CodeWarrior Development Studio for Microcontrollers Version 11.x Profiling and Analysis Tools Users Guide

Document Number: CWMCUSWAUG
Rev. 11.x, 07/2017

Contents

Section number **Title** **Page**

Section number	Title	Page

Section number Title Page

Section number	Title	Page

Chapter 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
- Accompanying Documentation

1.1 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.

1.2 Accompanying Documentation

The **Documentation** page describes the documentation included in the CodeWarrior Development Studio for Microcontrollers. You can access the **Documentation** page by:

- opening the START_HERE.html in the `<CWInstallDir>\MCU\Help` folder,
- selecting **Help > Documentation** from the IDE’s menu bar, or selecting **Start > Programs > Freescale CodeWarrior > CW for MCU v11.x > Documentation** from the Windows taskbar.

NOTE

To view the online help for the CodeWarrior tools, first select **Help > Help Contents** from the IDE's menu bar. Next, select required manual from the **Contents** list. For general information about the CodeWarrior IDE and debugger, refer to the *Codewarrior Common Features Guide* in this folder:

`<CWInstallDir>\MCU\Help\PDF`

Chapter 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 Interface
- Data Collection

2.1 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.

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+, ColdFire V4e, Kinetis, DSC (Digital Signal Controller), MPC56xx, and S12Z targets.
- Trace is collected by setting triggers and using various trigger conditions - Applicable for HCS08, ColdFire V1, Kinetis, DSC, and S12Z targets.
- Trace is collected even when no triggers are set - Applicable only for the HCS08 target.
- Profiling information is collected - Applicable for Coldfire V2-V4 and ColdFire V4e targets.

- Depending on the target selected, Software Analysis will extract trace through debugger using OSBDM/JTAG, P&E USB-ML-PPCNEXUS, P&E Universal MultiLink, EthernetTAP, USB TAP, J-Trace, or Tracelink.
- Remote launch support to collect trace by running Software Analysis scripts from Jython console.

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

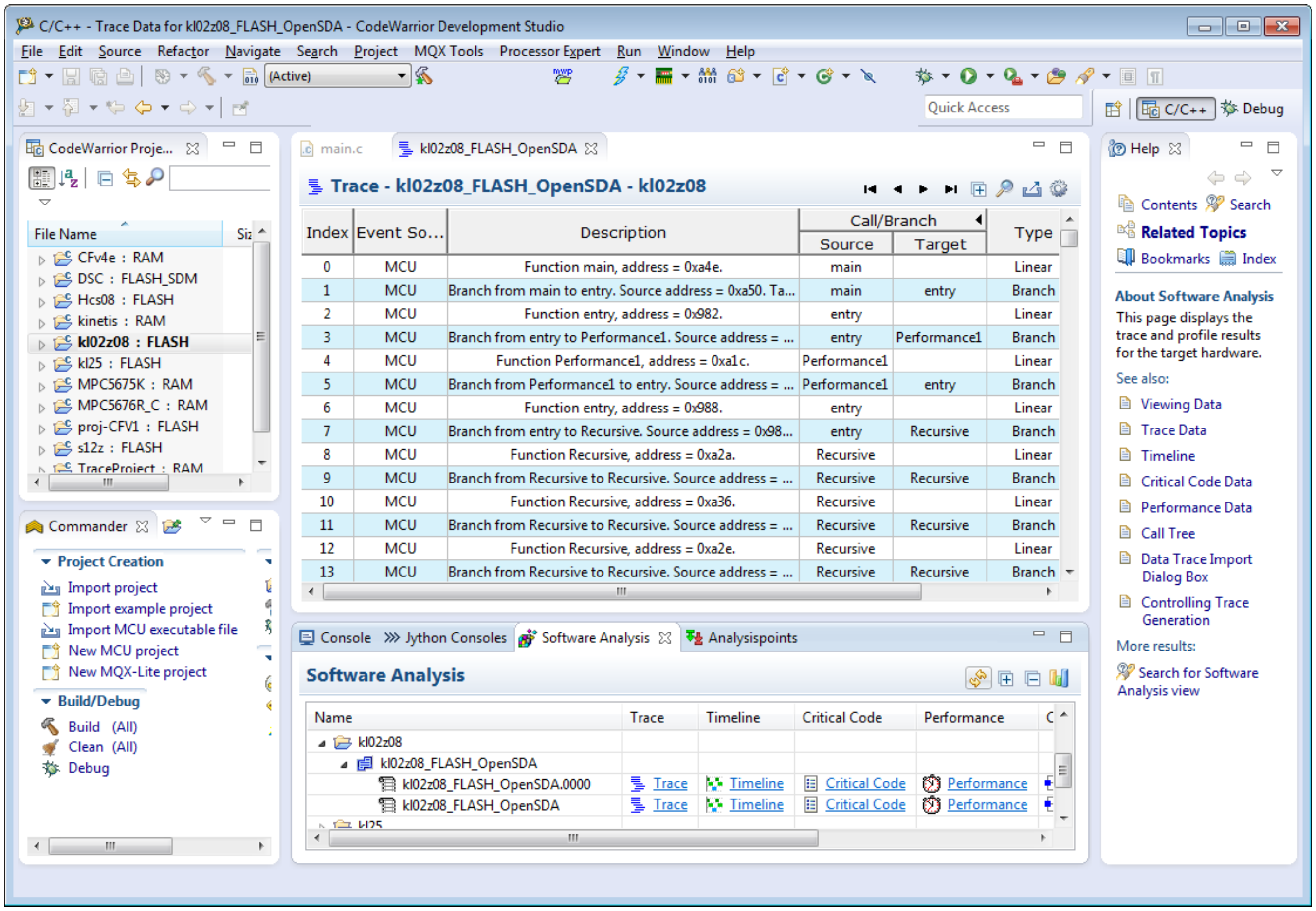
- step through trace data and the corresponding source code of that trace data simultaneously
- display results in an intuitive and user friendly manner in the **Trace Data** , **Critical Code** , **Timeline** , **Performance** , **Call Tree** , and **Simple Profiler Data** viewers
- export trace data, critical code data, profiling information into an Excel file
- copy and paste a line of the trace in a text file
- import trace data collected on all targets, HCS08, ColdFire V1-V4, ColdFire V4e, Kinetis, MPC56xx, DSC and S12Z.

NOTE

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

2.2 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.



Trace - kl02z08_FLASH_OpenSDA - kl02z08

Index	Event So...	Description	Call/Branch		Type
			Source	Target	
0	MCU	Function main, address = 0xa4e.	main		Linear
1	MCU	Branch from main to entry. Source address = 0xa50. Ta...	main	entry	Branch
2	MCU	Function entry, address = 0x982.	entry		Linear
3	MCU	Branch from entry to Performance1. Source address = ...	entry	Performance1	Branch
4	MCU	Function Performance1, address = 0xa1c.	Performance1		Linear
5	MCU	Branch from Performance1 to entry. Source address = ...	Performance1	entry	Branch
6	MCU	Function entry, address = 0x988.	entry		Linear
7	MCU	Branch from entry to Recursive. Source address = 0x98...	entry	Recursive	Branch
8	MCU	Function Recursive, address = 0xa2a.	Recursive		Linear
9	MCU	Branch from Recursive to Recursive. Source address = ...	Recursive	Recursive	Branch
10	MCU	Function Recursive, address = 0xa36.	Recursive		Linear
11	MCU	Branch from Recursive to Recursive. Source address = ...	Recursive	Recursive	Branch
12	MCU	Function Recursive, address = 0xa2e.	Recursive		Linear
13	MCU	Branch from Recursive to Recursive. Source address = ...	Recursive	Recursive	Branch

Software Analysis

Name	Trace	Timeline	Critical Code	Performance
kl02z08				
kl02z08_FLASH_OpenSDA				
kl02z08_FLASH_OpenSDA.0000	Trace	Timeline	Critical Code	Performance
kl02z08_FLASH_OpenSDA	Trace	Timeline	Critical Code	Performance
kl25				

About Software Analysis

This page displays the trace and profile results for the target hardware.

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.
- **Software Analysis** view provides a hierarchical view of the data sources, data files, and data sets of the project.
- **Simple Profiler Data Viewer** provides flat, tree, and class-based view of the profiler output.

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.

2.3 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
- Timeline Data
- Performance Data
- Call Tree
- Profiling Data - This does not require a target hardware, it uses the profiling system

2.3.1 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.

2.3.2 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.

2.3.3 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.

2.3.4 Performance Data

The performance data includes the metric and invocation information for each function that executes in the application. The performance data during measurement enables you to compare the relative efficiencies of various portions of your target program. Both exclusive and inclusive timing measurements are provided in the performance data.

The parent-child calling relationships between your program's functions are also provided. Each function pair consists of a caller and a callee with data provided for each.

2.3.5 Call Tree

The Call Tree data shows the general application flow in a hierarchical tree in which statistics are displayed for each function.

2.3.6 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.

Chapter 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:

- Creating New Project
- Configuring Launcher
- Collecting Data
- Viewing Data

3.1 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.

- Using HCS08 Target
- Using ColdFire V1 Target
- Using Kinetis Target
- Using ColdFire V4e
- Using MPC56xx Target

NOTE

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

3.1.1 Using HCS08 Target

To create a new Microcontrollers project using the HCS08 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.

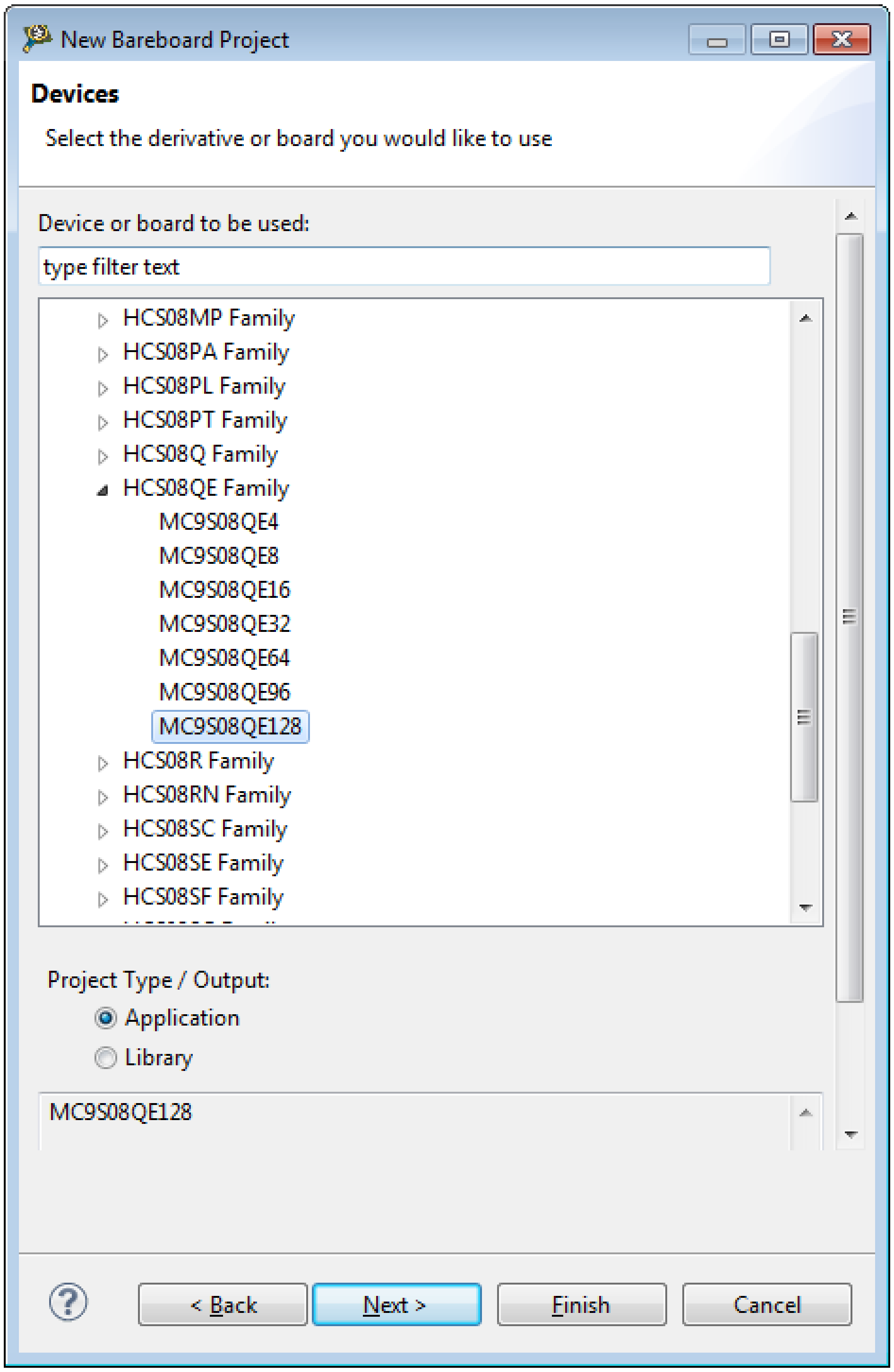
3. Enter the name of your project in the **Project Name** text box.

   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. The table below describes the Microcontrollers bareboard project settings.
5. Click **Next**.

   The **Devices** page appears.

6. Select the target device or board for your project from the HCS08 family. For example, select **S08 > HCS08QE Family > MC9SO8QE128**.



New Bareboard Project

Devices

Select the derivative or board you would like to use

Device or board to be used:

type filter text

- HCS08MP Family
- HCS08PA Family
- HCS08PL Family
- HCS08PT Family
- HCS08Q Family
- HCS08QE Family
  - MC9S08QE4
  - MC9S08QE8
  - MC9S08QE16
  - MC9S08QE32
  - MC9S08QE64
  - MC9S08QE96
  - MC9S08QE128
- HCS08R Family
- HCS08RN Family
- HCS08SC Family
- HCS08SE Family
- HCS08SF Family

Project Type / Output:

[✓ Application] Library

MC9S08QE128

Figure 3-1. Devices Page

7. Click **Next**.

   The **Connections** page appears.

8. Select the available connection.

NOTE

The Profiling supports all the connections except the simulator.

9. Click **Next**.

   The **Languages** page appears.

   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.

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

    The **Rapid Application Development** page appears.

12. Accept the default settings and click **Next**.
13. Click **Next**.

    The **C/C++ Options** page appears.

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

    The project *proj-hcs08* is created and appears in the **CodeWarrior Projects** view.

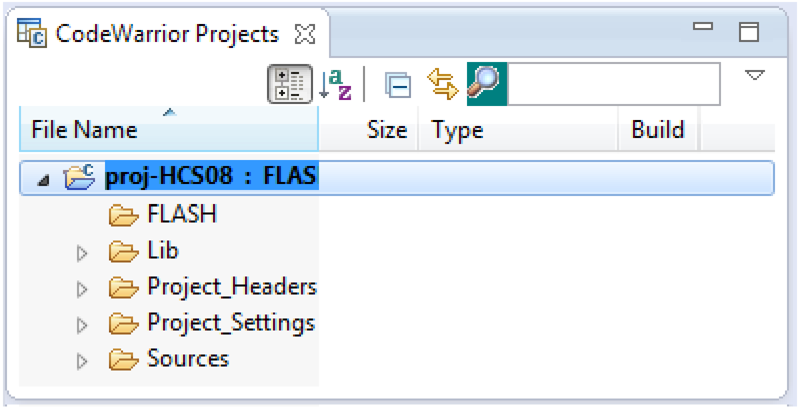


Figure 3-2. CodeWarrior Projects View - HCS08 Project

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

This creates a new Microcontrollers project for the HCS08 target.

NOTE

For details on creating a new MCU bareboard project, refer CodeWarrior Development Studio for Microcontrollers V11.x Targeting Manual located at <CWInstallDir>\CW MCU v11.x \MCU\Help\PDF, where <CWInstallDir> is the location where the CodeWarrior software is installed.

3.1.2 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.

