog box.
2. Click the **Trace and Profile** tab, and check the **Enable Trace and Profile** checkbox.
3. Select the **Expert** option from the **Trace Mode Options** group.
4. Click the **Configure Expert Settings** button.

   The **Configure Expert Settings** dialog box appears.

Figure 6.42 Configure Expert Settings Dialog Box



5. Specify the settings according to requirements.
6. Click **OK** to save the settings.
7. Click **Apply** in the **Debug Configurations** dialog box.
8. Click **Debug** and collect trace.

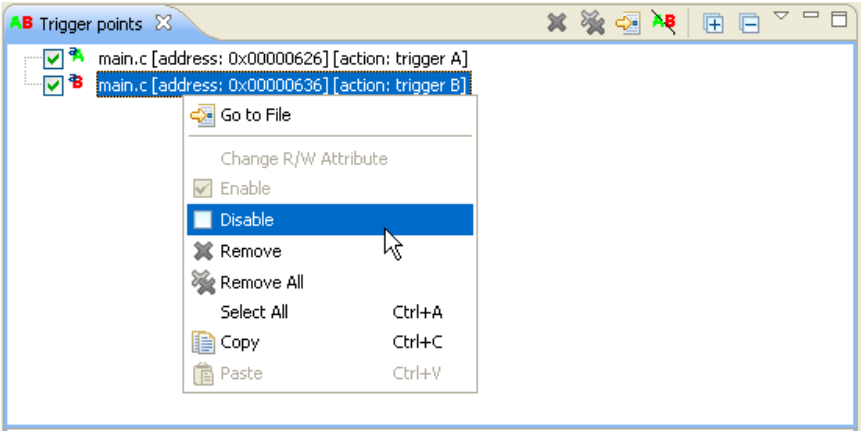
Enabling and Disabling the Tracepoints

If you want to enable the tracepoints, right-click on the marker bar where trigger A and trigger B are already set and in disabled state, select the **Enable Triggerpoint** option from the context menu.

If you want to disable the tracepoints, right-click on the marker bar where triggers are already set and enabled, select the **Trace Triggers > Disable Triggerpoint** option from the context menu. A disabled tracepoint will have no effect during the collection of trace data. You can also disable/enable the tracepoint from **Trigger points** view. Right-click on the selected attribute and select **Disable/Enable** option. The unchecked attribute indicates the disabled tracepoint.

You can also use the **Ignore all**,  option to disable all the tracepoints without manually selecting them in the **Trigger Points** view. You can click **Ignore All** again to enable the tracepoints.

Figure 6.43 Disabling/Enabling the Trigger from Trigger Points View



7

Setting Tracepoints (ColdFire V1)

Tracepoint is a point in the target program where start or stop triggers are set at a line of source code or assembly code and on the memory address. The start and stop tracepoints are triggers for enabling and disabling the trace output. The advantage of setting start and stop tracepoints is to capture the trace data from the specific part of the program. They solve the problem of tracing a very large application. A full trace would be extremely difficult to follow and would also require a large amount of target memory to store it. The trace data displays the trace result based on the tracepoints. This chapter explains how to set start and stop tracepoints and how to enable and disable a tracepoint.

The tracepoints for the ColdFire V1 target can be set on:

- addresses and source files

  The tracepoints on addresses and source files can be set in the editor area and the **Disassembly** view. The source line tracepoint is set in the editor area, and the address tracepoint is set in the **Disassembly** view.

- data and memory

  The tracepoints on data and memory can be set from the **Variables** and **Memory** views.

This chapter consists of the following topics:

Conditions for Starting/Stopping Triggers

In the ColdFire V1 target, the triggers, A, B, and C are used to start and stop the trace collection. The triggers, A and B are set on a function address and trigger C is set on a variable address. The trace collection starts or ends depending on the trace mode selected along with a combination of the following actions involving A, B, and C.

- **Trace is always enabled** — Trace remains enabled all the time during the application run. The trace data is collected from the beginning till end of the debug session.
- **Trace from Trigger A Onward** — Trace is disabled initially. Once the application starts and the execution reaches the instruction where trigger A is set, trace is automatically enabled by the hardware, without stopping the core. Trace remains enabled for the rest of the debug session, that is until you suspend the application.
- **Trace from Trigger A to Trigger B** — The trace gets enabled at trigger A and starts collecting from there. Once the execution reaches the instruction where trigger B is set, the trace is automatically disabled by the hardware. After the trace is disabled, it is not restarted. However, if the application is stopped either automatically or manually, the CodeWarrior resets the trace module. Therefore, if you resume the application later and the execution reaches trigger A again, trace automatically restarts and repeats the same cycle.
- **Trace from Trigger A to Trigger C** — Same as **Trace from Trigger A to Trigger B** except that trigger C is set on the variable addresses.
- **Trace from Trigger B Onward** — The application starts and enables trace at trigger B, which remains enabled through out the debug session.
- **Trace from Trigger B to Trigger A** — Trace enables at trigger B and disables at trigger A. The trace collection follows the same approach as **Trace from Trigger A to Trigger B**.
- **Trace from Trigger B to Trigger C** — Trace enables at trigger B and disables at trigger C. The trace collection follows the same approach as **Trace from Trigger A to Trigger B**.
- **Trace from Trigger C Onward** — The application starts and enables trace at trigger C, which remains enabled through out the debug session.
- **Trace from Trigger C to Trigger A** — Trace enables at trigger C and disables at trigger A. The trace collection follows the same approach as **Trace from Trigger A to Trigger B**.
- **Trace from Trigger C to Trigger B** — Trace enables at trigger C and disables at trigger B. The trace collection follows the same approach as **Trace from Trigger A to Trigger B**.

NOTE In the **Automatic (One-buffer)** mode, if you check the **Halt the Target when Trace Buffer Gets Full** checkbox, the trace starts collecting from trigger A till the buffer gets full. If you do not check the checkbox, the trace is collected till you suspend the application manually. And you will only see the last portion of trace in the **Trace Data** viewer because the internal trace buffer is overwritten. It holds for all the above options.

Trace Modes

The triggers can be set in the following trace modes:

- Continuous — Setting Triggers in Continuous Mode
- Automatic (One-buffer) — Setting Triggers in Automatic (One-buffer) Mode
- Profile-Only — Setting Triggers in Profile-Only Mode
- Expert — Setting Triggers in Expert Mode

Setting Triggers in Continuous Mode

The Continuous mode collects trace continuously till you suspend the target application. This topic explains how to set the following trace conditions in the editor area in the Continuous mode:

- Trace From Trigger A Onward
- Trace From Trigger A to Trigger B

Before setting the tracepoints, see Collecting Data chapter for the procedure of how to collect the trace and profiling data.

Trace From Trigger A Onward

To set trigger A in the editor area for the ColdFire V1 target:

1. In the **CodeWarrior Projects** view, select the **Sources** folder of your project.
2. Double-click on the source file, for example, `main.c` to display its contents in the editor area. Replace the source code of the `main.c` file with the source code shown in Listing 7.1.

Listing 7.1 Source code for trace collection

```
#include <hidef.h> /* for EnableInterrupts macro */
#include "derivative.h" /* include peripheral declarations */

volatile int a, b, c, i;

void f()
{
  c=1;
  c=2;
  c=3;
}

void f1()
```

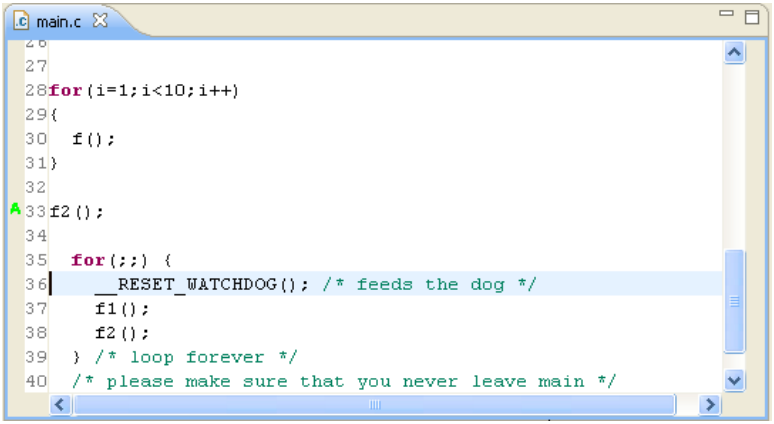
```
{
  b=1;
  b=2;
  b=3;
}

void f2()
{
  a=1;
  a=2;
  a=3;
}

void main(void) {
  EnableInterrupts;
  /* include your code here */
for(i=1;i<10;i++)
{
  f();
}
f2();
  for(;;) {
    __RESET_WATCHDOG();/* feeds the dog */
    f1();
    f2();
  } /* loop forever */
}
```

3. Save and build the project.
4. Open the **Debug Configurations** dialog box, and select your project in the tree structure.
5. Click the **Trace and Profile** tab, and check the **Enable Trace and Profile** checkbox.
6. Click **Apply** to save the settings, and close the **Debug Configurations** dialog box.
7. In the editor area, select the statement, `f2();`, in the following section of code.

   ```
   for(i=1;i<10;i++)
   { f();}
   f2();
   ```

8. Right-click on the marker bar, select the **Trace Triggers > Toggle Trace Trigger A** option from the context menu. The trigger A icon appears on the marker bar in green color (see Figure 7.1). The same option is also used to remove trigger A from the marker bar.

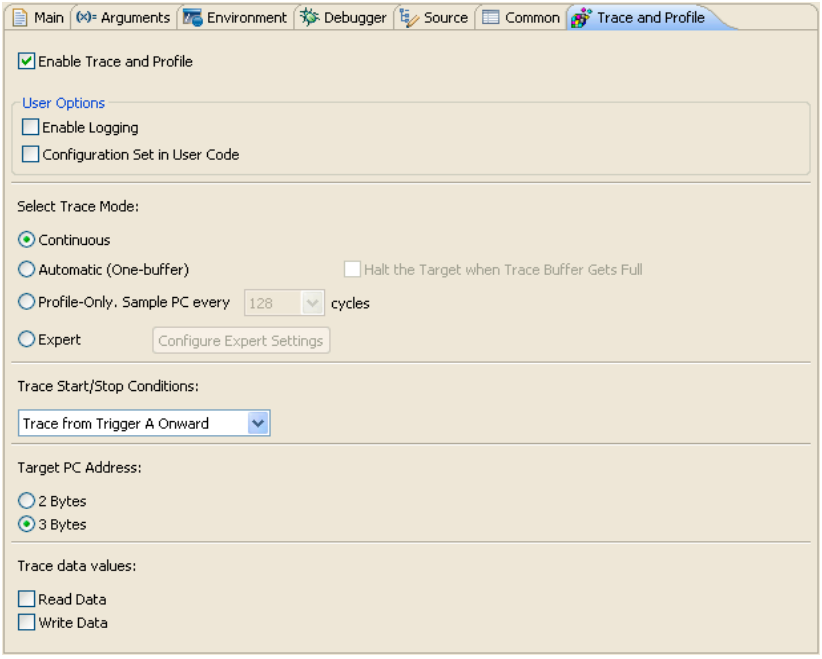
Figure 7.1 Trigger A Set in Editor Area



NOTE Do not set tracepoints on the statements containing only comments, brackets, and variable declaration with no value. If tracepoints are set on invalid statements, they are automatically disabled when the application is debugged.

9. Open the **Debug Configurations** dialog box, click the **Trace and Profile** tab.
10. Select the **Continuous** option from the **Select Trace Mode** group.
11. Select the **Trace from Trigger A Onward** option from the **Trace Start/Stop Conditions** drop-down list (see Figure 7.2).

Figure 7.2 Setting Trigger Conditions

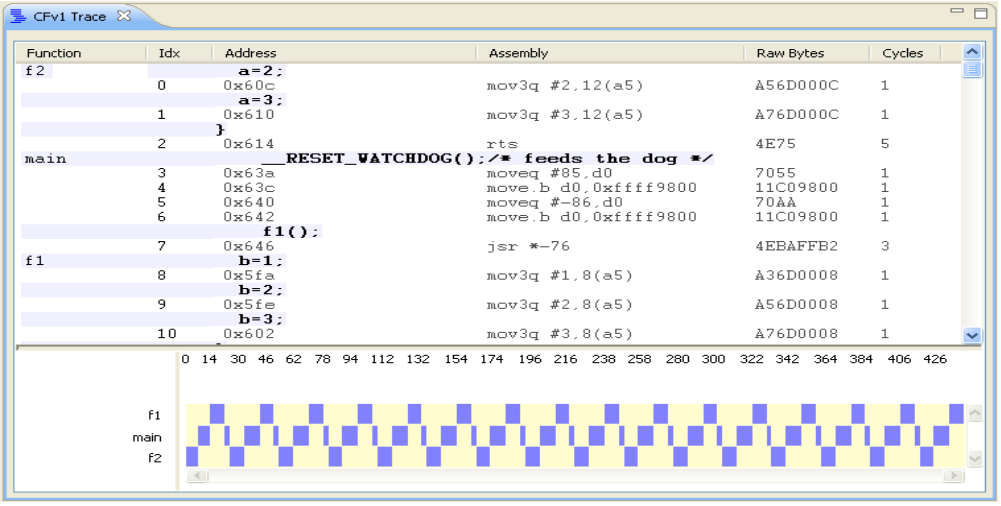


12. Click **Debug** to debug the application.
13. Click **Resume** to resume the application and after a short while, click **Suspend**.

    The trace data is collected in the data files.
14. Open the **Trace Data** viewer following the steps explained in the topic, Viewing Data for ColdFire V1 Target to view the trace results.

    Figure 7.3 shows the data file generated by the application in which the data has been collected after setting trigger A in the source code. In this data file, the **Function** field shows that trace starts collecting from `f2()`, where you set trigger A. Because you selected the **Trace from A Onward** option, the trace data starts collecting from `f2()` and stops when you suspend the application.

Figure 7.3 Trace Data After Setting *Trace From Trigger A Onward* in Continuous Mode



15. Click the **Terminate** icon in the **Debug** perspective to terminate the application.

Trace From Trigger A to Trigger B

To set trigger A and trigger B in the editor area in the Continuous mode:

1. In the **CodeWarrior Projects** view, select the **Sources** folder of your project.
2. Double-click on the source file, for example, `main.c` to display its contents in the editor area. Replace the source code of the `main.c` file with the source code shown in Listing 7.1.
3. Save and build the project.
4. Open the **Debug Configurations** dialog box, and select your project in the tree structure.
5. Click the **Trace and Profile** tab, and check the **Enable Trace and Profile** checkbox.
6. Click **Apply** to save the settings, and close the **Debug Configurations** dialog box.
7. In the editor area, select the statement, `f();`, in the following section of code.

```
for(i=1;i<10;i++)
{ f();}
f2();
```

8. Right-click on the marker bar, select the **Trace Triggers > Toggle Trace Trigger A** option from the context menu.

9. Right-click on the marker bar corresponding to the statement, `f2();`, and select **Trace Triggers > Toggle Trace Trigger B** from the context menu.

   The trigger B icon appears on the marker bar in red color (see Figure 7.4). The same option is also used to remove trigger B from the marker bar.

NOTE It is recommended to set both triggers in the same function so that the trace that is collected is meaningful.

Figure 7.4 Setting Triggers A and B

NOTE The mouse pointer over a trigger icon in the marker bar displays the attributes of the trigger on that line. For source lines, there can be multiple tracepoints mapping to the same line.

10. Open the **Debug Configurations** dialog box, and click the **Trace and Profile** tab.
11. Select the **Continuous** option from the **Select Trace Mode** group.
12. Select the **Trace from Trigger A to Trigger B** option from the **Trace Start/Stop Conditions** drop-down list.
13. Click **Apply** to save the settings.
14. Click **Debug** to debug the application.
15. Click **Resume** to collect the trace data.
16. Click **Suspend** to stop the target application.
17. Open the **Trace Data** viewer following the steps explained in the topic, Viewing Data for ColdFire V1 Target to view the trace results.

    Figure 7.5 shows the data files and the timeline graph that is generated by the application in which the data has been collected after setting triggers, A and B. The **Trace Data** viewer in Figure 7.5 shows that trace starts collecting from the `f()` function where you set trigger A.