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

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

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

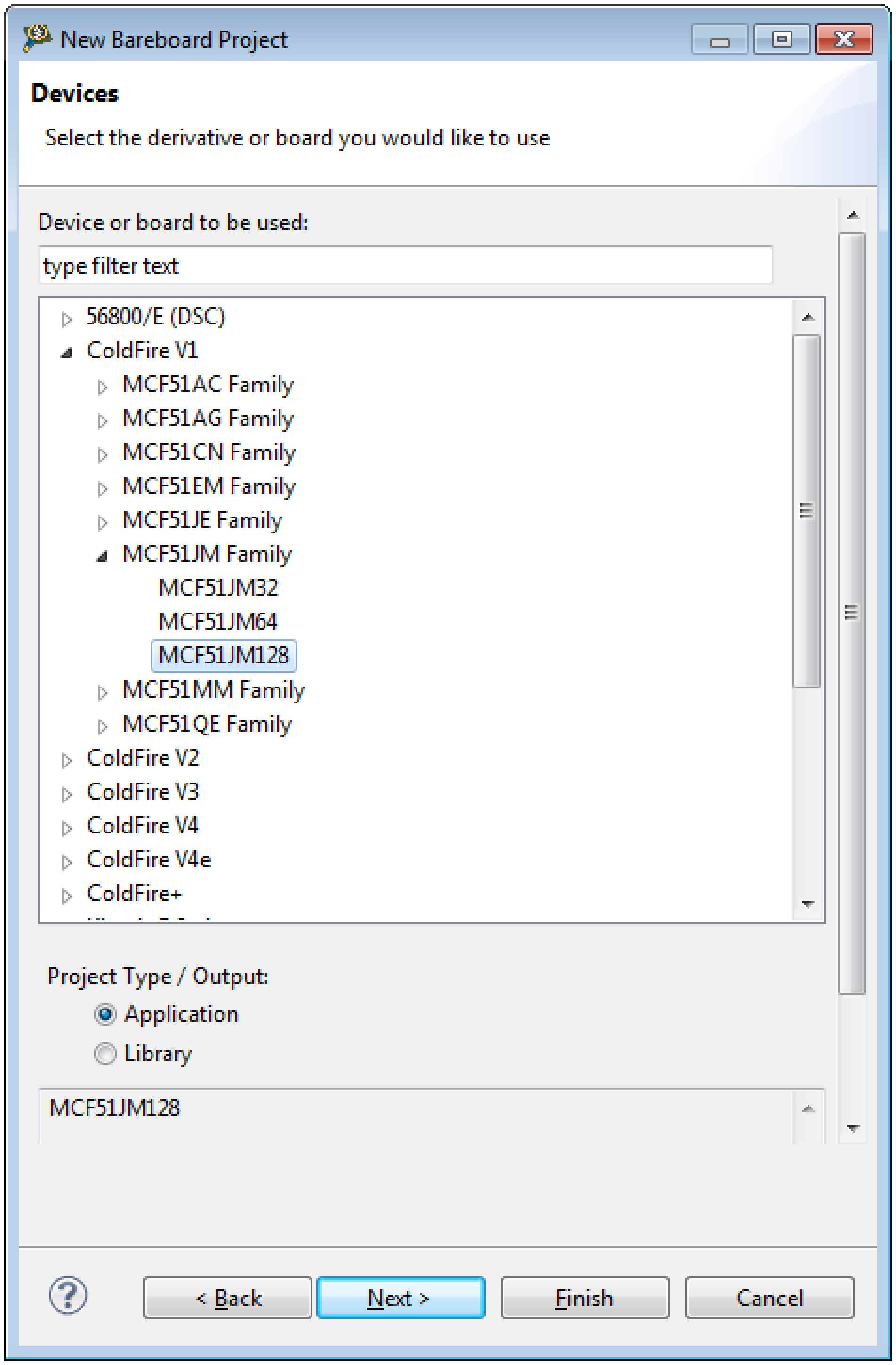
   NOTE

   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 ColdFire V1 family. For example, select **ColdFire V1 > MCF51JM Family > MCF51JM128**.


New Bareboard Project

Devices

Select the derivative or board you would like to use

Device or board to be used:

type filter text

- 56800/E (DSC)
- ColdFire V1
  - MCF51AC Family
  - MCF51AG Family
  - MCF51CN Family
  - MCF51EM Family
  - MCF51JE Family
  - MCF51JM Family
    - MCF51JM32
    - MCF51JM64
    - MCF51JM128
  - MCF51MM Family
  - MCF51QE Family
- ColdFire V2
- ColdFire V3
- ColdFire V4
- ColdFire V4e
- ColdFire+

Project Type / Output:

[✓ Application] Library

MCF51JM128

Figure 3-3. Devices Page

6. Click **Next**.

   The **Connections** page appears.

7. Select the available connection from the **Connections** page.

NOTE

The Profiling supports all the connections except the simulator.

8. Click **Next**.

   The **ColdFire Build Options** page appears.

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

    The **Rapid Application Development** page appears.

11. Accept the default settings and click **Finish**.
12. The project *proj-CFV1* is created and appears in the **CodeWarrior Projects** view.

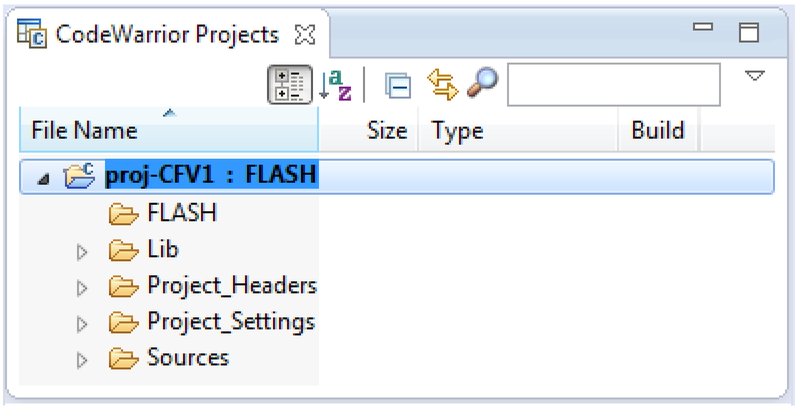


Figure 3-4. CodeWarrior Projects View - ColdFire V1 Project

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

This creates a new Microcontrollers project for the ColdFire V1 target.

NOTE

For details on creating a new MCU bareboard project, refer CodeWarrior Development Studio for Microcontrollers V11.x Targeting Manual located at <CWInstallDir>\CW MCU v11.x \MCU\Help\PDF, where <CWInstallDir> is the location where the CodeWarrior software is installed.

3.1.3 Using Kinetis Target

To create a new Microcontrollers project using the Kinetis 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 **Project Name** text box, for example, *TraceProject*, and specify the location of the project if you do not want to use the default location.

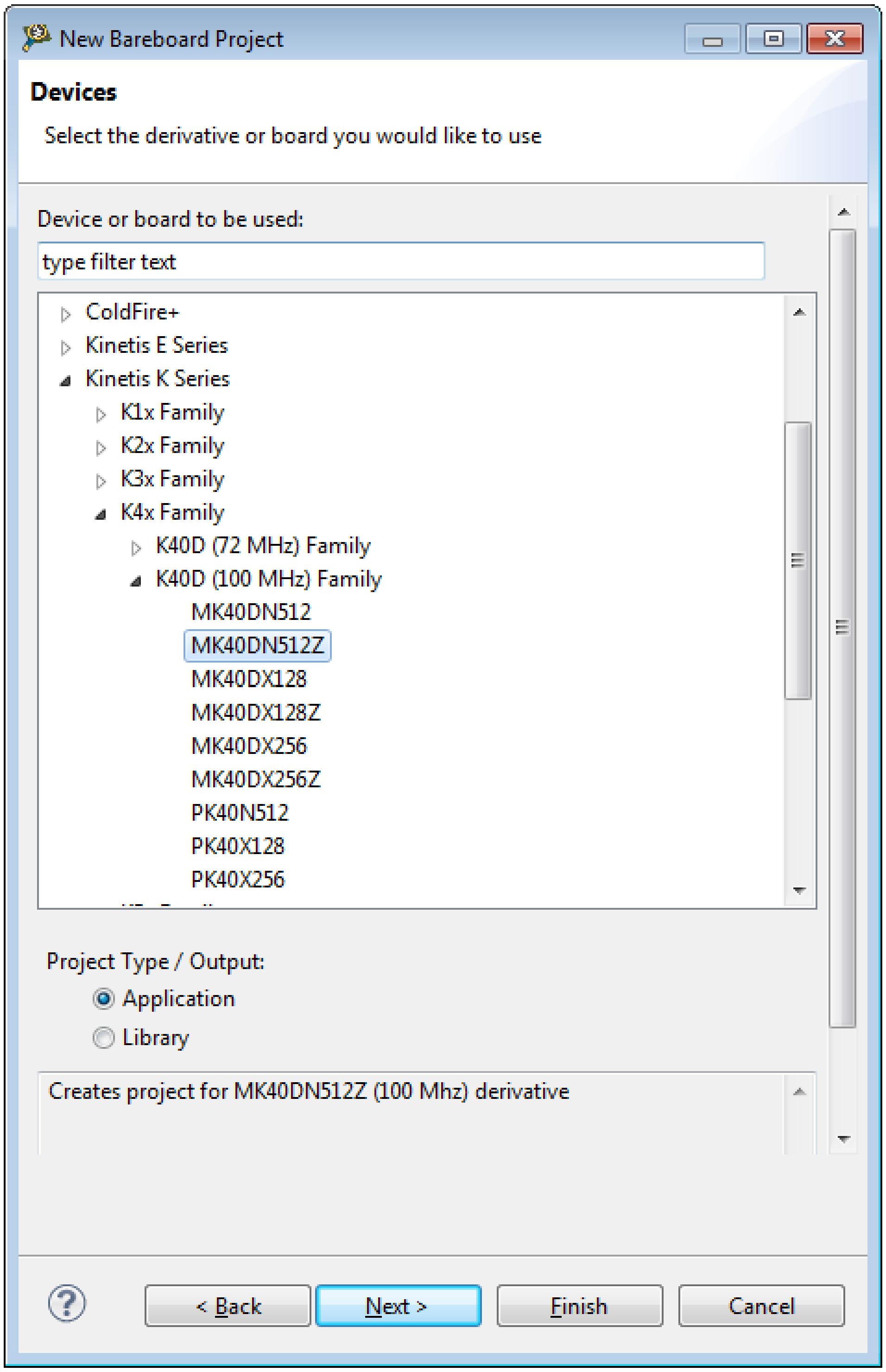
   NOTE

   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 K Series > K4x Family > K40D (100 MHz) Family > MK40DN512Z**.



New Bareboard Project

Devices

Select the derivative or board you would like to use

Device or board to be used:

type filter text

- ColdFire+
- Kinetis E Series
- Kinetis K Series
  - K1x Family
  - K2x Family
  - K3x Family
  - K4x Family
    - K40D (72 MHz) Family
    - K40D (100 MHz) Family
      - MK40DN512
      - MK40DN512Z
      - MK40DX128
      - MK40DX128Z
      - MK40DX256
      - MK40DX256Z
      - PK40N512
      - PK40X128
      - PK40X256

Project Type / Output:

[✓ Application] Library

Creates project for MK40DN512Z (100 Mhz) derivative

Figure 3-5. Devices Page

6. Click **Next**.

   The **Connections** page appears.

7. Select the available connection.

NOTE

The Profiling supports all the connections except the simulator.

8. Click **Next**.

   The **Language and Build Tools Options** page appears. By default, GCC ARM Build Tools is selected; you can select Freescale tools by selecting the **Freescale** option.

9. Do not change the default settings and click **Next**.

   The **Rapid Application Development** page appears.

10. Click **Finish**.

    The project *TraceProject* is created and appears in the **CodeWarrior Projects** view.

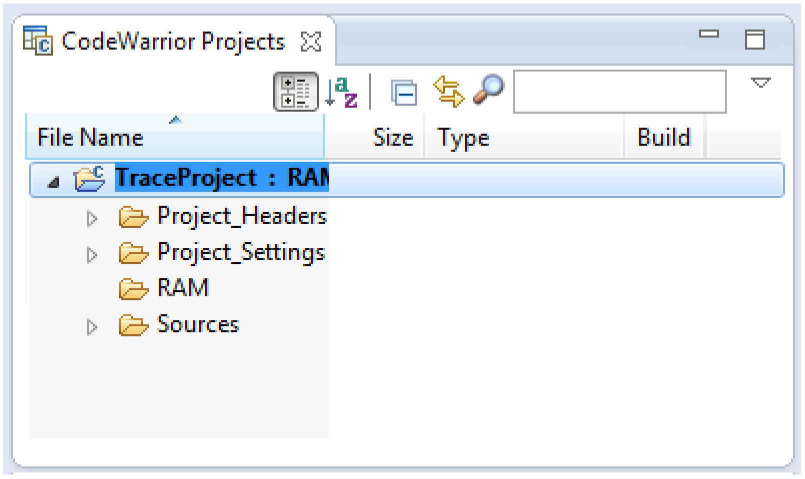


Figure 3-6. CodeWarrior Projects View - Kinetis Project

11. Select the project in the **CodeWarrior Projects** view.
12. Select **Project > Build Project** to build your project.

This creates a new Microcontrollers project for the Kinetis target.

NOTE

For details on creating a new MCU bareboard project, refer CodeWarrior Development Studio for Microcontrollers V11.x Targeting Manual located at <CWInstallDir>\CW MCU v11.x \MCU\Help\PDF, where <CWInstallDir> is the location where the CodeWarrior software is installed.

3.1.4 Using ColdFire V4e

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

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

   The **New Project** dialog box appears.

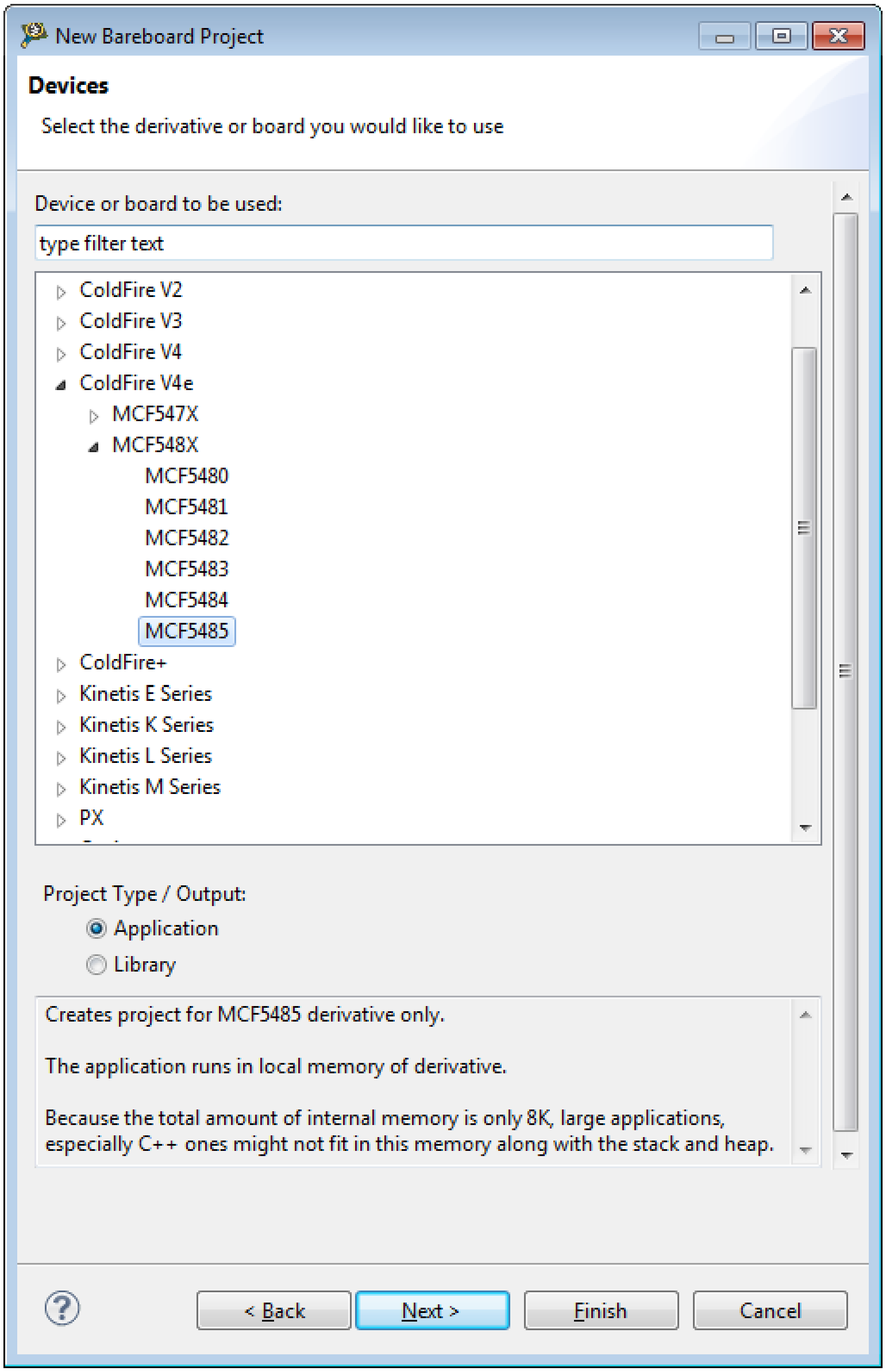
2. Select **Bareboard Project** and click **Next**.

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

3. Enter the name of your project in the **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 V4e > MCF548x > MCF5485**.



Project Type / Output: [✓ Application] Library

Creates project for MCF5485 derivative only.

The application runs in local memory of derivative.

Because the total amount of internal memory is only 8K, large applications, especially C++ ones might not fit in this memory along with the stack and heap.

Figure 3-7. Devices Page

6. Click **Next**.

   The **Connections** page appears.

7. Select the available connection.
8. Click **Next**.

   The **ColdFire Build Options** page appears.

9. Check the **Enable C++ Support** checkbox.
10. Click **Finish**.

    The project is created and appears in the **CodeWarrior Projects** view.

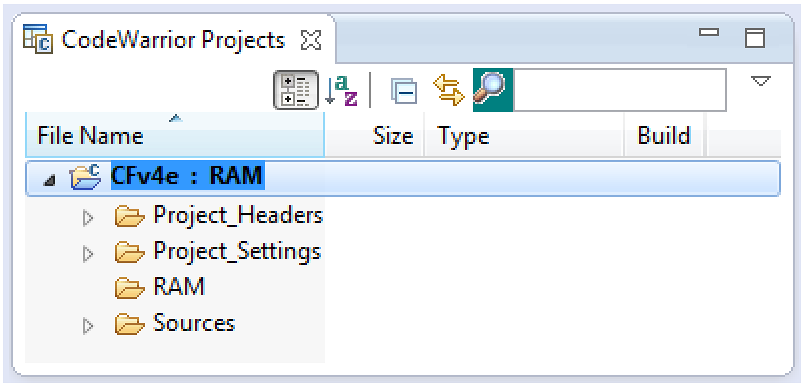


Figure 3-8. CodeWarrior Projects View - ColdFire V4e PRoject

11. Collect profiling information using the steps described in the chapter, Simple Instrumentation Profiling on ColdFire V2 - V4e Targets.

This creates a new Microcontrollers project for the ColdFire V4e target.

NOTE

For more details, refer CodeWarrior Development Studio for Microcontrollers V11.x Targeting Manual located at <CWInstallDir>\CW MCU v11.x\MCU\Help\PDF, where <CWInstallDir> is the location where the CodeWarrior software is installed.

3.1.5 Using MPC56xx Target

To create a new Microcontrollers project using the MPC56xx target:

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

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

2. Enter the name of your project in the **Project Name** text box.
3. Click **Next**.

The **Devices** page appears.

4. Select the target device or board for your project from the MPC56xx family. For example, select **Qorivva > MPC56xxK Family > MPC5675K**.
5. Click **Next**.

   The **Connections** page appears.

6. Select the available connection, for example, P&E USB Multilink PPCNEXUS.
7. Click **Next**.

   The **LSM/DPM configuration** page appears.

8. Click **Next** to display the **Language and Build Tools Options** page.
9. Click **Finish**.

   The project is created and appears in the **CodeWarrior Projects** view.

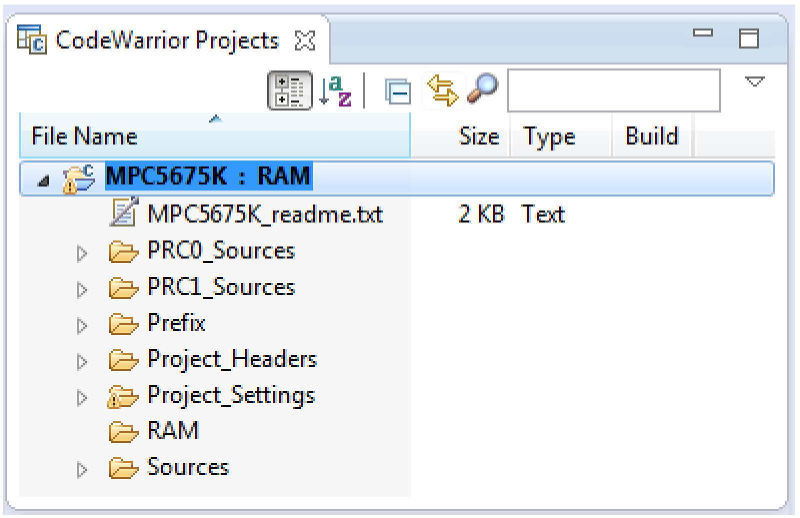


Figure 3-9. CodeWarrior Projects View - MPC56xx Project

10. Select the project in the **CodeWarrior Projects** view.
11. Select **Project > Build Project** to build your project.

NOTE

You can create projects in a similar way on the ColdFire V2- V4, DSC and e200 derivatives. For DSC, select the required device from the **56800/E (DSC)** family in the **Devices** page of the **New Project Wizard** . The e200 derivative is the core of MPC56xx target.

The MPC56xx target also supports MPC5668G and MPC5668E derivatives. The MPC5668G/E derivatives are compatible 32-bit microcontrollers built on Power Architecture technology. Both MPC5668G and MPC5668E have two e200 cores, e200z6 (Core 0) and e200z0 (Core 1). Only e200z6 core provides tracing capability with the following features:

- Trace is collected only in overwrite mode
- Timestamp is always zero in the **Trace Data** viewer
- Only **Trace Data** and **Timeline** viewers are available

While creating a MPC5668G/E project, select **Qorivva > MPC5668G/E Family > MPC5668G/E** to choose the required target and follow the New Project Wizard steps. In the **Power Architecture Core Configuration** page, select the required core configuration. Select **e200z6+e200z0h** if you want to work on both cores, else select **e200z6** to work on a single core.

3.2 Configuring Launcher

Before debugging an application, you need to configure the Trace and Profile settings used for the current debug launcher.

- Configure HCS08 Target
- Configure ColdFire V1 Target
- Configure Kinetis Target
- Configuring Advanced Settings on Kinetis
- Configure MPC56xx Target
- Configure S12Z Target
- Configure ColdFire V2-V4 Targets
- Configure DSC Target

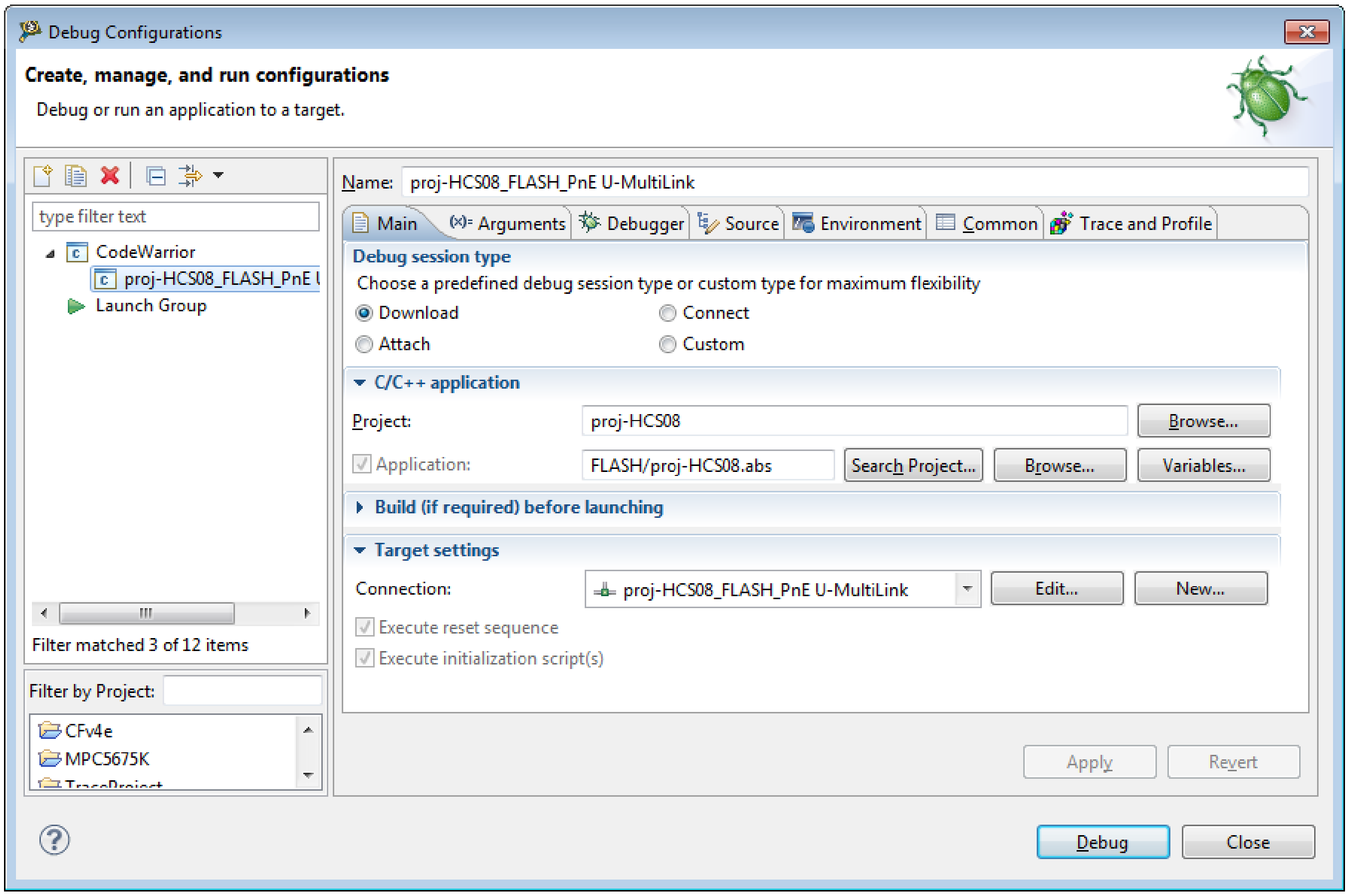
3.2.1 Configure HCS08 Target

To configure the launch configuration for the HCS08 target:

1. In the **CodeWarrior Projects** view, right-click 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** 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_FLASH_PnE U-MultiLink*.
3. On the **Main** tab page, 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.


Debug Configurations

Create, manage, and run configurations

Debug or run an application to a target.

type filter text

CodeWarrior
proj-HCS08_FLASH_PnE U
Launch Group

Filter matched 3 of 12 items

Filter by Project:
CFv4e
MPC5675K
TraceProject

Name: proj-HCS08_FLASH_PnE U-MultiLink

Main | Arguments | Debugger | Source | Environment | Common | Trace and Profile

Debug session type
Choose a predefined debug session type or custom type for maximum flexibility
[✓ Download] Connect
Attach Custom

C/C++ application
Project: proj-HCS08 Browse...
Application: FLASH/proj-HCS08.abs Search Project... Browse... Variables...

Build (if required) before launching

Target settings
Connection: proj-HCS08_FLASH_PnE U-MultiLink Edit... New...
[✓] Execute reset sequence
[✓] Execute initialization script(s)

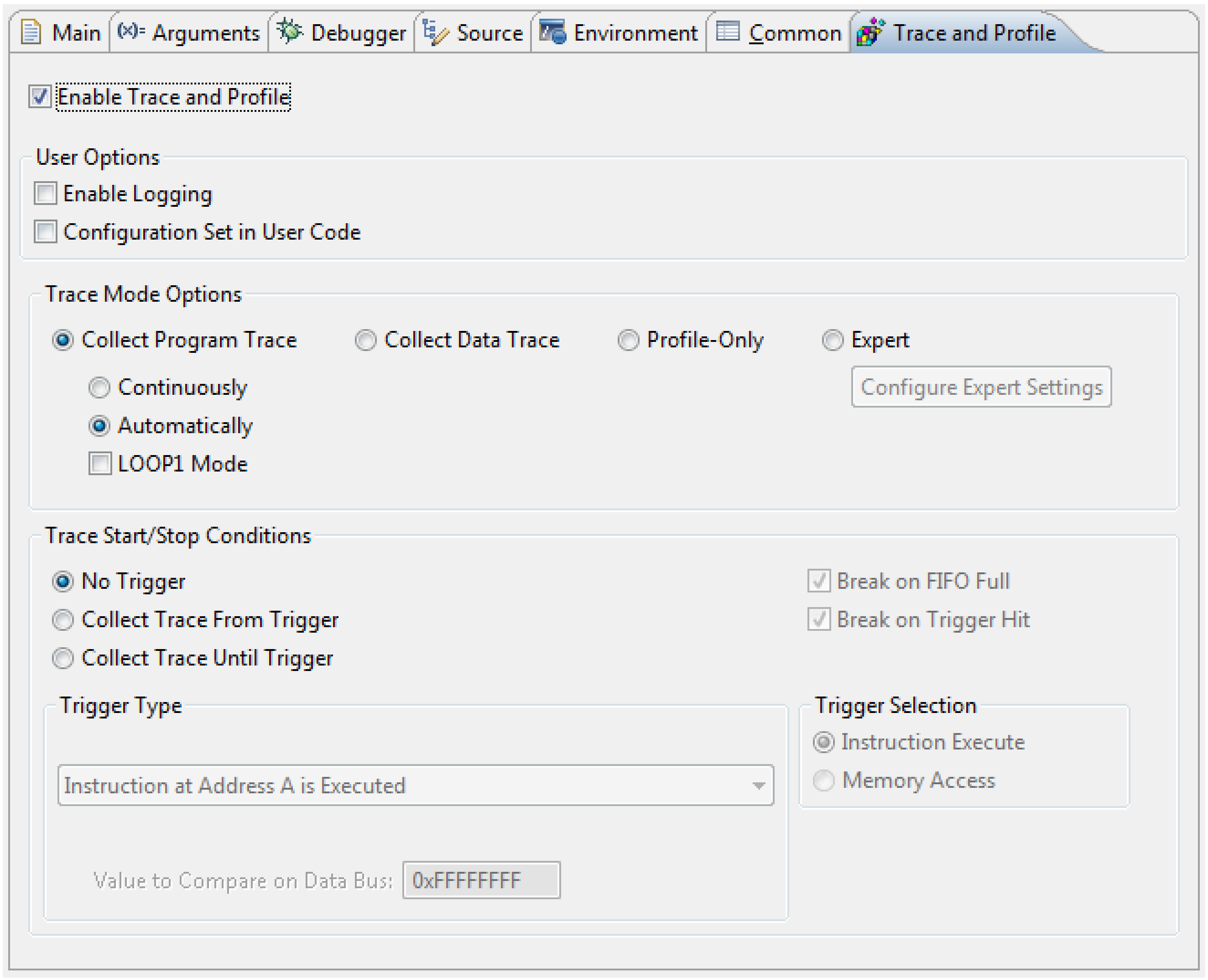
Apply Revert

Debug Close

Figure 3-10. 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.