Figure 7.5 Trace Data View After Setting *Trace From Trigger A to Trigger B* in Continuous Mode — Begin

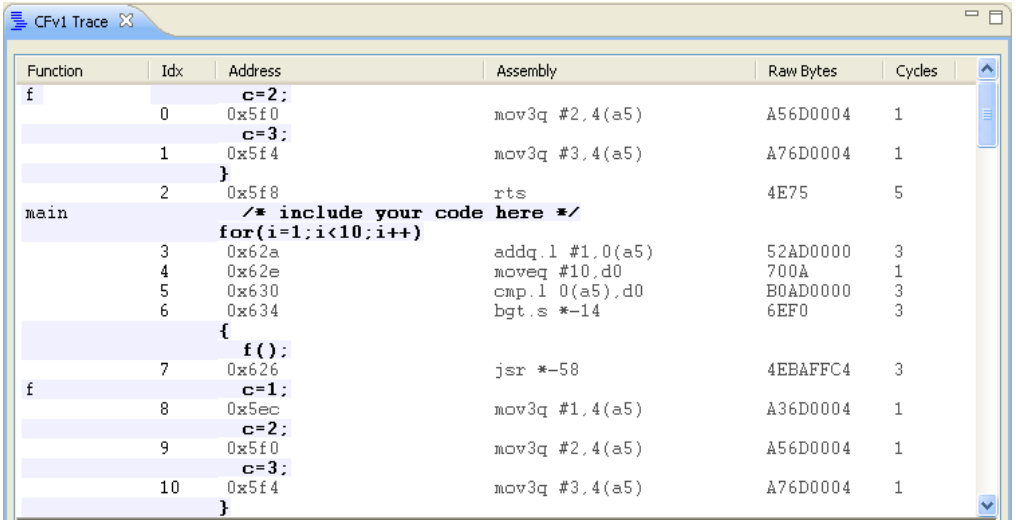
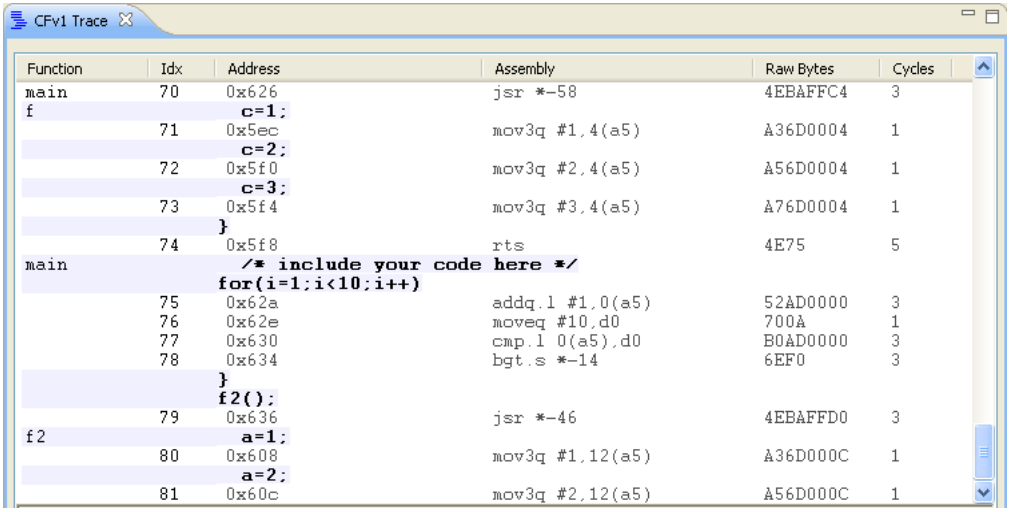


Figure 7.6 shows that trace stops at the `f2()` function where you set trigger B.

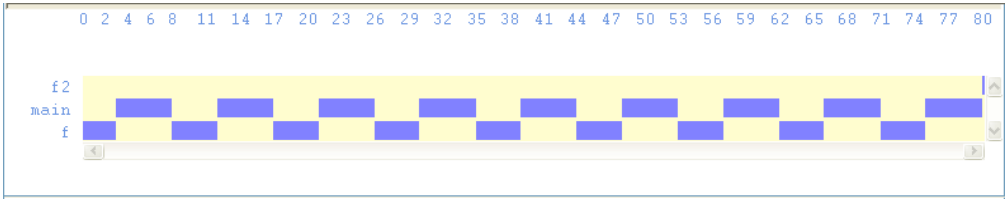
Figure 7.6 Trace Data View After Setting *Trace From Trigger A to Trigger B* in Continuous Mode — End



NOTE If trigger A and trigger B have less than three executed instructions between them, the stop point is not taken into consideration by the hardware. That is, the hardware misses trigger B if it is set close (less than three executed instructions) to trigger A. This is because there is a three-instruction delay in the hardware pipeline; therefore trigger A is acknowledged at the third executed instruction from where it is actually set.
However, incase triggers are set in a loop, hardware will acknowledge trigger B when the loop is executed second time.
A three-instruction delay during trace collection also occurs when a breakpoint is set on a `jsr` instruction in the **Disassembly** view. This stops the trace abruptly; therefore a red line is displayed in the **Trace Data** viewer to mark the points where the trace is broken. For more information, refer Trace Collection with Breakpoints in Appendix A.

The graph in Figure 7.7 shows the timeline of the trace data. In this graph, you can see that the trace data starts collecting from `f()`, that is trigger A, and stops at `f2()`, that is trigger B. To have a clearer view of the graph, you can zoom-in or zoom-out in the graph by scrolling the mouse up or down.

Figure 7.7 Graph Displaying Timeline of Trace Data



18. Click the **Terminate** icon in the **Debug** perspective to terminate the application.

Setting Triggers in Automatic (One-buffer) Mode

The **Automatic (One-buffer)** mode collects trace till the trace buffer gets full. This topic explains how to set the following trace conditions in the editor area in the **Automatic (One-buffer)** mode:

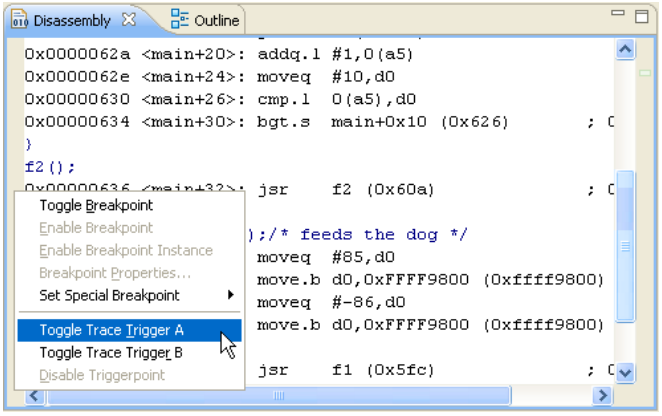
- Trace From Trigger A Onward
- Trace From Trigger A to Trigger B

Trace From Trigger A Onward

To set trigger A in the **Automatic (One-buffer)** mode in the **Disassembly** view:

1. In the **CodeWarrior Projects** view, select the **Sources** folder of your project.
2. Double-click on the source file, for example, `main.c` to display its contents in the editor area. Replace the source code of the `main.c` file with the source code shown in Listing 7.1.
3. Save and build the project.
4. Open the **Debug Configurations** dialog box, and select your project in the tree structure.
5. Click the **Trace and Profile** tab, and check the **Enable Trace and Profile** checkbox.
6. Select the **Automatic (One-buffer)** option from the **Select Trace Mode** group.
7. Check the **Halt the Target when Trace Buffer Gets Full** checkbox.
8. Select **Trace from Trigger A Onward** from the **Trace Start/Stop Conditions** drop-down list.
9. Click **Apply** to save the settings.
10. Click **Debug** to debug the application.
11. In the **Disassembly** view, right-click on the marker bar corresponding to the statement, `f2();` (see Figure 7.8).

Figure 7.8 Setting Trigger A in Disassembly View



12. Select the **Trace Triggers > Toggle Trace Trigger A** option from the context menu. The trigger A icon appears in green color on the marker bar, in the editor area and the **Disassembly** view.
13. Click **Resume**. The application stops automatically after some time.

NOTE Because you selected the **Halt the Target when Trace Buffer Gets Full** check box in the Trace and Profile tab, the application will stop automatically after the trigger hit. You do not need to stop it manually by clicking **Suspend**.

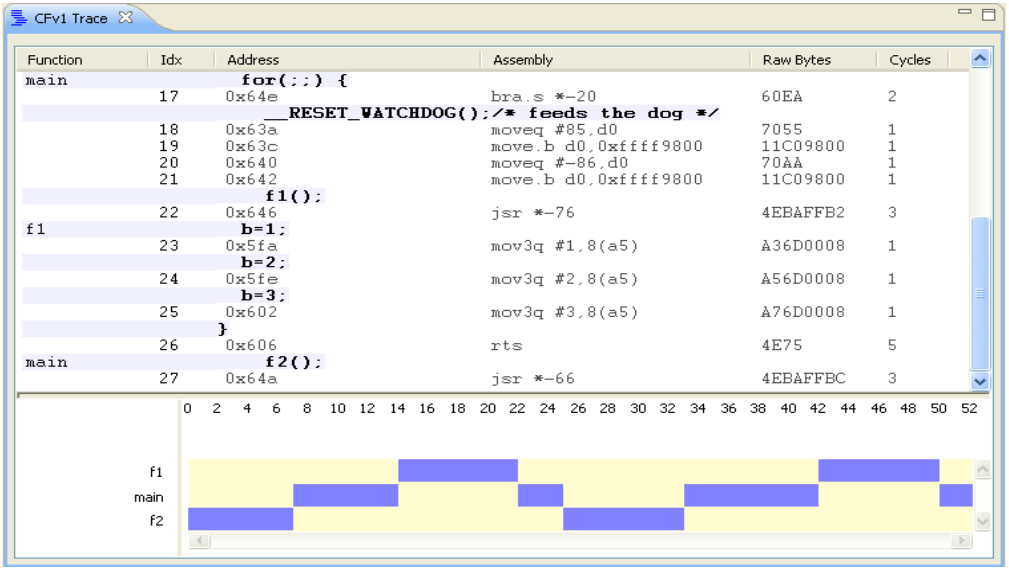
14. Open the **Trace Data** viewer following the steps explained in the topic, Viewing Data for ColdFire V1 Target to view the trace results.

Figure 7.9 shows the data files that are generated by the application in which the data has been collected after setting trigger A in **Disassembly** view, and selecting the **Automatic (One-buffer)** mode. Figure 7.9 shows that the trace data starts collecting from trigger A. The application stops automatically when the trace buffer gets full and the trace data is collected till that trace buffer.

The graph in Figure 7.9 shows the timeline of the trace data. In this graph, you can see that the trace data starts collecting from the `f2()` function and stops when the trace buffer gets full.

NOTE If you choose to not check the **Halt the Target when Trace Buffer Gets Full** checkbox, the trace buffer gets overwritten. Therefore, only the last part of the trace data executed before the application suspends is visible in the **Trace Data** viewer.

Figure 7.9 Trace Data After Setting *Trace From Trigger A Onward* in Automatic Mode



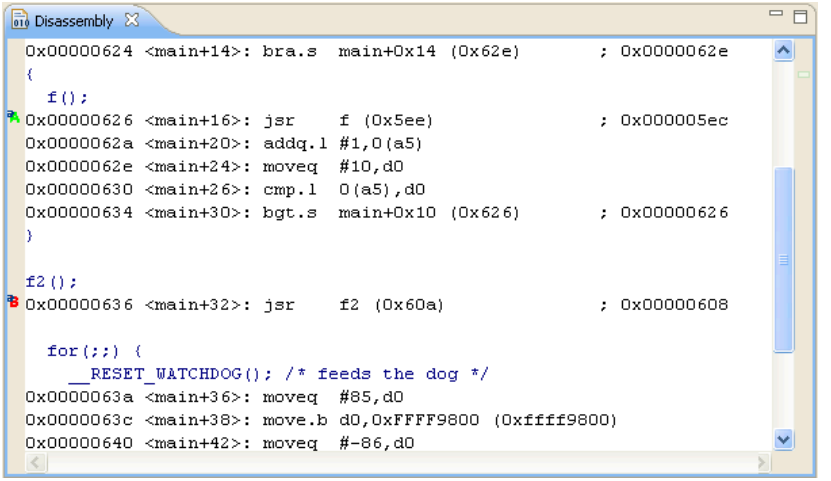
NOTE Similarly, you can set the **Trace from Trigger B Onward** trace condition in both Continuous and Automatic modes and collect the trace data.

Trace From Trigger A to Trigger B

To set triggers, A and B in the **Automatic (One-buffer)** mode in the **Disassembly** view:

1. In the **CodeWarrior Projects** view, select the **Sources** folder of your project.
2. Double-click on the source file, for example, `main.c` to display its contents in the editor area. Replace the source code of the `main.c` file with the source code shown in Listing 7.1.
3. Save and build the project.
4. Open the **Debug Configurations** dialog box, and select your project in the tree structure.
5. Click the **Trace and Profile** tab, and check the **Enable Trace and Profiling** checkbox.
6. Select the **Automatic (One-buffer)** option from the **Select Trace Mode** group.
7. Check the **Halt the Target when Trace Buffer Gets Full** checkbox.
8. Select **Trace from Trigger A to Trigger B** from the **Trace Start/Stop Conditions** drop-down list.
9. Click **Apply** to save the settings.
10. Click **Debug** to debug the application.
11. Open the **Disassembly** view.
12. Right-click on the marker bar corresponding to the statement, `f();`.
13. Select **Trace Triggers > Toggle Trace Trigger A** from the context menu. The trigger A icon appears in green color on the marker bar, in the editor area and the **Disassembly** view (see Figure 7.10).
14. Right-click on the marker bar corresponding to the statement, `f2();`.
15. Select **Trace Triggers > Toggle Trace Trigger B** from the context menu. The trigger B icon appears in red color on the marker bar, in the editor area and the **Disassembly** view (see Figure 7.10).

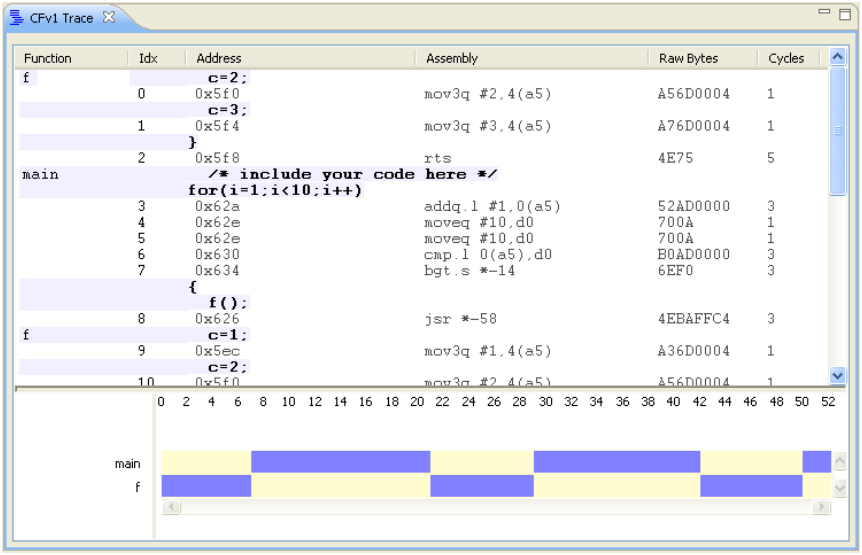
Figure 7.10 Setting Triggers A and B in Disassembly View



16. Click **Resume**. The application stops automatically after some time and trace data is collected.
17. Open the **Trace Data** viewer following the steps explained in the topic, Viewing Data for ColdFire V1 Target to view the trace results.

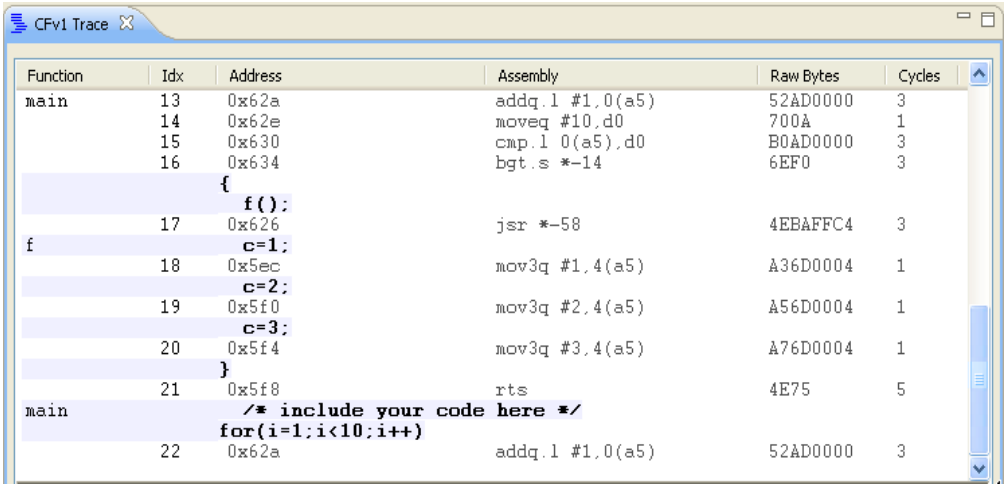
Figure 7.11 shows the data files that are generated by the application in which the data has been collected after setting triggers, A and B, in the **Disassembly** view and selecting the **Automatic (One-buffer)** mode. The **Trace Data** viewer in Figure 7.11 shows that the trace data starts collecting from the `f()` function where you set trigger A. The application stops automatically when the trace buffer gets full and the trace data is collected till that trace buffer.

Figure 7.11 Trace Data View After Setting *Trace From Trigger A to Trigger B* in Automatic Mode — Begin Data



In this example, the trace buffer got full before trigger B was executed, therefore, the **Trace Data** viewer in Figure 7.12 displays only the trace data collected from trigger A till first buffer full. If you resume the application at this point, trace will continue to collect till other part of the trace buffer gets full and so on till trigger B is hit. Once trigger B is hit, trace stops collecting.