[✓ Enable Trace and Profile]

User Options
- Enable Logging
- Configuration Set in User Code

Trace Mode Options
- [✓ Collect Program Trace] Collect Data Trace Profile-Only Expert
  - Continuously
  - [✓ Automatically]
  - LOOP1 Mode

Trace Start/Stop Conditions
- [✓ No Trigger]
- Collect Trace From Trigger
- Collect Trace Until Trigger
- [✓ Break on FIFO Full]
- [✓ Break on Trigger Hit]

Trigger Type: Instruction at Address A is Executed

Trigger Selection: [✓ Instruction Execute] Memory Access

Value to Compare on Data Bus: 0xFFFFFFFF

Figure 3-11. Debug Configurations - Trace and Profile Page (HCS08)

The table below describes the various **Trace and Profile** options.

Table 3-1. 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.

Table continues on the next page...

Table 3-1. Trace and Profile Options for HCS08 (continued)

Group	Options	Descriptions
		To understand how to configure trace registers in the application for the HCS08 target, refer Configuring Trace Registers in Source Code.
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, refer *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.

Table continues on the next page...

Table 3-1. Trace and Profile Options for HCS08 (continued)

Group	Options	Descriptions
	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/
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. For more information on tracepoints, refer 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. For more information on tracepoints, refer Setting Tracepoints (HCS08).
	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. For more information on tracepoints, refer 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.
Trigger Type		Contains various conditions of triggers, A and B for starting/stopping trace collection. For more information on tracepoints, refer Setting Tracepoints (HCS08).
	Instruction at Address A is Executed	Starts trace from the address or source line corresponding to trigger A. For more information, refer 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.

Table continues on the next page...

Table 3-1. Trace and Profile Options for HCS08 (continued)

Group	Options	Descriptions
		For more information, refer 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, refer Instruction Inside Range from Address A to Address B is Executed. **NOTE:** For the MC9S08PT60 target, the **Instruction Inside Range from Address A to Address B is Executed** trigger will hit when any instruction inside the range between trigger address A and trigger address B matches with the data on the bus or program address inside the range.
	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, refer Instruction Outside Range from Address A to Address B is Executed. **NOTE:** For the MC9S08PT60 target, the **Instruction Outside Range from Address A to Address B is Executed** trigger will hit when any instruction outside the range between trigger address A and trigger address B matches with the data on the bus or program address outside the range.
	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, refer 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, refer Instruction at Address A is Executed and Value on Data Bus Match.

Table continues on the next page...

Table 3-1. Trace and Profile Options for HCS08 (continued)

Group	Options	Descriptions
		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, refer 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, refer 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, refer Capture Read/Write Values at Address B, After Access at Address A.
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, refer Memory Access Triggers.

NOTE

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

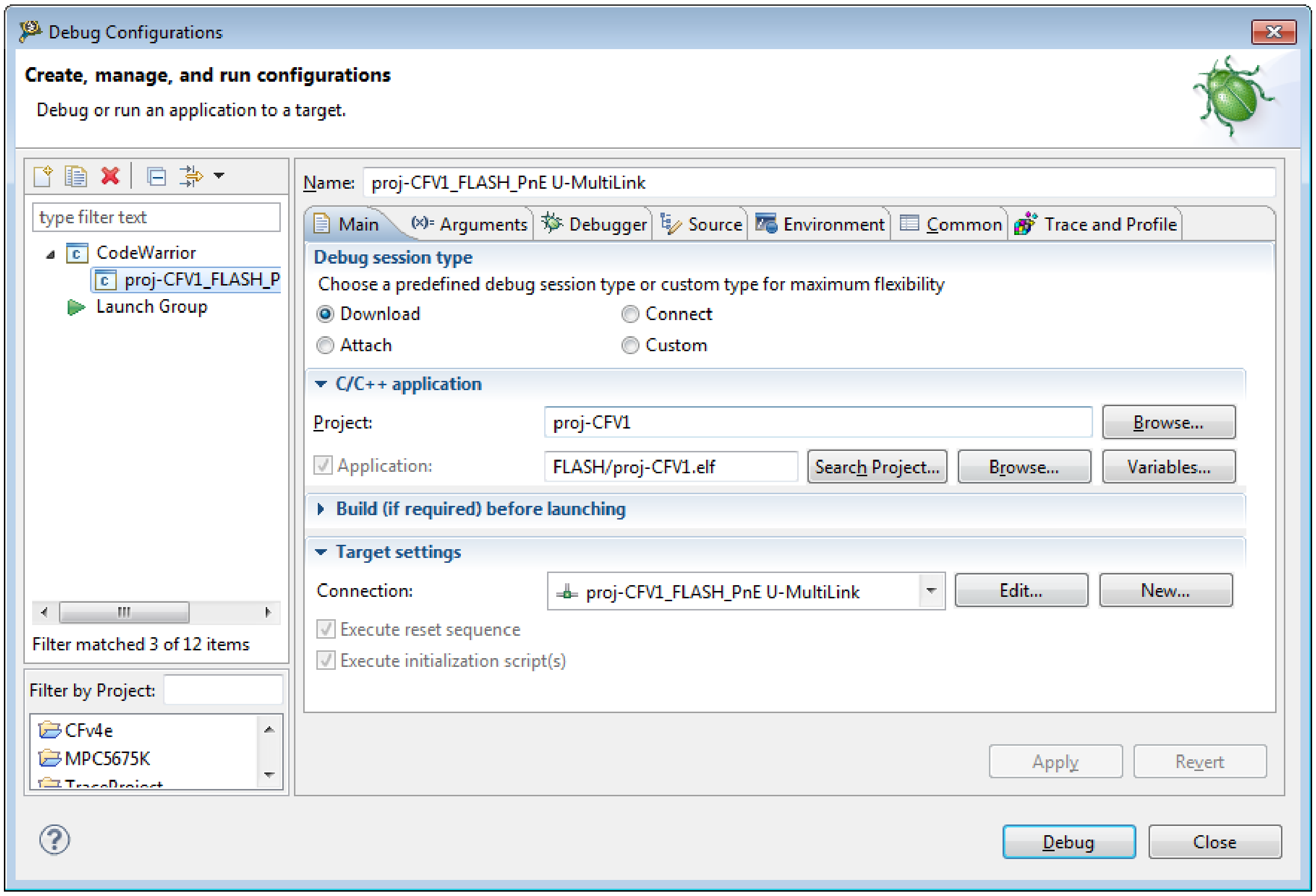
3.2.2 Configure ColdFire V1 Target

To configure the launch configuration for the ColdFire V1 target:

1. In the **CodeWarrior Projects** view, right-click 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_FLASH_PnE U-MultiLink*.
3. On the **Main** tab page, 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.


Debug Configurations

Create, manage, and run configurations

Debug or run an application to a target.

Name: proj-CFV1_FLASH_PnE U-MultiLink

Main | Arguments | Debugger | Source | Environment | Common | Trace and Profile

Debug session type

Choose a predefined debug session type or custom type for maximum flexibility

[✓ Download] Connect

Attach Custom

C/C++ application

Project: proj-CFV1

Application: FLASH/proj-CFV1.elf

Build (if required) before launching

Target settings

Connection: proj-CFV1_FLASH_PnE U-MultiLink

[✓] Execute reset sequence

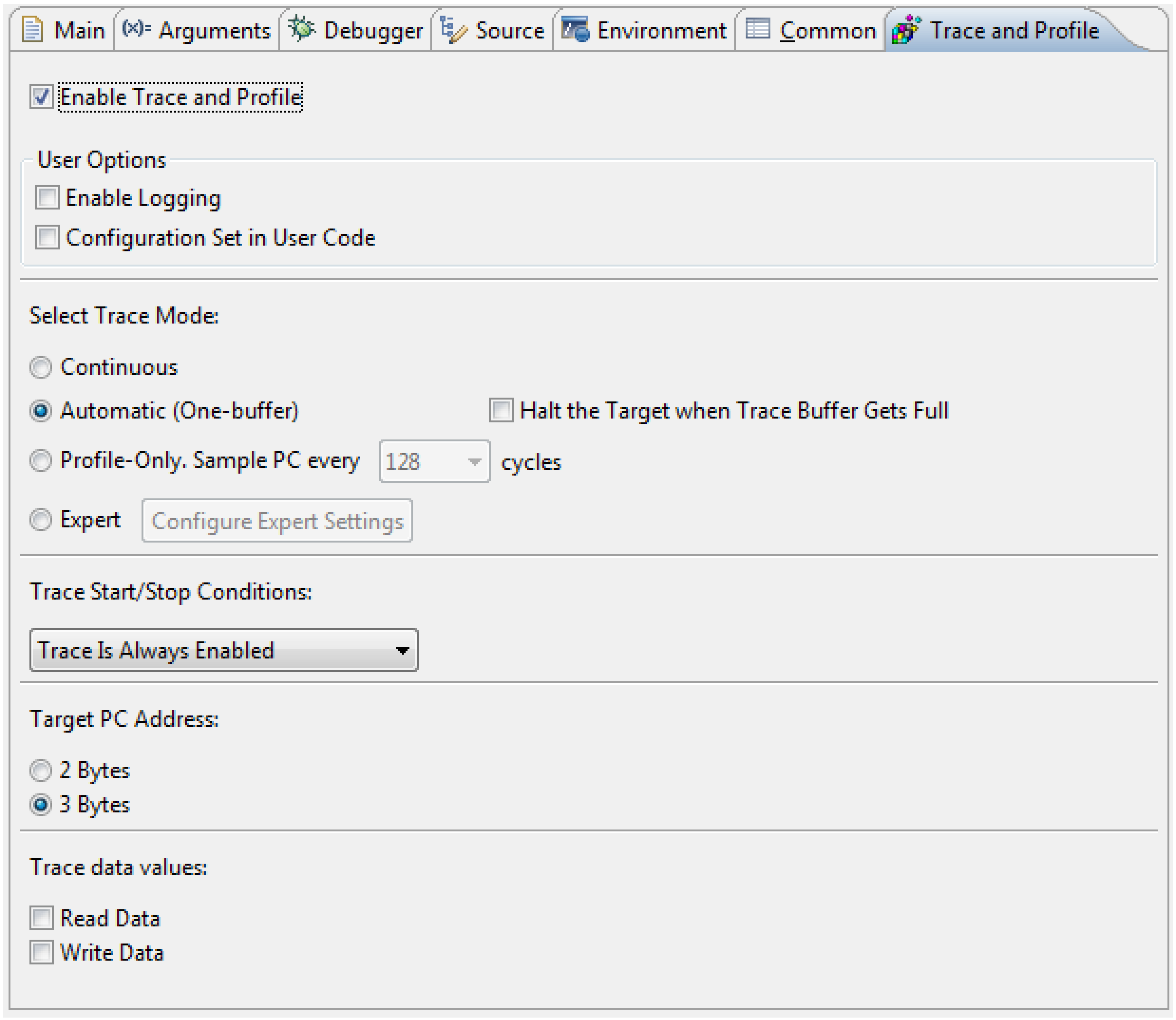
[✓] Execute initialization script(s)

Filter matched 3 of 12 items

Figure 3-12. 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.



[✓] Enable Trace and Profile

User Options
[ ] Enable Logging
[ ] Configuration Set in User Code

Select Trace Mode:
( ) Continuous
(●) Automatic (One-buffer) [ ] Halt the Target when Trace Buffer Gets Full
( ) Profile-Only. Sample PC every 128 cycles
( ) Expert Configure Expert Settings

Trace Start/Stop Conditions:
Trace Is Always Enabled

Target PC Address:
( ) 2 Bytes
(●) 3 Bytes

Trace data values:
[ ] Read Data
[ ] Write Data

Figure 3-13. Debug Configurations - Trace and Profile Page (ColdFire V1)

The table below describes the various **Trace and Profile** options.

Table 3-2. 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.

Table continues on the next page...

Table 3-2. Trace and Profile Options for ColdFire V1 (continued)

Group	Options	Descriptions
		To understand how to configure trace registers in the application for the ColdFire V1 target, refer Configuring Trace Registers in Source Code.
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 more information, refer Conditions for Starting/Stopping Triggers.
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).

Table continues on the next page...

Table 3-2. Trace and Profile Options for ColdFire V1 (continued)

Group	Options	Descriptions
		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.

3.2.3 Configure Kinetis Target

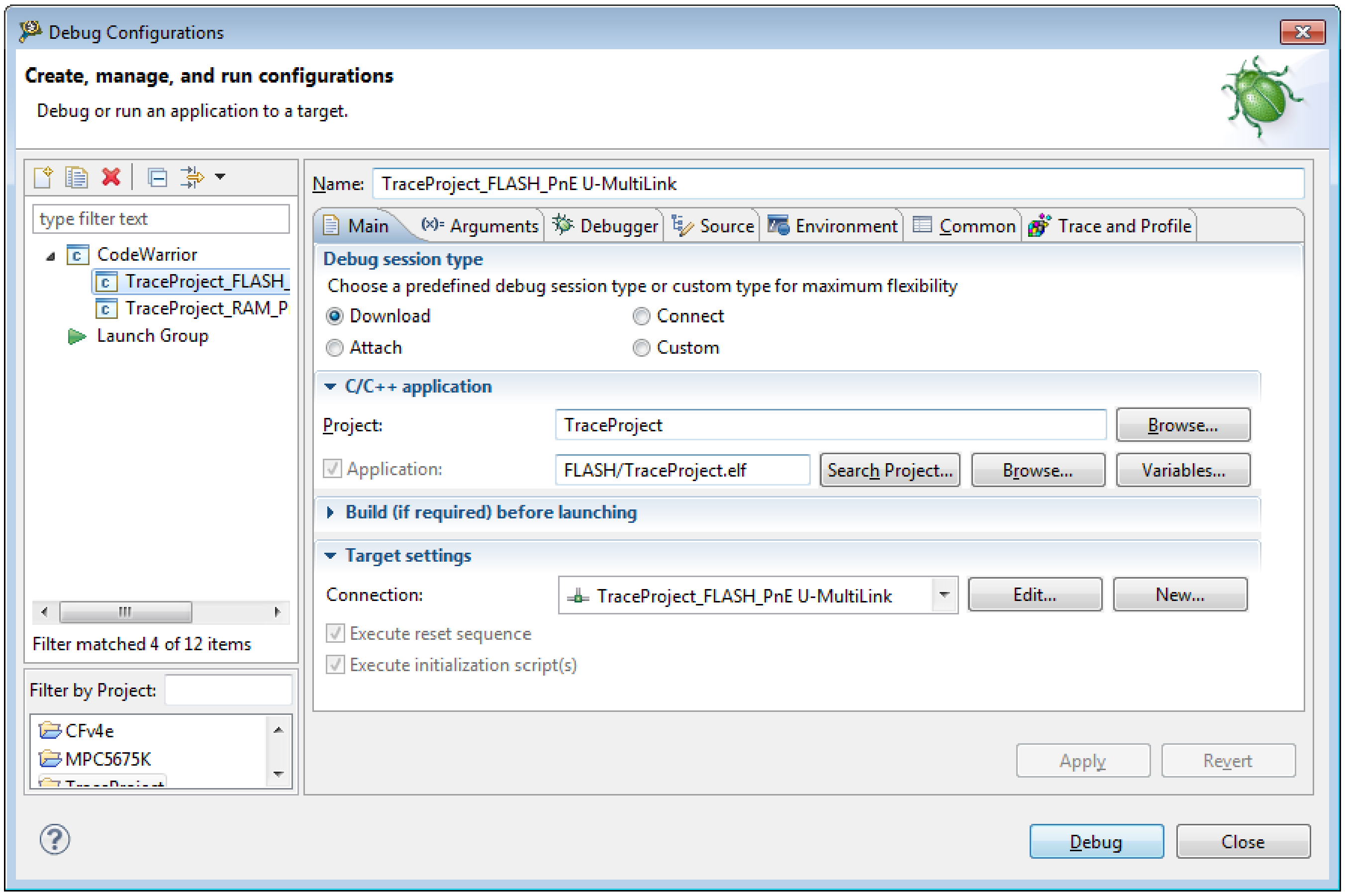
The Kinetis target has different cores and trace modules. For example, Kinetis K series has Cortex M4 core and Kinetis L series has Cortex M0+ core.

To configure the launch configuration for the Kinetis Cortex M4 core:

1. In the **CodeWarrior Projects** view, right-click 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_RAM_PnE U-MultiLink*.
3. On the **Main** tab page, 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.