Figure 7.12 Trace Data View After Setting *Trace From Trigger A to Trigger B* in Automatic Mode — End Data



Similarly, you can set the trace conditions, **Trace from Trigger B to Trigger A**, **Trace from Trigger A to Trigger C**, **Trace from Trigger C to Trigger A**, **Trace from Trigger B to Trigger C**, **Trace from Trigger C to trigger B** in both Continuous and Automatic modes and collect the trace data accordingly.

The Tracepoints on Data and Memory topic explains how to set trigger C.

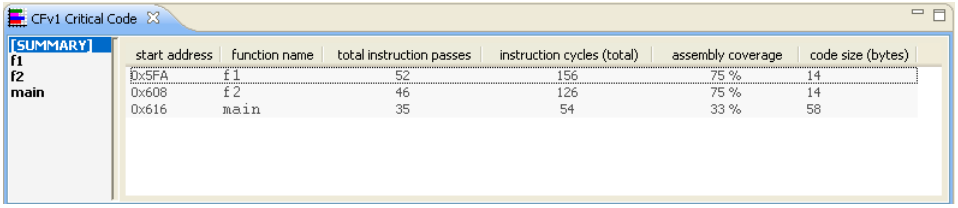
Setting Triggers in Profile-Only Mode

The Profile-Only mode does not collect the trace data; it only profiles the data. Trace is empty in this mode; you can only see the profiling information in the **Critical Code Data** viewer. To set a trigger in Profile-Only mode:

1. Set trigger A in the source code following the steps 1– 8 described in the topic, Trace From Trigger A Onward.
2. Open the **Debug Configurations** dialog box, and click the **Trace and Profile** tab.
3. Select the **Profile-Only. Sample PC every 512 cycles** option from the **Select Trace Mode** group.
4. Select the **Trace from Trigger A Onward** trace condition.
5. Click **Apply** to save the settings.
6. Click **Debug** to debug the application.
7. Click **Resume** and after a short while, click **Suspend**. The application halts and data is collected.

8. Open the **Critical Code Data** viewer following the steps explained in the topic, Viewing Data for ColdFire V1 Target to view the critical code data results.

Figure 7.13 Critical Code Data — Profile-Only Mode of ColdFire V1



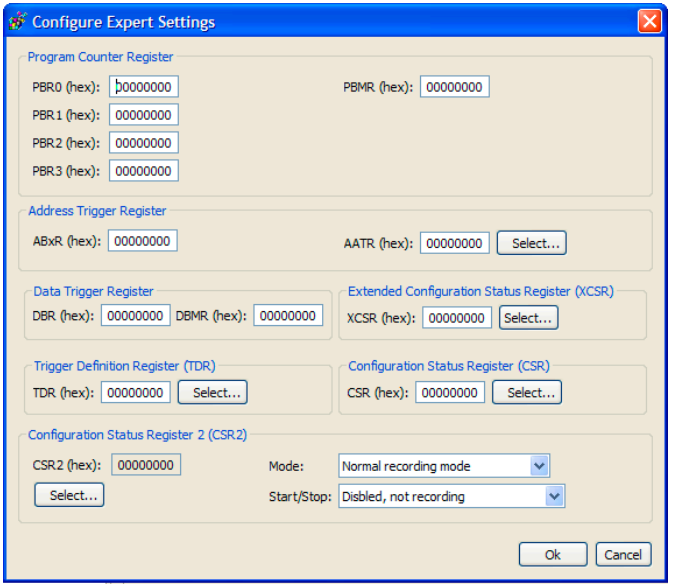
Setting Triggers in Expert Mode

The **Expert** mode lets you configure the ColdFire V1 trace and debug registers directly. This mode provides you full control over the trace data to be collected. To collect trace in the **Expert** mode:

1. Open the **Debug Configurations** dialog box.
2. Click the **Trace and Profile** tab, and check the **Enable Trace and Profile** checkbox.
3. Select the **Expert** option from the **Select Trace Mode** group.
4. Click the **Configure Expert Settings** button.

   The **Configure Expert Settings** dialog box appears.

Figure 7.14 Configure Expert Settings Dialog Box



5. Specify the settings according to requirements.
6. Click **OK** to save the settings.
7. Click **Apply** in the **Debug Configurations** dialog box.
8. Click **Debug** to debug the application and collect the trace data.

Tracepoints on Data and Memory

This topic explains how to collect trace data after setting tracepoints on data and memory from both the **Variables** and the **Memory** views. You can set only trigger C on data and memory on a variable address. When you set trigger C on a variable address, the trace data collection starts when the first time that variable is accessed in the execution. This topic uses the same source code as displayed in Listing 7.1.

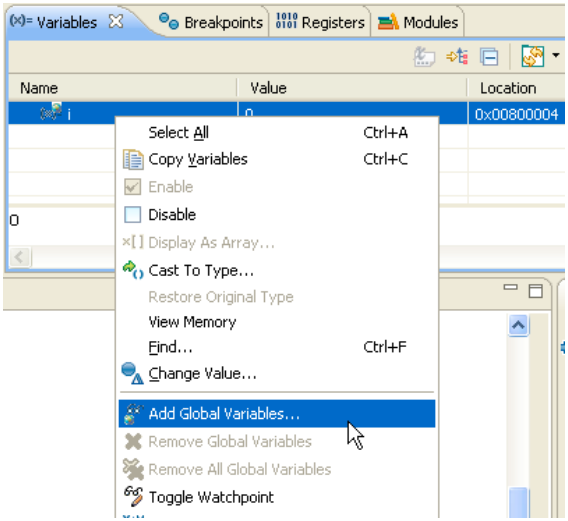
- From Variables View
- From Memory View

From Variables View

To set trigger C from the **Variables** view:

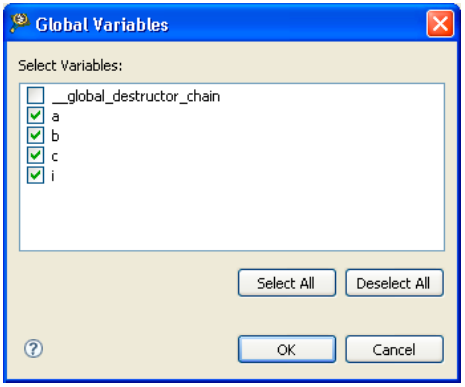
1. In the **CodeWarrior Projects** view, select the **Sources** folder of your project.
2. Double-click on the source file, for example, `main.c` to display its contents in the editor area. Replace the source code of the `main.c` file with the source code shown in Listing 7.1.
3. Save and build the project.
4. Open the **Debug Configurations** dialog box, and select your project in the tree structure.
5. Click the **Trace and Profile** tab, and check the **Enable Trace and Profile** checkbox.
6. Select the **Continuous** option from **Select Trace Mode**.
7. Select the **Trace from Trigger C Onward** option from the **Trace Start/Stop Conditions** group.
8. Click **Apply** to save the settings.
9. Click **Debug** to debug the application.
10. In the **Variables** view, right-click on a cell in the **Name** column (see Figure 7.15).

Figure 7.15 Variables View



11. Select the **Add Global Variables** option from the context menu. The **Global Variables** dialog box appears.

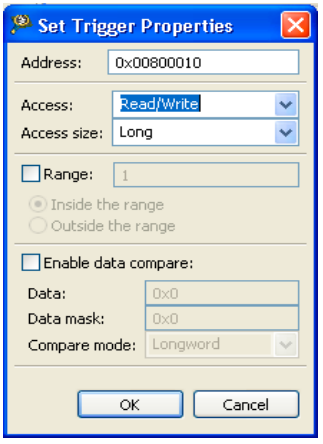
Figure 7.16 Global Variables Dialog Box



12. Click **Select All** and then **OK**. All the variables of the program get added to the **Variables** view.
13. Right-click on the variable, `a`, in the **Name** column of the **Variables** view.
14. Select **Toggle Triggers > Set CFv1 Trace Trigger C** option from the context menu.

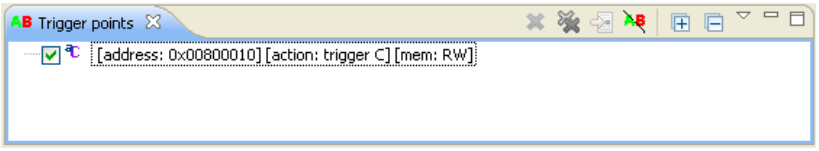
    The **Set Trigger Properties** dialog box appears (see Figure 7.17).
15. Select the **Read/Write** option from the **Access** drop-down list.

Figure 7.17 Set Trigger Properties Dialog Box



16. Click **OK**.
17. Select **Window > Show View > Other > Analysis > Trigger points** to open the **Trigger points** view. The **Trigger points** view displays the trigger C set on the variable address (see Figure 7.18).

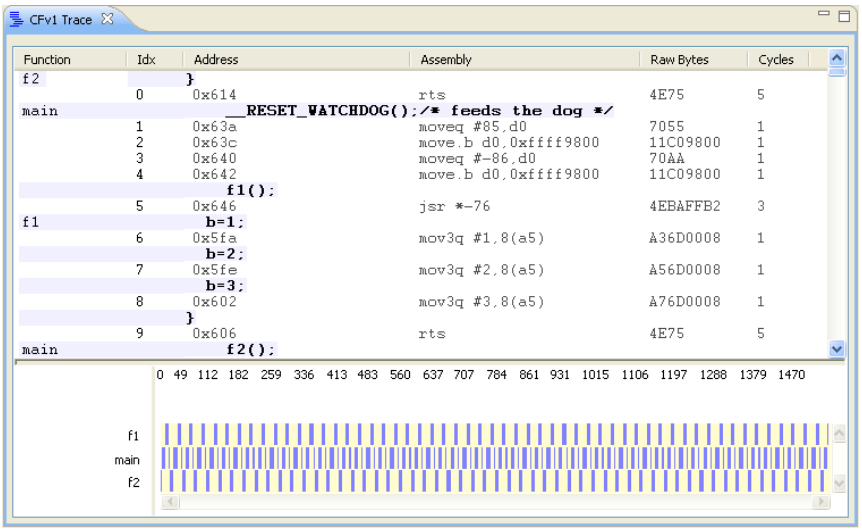
Figure 7.18 Trigger Points View



18. Click **Resume** to collect trace.
19. After a short while, click **Suspend** to stop the target application.
20. Open the **Trace Data** viewer following the steps explained in the topic, Viewing Data for ColdFire V1 Target to view the trace results.

Figure 7.19 shows the data files generated by the application after setting trigger C in the Continuous mode on the variable address.

Figure 7.19 Trace Data After Setting *Trace from Trigger C* in Continuous Mode



The variable, `a` is accessed first time in the `f2()` function. Therefore, after setting trigger C on the address of `a`, the trace data starts collecting from `f2()`. The trace data continues till you suspend the application.

NOTE When trigger C is hit, trace starts from the instruction where the variable on which trigger C has been set is first accessed. However, trace misses the first few instructions due to a delay from the hardware. Therefore, the **Trace Data** viewer does not display the first instruction of the variable. Instead, it displays

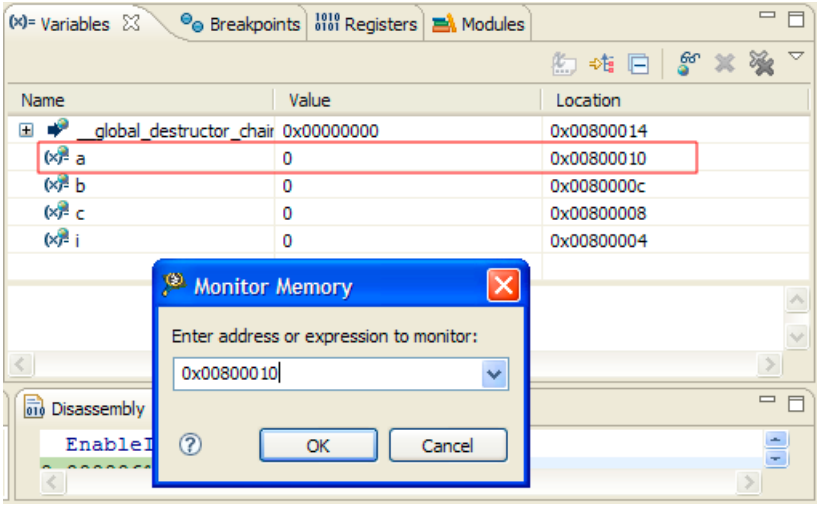
the trace data starting from the instructions that are executed after the first instruction of the variable.

From Memory View

To set trigger C from the **Memory** view:

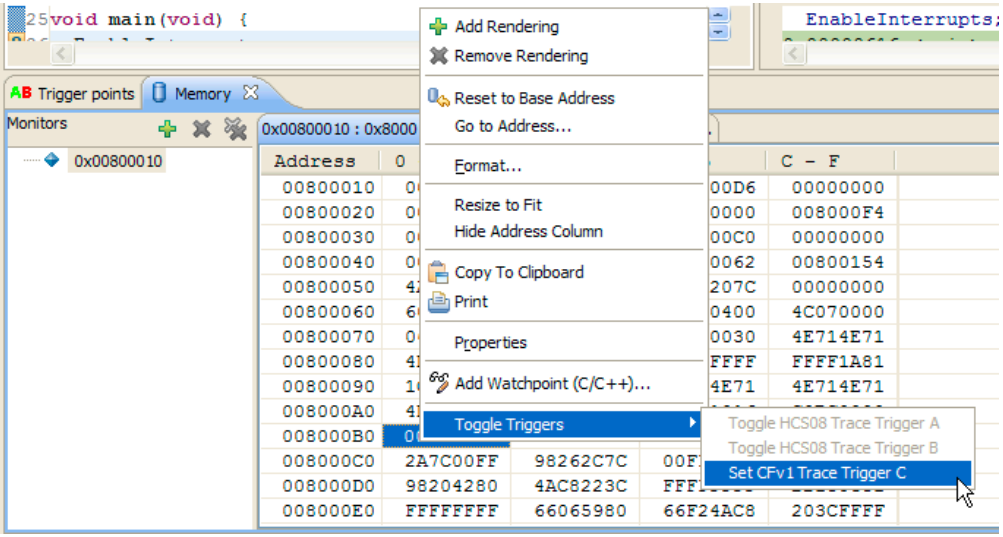
1. Perform steps 1 – 12 as explained in From Variables View
2. Select **Window > Show View > Memory** to open the **Memory** view.
3. Click  in the **Memory** view to open the **Monitor Memory** dialog box (see Figure 7.20).

Figure 7.20 Monitor Memory Dialog Box



4. Enter the address of the variable, `a` in the **Enter address or expression to monitor** text box and click **OK**.
5. In the **Memory** view, right-click on any cell and select **Toggle Triggers > Set CFV1 Trace Trigger C** (see Figure 7.21).

Figure 7.21 Setting Trigger C From Memory View



The **Set Trigger Properties** dialog box appears.

6. Enter the address of the variable, `a` in the **Address** text box.
7. Select the **Read/Write** option from the **Access** drop-down list and click **OK**.
8. Click **Resume** to collect trace.
9. After a short while, click **Suspend** to stop the target application.

The data files are generated by the application, as shown in Figure 7.19, after setting trigger C in the Continuous mode on the variable address.

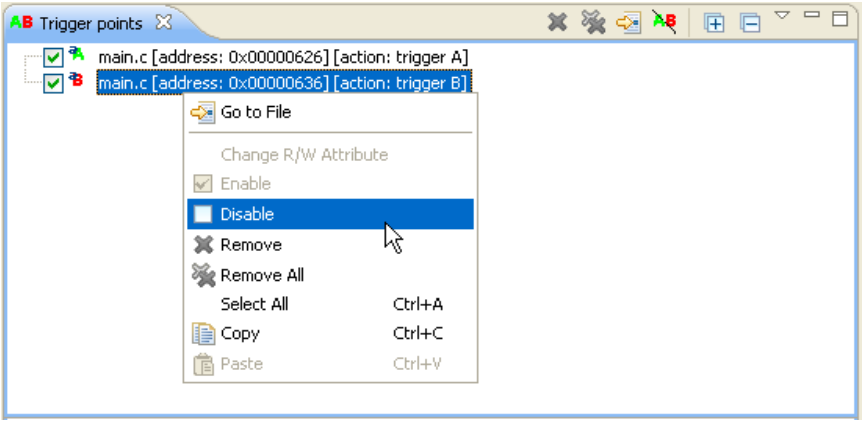
Enable and Disable Tracepoints

If you want to enable the tracepoints, right-click on the Marker bar where triggers are already set and in disabled state, select the **Enable Triggerpoint** option from the context menu.

If you want to disable the tracepoints, right-click on the Marker bar where triggers are already set and enabled, select the **Trace Triggers > Disable Triggerpoint** option from the context menu. A disabled tracepoint will have no effect during the collection of trace data. You can also disable/enable the tracepoint from **Trigger points** view. Right-click on the selected attribute and select **Disable/Enable** option. The unchecked attribute indicates the disabled tracepoint.

You can also use the **Ignore all**,  option to disable all the tracepoints without manually selecting them in the **Trigger Points** view. You can click **Ignore All** again to enable the tracepoints.

Figure 7.22 Disabling/Enabling the Trigger from Trigger Points View



8

Setting Tracepoints (Kinetis)

The Kinetis target supports hardware and software tracepoints for trace collection. Hardware tracepoints use hardware resources to start and stop trace. Hardware tracepoints allow only four comparators to be set for trace collection because they use DWT comparators to start or stop the trace collection. Software tracepoints on the other hand do not use hardware resources and generate interrupts from software to start and stop trace. They allow you to install infinite number of comparators for trace collection and are more intrusive.

This chapter consists of the following topics:

Setting Hardware Tracepoints

You can set hardware tracepoints from the source code as well as from the **Disassembly** view. A hardware tracepoint is set from the source code on an instruction; while it is set from the **Disassembly** view on an address. You can also set hardware tracepoints from the **Trace and Profile** tab of the **Debug Configurations** window by configuring the DWT settings. Configuring DWT settings includes setting comparators as triggers along with their reference value and selecting the event that generates the comparators match. When this event occurs, triggers are fired and trace is collected.

From Source Code

To set hardware tracepoints in the source code and collect trace data on the Kinetis target:

1. In the **CodeWarrior Projects** view, select the **Sources** folder of your project.
2. Double-click on the source file, for example, `main.c` to display its contents in the editor area. Replace the source code of the `main.c` file with the source code shown in Listing 8.1.

Listing 8.1 Source code for trace collection

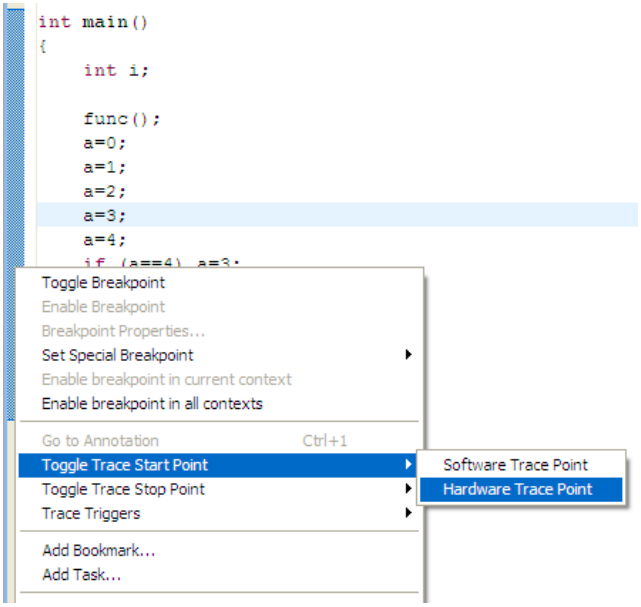
```
#include <stdio.h>
volatile int a;

void func() {
     a=0;
}
int main()
{
   int i;

   func();
   a=0;
   a=1;
   a=2;
   a=3;
   a=4;
   if (a==4) a=3;
   if (a==3) a=1;
   if (a==1) a=4;
   if (a==4) a=5;
   func();
   return 0;
}
```

3. Save and build the project.
4. Open the **Debug Configurations** dialog box, and select your project in the tree structure.
5. Click the **Trace and Profile** tab, and check the **Enable Trace and Profile** checkbox.
6. Click **Apply** to save the settings, and close the **Debug Configurations** dialog box.
7. In the editor area, select the following statement:

   ```
   if (a==4) a=3;
   ```

8. Right-click on the marker bar, select the **Toggle Trace Start Point > Hardware Trace Point** option from the context menu (see Figure 8.1). The same option is also used to remove the start trigger from the marker bar.

Figure 8.1 Setting Hardware Start Tracepoint From Source Code



The start trigger icon appears on the marker bar in green color.

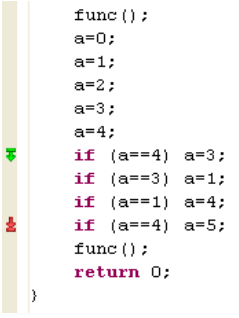
9. In the editor area, select the following statement:

   ```
   if (a==4) a=5;
   ```

10. Right-click on the marker bar, select the **Toggle Trace Stop Point > Hardware Trace Point** option from the context menu.

    The stop trigger icon appears on the marker bar in red color (see Figure 8.2). The same option is also used to remove the stop trigger from the marker bar.

Figure 8.2 Hardware Start and Stop Tracepoints set in Source Code



11. Debug your application.

    The application halts at the beginning of the `main()` function.

12. Click **Resume** to resume the program execution.

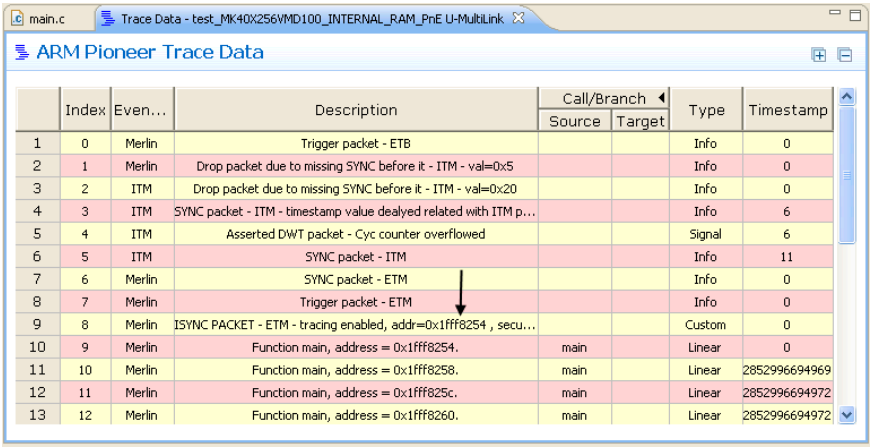
13. Wait for some time to let the application terminate.

14. From the menu bar, select **Profiler > ARM - Trace and Profile Results** to open the **Profile Results** view.

15. Expand the project name and click on the **Trace** hyperlink.

    The **ARM Pioneer Trace Data** page appears, as shown in Figure 8.3 and Figure 8.4.

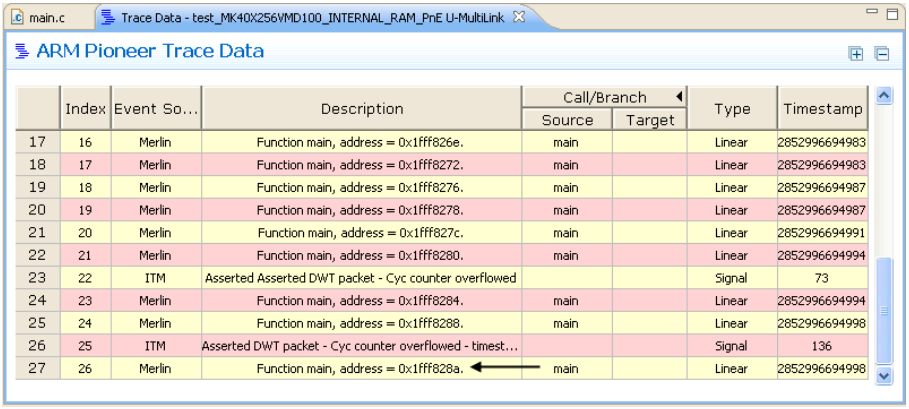
Figure 8.3 Trace Results After Setting Hardware Tracepoints in Source Code — Start Address



	Index	Even...	Description	Call/Branch Source	Call/Branch Target	Type	Timestamp
1	0	Merlin	Trigger packet - ETB			Info	0
2	1	Merlin	Drop packet due to missing SYNC before it - ITM - val=0x5			Info	0
3	2	ITM	Drop packet due to missing SYNC before it - ITM - val=0x20			Info	0
4	3	ITM	SYNC packet - ITM - timestamp value dealyed related with ITM p...			Info	6
5	4	ITM	Asserted DWT packet - Cyc counter overflowed			Signal	6
6	5	ITM	SYNC packet - ITM			Info	11
7	6	Merlin	SYNC packet - ETM			Info	0
8	7	Merlin	Trigger packet - ETM			Info	0
9	8	Merlin	ISYNC PACKET - ETM - tracing enabled, addr=0x1fff8254 , secu...			Custom	0
10	9	Merlin	Function main, address = 0x1fff8254.	main		Linear	0
11	10	Merlin	Function main, address = 0x1fff8258.	main		Linear	2852996694969
12	11	Merlin	Function main, address = 0x1fff825c.	main		Linear	2852996694972
13	12	Merlin	Function main, address = 0x1fff8260.	main		Linear	2852996694972

Figure 8.3 and Figure 8.4 shows the data file generated by the application in which the data has been collected after setting start and stop hardware tracepoints in the source code. In this data file, the **Description** field shows that trace starts collecting from the address, where you set start tracepoint, and trace collection stops before the address where you set stop tracepoint.

Figure 8.4 Trace Results After Setting Hardware Tracepoints in Source Code — Stop Address



	Index	Event So...	Description	Call/Branch Source	Call/Branch Target	Type	Timestamp
17	16	Merlin	Function main, address = 0x1fff826e.	main		Linear	2852996694983
18	17	Merlin	Function main, address = 0x1fff8272.	main		Linear	2852996694983
19	18	Merlin	Function main, address = 0x1fff8276.	main		Linear	2852996694987
20	19	Merlin	Function main, address = 0x1fff8278.	main		Linear	2852996694987
21	20	Merlin	Function main, address = 0x1fff827c.	main		Linear	2852996694991
22	21	Merlin	Function main, address = 0x1fff8280.	main		Linear	2852996694994
23	22	ITM	Asserted Asserted DWT packet - Cyc counter overflowed			Signal	73
24	23	Merlin	Function main, address = 0x1fff8284.	main		Linear	2852996694994
25	24	Merlin	Function main, address = 0x1fff8288.	main		Linear	2852996694998
26	25	ITM	Asserted DWT packet - Cyc counter overflowed - timest...			Signal	136
27	26	Merlin	Function main, address = 0x1fff828a. ←	main		Linear	2852996694998

NOTE Similarly, you can set hardware tracepoints from the **Disassembly** view. The only difference is that tracepoints are set when the program is in debug mode. After setting the tracepoints, you click **Resume** and collect trace and flat profile data.

From Trace and Profile Tab

This topic explains how to set comparator 1 and comparator 2 as hardware tracepoints for trace collection. You can take any combination of comparators to be used as hardware tracepoints. You simply need to specify the correct corresponding reference value against them.

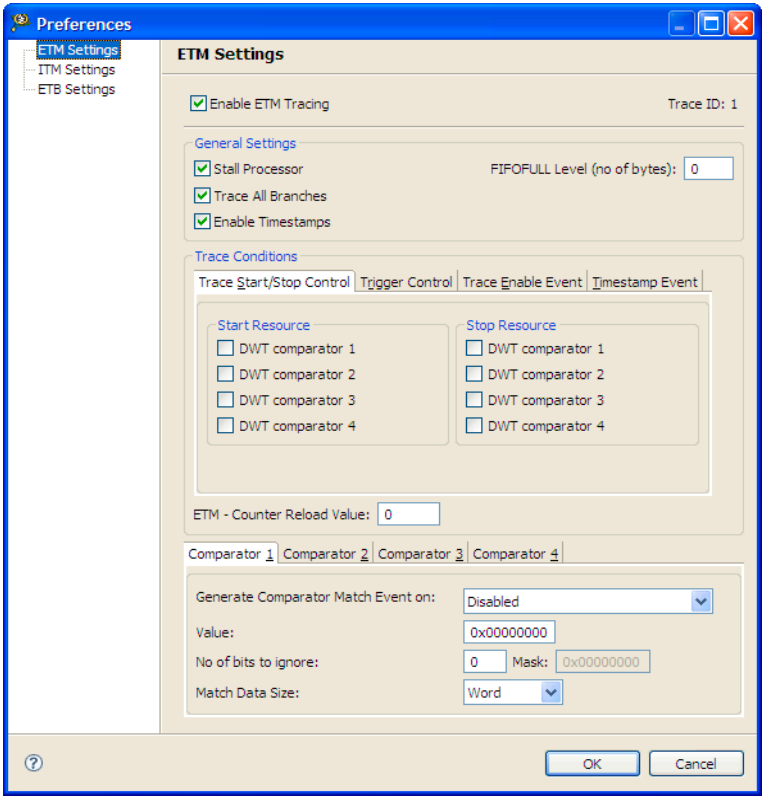
This topic uses the addresses of the instructions, of Listing 8.1, at which you set start and stop hardware tracepoint. These addresses will be set against comparator 1 and comparator 2 to collect the same trace results that appear on setting hardware tracepoints from the source code.

To set hardware tracepoints using the **Trace and Profile** tab and collect trace:

1. Build your project.
2. Open the **Debug Configurations** dialog box, and select your project in the tree structure.
3. Click the **Trace and Profile** tab, and check the **Enable Trace and Profile** checkbox.
4. Click the **Advanced Settings** button.

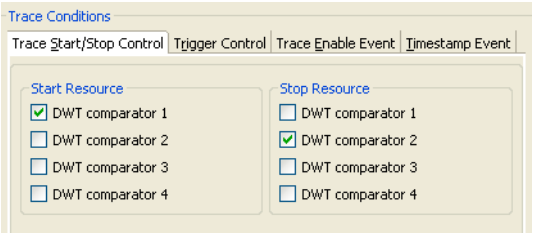
   The **Preferences** dialog box appears, as shown in Figure 8.5.

Figure 8.5 Preferences Dialog Box



5. In the **Trace Start/Stop Control** group, specify the following settings:
   a. Check the **DWT comparator 1** checkbox in the **Start Resource** group.
   b. Check the **DWT comparator 2** checkbox in the **Stop Resource** group.

Figure 8.6 Trace Start/Stop Control Settings

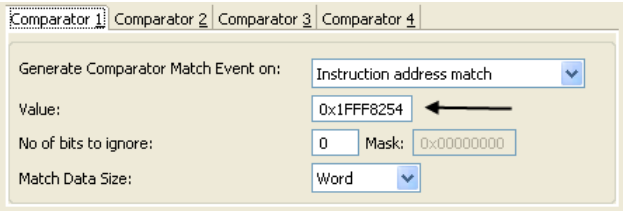


6. In the **Comparator 1** tab, specify the following settings:

a. Select the **Instruction address match** option in the **Generate Comparator Match Event on** drop-down list.

b. In the **Value** text box, specify the address of the instruction on which you want to set comparator 1. For example, set the address of the instruction where you set the start hardware tracepoint in Listing 8.1.

Figure 8.7 Setting Comparator 1



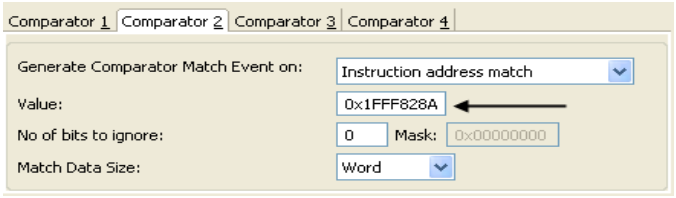
NOTE You can view the address of an instruction in the **Disassembly** view. Open the **Disassembly** view while the application is running, navigate to the instruction, and take the first address.

7. In the **Comparator 2** tab, specify the following settings:

   a. Select the **Instruction address match** option in the **Generate Comparator Match Event on** drop-down list.

   b. In the **Value** text box, specify the address of the instruction on which you want to set comparator 2. For example, set the address of the instruction where you set the stop hardware tracepoint in Listing 8.1.

Figure 8.8 Setting Comparator 2



8. Click **OK** to close the **Preferences** dialog box.
9. Click **Apply** to save the settings.
10. Click **Debug** to debug the application.
11. Click **Resume** to resume the program execution.
12. Wait for some time to let the application terminate.

13. From the menu bar, select **Profiler > ARM - Trace and Profile Results** to open the **Profile Results** view.
14. Expand the project name and click on the **Trace** hyperlink.

    The **ARM Pioneer Trace Data** page appears displaying the same results, as shown in Figure 8.3 and Figure 8.4.

Setting Software Tracepoints

Software tracepoints use an interrupt to start and stop trace. To use software tracepoints for collecting trace data on the Kinetis target, you need to add the `sa_handle.c` file to your project and modify the `kinetis_sysinit.c` file. The `sa_handle.c` file contains the handler for the interrupt inside the code which enables/disables trace collection. The `kinetis_sysinit.c` file contains the code that instructs the hardware to use a specific handler for each type of interrupt of the application.

You also need to edit the linker control file of your project in which you add the `.swtp` section. These sections are used by the software tracepoints mechanism to reserve a memory zone to save a table with tracepoints records which are parsed inside the `sa_handler` to determine whether the interrupt occurred for a start or stop tracepoint.

To set software tracepoints in the source code and collect trace data on the Kinetis target:

1. Create a stationary Kinetis project.
2. Add the `sa_handler.h` file to the project.
   a. In the **CodeWarrior Projects** view, select the **Project_Headers** folder of your project.
   b. Right-click and select the **Add Files** option form the context menu.