Debug Configurations

Create, manage, and run configurations

Debug or run an application to a target.

type filter text

- CodeWarrior
  - TraceProject_FLASH_
  - TraceProject_RAM_P
- Launch Group

Filter matched 4 of 12 items

Filter by Project:

- CFv4e
- MPC5675K

Name: TraceProject_FLASH_PnE U-MultiLink

Tabs: Main | Arguments | Debugger | Source | Environment | Common | Trace and Profile

Debug session type

Choose a predefined debug session type or custom type for maximum flexibility

[✓ Download] Connect

Attach Custom

C/C++ application

Project: TraceProject

Application: FLASH/TraceProject.elf

Build (if required) before launching

Target settings

Connection: TraceProject_FLASH_PnE U-MultiLink

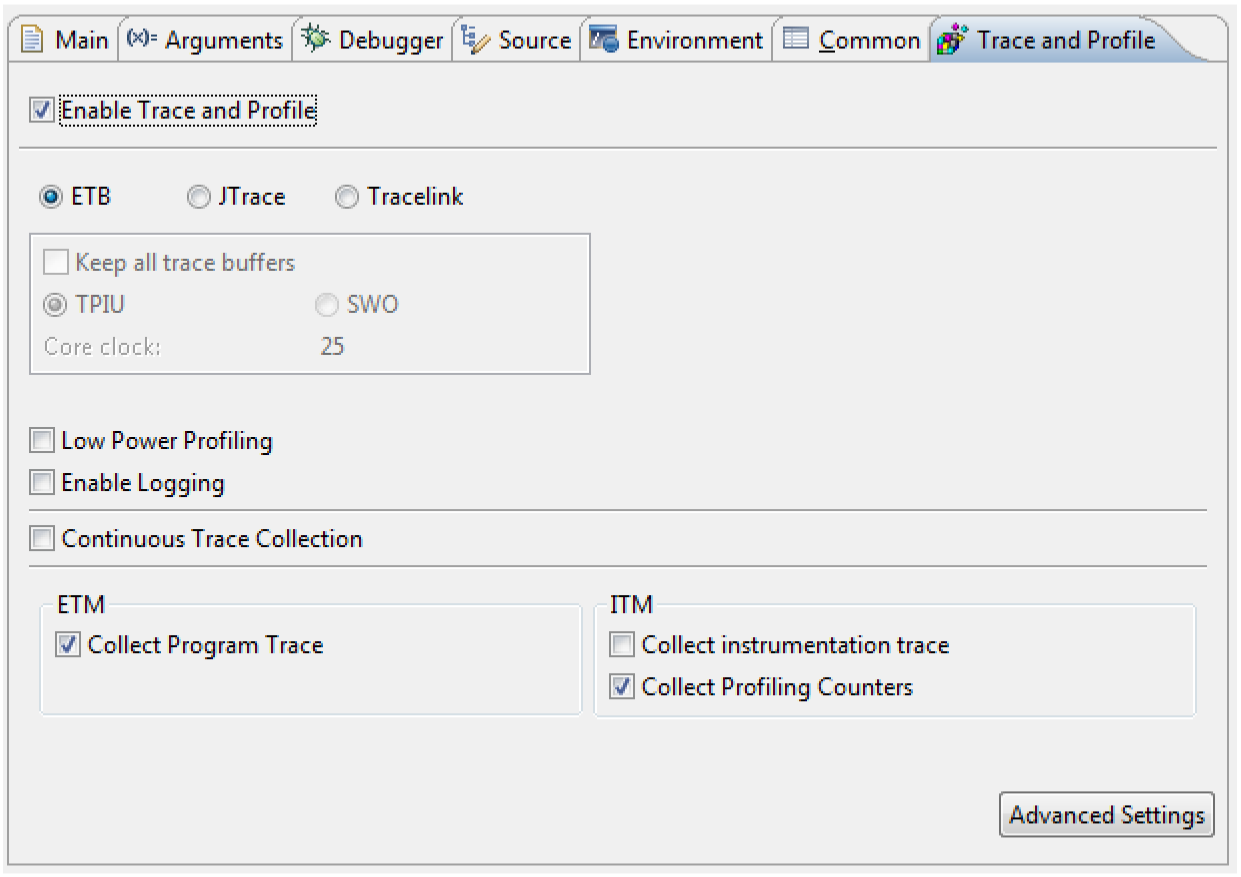
[✓] Execute reset sequence

[✓] Execute initialization script(s)

Figure 3-14. 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.



[✓] Enable Trace and Profile

[● ETB] ○ JTrace ○ Tracelink

[ ] Keep all trace buffers
[● TPIU] ○ SWO
Core clock: 25

[ ] Low Power Profiling
[ ] Enable Logging
[ ] Continuous Trace Collection

ETM
[✓] Collect Program Trace

ITM
[ ] Collect instrumentation trace
[✓] Collect Profiling Counters

Figure 3-15. Debug Configurations - Trace and Profile Page (Kinetis Cortex M4 Core)

The table below describes the various **Trace and Profile** options.

Table 3-3. Trace and Profile Options for Kinetis Cortex M4 Core

Group	Option	Description
ETB		Embedded Trace Buffer where collected trace data is stored. For more information, refer Embedded Trace Buffer (ETB). **NOTE:** For K11D, K12D, K21D and K22D devices, ETB option is disabled because these devices do not have internal ETB buffer.
JTrace		Enables trace collection by using the Segger/J-Trace probe. For more information, refer J-Trace.
	Keep all trace buffers	Keeps all trace collected by the J-Trace or Tracelink (if **Tracelink** is selected) probe and not only the last buffer collected. **NOTE:** If you check this checkbox and collect trace, you might see some trace missing between buffers because of

Table continues on the next page...

Table 3-3. Trace and Profile Options for Kinetis Cortex M4 Core (continued)

Group	Option	Description
		buffer overflow. A trace entry of *Trace buffer read finished* in the **Trace Data** viewer marks the point where an individual buffer ends. **NOTE:** This option is not the equivalent of continuous mode; in the **Keep all trace buffers** mode, all buffers are kept and concatenated, independent of one another. All data (including timestamps) is saved as if each buffer was collected alone, and timestamps for each appended buffer start from 0.
	TPIU	Trace Port Interface Unit - Collects ETM and ITM trace into the internal probe buffer of size 4MB. TPIU is a block on the processor that manages the output of trace. For more information, refer J-Trace.
	SWO	Serial Wire Output - Single pin serial output that collects only ITM trace into the buffer of size 4MB. SWO uses the Serial Wire Debug (SWD) debug connection. If selected, the **Debug Port Interface** should be set as **SWD**.
	Core clock	ARM core clock in Mhz needed for the serial connection setup. The core clock can change due to multiple settings. For example, when started, a K60 processor rated at 100Mh works at 25Mhz, which is the default value. You can change the core clock value according to the requirements.
Tracelink		Provides support of trace collection for 32-bit device architectures using the Tracelink (or multilink trace) probe. When selected, it provides the same options as **JTrace**.
Low Power Profiling		Allows monitoring of low power Wait states. This state lets peripherals to function, while allowing CPU to go to sleep reducing power. For more information, refer Low Power WAIT Mode.
Enable Logging		If checked, creates a log file that keeps details of the actions that took place while executing the target to collect trace data. For example, when the debug session terminated, when the target execution resumed or stopped.

Table continues on the next page...

Table 3-3. Trace and Profile Options for Kinetis Cortex M4 Core (continued)

Group	Option	Description
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.
ETM		Enables/disables trace output from the Embedded Trace Macrocell (ETM) block. It controls the ETM port selection bit from ETM's control register. For more information, refer Embedded Trace Macrocell (ETM).
	Collect Program Trace	Collects program trace.
ITM		Enables/disables trace output from the Instrumentation Trace Macrocell (ITM) block. For more information, refer Instrumentation Trace Macrocell (ITM).
	Collect Instrumentation trace	Collects instrumentation trace.
	Collect Profiling Counters	Enables/disables the following profiling counters at once: • Cycle Count Event Generation - increments and generates synchronization and count events. • Exception Trace - traces exception entry, exit and return to a pre-empted handler or thread. • Exception Overhead Count - exception overhead counter counts the total cycles spent in exception processing. For example, entry stacking, return unstacking, or preemption. An event is emitted on counter overflow which occurs after every 256 cycles. • CPI Count • Sleep Overhead Count • LSU Count - increments on the additional cycles required to execute all load and store instructions. • Folded Instruction Count - increments on any instruction that executes in zero cycles.

The Kinetis K10/K20 50 Mhz and 72 MHz derivatives do not support ETB or ETM; they collect ITM trace only. Also, these derivatives do not support TPIU module; the only mode of collecting ITM trace is through SWO.

The probes that can collect trace on Kinetis K10/K20 50 Mhz and 72 MHz derivatives are J-Trace and J-link. You can debug but can not collect trace with P&E ARM Multilink. The **Trace and Profile** tab for these derivatives has only **JTrace** and **ITM** options enabled.

This topic contains the following sub-topics:

- Configuring Kinetis Cortex M0+ Core
- Difference Between Kinetis Cores Cortex M4 and Cortex M0+
- Trace Collection on Kinetis

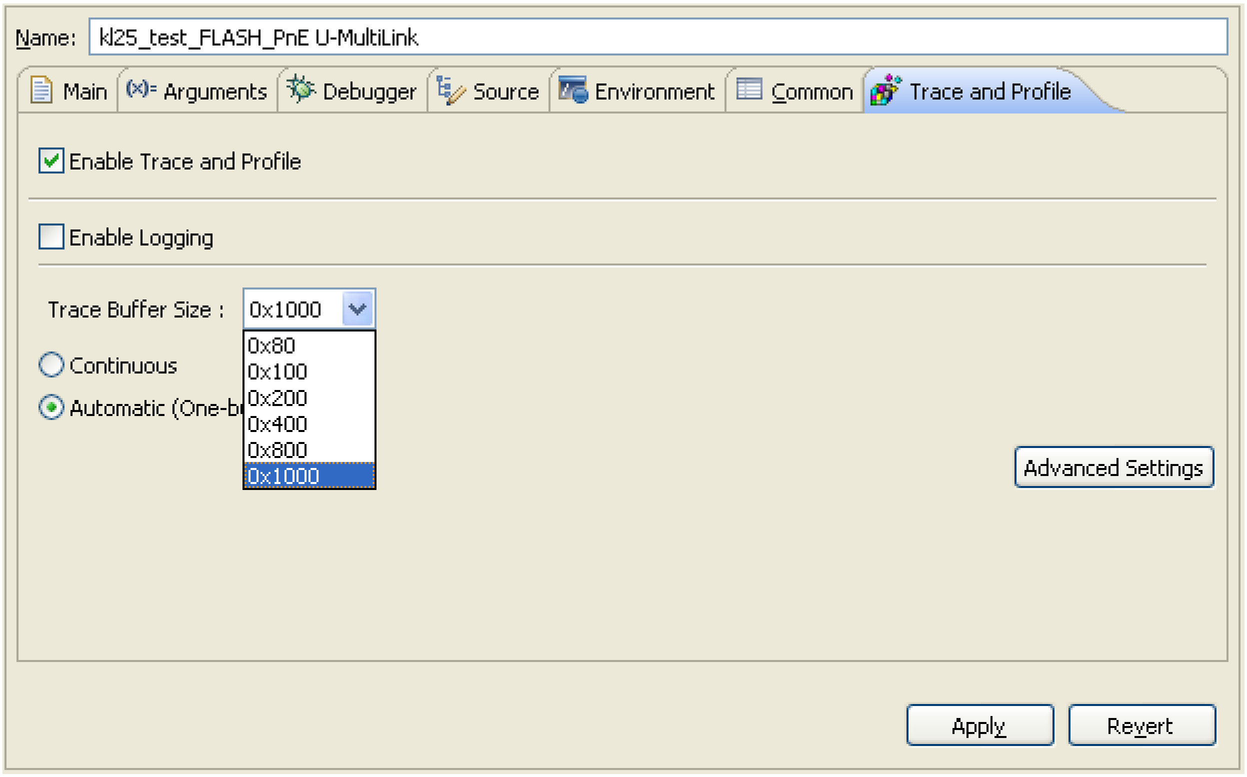
3.2.3.1 Configuring Kinetis Cortex M0+ Core

Perform the following steps to configure Kinetis Cortex M0+ core:

1. In the **CodeWarrior Projects** view, right-click 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 *Kl25_test_FLASH_PnE U-MultiLink*.
3. Check that the file `sa_mtb.c` exists in the **Sources** folder of the **CodeWarrior Projects** view.
4. Select the **Trace and Profile** tab.
5. Check the **Enable Trace and Profile** checkbox.
6. In the **Trace Buffer Size** drop-down box, change the value of the trace buffer size to *0x100* for the KL05 device and *0x200* for the KL25 device. There are a variable number of predefined sizes depending on the target. The maximum value varies from *0x80* on KL05Z to *0x1000* on KL25Z128.



Name: kl25_test_FLASH_PnE U-MultiLink

Main | Arguments | Debugger | Source | Environment | Common | Trace and Profile

[✓] Enable Trace and Profile

[ ] Enable Logging

Trace Buffer Size : 0x1000

0x80
0x100
0x200
0x400
0x800
0x1000

( ) Continuous

(●) Automatic (One-b

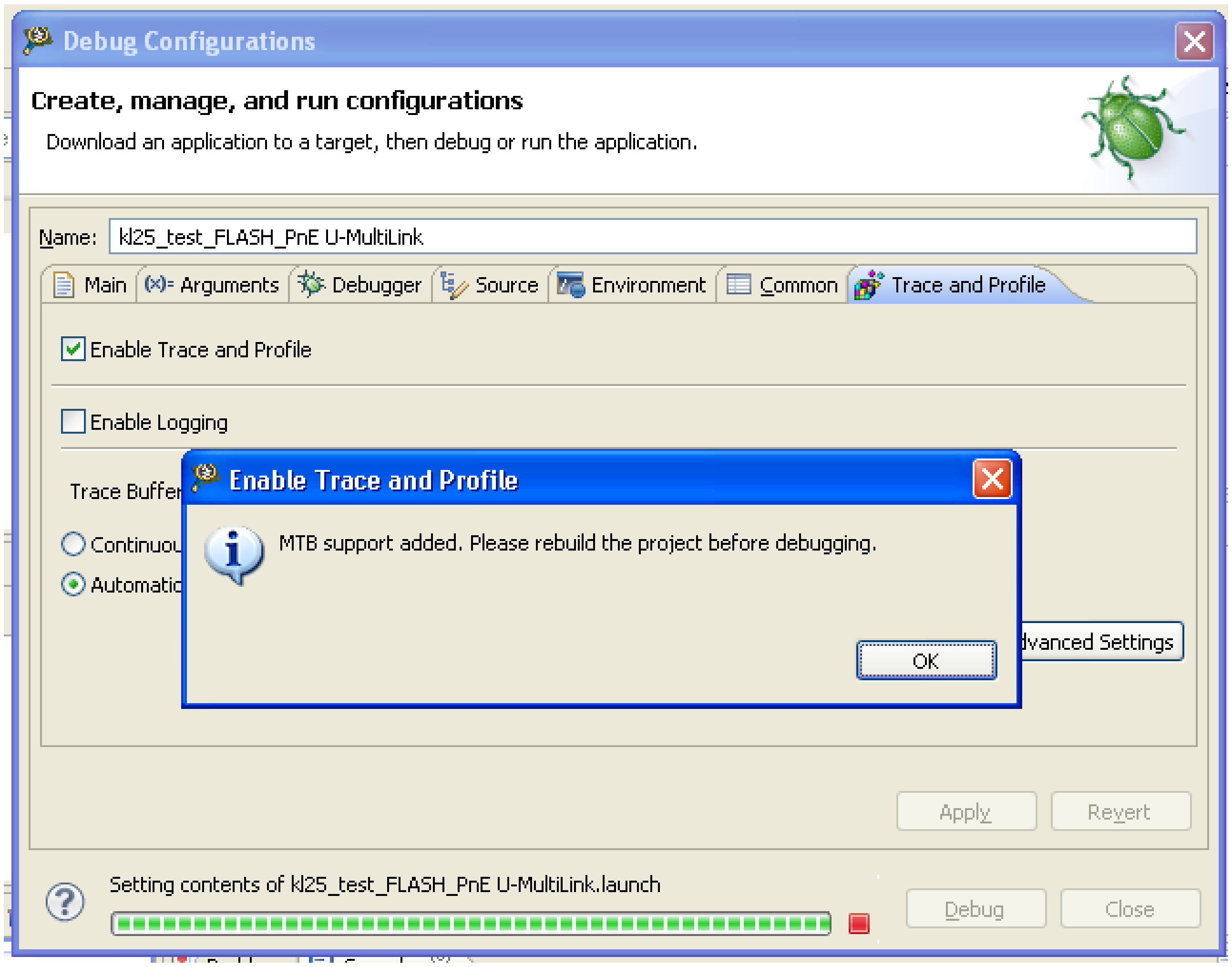
Advanced Settings

Apply | Revert

Figure 3-16. Setting Trace Buffer Size

7. Click **Apply**.

   The **Enable Trace and Profile** message box appears displaying that MTB support is enabled.