      The **Open** dialog box appears.
   c. Browse to the `<MCU CW Installation Folder>\MCU\(CodeWarrior_Examples)\ARM_Examples\sa_software_tracepoints_marconi_x256\Project_Headers` location.
   d. Select `sa_handler.h` and click **Open**.

      The `sa_handler.h` file gets added to the project.
3. Add the `sa_handle.c` file to your project.
   a. In the **CodeWarrior Projects** view, select the **Sources** folder of your project.
   b. Right-click and select the **Add Files** option form the context menu.
   c. Browse to the `<MCU CW Installation Folder>\MCU\morpho_sa\sasdk\support\swtp` location.
   d. Select the `sa_handle.c` file and click **Open**.

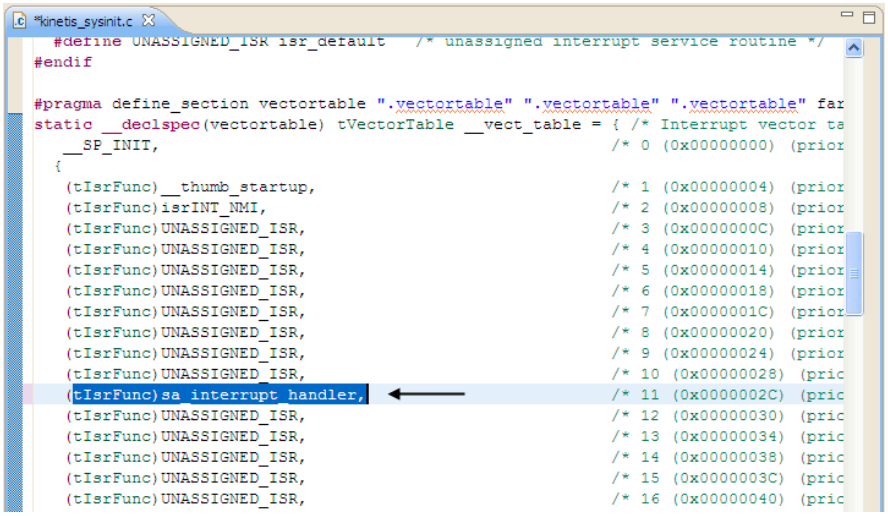
4. Expand the **Project_Settings > Startup_Code** folder of your project and double-click on the `kinetis_sysinit.c` file to display its contents in the editor area.
5. Edit the `kinetis_sysinit.c` file.
   a. Include the following header file in the source code:

      ```
      #include "sa_handler.h"
      ```

   b. Add the following line in the interrupt vector section (see Figure 8.9):

      ```
      (tIsrFunc)sa_interrupt_handler,
      ```

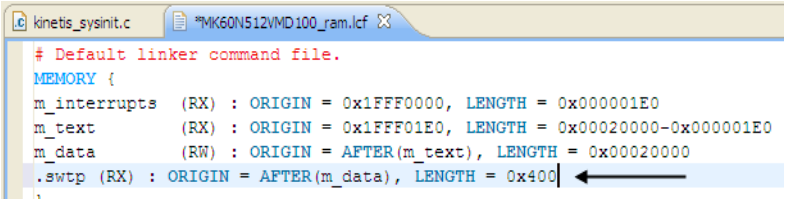
Figure 8.9 Interrupt Vector Section



6. Save the `kinetis_sysinit.c` file.
7. Edit the linker control file of your project.
   a. Expand **Project_Settings > Linker_Files** in the **CodeWarrior Projects** view.
   b. Double-click the `<project>_ram.lcf` file to open its contents in the editor area.
   c. Add the following statement in the memory section (see Figure 8.10):

      ```
      .swtp (RX) : ORIGIN = AFTER(m_data), LENGTH = 0x400
      ```

Figure 8.10 Memory Section of Linker File



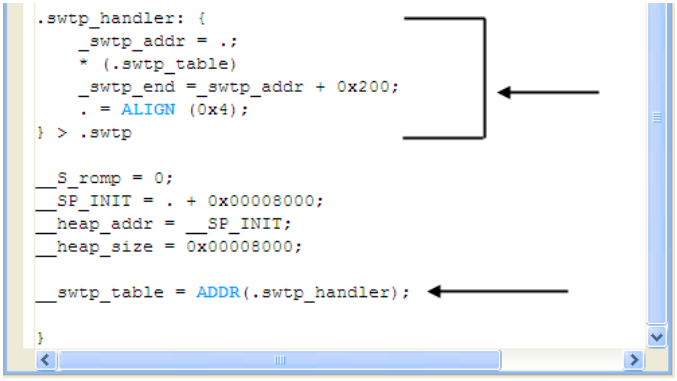
d. Add the `.swtp` section containing the following statements (see Figure 8.11)

```
.swtp_handler: {
 _swtp_addr = .;
    * (.swtp_table)
    _swtp_end =_swtp_addr + 0x200;
    . = ALIGN (0x4);
} > .swtp
```

e. Add the following statement at the end of the file (see Figure 8.11). This will store the address of the software tracepoint handler.

```
__swtp_table = ADDR(.swtp_handler);
```

Figure 8.11 `.swtp` Section of Linker File



f. Save `<project>_ram.lcf`.

8. Save and build the project.
9. Enable Trace and Profile.

a. Open the **Debug Configurations** dialog box, and select your project in the tree structure.

b. Click the **Trace and Profile** tab, and check the **Enable Trace and Profile** checkbox.

c. Keep **Enable ETM Tracing** checked.

d. Check **Enable Continuous Trace collection** and uncheck all other checkboxes.

e. Click **Apply** to save the settings, and close the **Debug Configurations** dialog box.

10. Set start and stop software tracepoints.

    a. In the editor area, select the following statement:

    ```
    int counter = 0;
    ```

    b. Right-click on the marker bar, select the **Toggle Trace Start Point > Software Trace Point** option from the context menu. The same option is also used to remove the start trigger from the marker bar.

    c. In the editor area, select the following statement:

    ```
    for(;;)
    ```

    d. Right-click on the marker bar, select the **Toggle Trace Stop Point > Software Trace Point** option from the context menu.

NOTE It is recommended to set Inlining as **Off** before you debug the project. To set Inlining as **Off**, right-click on your project and select the **Properties** option. The **Properties** page of you project appears. Expand **C/C++ Build** and select **Settings**. In the **Tool Settings** tab, expand **ARM Compiler** and select **Optimization**. Select **Off** from the **Inlining** drop-down box on the right-side of the tab.

11. Debug the application.

12. Collect trace and profile results.

    a. Click **Resume**.

    b. Click **Suspend** after a few seconds.

13. From the menu bar, select **Profiler > ARM - Trace and Profile Results** to open the **Profile Results** view.

14. Expand the project name and click on the **Trace** hyperlink.

    The **ARM Pioneer Trace Data** page appears, as shown in Figure 8.12.

Figure 8.12 Trace Results After Setting Software Tracepoints — Beginning of Trace

	Index	Event So...	Description	Call/Branch		Type	Timestamp
				Source	Target		
1	0	Merlin	Trigger packet - ETB			Info	0
2	1	Merlin	SYNC packet - ETM			Info	0
3	2	Merlin	ISYNC PACKET - ETM - tracing enabled, addr=0x1fff832e , ...			Custom	0
4	3	Merlin	Trigger packet - ETM			Info	0
5	4	Merlin	Function setupTracepoints, address = 0x1fff832e.	setupTracepo...		Linear	0
6	5	Merlin	Function setupTracepoints, address = 0x1fff8330.	setupTracepo...		Linear	131018785
7	6	Merlin	Function setupTracepoints, address = 0x1fff8336.	setupTracepo...		Linear	131018785
8	7	Merlin	Function setupTracepoints, address = 0x1fff8338.	setupTracepo...		Linear	131018791
9	8	Merlin	Function setupTracepoints, address = 0x1fff833a.	setupTracepo...		Linear	131018791
10	9	Merlin	Branch from setupTracepoints to setupTracepoints. Source ...	setupTracepo...	setupTracepo...	Branch	131018791
11	10	Merlin	Function setupTracepoints, address = 0x1fff837a.	setupTracepo...		Linear	131018797
12	11	Merlin	Function setupTracepoints, address = 0x1fff837c.	setupTracepo...		Linear	131018797
13	12	Merlin	Branch from setupTracepoints to sa_interrupt_handler. Sou...	setupTracepo...	sa_interrupt_...	Branch	131018809
14	13	Merlin	Function sa_interrupt_handler, address = 0x1fff8390.	sa_interrupt_...		Linear	131018812
15	14	Merlin	Function sa_interrupt_handler, address = 0x1fff8394	sa_interrupt_...		Linear	131018812

Figure 8.12 and Figure 8.13 show the data file generated by the application in which the data has been collected after setting start and stop software tracepoints in the source code.

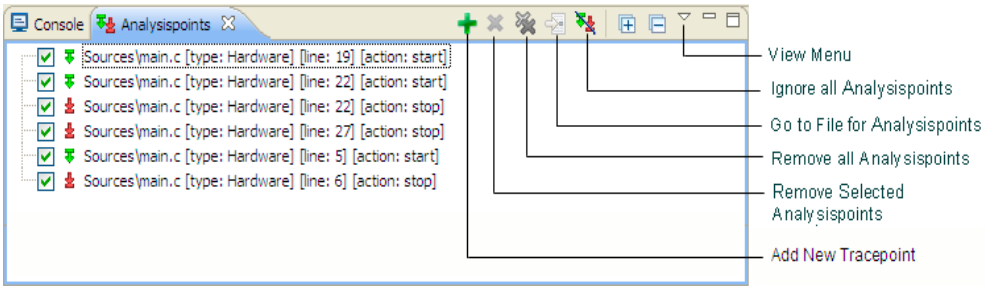
Figure 8.13 Trace Results After Setting Software Tracepoints — End of Trace

	Index	Event So...	Description	Call/Branch		Type	Timestamp
				Source	Target		
1068	1067	Merlin	Function sa_interrupt_handler, address = 0x1fff8388.	sa_interrupt_...		Linear	337644438
1069	1068	Merlin	Function sa_interrupt_handler, address = 0x1fff838a.	sa_interrupt_...		Linear	337644438
1070	1069	Merlin	Branch from sa_interrupt_handler to setupTracepoints. Sou...	sa_interrupt_...	setupTracepo...	Branch	337644438
1071	1070	Merlin	Function setupTracepoints, address = 0x1fff8224.	setupTracepo...		Linear	337644442
1072	1071	Merlin	Function setupTracepoints, address = 0x1fff8228.	setupTracepo...		Linear	337644452
1073	1072	Merlin	Function setupTracepoints, address = 0x1fff8230.	setupTracepo...		Linear	337644455
1074	1073	Merlin	Function setupTracepoints, address = 0x1fff823c.	setupTracepo...		Linear	337644458
1075	1074	Merlin	Function setupTracepoints, address = 0x1fff8246.	setupTracepo...		Linear	337644461
1076	1075	Merlin	Function setupTracepoints, address = 0x1fff824a.	setupTracepo...		Linear	337644461
1077	1076	Merlin	Function setupTracepoints, address = 0x1fff824e.	setupTracepo...		Linear	337644465
1078	1077	Merlin	Function setupTracepoints, address = 0x1fff825a.	setupTracepo...		Linear	337644468
1079	1078	Merlin	Function setupTracepoints, address = 0x1fff825c.	setupTracepo...		Linear	337644468
1080	1079	Merlin	Function setupTracepoints, address = 0x1fff825e.	setupTracepo...		Linear	337644473
1081	1080	Merlin	Function setupTracepoints, address = 0x1fff8262.	setupTracepo...		Linear	337644473
1082	1081	Merlin	Trace buffer read finished.			Info	337644473

Viewing Tracepoints

You can view tracepoints in the **Analysispoints** view that displays the attributes of the tracepoints set in source code or/and assembly code. To view the attributes of the tracepoints, select **Window > Show View > Other > Software Analysis > Analysispoints**. The **Analysispoints** view is displayed, as shown in Figure 8.14.

Figure 8.14 Analysispoints View



The **Analysispoints** view displays the following attributes of the tracepoints set from the source code:

- Name of the file where tracepoint is set
- Type of the tracepoint, that is software or hardware
- Line number where tracepoint is set
- Action of the tracepoint, that is start or stop

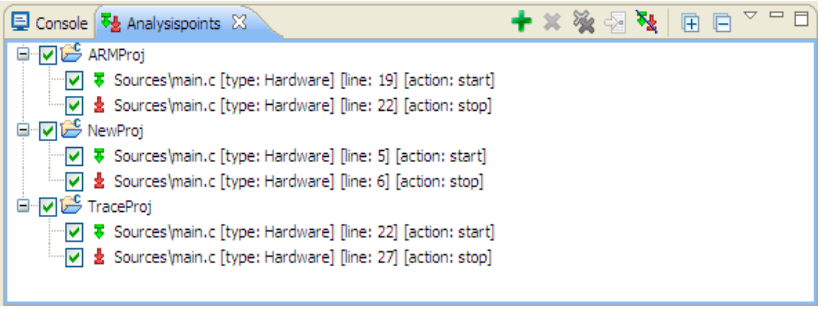
The **Analysispoints** view displays the following attributes of the tracepoints set from the **Disassembly** view:

- Name of the file where tracepoint is set
- Type of the tracepoint, that is software or hardware
- Address where tracepoint is set
- Line number where tracepoint is set
- Action of the tracepoint, that is start or stop

You can use the **Analysispoints** view to perform the following actions:

- View full path of tracepoint attribute — To view the complete path of the files of the tracepoint attribute, click **View Menu** in the **Analysispoints** view and select the **Show Full Paths** option. Select the option again to hide the complete path.
- Group tracepoints — You can group the attributes of the tracepoints by tracepoint type, files, projects, or working sets. To group the attributes, click **View Menu > Group By** and select the necessary option. Figure 8.15 shows the tracepoint attributes in the **Analysispoints** view which are grouped by projects. You can ungroup the attributes by selecting the same option again.

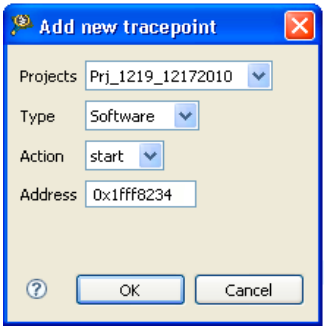
Figure 8.15 Analysispoints View — Group by Projects



- Add new tracepoint — You can use the **Analysispoints** view to add a tracepoint on the address of an instruction. To add an address tracepoint:

a. Click the **Add new tracepoint** icon to display the **Add new tracepoint** dialog box.

Figure 8.16 Add New Tracepoint Dialog Box



b. Select the required project from the **Projects** text box.

c. Select the type of the tracepoint from the **Type** text box.

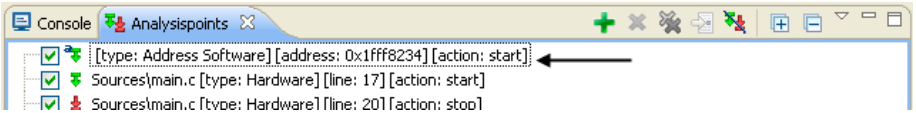
d. Select the action of the tracepoint, that is whether start or stop, from the **Action** text box.

e. Enter the address where you want to set the tracepoint in the **Address** text box.

f. Click **OK**.

The tracepoint is set and appears in the **Analysispoints** view.

Figure 8.17 Address Tracepoint Set in Analysispoints View

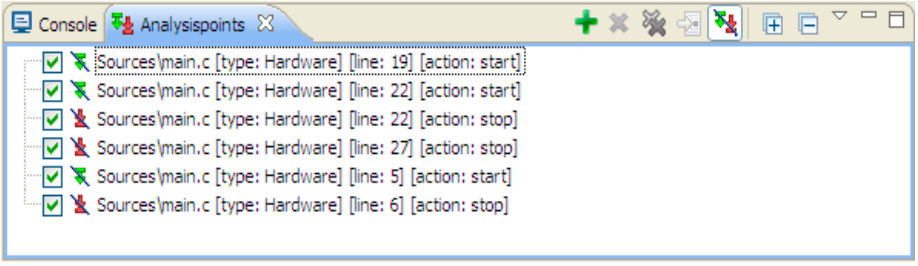


NOTE The address tracepoint set from **Analysispoints** view does not have a source file associated with it, therefore, the tracepoint when set is not displayed in the source code. You can view it only in the **Analysispoints** view.

- Enable/Disable tracepoints — You can enable/disable a tracepoint from **Analysispoints** view by checking/clearing it. The unchecked attribute indicates the disabled tracepoint. A disabled tracepoint will have no effect during the collection of trace data.

  You can also use the **Ignore all Analysispoints** icon to disable all the tracepoints without manually selecting them (see Figure 8.18). Click **Ignore All Analysispoints** again to enable the tracepoints.