Debug Configurations

Create, manage, and run configurations

Download an application to a target, then debug or run the application.

Name: kl25_test_FLASH_PnE U-MultiLink

☑ Enable Trace and Profile

☐ Enable Logging

Enable Trace and Profile

MTB support added. Please rebuild the project before debugging.

Setting contents of kl25_test_FLASH_PnE U-MultiLink.launch

Figure 3-17. Enable Trace and Profile Message Box

8. Click **OK** in the **Enable Trace and Profile** message box.
9. Build the project again.

NOTE

You can remove the MTB support from the project by using the **CodeWarrior Projects** view. Right-click the project in the **CodeWarrior Projects** view, and select **Profiler > Remove MTB support** to remove the MTB support. You can select **Profiler > Add MTB support** to add it again.

3.2.3.2 Difference Between Kinetis Cores Cortex M4 and Cortex M0+

The K series and L series of the Kinetis target have different cores, trace modules, and raw trace format. However, the results available in the **Software Analysis** view look identical.

The K series of the Kinetis target has the following features:

- Cortex M4 core
- ETM and ITM trace with an internal ETB buffer (2Kb) located in RAM
- DWT module for trace control/HW Tracepoints with four address comparators and a large variety of resource counters

The L series of the Kinetis target has the following features:

- Cortex M0+ core
- Micro Trace Buffer (MTB) trace with an internal buffer, which is configurable up to 32Kb. MTB is located in SRAM. MTB trace is less compressed than ETM trace having a ratio of 128 branches for 1Kb of buffer. Like ETM, MTB also supports automatic and continuous trace.
- DWT module for trace control/HW Tracepoints with two address comparators and no resource counter.

The differences between Kinetis Cortex M4 core and Cortex M0+ core with respect to ETM and MTB are:

- MTB allows you to configure buffer size while ETB size is fixed.
- MTB shares the same SRAM memory as program and data, therefore the linker control file is updated with a new section for MTB so that trace collection does not interfere with program or data.
- MTB trace is not encoded with relative addresses (as ETM trace is), so it does not require SYNC information. Therefore, in the automatic mode, none of the existing trace will be lost doe to missing SYNC information.
- The K series supports real time trace collection by an external probe due to its TPIU module. There are probes with large internal buffers which can collect this external trace, for example, Segger J-Trace with 2-16Mb internal buffer; P&E Multilink Trace with 128 Mb internal buffer. The L series does not support this feature.

3.2.3.3 Trace Collection on Kinetis

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

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

3.2.3.3.1 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.

3.2.3.3.1.1 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.

3.2.3.3.2 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 is an optional application-driven trace source that supports printf style debugging to trace operating system and application events, and generates diagnostic system information.

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.

NOTE

If you collect ITM trace through TPIU (that is not using ETB buffer), you may lose ITM packets when ITM trace throughput is very high, especially when the executed code includes many branches with less sequential instructions and there is a lot of ETM trace generating. Since ETM has higher priority than ITM and when the trace generation exceeds the trace port bandwidth, the trace with the smaller priority is lost. To ensure enough bandwidth for ITM trace, disable ETM and ITM profiling counters and collect only ITM instrumentation trace with stimulus register 1.

ITM supports Timestamps and Synchronization.

3.2.3.3.2.1 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.

3.2.3.3.2.2 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.

3.2.3.3.3 Embedded Trace Buffer (ETB)

ETB stores data that ETM or ITM or both 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.

The ETM and ITM blocks generate trace. The processor dumps the generated trace into ETB, which is a memory buffer from the on-chip memory. The bottleneck here is how to transfer the collected trace data from the chip's memory to the computer. This is where probes are useful which implement a protocol that allows communication between the chip and the computer. These probes have the ability to read registers and chip's memory.

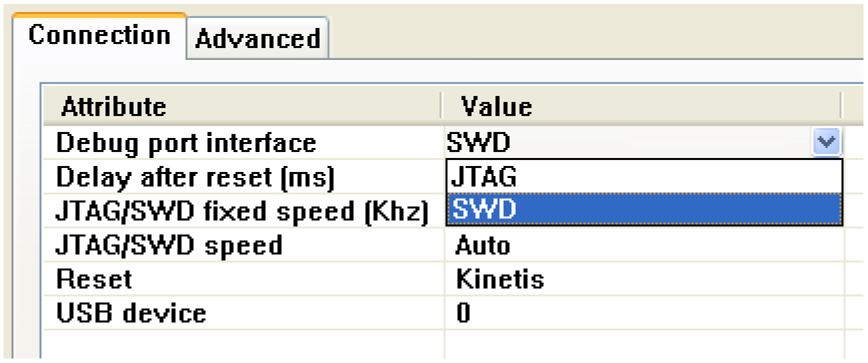
ETB stores only 2KB of trace data and it transfers data at a very slow speed. An alternative to ETB is to output the trace data to a port instead of storing it in the chip's memory. There is a block on the processor that manages the output of trace called Trace Port Interface Unit (TPIU). If the trace is output to an external port, it must be stored in real-time, and therefore, a probe is necessary.

3.2.3.3.4 J-Trace

The J-Trace probe has an internal memory buffer of 4MB where it can store trace data. It supports two modes of trace collection, TPIU (or Rawtrace) and Serial Wire Output (SWO), depending on the configuration of the processor. TPIU can output complex trace, that is both ETM and ITM. SWO is a lightweight standard and can only output ITM trace. The benefits of using J-Trace probe is the bigger memory size and the better collection speed. The drawback is that you cannot collect continuous trace, only the last 4MB of trace is stored.

To collect trace data on the Kinetis target using the JTrace connection:

1. Create a Kinetis project with **Segger J-Link** connection selected in the **Connections** screen.
2. Build the project.
3. Open the **Debug Configurations** dialog box.
4. Select your project in the tree structure on the left.
5. In the **Main** tab, click **Edit** . The **Properties** dialog box appears.
6. In the **Connection** tab, select **SWD** from the **Debug port interface** drop-down list.



Attribute	Value
Debug port interface	SWD
Delay after reset (ms)	
JTAG/SWD fixed speed (Khz)	
JTAG/SWD speed	Auto
Reset	Kinetis
USB device	0

Figure 3-18. Selecting Debug Port Interface for JTrace

7. Click **OK**.
8. Select the **Trace and Profile** tab and enable tracing and profiling.
9. Select the **JTrace** option and then select the **SWO** option.

NOTE

If you choose **TPIU** in the **Trace and Profile** tab for JTrace, you can select either **JTAG** or **SWD** (Serial Wire Debug) as debug port interface.

10. Click **Apply**.
11. Debug the application and collect trace.

Refer Collecting Data to know how to collect data on Kinetis target. Refer Viewing Data to know how to view collected trace data.

3.2.4 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. For Kinetis Cortex M4, the default page is **ETM Settings**.

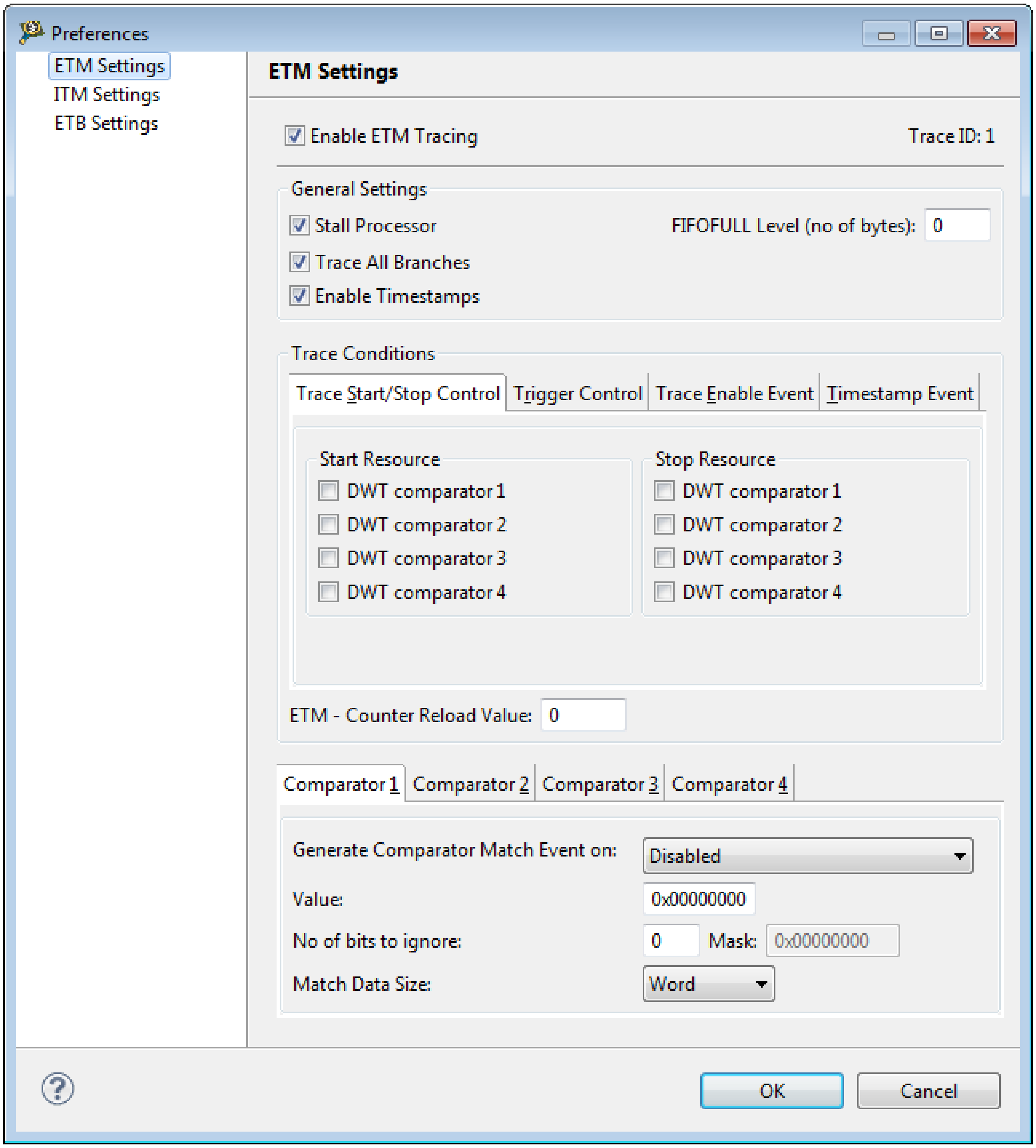
NOTE

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

NOTE

If you are configuring Kinetis Cortex M4 project, the **Preferences** dialog box displays ETM, ITM, and ETB trace settings since Kinetis Cortex M4 supports ETM and ITM trace. Kinetis Cortex M0+ supports MTB trace, therefore, the **Preferences** dialog box for a Kinetis Cortex M0+ project displays MTB trace settings only.

2. If you are configuring settings for Kinetis Cortex M4, ensure that the **Enable ETM Tracing** checkbox is checked.



ETM Settings

[✓] Enable ETM Tracing — Trace ID: 1

General Settings

- [✓] Stall Processor — FIFOFULL Level (no of bytes): 0
- [✓] Trace All Branches
- [✓] Enable Timestamps

Trace Conditions

Trace Start/Stop Control | Trigger Control | Trace Enable Event | Timestamp Event

Start Resource	Stop Resource
[ ] DWT comparator 1	[ ] DWT comparator 1
[ ] DWT comparator 2	[ ] DWT comparator 2
[ ] DWT comparator 3	[ ] DWT comparator 3
[ ] DWT comparator 4	[ ] DWT comparator 4

ETM - Counter Reload Value: 0

Comparator 1 | Comparator 2 | Comparator 3 | Comparator 4

Generate Comparator Match Event on:	Disabled
Value:	0x00000000
No of bits to ignore:	0 — Mask: 0x00000000
Match Data Size:	Word

Figure 3-19. Preferences Dialog Box

The table below describes the various options available on the **ETM Settings** page.

Table 3-4. 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. **NOTE:** The same settings are displayed for configuring MTB trace in case you are using a Kinetis M0+ project. The difference is that in ETM, there are 4 address comparators, and in MTB, there are only 2.
	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.

Table continues on the next page...

Table 3-4. Options Available in ETM Settings Page (continued)

Group	Option	Description
	Stop Resource	Selects the corresponding DWT comparator to control trace stop.
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.
	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**

Table continues on the next page...

Table 3-4. Options Available in ETM Settings Page (continued)

Group	Option	Description
		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.
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

Table continues on the next page...

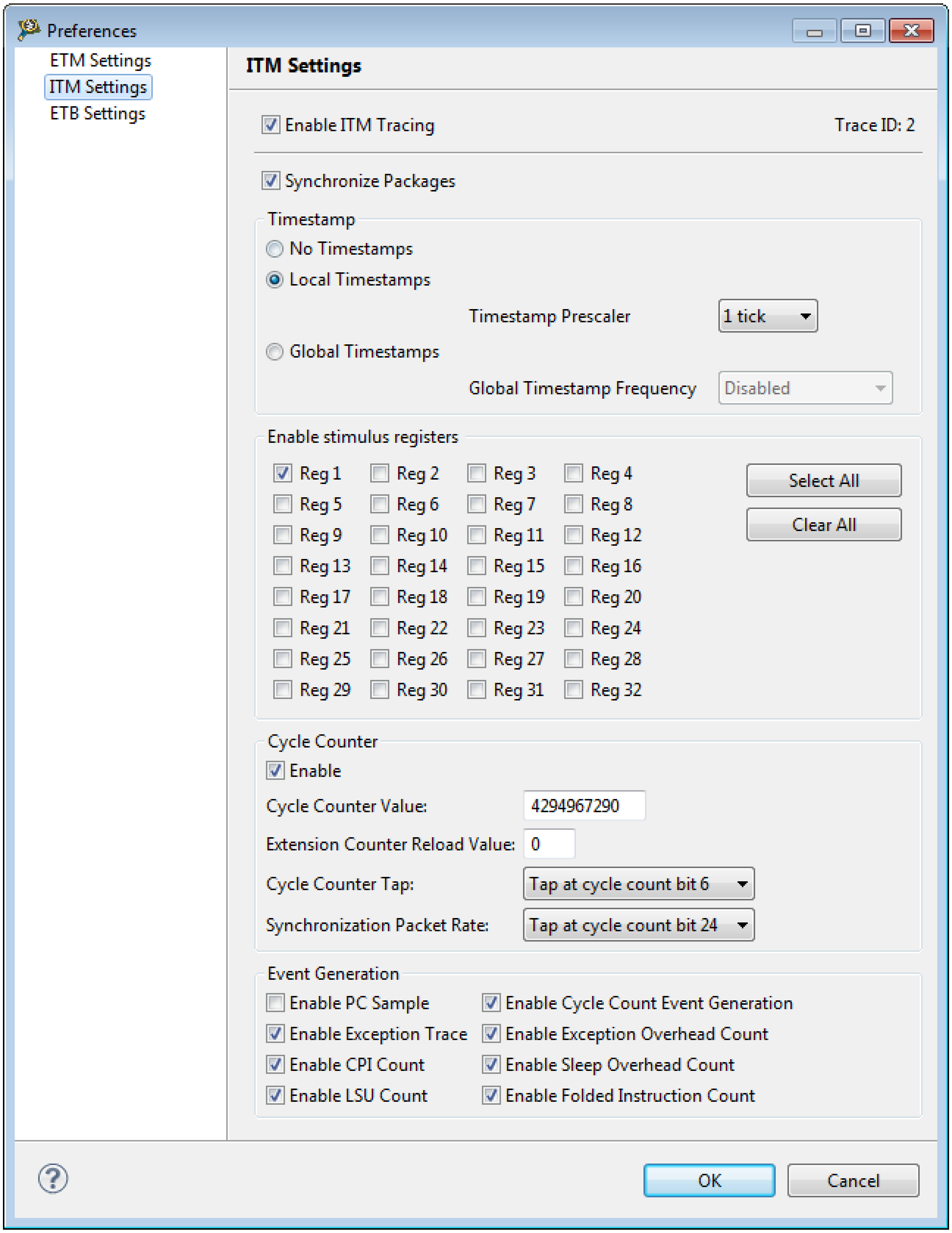
Table 3-4. Options Available in ETM Settings Page (continued)

Group	Option	Description
		• 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.
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.