Figure 8.18 Disabling Tracepoints from Analysispoints View



- Navigate to tracepoint line — The **Analysispoints** view shows where start and stop tracepoints are set in the editor area or the **Disassembly** view. It helps you navigate to the source code or assembly code where the tracepoint is set. If you want to go to the specific line of the source code where the tracepoint is set, select the tracepoint attribute in the **Analysispoints** view and click the **Go To File for Analysispoints** icon. This option will navigate you to that line.
- Remove tracepoints — To remove a tracepoint, select the tracepoint attribute and click the **Remove Selected Analysispoint** icon. To remove all the tracepoints, click the **Remove All Analysispoints** icon.

9

Profiling on ColdFire V4e Target

The ColdFire V2 - V4 targets collect the profiling information by using the profiling system. These targets do not have the hardware capability to collect trace. The profiling system consists of three main components:

- The statically-linked code library of compiled code containing the profiler. This library is created and distributed within the CodeWarrior product.
- An Application Programming Interface (API) to control the profiler
- The Simple Profiler Viewer to view and analyze the profile results

The profiling system collects information using a profiler, which analyzes the amount of time a program spends performing various tasks and detects bottlenecks between the functions/routines. This type of information-tracking can be useful for determining the cost of calling a routine. The cost of a routine call is not only the time spent in the routine, it is also the time spent in its children, that is the subsidiary routines it calls, the routines they call, and so on.

To use the profiler for profiling an application, perform the following actions:

- Include the profiler library and files in the CodeWarrior project — Include Profiler Library and Files
- Configure your project to turn on profiling — Configure Project for Profiling
- Modify the application source code to make use of the profiler API — Modify Source Code
- Debug the application and collect profiling information — Debug Application and Collect Profiling Information
- Open the Simple Profiler Viewer to view the results — View Profiling Results

This process of profiling gets you all the data you need to perform a professional-level analysis of the runtime behavior of your application.

NOTE The process of profiling is same for ColdFire V2 - V4 targets.

Include Profiler Library and Files

The profiling code that keeps track of the time spent in a routine exists in a series of libraries. You need to add a profiler library and four profiler files to your project to use the profiler.

To add the profiler files:

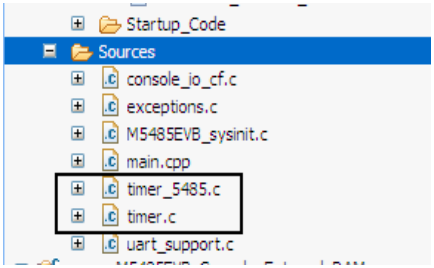
1. Select the **Sources** folder of your project in the **CodeWarrior Projects** view.
2. Right-click and select the **Add Files** option.

   The **Open** dialog box appears.
3. Browse to the `<CodeWarrior_Installation_Folder>\MCU\ColdFire_Support\Profiler\Support\` location
4. Select `timer.c` and `timer_5485.c` with `Ctrl` key pressed and click **Open**.

   The **File and Folder Import** dialog box appears.
5. Select the **Copy the files and directories** option and click **OK**.

   The files are added to the **Sources** folder (see Figure 9.1).

Figure 9.1 Profiler Files Added to Sources Folder



6. Select the **Project_Headers** folder of your project in the **CodeWarrior Projects** view.
7. Right-click and select the **Add Files** option.

   The **Open** dialog box appears.
8. Browse to the `<CodeWarrior_Installation_Folder>\MCU\ColdFire_Support\Profiler\include\` location.
9. Select `Profiler.h` and `ProfilerTimer.h` file with `Ctrl` key pressed and click **Open**.

   The **File and Folder Import** dialog box appears.

10. Select the **Copy the files and directories** option and click **OK**.

    The files are added to the **Project_Headers** folder.

To add the profiler library, `ProfileLibrary_CF_Runtime.a`:

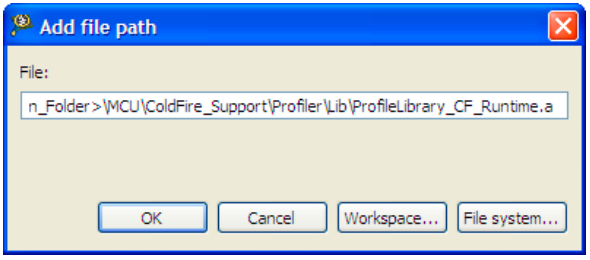
1. Select your project in the **CodeWarrior Projects** view.
2. Right-click and select the **Properties** option from the context menu.

   The **Properties for <project name>** dialog box appears.
3. Expand the **C/C++ Build** node in the tree structure on the left, and select the **Settings** option.
4. In the **Tool Settings** tab page, expand the **ColdFire Linker** node in the left tree structure.
5. Select the **Input** option.
6. In the **Library Files** section on the right side of the tab page, click  button to open the **Add file path** dialog box.
7. Click the **File system** button.

   The **Open** dialog box appears.
8. Browse to the `<CodeWarrior_Installation_Folder>\MCU\ColdFire_Support\Profiler\Lib` location.
9. Select the `ProfileLibrary_CF_Runtime.a` file and click **Open**.

   The complete path of the file appears in the **Add file path** dialog box (see Figure 9.2).

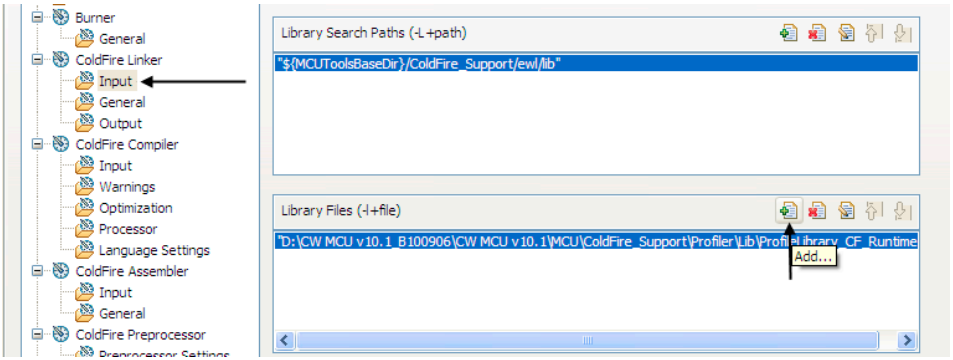
Figure 9.2 Add File Path Dialog Box



10. Click **OK** in the **Add file path** dialog box.

    The profiler library gets added to the **Library Files** section (see Figure 9.3).

Figure 9.3 Tool Settings Page of Properties Dialog Box



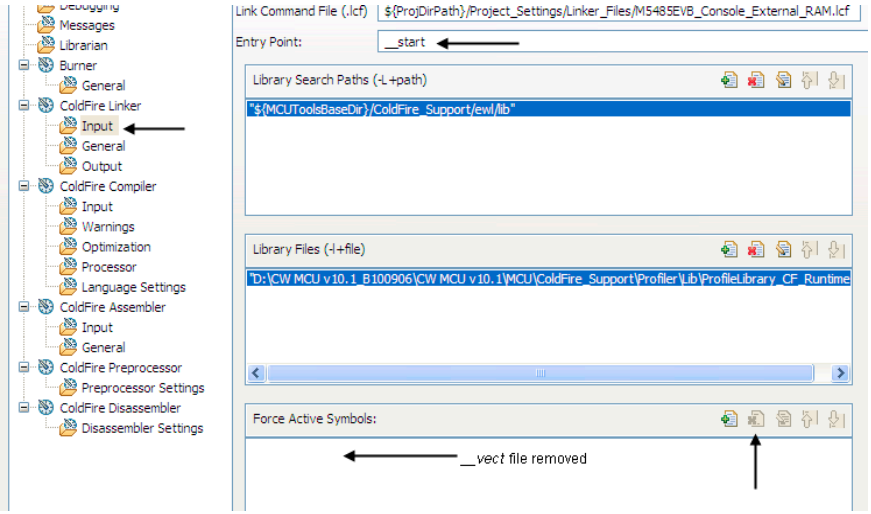
Configure Project for Profiling

You need to configure the compiler and linker settings of your project. The compiler and linker using the profiler library generate a new version of your program ready for profiling. While it runs, the profiler generates data.

To configure the project for profiling:

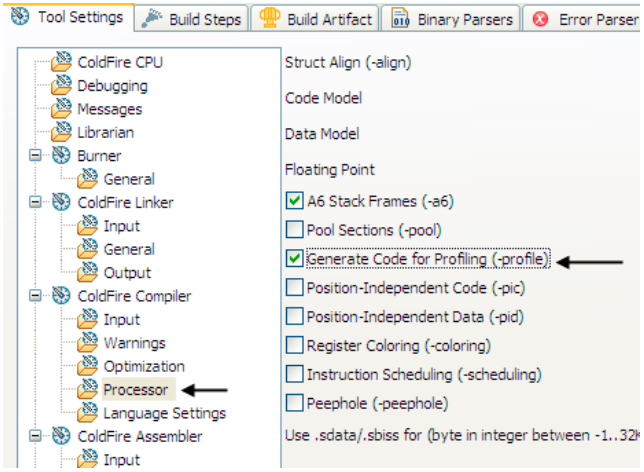
1. In the **Tool Settings** tab page, modify the **Entry Point** value as `__start`.
2. In the **Force Active Symbols** section, remove the entry for `__vect` by clicking  (see Figure 9.4).

Figure 9.4 Configuration of ColdFire Linker Properties



3. Select the **ColdFire Compiler** node in the left tree structure.
4. Select the **Processor** option.
5. Check the **Generate Code for Profiling (-profile)** checkbox (see Figure 9.5).

Figure 9.5 Configuration of ColdFire Compiler Properties

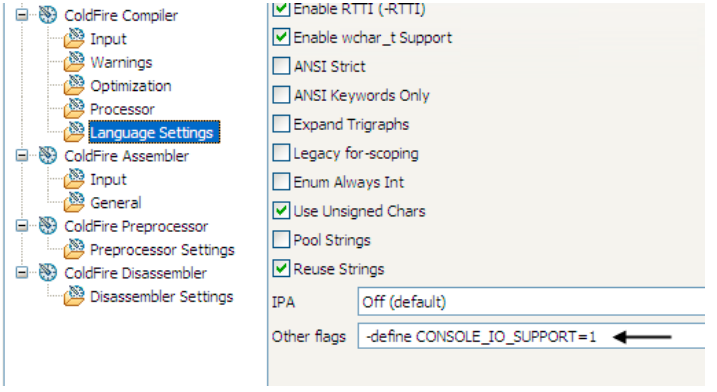


6. Select the **Language Settings** option in the left tree structure.

7. Ensure that the **Other flags** text box on the right side has the following entry:

   ```
   -define CONSOLE_IO_SUPPORT=1
   ```

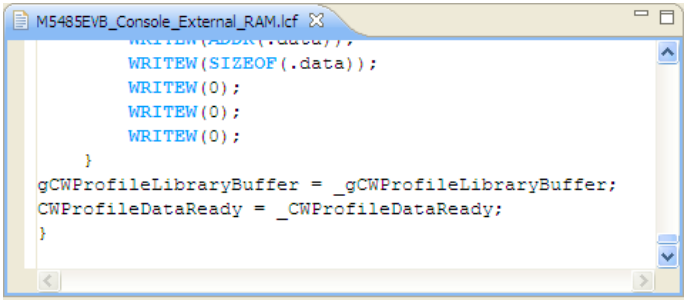
Figure 9.6 Configuring Language Settings



8. Click **Apply** to apply the settings.
9. Click **OK** to close the **Properties of <project name>** dialog box.
10. Expand the **Project_Settings > Linker_Files** node of your project in the **CodeWarrior Projects** view.
11. Double-click the `M5485EVB_Console_External_RAM.lcf` file to open its contents in the editor area.
12. Add the following statements at the end of the file before the last closing parenthesis:

    ```
    gCWProfileLibraryBuffer = _gCWProfileLibraryBuffer;
    CWProfileDataReady = _CWProfileDataReady;
    ```

Figure 9.7 Modification in M5485EVB_Console_External_RAM.lcf File



13. Save and close the file.

Modify Source Code

Replace the source code of the `main.cpp` file of your project with the source code of Listing 9.1. To open the `main.cpp` file:

1. Expand the **Sources** node of your project in the **CodeWarrior Projects** view.
2. Double-click on `main.cpp` to open its contents in the editor area.

You need to modify the source code such that it uses the profiler API to do profiling. This source code uses:

- The `ProfilerInit()` function to initialize the profiler — With profiling on, the compiler generates all the necessary code so that every routine calls the profiler.
- The `ProfilerDump()` function to write the profiling data into the `profiledump.mwp` file
- The `ProfilerTerm()` function to terminate the profiler — If you initialize the profiler and then exit the program without terminating the profiler, timers may be left running that could crash the machine.

The source files that make calls to the profiler API must include the appropriate header file for your target. The header file that the ColdFire V4e target uses for profiling is:

```
#include <Profiler.h>
```

Listing 9.1 Sample source code used for profiling

```
#include <stdio.h>
#include <Profiler.h>

/*---------------Loop constant definition--------------------*/
# define LOOP_1  0x0001FFFF
# define LOOP_2  0x0002FFFF
# define LOOP_3  0x0003FFFF
# define LOOP_4  0x0001FFFF
# define LOOP_5  0x0002FFFF
# define LOOP_6  0x0003FFFF
# define LOOP_7  0x0001FFFF
# define LOOP_8  0x0002FFFF
# define LOOP_9  0x0003FFFF
# define LOOP_10 0x0001FFFF

/*----------------------Global function Declaration-------------*/
void fn1(int loop);
void fn2(int loop);
void fn3(int loop);
void fn4(int loop);
void fn5(int loop);
void fn6(int loop);
```

```
volatile int fn7( int a );

/*-----------------Class One Definition-----------------------*/
class One
{
public:
       void one_fn1(int loop);
};

/*---------------Class One's function Definition--------------*/
void One::one_fn1(int loop)
{
   unsigned int i,j;
   printf("In function : One::one_fn1 \n\r");
   for( i = 0 ; i < loop ; i++)
   {
      for( j = 0; j < 0x12 ; j++ )
      {}
   }
}
/*----------------Class Two Defination-------------------------*/
class Two
{
private:
        One test;
public:
        void two_fn1(int loop);
        void two_fn2(int loop);
};

/*---------------Class Two's function Defination----------------*/
void Two::two_fn1(int loop)
{
    unsigned int i,j;
    printf("In function : Two::two_fn1 \n\r");
    for( i = 0 ; i < loop ; i++)
    {
       for( j = 0; j < 0x12 ; j++ )
       {}
    }
    two_fn2(LOOP_9);
}

void Two::two_fn2(int loop)
{
      unsigned int i,j;
      printf("In function : Two::two_fn2 \n\r");
      for( i = 0 ; i < loop ; i++)
```

```
      {
          for( j = 0; j < 0x12 ; j++ )
          {}
      }
      test.one_fn1(LOOP_10);
}

/*----------------Global function Defination--------------------*/
volatile int fn7( int a )
{
  int i;
  printf("In function : fn7 \n\r");
  if( a == 1 )
  {
    for( i = 0; i < 240000; i ++ );
    return 1;
  }
  else
    return a * fn7( a - 1 );
}

void fn6(int loop)
{
    unsigned int i,j;
    printf("In function : fn6 \n\r");
    for( i = 0 ; i < loop ; i++)
    {
        for( j = 0; j < 0x12 ; j++ )
        {}
    }
}

void fn5(int loop)
{
     unsigned int i,j;
     printf("In function : fn5 \n\r");
     for( i = 0 ; i < loop ; i++)
     {
        for( j = 0; j < 0x12 ; j++ )
        {}
     }
}

void fn4(int loop)
{
     unsigned int i,j;
     printf("In function : fn4 \n\r");
    for( i = 0 ; i < loop ; i++)
```

```
    {
       for( j = 0; j < 0x12 ; j++ )
       {}
    }
}

void fn3(int loop)
{
    unsigned int i,j;
    printf("In function : fn3 \n\r");
    for( i = 0 ; i < loop ; i++)
    {
       for( j = 0; j < 0x12 ; j++ )
       {}
    }
}

void fn2(int loop)
{
    unsigned int i,j;
    printf("In function : fn2 \n\r");
    for( i = 0 ; i < loop ; i++)
    {
       for( j = 0; j < 0x12 ; j++ )
       {}
    }
    fn3(LOOP_3);
}

void fn1(int loop)
{
    unsigned int i,j;
    printf("In function : fn1 \n\r");
    for( i = 0 ; i < loop ; i++)
    {
       for( j = 0; j < 0x12 ; j++ )
       {}
    }
    fn2(LOOP_2);
}

/*---------------main function Definition----------------------*/

int main()
{
   One a;
   Two b;
   printf("Profiler yet to start C\n\r");
```

```
// Following section of code initialises the profiler
   ProfilerInit(collectDetailed, bestTimeBase, 5,20);
   ProfilerClear();
   ProfilerSetStatus(1);

   printf("Profiler has just started \n\r");

// Code to be profiled
   fn1(LOOP_1);
   fn4(LOOP_4);
   fn5(LOOP_5);
   fn6(LOOP_6);
   fn7(4);
   a.one_fn1(LOOP_7);
   b.two_fn1(LOOP_8);

   printf("Profiling is just to end \n\r");

// Following section of code terminates the profiler
   ProfilerSetStatus(0);
   ProfilerDump("profiledump");
   ProfilerTerm();

   printf("I have all the Profile Data \n\r");

   return 0;
}
```

Debug Application and Collect Profiling Information

After configuring the project and modifying the source code to use the profiler API, build and debug your project.