Preferences

- ETM Settings
- ITM Settings
- ETB Settings

ITM Settings

[✓] Enable ITM Tracing — Trace ID: 2

[✓] Synchronize Packages

Timestamp

- ( ) No Timestamps
- (●) Local Timestamps
  - Timestamp Prescaler: 1 tick
- ( ) Global Timestamps
  - Global Timestamp Frequency: Disabled

Enable stimulus registers

[✓] Reg 1	[ ] Reg 2	[ ] Reg 3	[ ] Reg 4
[ ] Reg 5	[ ] Reg 6	[ ] Reg 7	[ ] Reg 8
[ ] Reg 9	[ ] Reg 10	[ ] Reg 11	[ ] Reg 12
[ ] Reg 13	[ ] Reg 14	[ ] Reg 15	[ ] Reg 16
[ ] Reg 17	[ ] Reg 18	[ ] Reg 19	[ ] Reg 20
[ ] Reg 21	[ ] Reg 22	[ ] Reg 23	[ ] Reg 24
[ ] Reg 25	[ ] Reg 26	[ ] Reg 27	[ ] Reg 28
[ ] Reg 29	[ ] Reg 30	[ ] Reg 31	[ ] Reg 32

Select All | Clear All

Cycle Counter

[✓] Enable

Cycle Counter Value: 4294967290

Extension Counter Reload Value: 0

Cycle Counter Tap: Tap at cycle count bit 6

Synchronization Packet Rate: Tap at cycle count bit 24

Event Generation

[ ] Enable PC Sample	[✓] Enable Cycle Count Event Generation
[✓] Enable Exception Trace	[✓] Enable Exception Overhead Count
[✓] Enable CPI Count	[✓] Enable Sleep Overhead Count
[✓] Enable LSU Count	[✓] Enable Folded Instruction Count

OK | Cancel

Figure 3-20. ITM Settings Page

The table below describes the various options available on the **ITM Settings** page.

Table 3-5. Options Available in ITM Settings Page

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

Table continues on the next page...

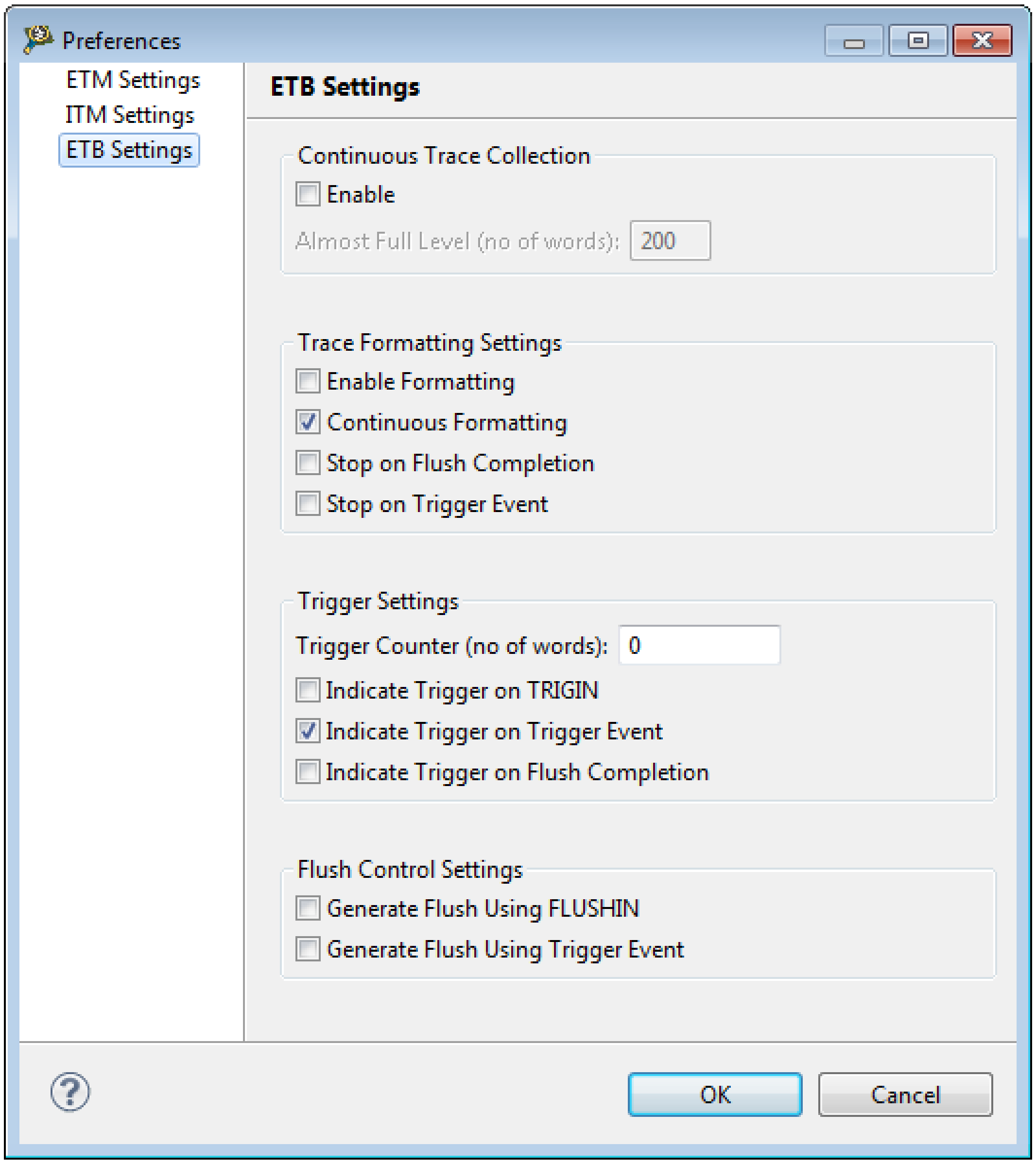
Table 3-5. Options Available in ITM Settings Page (continued)

Group	Option	Description
	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).
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.

Table 3-5. Options Available in ITM Settings Page

Group	Option	Description
		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.



ETB Settings

Continuous Trace Collection
- [ ] Enable

Almost Full Level (no of words): 200

Trace Formatting Settings
- [ ] Enable Formatting
- [x] Continuous Formatting
- [ ] Stop on Flush Completion
- [ ] Stop on Trigger Event

Trigger Settings

Trigger Counter (no of words): 0
- [ ] Indicate Trigger on TRIGIN
- [x] Indicate Trigger on Trigger Event
- [ ] Indicate Trigger on Flush Completion

Flush Control Settings
- [ ] Generate Flush Using FLUSHIN
- [ ] Generate Flush Using Trigger Event

Figure 3-21. ETB Settings Page

The table below describes the various options available on the **ETM Settings** page.

Table 3-6. Options Available in ETB Settings Page

Group	Option	Description
Enable ETB Trace Capture		Enables ETB tracing.
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*.
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 continues on the next page...

Table 3-6. Options Available in ETB Settings Page (continued)

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.
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.

Table continues on the next page...

Table 3-6. Options Available in ETB Settings Page (continued)

Group	Option	Description
	Generate Flush Using FLUSHIN	Generates flush using the FLUSHIN interface.
	Generate Flush Using Trigger Event	Generates flush using the trigger event.

3.2.5 Configure MPC56xx Target

To configure the launch configuration for the MPC56xx target:

1. In the **CodeWarrior Projects** view, right-click 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 *MPC5675K_RAM_PnE USB-ML-PPCNEXUS*.

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.

This configures the launch configuration for the MPC56xx target.

NOTE

Tracing is not supported for MPC56xx processors with z0 core as these processors do not have a trace buffer.

3.2.6 Configure S12Z Target

To configure the launch configuration for the S12Z target:

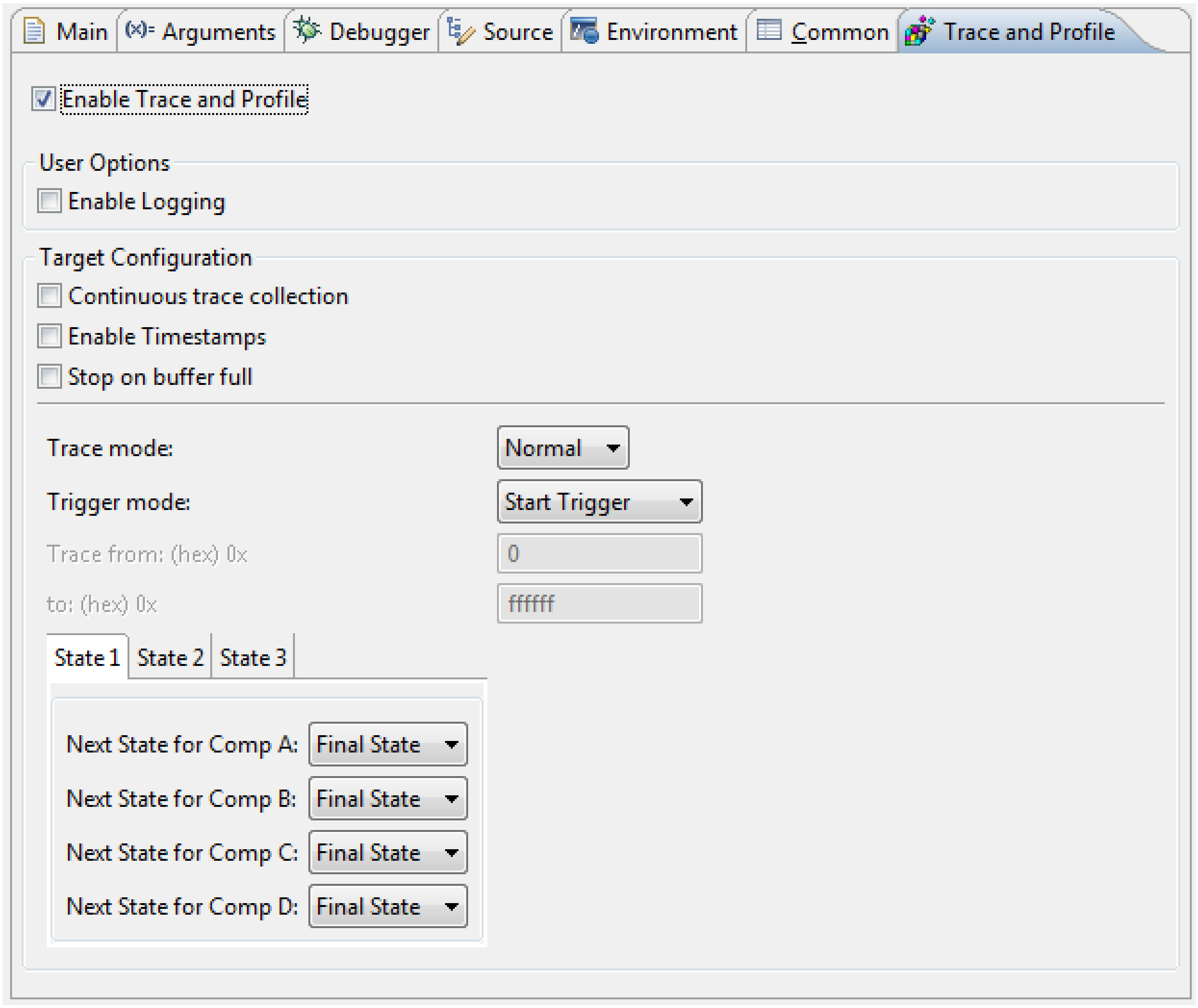
1. In the **CodeWarrior Projects** view, right-click the project and select **Debug As > Debug Configurations** from the context menu.

   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 *S12z_FLASH_PnE U-MultiLink*.

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.



☑ Enable Trace and Profile

User Options

☐ Enable Logging

Target Configuration

☐ Continuous trace collection

☐ Enable Timestamps

☐ Stop on buffer full

Trace mode: Normal

Trigger mode: Start Trigger

Trace from: (hex) 0x 0

to: (hex) 0x ffffff

State 1 | State 2 | State 3

Next State for Comp A: Final State

Next State for Comp B: Final State

Next State for Comp C: Final State

Next State for Comp D: Final State

Figure 3-22. Trace and Profile Tab Options of S12Z

The table below describes the various **Trace and Profile** options for S12Z architecture.

Table 3-7. Trace and Profile Options for S12Z

Group	Option	Description
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.
Target Configuration	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.
	Enable Timsestamps	Enables time stamping when checked.
	Stop on buffer full	Suspends the target automatically and stops trace collection when buffer is full.
Trace Mode		Allows you to select any one of the following trace modes: • Normal - collects trace in normal mode and stores change of flow (COF) program counter (PC) addresses. • Loop1 - lets you collect trace 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. • Detail - allows you to capture specific values using triggers. When selected, enables **Trace from: 0x** and **to: 0x** where you can specify the limit of the addresses for which you want to collect trace. • Pure PC - collects trace in normal mode without timestamps.
Trigger Mode		Allows you to select any one of the following triggers for the trigger set in the source code: • End Trigger - collects trace data after the trigger is hit • Start Trigger - collects trace data until the trigger is hit. • Middle Trigger - collects some data until the trigger is hit and some data after the trigger hits.

Table continues on the next page...

Table 3-7. Trace and Profile Options for S12Z (continued)

Group	Option	Description
Trace from: 0x		Specifies the address from which you want to collect trace data.
to: 0x		Specifies the address till which you want to collect trace data.
State 1, 2, 3	Next State for Comp A/B/C/D	Specify the logic for starting the trace collection. The trace collection starts when the target enters the **Final State** . The drop-down lists for each state allows you to select the next state when the corresponding comparator is hit. For example, in the **State 1** tab, you select **Next State for Comp A** as **State 2** , that is when comparator A is hit you want to enter **State 2** . Then in the **State 2** tab, when comparator B is hit, you want to enter **State 3** . Then in **State 3** when comparator C is hit, you want to enter the **Final State** and begin collecting trace. **NOTE:** The comparators are the triggers A,B,C,D, which you can set in the source code, on variables, or in memory. To set triggers in the source code, select **Trace Triggers > Toggle Trace Trigger A/B/C/D**.

NOTE

On S12z platform, breakpoints and triggers are mutually exclusive. You may use only breakpoints or only triggers but never both during debugging a S12z application.