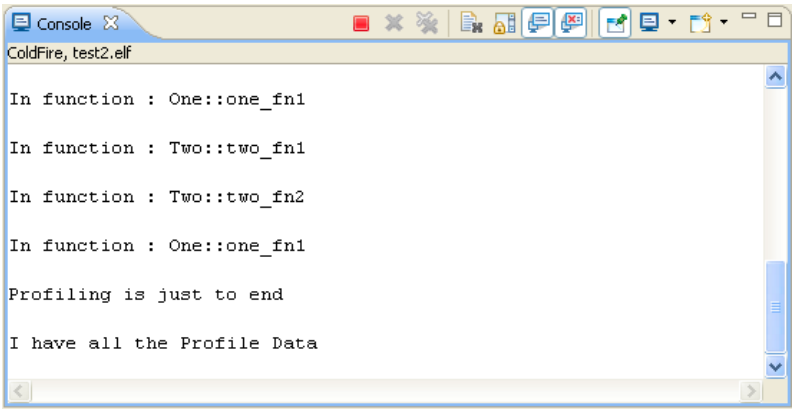
The project halts at the `main()` function.

NOTE If you are using the **Debug Configurations** dialog box to debug the project, select the Console External RAM configuration in the tree structure. For example, <*project_name*>*_M5485EVB_Console_External_RAM_PnE USB BDM*.

To collect profiling information:

1. Select the **Console** view and click the **Pin Console** button.
2. Click **Resume** to resume the program execution.
3. Notice the program output messages in the **Console** view and wait until the `I have all the Profile Data` message is displayed (see Figure 9.8).
4. Click **Terminate** to stop the program execution.

Figure 9.8 Console View



View Profiling Results

You use the **Simple Profiler Data Viewer** to view the profiler results. This viewer helps analyze the data of the executed program and determine what changes are appropriate to improve the performance of the application. Using the data display, you can:

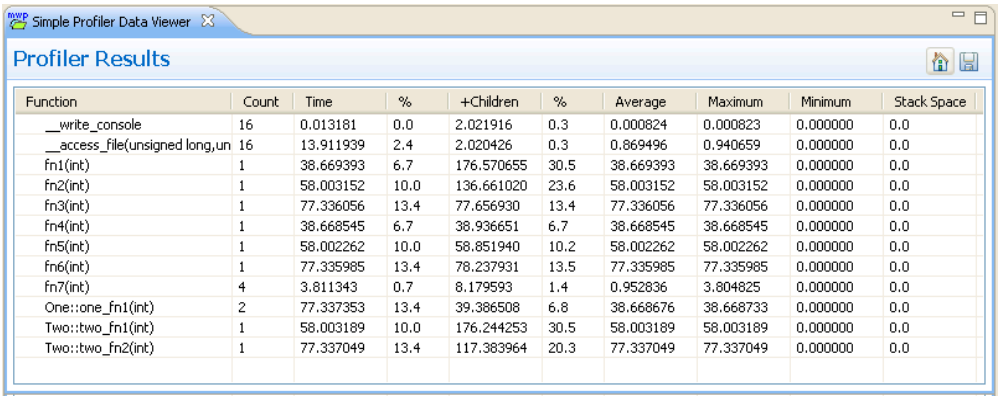
- open multiple profiles simultaneously to compare different versions of the profiled source code
- identify trouble spots in the source code
- view flat, tree (detailed), or class-based data

To view the profiling results:

1. Select your project in the **CodeWarrior Projects** view and press the `F5` key to refresh it.
2. Expand the **M5485EVB_Console_External_RAM** folder and double-click the `profiledump.mwp` file.

   The **Simple Profiler Data Viewer** appears, as shown in Figure 9.9.

Figure 9.9 Simple Profiler Data Viewer



Function	Count	Time	%	+Children	%	Average	Maximum	Minimum	Stack Space
__write_console	16	0.013181	0.0	2.021916	0.3	0.000824	0.000823	0.000000	0.0
__access_file(unsigned long,un	16	13.911939	2.4	2.020426	0.3	0.869496	0.940659	0.000000	0.0
fn1(int)	1	38.669393	6.7	176.570655	30.5	38.669393	38.669393	0.000000	0.0
fn2(int)	1	58.003152	10.0	136.661020	23.6	58.003152	58.003152	0.000000	0.0
fn3(int)	1	77.336056	13.4	77.656930	13.4	77.336056	77.336056	0.000000	0.0
fn4(int)	1	38.668545	6.7	38.936651	6.7	38.668545	38.668545	0.000000	0.0
fn5(int)	1	58.002262	10.0	58.851940	10.2	58.002262	58.002262	0.000000	0.0
fn6(int)	1	77.335985	13.4	78.237931	13.5	77.335985	77.335985	0.000000	0.0
fn7(int)	4	3.811343	0.7	8.179593	1.4	0.952836	3.804825	0.000000	0.0
One::one_fn1(int)	2	77.337353	13.4	39.386508	6.8	38.668676	38.668733	0.000000	0.0
Two::two_fn1(int)	1	58.003189	10.0	176.244253	30.5	58.003189	58.003189	0.000000	0.0
Two::two_fn2(int)	1	77.337049	13.4	117.383964	20.3	77.337049	77.337049	0.000000	0.0

The **Simple Profiler Data Viewer** displays profiling information in the form of a table. Table 9.1 describes the fields of the profiling information.

Table 9.1 Description of Profiler Results

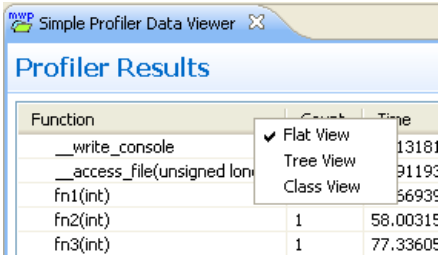
Name	Description
Function	Name of the function or routine.
Count	Number of times the function was called.
Time	Time spent in the function itself without counting any time in functions called by this function.
%	Percentage of the total time spent in the function.
+Children	Time spent in the function and all the functions it calls.
%	Percentage of the total time spent in the function and all the functions it calls.
Average	Average time for each function invocation, that is **Time** divided by the number of times the function was called.
Maximum	Longest time for an invocation of the function.
Minimum	Shortest time for an invocation of the function.
Stack Space	Largest size (in bytes) of the stack when the function is called.

The **Simple Profiler Data Viewer** displays profiling information in three different ways:

- Flat View
- Tree View
- Class View (relevant only for C++ projects)

To switch to these views, right-click on any column of the **Simple Profiler Data Viewer** table and select the necessary option from the **View** context menu (see Figure 9.10).

Figure 9.10 Different Views of Profile Results



Flat View

The **Flat** view is the default view which displays the summary of a complete and non-hierarchical list of each function profiled. No matter what calling path was used to reach a function, the profiler combines all the data for the same function and displays it on a single line. Figure 9.9 shows the **Flat** view of the profiler results.

The **Flat** view is particularly useful for comparing functions to check which function takes the longest time to execute. The **Flat** view is also useful for finding a performance problem with a small function that is called from many different places in the program. This view helps you look for the functions that make heavy demands in time or raw number of calls.

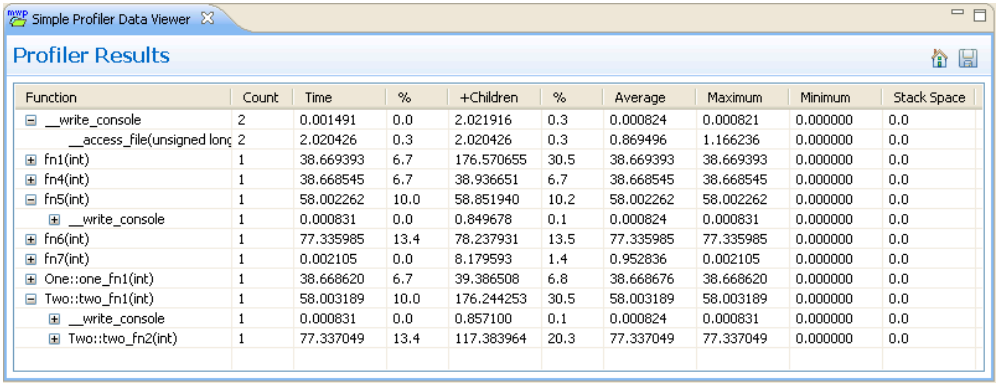
Tree View

The **Tree** view displays the detailed profile data, as shown in Figure 9.10. For example, details of the functions called by a particular function, or the details of the instructions executed in a function. This means that a function may appear more than once in the profile if it called from different functions.

The **Tree** view is useful for detecting design problems in code. It lets you see the functions that are called from other functions and also how often those functions are called. Armed with knowledge of your code's underlying design, you may discover flow-control problems.

You can use the **Expand All** or **Collapse All** options in the **View** context menu to open or close the entire hierarchy at once. These options are available only when you select the **Tree View** or **Class View** options.

Figure 9.11 Simple Profiler Data Viewer — Tree View



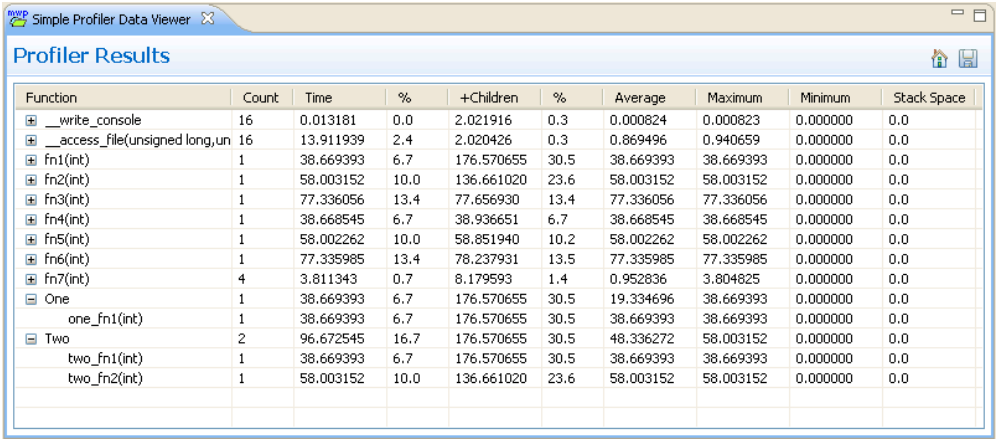
Function	Count	Time	%	+Children	%	Average	Maximum	Minimum	Stack Space
__write_console	2	0.001491	0.0	2.021916	0.3	0.000824	0.000821	0.000000	0.0
__access_file(unsigned long	2	2.020426	0.3	2.020426	0.3	0.869496	1.166236	0.000000	0.0
fn1(int)	1	38.669393	6.7	176.570655	30.5	38.669393	38.669393	0.000000	0.0
fn4(int)	1	38.668545	6.7	38.936651	6.7	38.668545	38.668545	0.000000	0.0
fn5(int)	1	58.002262	10.0	58.851940	10.2	58.002262	58.002262	0.000000	0.0
__write_console	1	0.000831	0.0	0.849678	0.1	0.000824	0.000831	0.000000	0.0
fn6(int)	1	77.335985	13.4	78.237931	13.5	77.335985	77.335985	0.000000	0.0
fn7(int)	1	0.002105	0.0	8.179593	1.4	0.952836	0.002105	0.000000	0.0
One::one_fn1(int)	1	38.668620	6.7	39.386508	6.8	38.668676	38.668620	0.000000	0.0
Two::two_fn1(int)	1	58.003189	10.0	176.244253	30.5	58.003189	58.003189	0.000000	0.0
__write_console	1	0.000831	0.0	0.857100	0.1	0.000824	0.000831	0.000000	0.0
Two::two_fn2(int)	1	77.337049	13.4	117.383964	20.3	77.337049	77.337049	0.000000	0.0

Class View

The **Class** view displays the summary information sorted by class. Beneath each class, the methods are listed in a hierarchy. You can open and close a class to show or hide its methods.

The **Class** view lets you study the performance impact of substituting one implementation of a class for another. You can run profiles on the two implementations, and view the behavior of the different objects side by side. You can do the same with the **Flat** view on a function-by-function basis, but the **Class** view offers a more natural way of accessing object-based data. It also lets you gather all the object methods together and view them simultaneously, revealing the effect of interactions between the methods of the object. Figure 9.12 shows the **Class** view of the profiler results.

Figure 9.12 Simple Profiler Data Viewer — Class View



Function	Count	Time	%	+Children	%	Average	Maximum	Minimum	Stack Space
⊞ __write_console	16	0.013181	0.0	2.021916	0.3	0.000824	0.000823	0.000000	0.0
⊞ __access_file(unsigned long,un	16	13.911939	2.4	2.020426	0.3	0.869496	0.940659	0.000000	0.0
⊞ fn1(int)	1	38.669393	6.7	176.570655	30.5	38.669393	38.669393	0.000000	0.0
⊞ fn2(int)	1	58.003152	10.0	136.661020	23.6	58.003152	58.003152	0.000000	0.0
⊞ fn3(int)	1	77.336056	13.4	77.656930	13.4	77.336056	77.336056	0.000000	0.0
⊞ fn4(int)	1	38.668545	6.7	38.936651	6.7	38.668545	38.668545	0.000000	0.0
⊞ fn5(int)	1	58.002262	10.0	58.851940	10.2	58.002262	58.002262	0.000000	0.0
⊞ fn6(int)	1	77.335985	13.4	78.237931	13.5	77.335985	77.335985	0.000000	0.0
⊞ fn7(int)	4	3.811343	0.7	8.179593	1.4	0.952836	3.804825	0.000000	0.0
⊟ One	1	38.669393	6.7	176.570655	30.5	19.334696	38.669393	0.000000	0.0
one_fn1(int)	1	38.669393	6.7	176.570655	30.5	38.669393	38.669393	0.000000	0.0
⊟ Two	2	96.672545	16.7	176.570655	30.5	48.336272	58.003152	0.000000	0.0
two_fn1(int)	1	38.669393	6.7	176.570655	30.5	38.669393	38.669393	0.000000	0.0
two_fn2(int)	1	58.003152	10.0	136.661020	23.6	58.003152	58.003152	0.000000	0.0

A

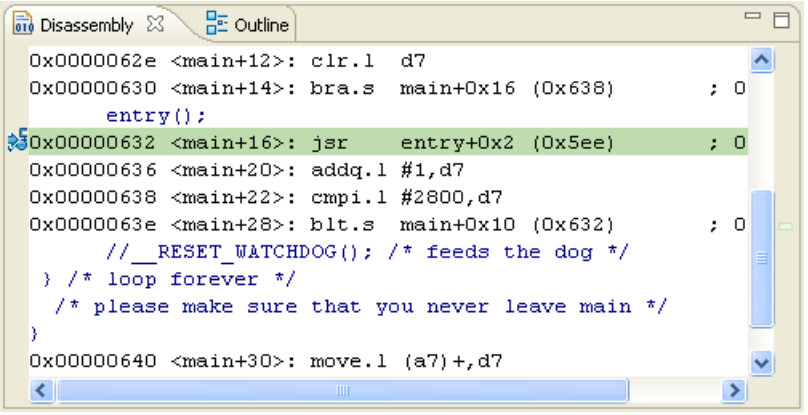
Trace Collection with Breakpoints

A breakpoint suspends the debug session automatically when the instruction on which it is set is executed. When a breakpoint is set in trace collection, it halts the application at a particular source line and stops collecting the trace data. You can start collecting trace again from that source line by resuming the debug session.

You can set breakpoints either in the editor area or in the **Disassembly** view. To set a breakpoint in the **Disassembly** view and collect trace data:

1. Open the **Debug Configurations** dialog box, and select your project in the tree structure.
2. Click the **Trace and Profile** tab, and check the **Enable Trace and Profile** checkbox.
3. Select the **Continuous** option from the **Select Trace Mode** group.
4. Ensure that the **Trace is Always Enabled** option is selected in the **Trace Start/Stop Conditions** drop-down list.
5. Click **Apply** to save the settings.
6. Click **Debug** to start the debug session.
7. In the **Disassembly** view, double-click on the marker bar corresponding to the `jsr` instruction on which you want to set the breakpoint.

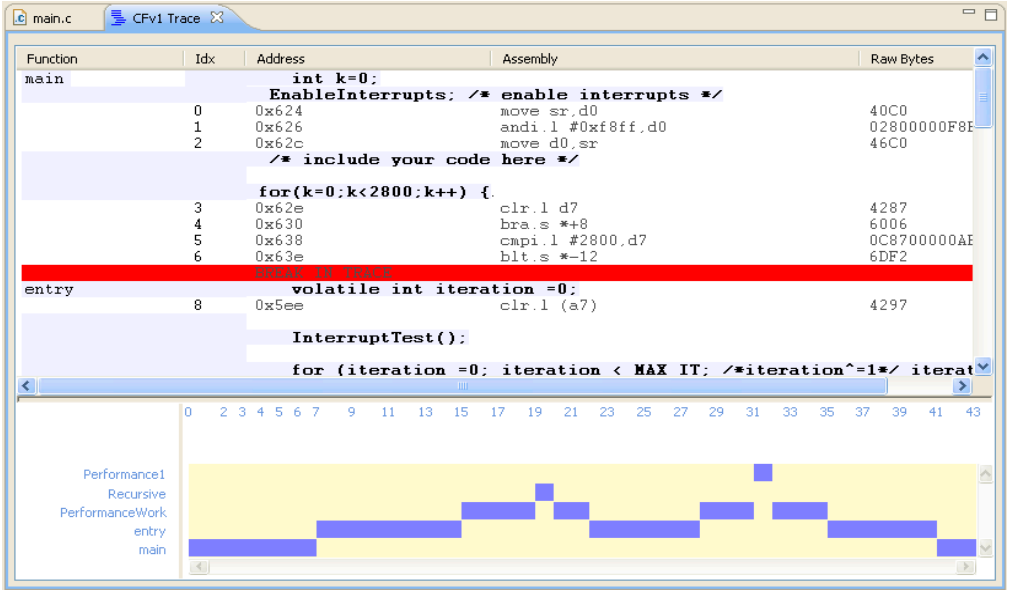
Figure A.1 Setting Breakpoint in Disassembly View



8. Click **Resume**. The debug session stops when the breakpoint instruction is executed.
9. Open the **Trace Data** viewer following the steps explained in the topic, Viewing Data for ColdFire V1 Target to view the trace data.
10. Press **Resume** to resume trace collection.

    When a breakpoint is set on the source code or on a `jsr` instruction in the **Disassembly** view, and you resume trace after the application halts, a three-instruction delay occurs in trace collection. The **Trace Data** viewer highlights this delay by displaying a message, 'BREAK IN TRACE' in red color (see Figure A.2).

Figure A.2 Break in Trace



NOTE Similarly, you can set breakpoints in the editor area and collect the trace data.

B

Configuring Trace Registers in Source Code

You can the configure trace registers in the source code without using the CodeWarrior IDE for both HCS08 and ColdFire V1 targets.

HCS08

To setup the trace mode for HCS08, you must configure the DBGT and DBGC registers in the source code, preferably within the `main()` function and before any processing and/or functions calls. This is not mandatory but recommended as the `main()` function is called immediately after processor is reset. Therefore, to collect trace from this point forward, you must configure these registers in the beginning of the `main()` function.

You can configure only the **Automatically** mode for the HCS08 target. The **Continuously** and **Profile-Only** modes are implemented in the host software and there are no registers on the target associated with these modes.

For HCS08, the trace registers are mapped in the memory, so they have addresses associated. Therefore, you need to simply write desired values to these registers in the source code. An example to configure the trace registers in the source code of the HCS08 target is discussed below.

1. Create a stationary project.
2. Open the source code editor area.
3. Replace the source code written in the `main()` function with the source code shown in Listing B.1:

Listing B.1 Configuration of DBGC and DBGT registers in main() function for HCS08 target

```
void main(void)
{
   EnableInterrupts;

   DBGT = 0x80;    // write debug trigger register
   DBGC = 0xC0;    // write debug control register
```

```
  for(;;) {
        __RESET_WATCHDOG();
        foo();
    }
}
```

4. Save and build the project.
5. Open the **Debug Configurations** dialog box, and select your project in the tree structure.
6. Click the **Trace and Profile** tab, and check the **Enable Trace and Profile** checkbox.
7. Check the **Configuration Set in User Code** checkbox in the **User Options** group. The rest of the controls on the page turn disabled.
8. Click **Apply** to save the settings.
9. Click **Debug** to debug the application.
10. Click **Resume** to resume the execution and begin measurement. Let the application run for several seconds.
11. Click **Suspend**.
12. Open the **Trace Data** viewer following the steps explained in the topic, Viewing Data for HCS08 Target to view the collected data.

You can also set triggers at the required addresses by writing the following statements in the `main()` function of your source code:

```
DBGCA = 0xE1CD; //write comparator A; sets trigger A at
0xE1CD

DBGCB = 0xE1DB; //write comparator B; sets trigger B at
0xE1DB
```

ColdFire V1

For ColdFire V1, the trace registers are not mapped in the memory space. Therefore, the only way to access these registers is by using the `wdebug` instruction, while the processor is running in the supervisor mode.

To configure the trace registers in the source code in the Automatic mode on the ColdFire V1 target:

1. Create a stationary project.
2. Open the source code editor area.
3. Replace the source code of `main.c` with the source code shown in Listing B.2:

Listing B.2 Configuration of trace registers for ColdFire V1

```
#include <hidef.h> /* for EnableInterrupts macro */
#include "derivative.h" /* include peripheral declarations */
#include <stdio.h>
#include <ctype.h>

/* Define the used DRc */
#define MCFDEBUG_CSR 0x0  /* Configuration status*/
#define MCFDEBUG_XCSR 0x1 /* Extended configuration status register*/
#define MCFDEBUG_CSR2 0x2 /* Configuration status register 2 */

#define TRACE_AUTOMATIC 0
#define TRACE_CONTINUOUS 1
#define TRACE_PCSYNC 2
#define TRACE_NONE 3

#define TRACE_MODE TRACE_AUTOMATIC

volatile unsigned short dbg_spc[6];
volatile unsigned short *dbg;

inline void wdebug(int reg, unsigned long data) {
 // Force alignment to long word boundary
 dbg = (unsigned short *)((((unsigned long)dbg_spc) + 3) & 0xfffffffc);

 // Build up the debug instruction
 dbg[0] = 0x2c80 | (reg & 0xf);
 dbg[1] = (data >> 16) & 0xffff;
 dbg[2] = data & 0xffff;
 dbg[3] = 0;
 asm("    MOVE.L dbg ,A1");
 asm("    WDEBUG (A1) ");
}
inline void setSupervisorModel(void)
{
  asm ( "    MOVE.W #0x2000,D0" );
  asm ( "    MOVE.W D0,SR" );
}

void main(void) {
  EnableInterrupts;
  /* include your code here */
  setSupervisorModel(); /* set CPU supervisor programming model */

#if TRACE_MODE == TRACE_AUTOMATIC  /* set automatic trace mode */
  wdebug(MCFDEBUG_CSR, 0x200);
  wdebug(MCFDEBUG_XCSR, 0x1);
```

```
  wdebug(MCFDEBUG_CSR2, 0xC1);
#elif TRACE_MODE == TRACE_CONTINUOUS  /* set continuous trace mode */
  wdebug(MCFDEBUG_CSR, 0x200);
  wdebug(MCFDEBUG_XCSR, 0x0);
  wdebug(MCFDEBUG_CSR2, 0x89);
#elif TRACE_MODE == TRACE_PCSYNC  /* set PCSync trace mode */
  wdebug(MCFDEBUG_CSR, 0x200);
  wdebug(MCFDEBUG_XCSR, 0x3);
  wdebug(MCFDEBUG_CSR2, 0x99);
#endif

  for(;;) {
    __RESET_WATCHDOG(); /* feeds the dog */

  }
}
```

4. Save and build the project.
5. Open the **Debug Configurations** dialog box, and select your project in the tree structure.
6. Click the **Trace and Profile** tab, and check the **Enable Trace and Profile** checkbox.
7. Check the **Configuration Set in User Code** checkbox. The rest of the controls on the page turn disabled.
8. Click **Apply** to save the settings.
9. Click **Debug** to debug the application.
10. Click **Resume** to resume the execution and begin measurement. Let the application run for several seconds.
11. Click **Suspend**.
12. Open the **Trace Data** viewer following the steps explained in the topic, Viewing Data for ColdFire V1 Target to view the collected data.

You can also set triggers at the required addresses using the source code. To set triggers, enable the triggers by including the following definition in your source code.

```
#define ENABLE_TRIGGERS True  /* Enabling triggers */
```

Also, include the statements shown in Listing B.3 in the `main()` function.

Listing B.3 Setting triggers using source code in ColdFire V1

```
#if TRACE_MODE == TRACE_AUTOMATIC  /* set automatic trace mode */
 #if ENABLE_TRIGGERS == True  // with trigger points
  wdebug(MCFDEBUG_PBR0, 0x5F2); //set PBR0 register;trigger A at 0x5F2
  wdebug(MCFDEBUG_PBR1, 0x6C8); //set PBR1 register;trigger B at 0x6C8
```

```
    wdebug(MCFDEBUG_PBMR, 0x0);
    wdebug(MCFDEBUG_AATR, 0xE401);
    wdebug(MCFDEBUG_DBR, 0x0);
    wdebug(MCFDEBUG_DBMR, 0x0);
    wdebug(MCFDEBUG_TDR, 0x40006002);
 #endif
    // without trigger points
    wdebug(MCFDEBUG_CSR, 0x200);
    wdebug(MCFDEBUG_XCSR, 0x1);
    wdebug(MCFDEBUG_CSR2, 0xC1);
#elif TRACE_MODE == TRACE_CONTINUOUS /* set continuous trace mode */
 #if ENABLE_TRIGGERS == True  // with trigger points
   wdebug(MCFDEBUG_PBR0, 0x5F2); //set PBR0 register;trigger A at 0x5F2
   wdebug(MCFDEBUG_PBR1, 0x6C8); //set PBR1 register;trigger B at 0x6C8
    wdebug(MCFDEBUG_PBMR, 0x0);
    wdebug(MCFDEBUG_AATR, 0xE401);
    wdebug(MCFDEBUG_DBR, 0x0);
    wdebug(MCFDEBUG_DBMR, 0x0);
    wdebug(MCFDEBUG_TDR, 0x40006002);
 #endif
  // without trigger points
  wdebug(MCFDEBUG_CSR, 0x200);
  wdebug(MCFDEBUG_XCSR, 0x0);
  wdebug(MCFDEBUG_CSR2, 0x89);
#elif TRACE_MODE == TRACE_PCSYNC  /* set PCSync trace mode */
  wdebug(MCFDEBUG_CSR, 0x200);
  wdebug(MCFDEBUG_XCSR, 0x3);
  wdebug(MCFDEBUG_CSR2, 0x99);
#else
  /* None */
  wdebug(MCFDEBUG_CSR, 0x0);
  wdebug(MCFDEBUG_XCSR, 0x0);
  wdebug(MCFDEBUG_CSR2, 0x0);
#endif
```

NOTE The default setting of the `TRACE_MODE` compiler switch is `TRACE_AUTOMATIC`. To configure trace in the Continuous mode, set the compiler switch to `TRACE_CONTINUOUS` in the source code shown in Listing B.2. To configure trace in the Profile-Only mode, set the compiler switch to `TRACE_PCSYNC` in the source code shown in Listing B.2.

C

Low Power WAIT Mode

Currently, CodeWarrior supports two low power modes: normal WAIT and normal STOP. The Microcontrollers Software Analysis Tools component provides support for the normal WAIT state. This state allows peripherals to function, while allowing CPU to go to sleep reducing power.

The Wait For Interrupt (WFI) instruction is used to enter the low power WAIT state. When an interrupt request occurs, the CPU exits the WAIT mode and resumes processing, beginning with the stacking operations leading to the interrupt service routine.

NOTE When a processor issues a WFI instruction, it can suspend execution and enter a low power state. The processor can remain in that state until it detects a reset or one of the following WFI wake-up events:

- an asynchronous exception at a priority that preempts any currently active exceptions.
- a debug event with debug enabled.

When the hardware detects a WFI wake-up event, or earlier if the implementation chooses, the WFI instruction completes.

To activate low power mode monitoring and view results, you need to:

1. Configure Low Power WAIT State
2. Debug the project and collect trace data
3. View Low Power WAIT Results

Configure Low Power WAIT State

The low power WAIT state requires continuous trace to be enabled and ITM and DWT tracing to be disabled.

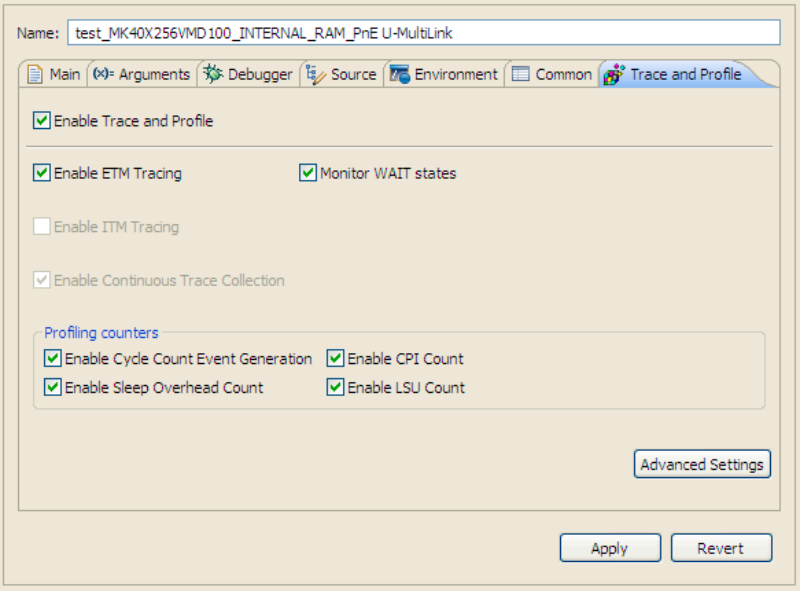
To configure the low power WAIT state:

1. Open the **Debug Configurations** dialog box.
2. Select the **Trace and Profile** tab.

3. Check the **Monitor WAIT states** checkbox.

   The **Enable ITM Tracing** checkbox will get disabled automatically. Also, the **Enable Continuous Trace Collection** checkbox will get checked and disabled (see Figure C.1).

Figure C.1 Configuration of Low Power WAIT Mode



4. Click **Apply** to save the settings.

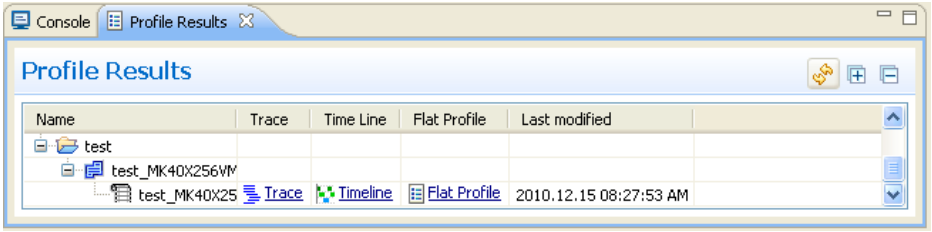
View Low Power WAIT Results

After the project is debugged and trace is collected, you can view the low power WAIT results in the timeline viewer of the **Profile Results** view. To view the low power WAIT results:

1. Open the **Profile Results** view.
2. Expand the project name.

   The data source is listed under the project name.

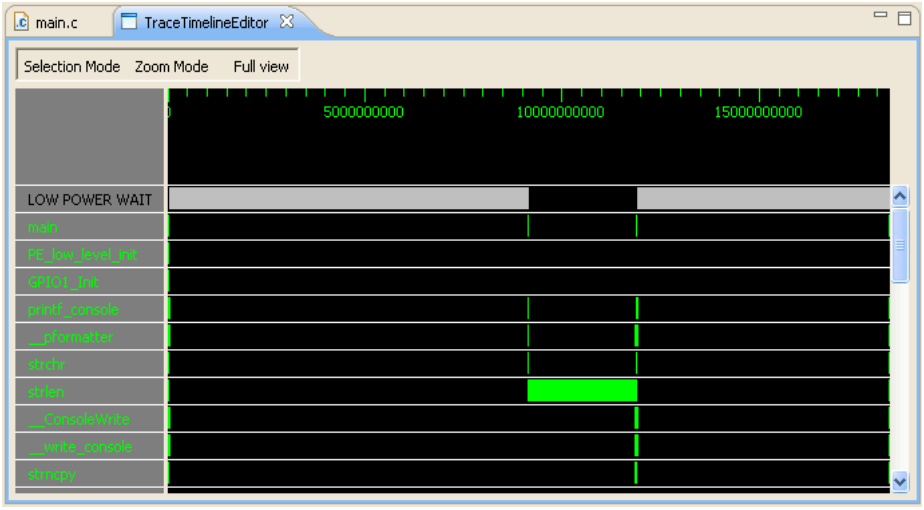
Figure C.2 Profile Results View



3. Click on the **Timeline** hyperlink.

   The **TraceTimelineEditor** viewer appears displaying the low power WAIT results as shown in Figure C.3.

Figure C.3 Timeline View



The list of functions in the left panel of the **TraceTimelineEditor** viewer displays LOW POWER WAIT at the first position. The corresponding gray-colored bars in the right panel shows for how long the application was in WAIT mode. The bars appear twice, which indicates that the application entered the WAIT state twice.

Index

F

G

H

I

K

L

M

N

P

R

S

T

U

V