3.2.7 Configure ColdFire V2-V4 Targets

To configure the launch configuration for the ColdFire V2-V4 targets:

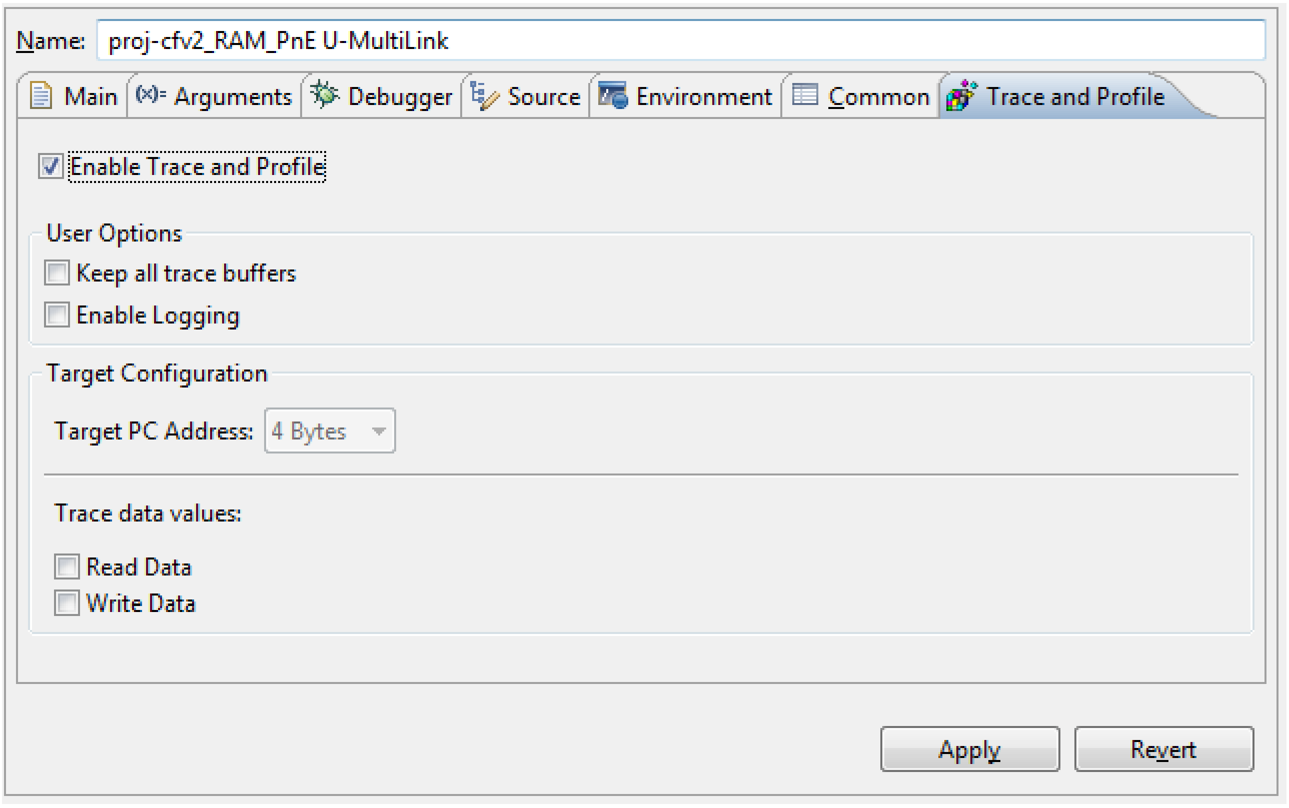
1. In the **CodeWarrior Projects** view, right-click the project and select **Debug As > Debug Configurations** from the context menu.

   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-cfv2_RAM_PnE U-MultiLink*.

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.


Name: proj-cfv2_RAM_PnE U-MultiLink

Tabs: Main, Arguments, Debugger, Source, Environment, Common, Trace and Profile

[✓] Enable Trace and Profile

User Options
[ ] Keep all trace buffers
[ ] Enable Logging

Target Configuration
Target PC Address: 4 Bytes

Trace data values:
[ ] Read Data
[ ] Write Data

Figure 3-23. Trace and Profile Tab Options of ColdFire V2-V4

The table below describes the various **Trace and Profile** options for ColdFire V2-V4 targets.

Table 3-8. Trace and Profile Options for ColdFire V2-V4

Group	Option	Description
Enable Trace and Profile		Enables tracing and profiling for the application.
User Options	Keep all trace buffers	Lets you keep all the data received from the probe. On Coldfire V2-V4 targets, you can collect trace with external buffer only using Tracelink probes. The Tracelink (or multilink trace) probes cannot collect trace continuously, but they have a very large buffer. So suspending the application repeatedly with this option enabled will append the trace buffers, as if each buffer was collected alone, resulting in a very large trace collection.

Table continues on the next page...

Table 3-8. Trace and Profile Options for ColdFire V2-V4 (continued)

Group	Option	Description
	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.
Target Configuration	Target PC Address	Specifies the number of bytes that the trace hardware uses to store the addresses in the raw trace. The value is currently fixed at 4 bytes.
Trace data values	Read Data	Traces the values of data operands being read from the memory.
	Write Data	Traces the values of data operands being written to the memory.

3.2.8 Configure DSC Target

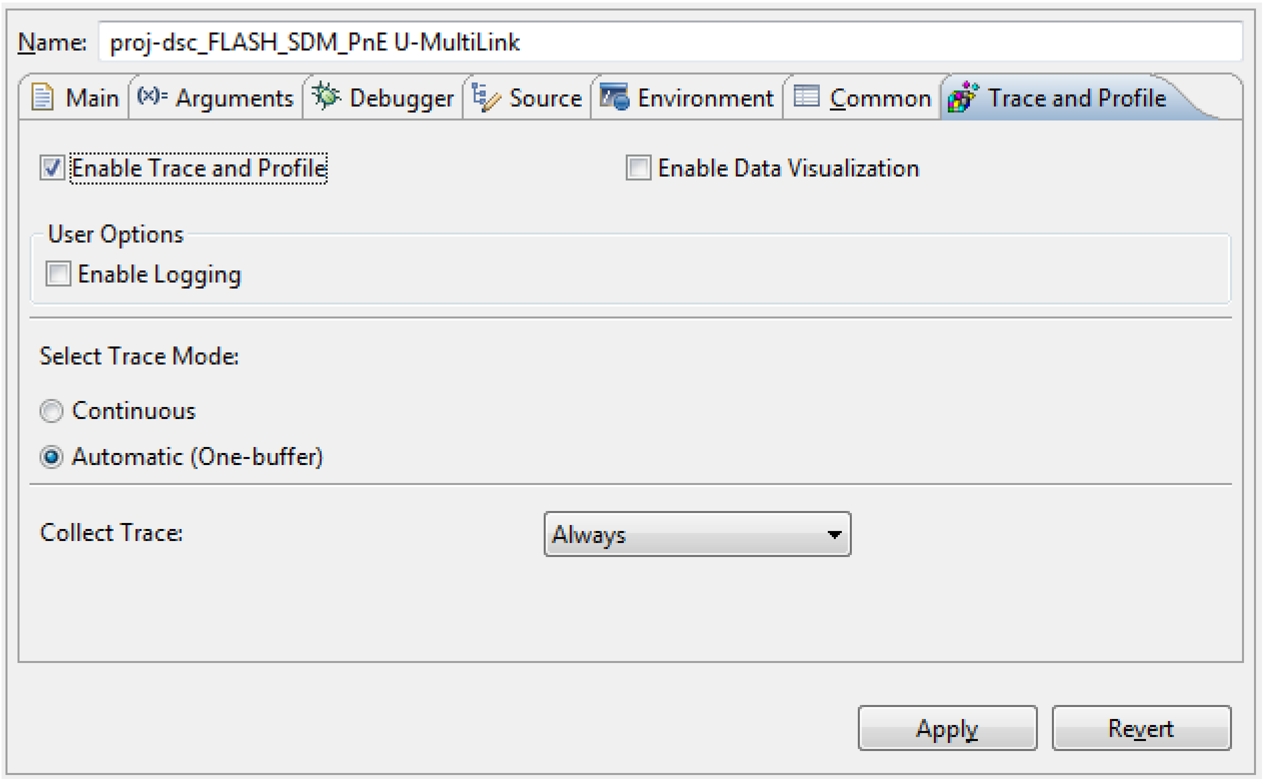
To configure the launch configuration for the DSC target:

1. In the **CodeWarrior Projects** view, right-click 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-dsc_FLASH_SDM_PnE U-MultiLink*.

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.



Name: proj-dsc_FLASH_SDM_PnE U-MultiLink

Main | Arguments | Debugger | Source | Environment | Common | Trace and Profile

[✓] Enable Trace and Profile
[ ] Enable Data Visualization

User Options
[ ] Enable Logging

Select Trace Mode:
( ) Continuous
(●) Automatic (One-buffer)

Collect Trace: Always

Apply | Revert

Figure 3-24. Trace and Profile Tab Options of DSC

The table below describes the various **Trace and Profile** options for DSC target.

Table 3-9. Trace and Profile Options for DSC

Group	Option	Description
Enable Trace and Profile		Enables trace and profiling for the DSC application.
Enable Data Visualization		Enables data visualization. Data Visualization and tracing cannot be used simultaneously. For details, refer Data Visualization.
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.
Select Trace Mode	Continuous	When selected, collects the trace data continuously.
	Automatic (One-buffer)	When selected, captures only the last instructions executed before the target gets suspended.
Collect Trace		Collects trace with any of the following trigger conditions selected: • Always — captures all executed instructions in the application

Table 3-9. Trace and Profile Options for DSC

Group	Option	Description
		• From Trigger A Onward — starts trace collection from where trigger A is set in the application • From Trigger B Onward — starts trace collection from where trigger B is set in the application • Until Trigger A — collects trace till Trigger A is set in the application • Until Trigger B — collects trace till Trigger B is set in the application

NOTE

On DSC platform, breakpoints and triggers are mutually exclusive. You may use only breakpoints or only triggers but never both during debugging a DSC application.

NOTE

The triggers set on the DSC target work on program prefetch and not on the program execution. Therefore, triggering occurs a bit earlier than the execution of the instruction on which the trigger has been set. To prevent this, if you don't have sequential instructions (statements) before the line where you want to set a trigger, you can set it in the **Disassembly** view on the instructions following one or two sequential instructions so that beginning of the trace is not affected.

NOTE

The DSC hardware traces destinations for only a subset of the change of flow instructions (please see the EOnCE manual, section "11.6.1 Trace Buffer Control Register"). Indirect change of flow instructions, such as returns (`rts`), indirect calls (such as `JSR R0`), and some direct branches (such as *BRA*) are not traced. As a result, in some cases, the trace data collected for a DSC project may contain *<no debug info>* when a return address cannot be estimated, or entire functions may be missing when they are called by a return or a indirect call. Also, collecting data in the automatic mode or setting triggers in the middle of the program may result into more returns than calls, which makes processing break and display *<no debug info>* instead of an address or "Break in trace!" messages in the collected trace data. Because of the above limitations, the **Call Tree** and

Performance hyperlinks do not appear in the **Software Analysis** view for DSC applications.

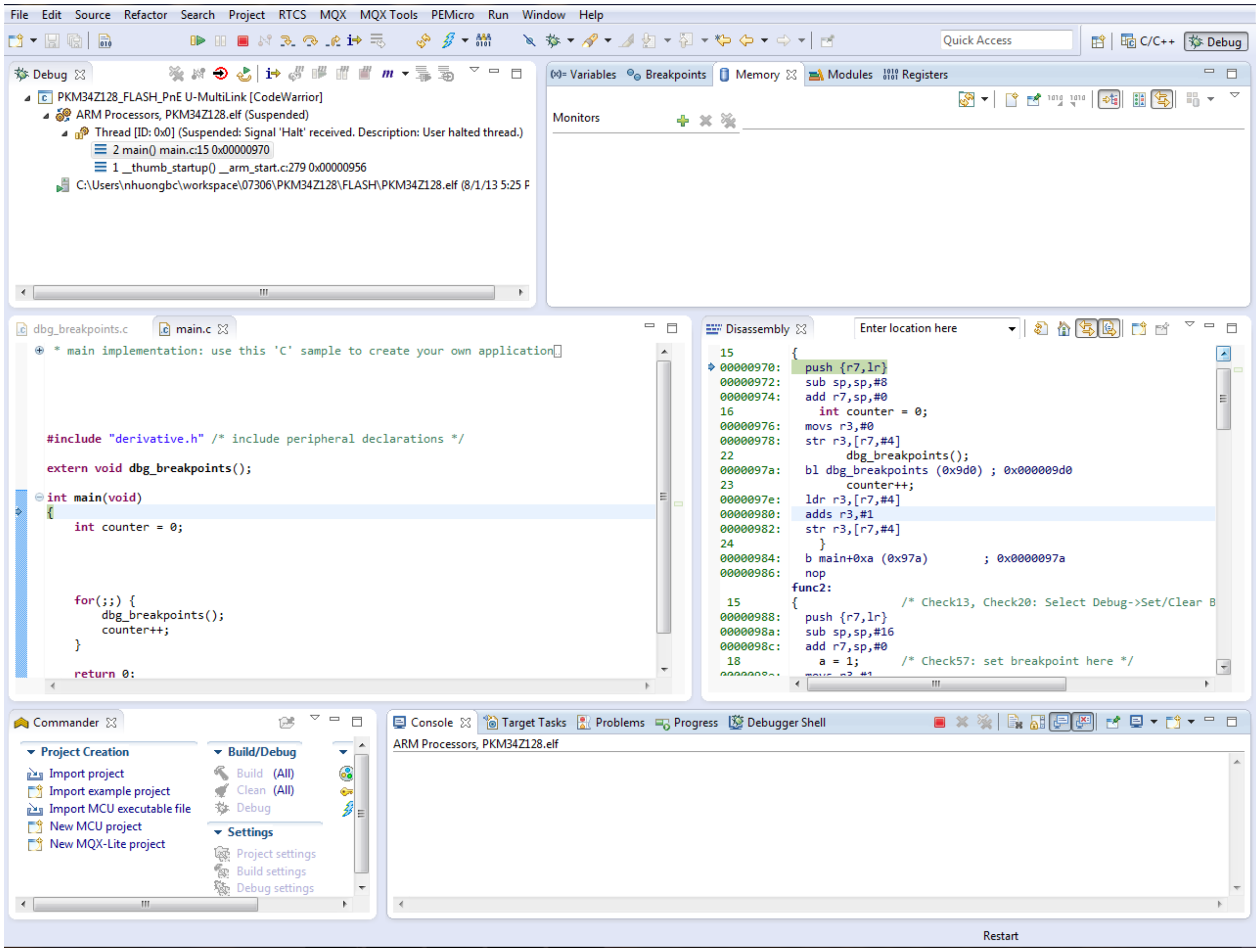
3.3 Collecting Data

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

To collect data on any target such as HCS08, ColdFire V1, Kinetis, MPC56xx, or DSC:

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

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



```
PKM34Z128_FLASH_PnE U-MultiLink [CodeWarrior]
  ARM Processors, PKM34Z128.elf (Suspended)
    Thread [ID: 0x0] (Suspended: Signal 'Halt' received. Description: User halted thread.)
      2 main() main.c:15 0x00000970
      1 __thumb_startup() __arm_start.c:279 0x00000956
  C:\Users\nhuongbc\workspace\07306\PKM34Z128\FLASH\PKM34Z128.elf (8/1/13 5:25 P
```

```
 * main implementation: use this 'C' sample to create your own application




#include "derivative.h" /* include peripheral declarations */

extern void dbg_breakpoints();

int main(void)
{
    int counter = 0;



    for(;;) {
        dbg_breakpoints();
        counter++;
    }

    return 0:
```

```
15          {
00000970:     push {r7,lr}
00000972:     sub sp,sp,#8
00000974:     add r7,sp,#0
16              int counter = 0;
00000976:     movs r3,#0
00000978:     str r3,[r7,#4]
22                   dbg_breakpoints();
0000097a:     bl dbg_breakpoints (0x9d0) ; 0x000009d0
23                   counter++;
0000097e:     ldr r3,[r7,#4]
00000980:     adds r3,#1
00000982:     str r3,[r7,#4]
24              }
00000984:     b main+0xa (0x97a)        ; 0x0000097a
00000986:     nop
            func2:
 15         {              /* Check13, Check20: Select Debug->Set/Clear B
00000988:     push {r7,lr}
0000098a:     sub sp,sp,#16
0000098c:     add r7,sp,#0
 18             a = 1;     /* Check57: set breakpoint here */
```

```
ARM Processors, PKM34Z128.elf
```

Figure 3-25. Debug Perspective Page

2. Click the **Resume** button 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** button. Click the **Terminate** button to stop the measurement.

If you are collecting trace on MPC5668G/E targets with both cores (e200z6 and e200z0), click the **Multicore Resume** button to start trace collection and click **Multicore Suspend** to suspend it.

NOTE

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

NOTE

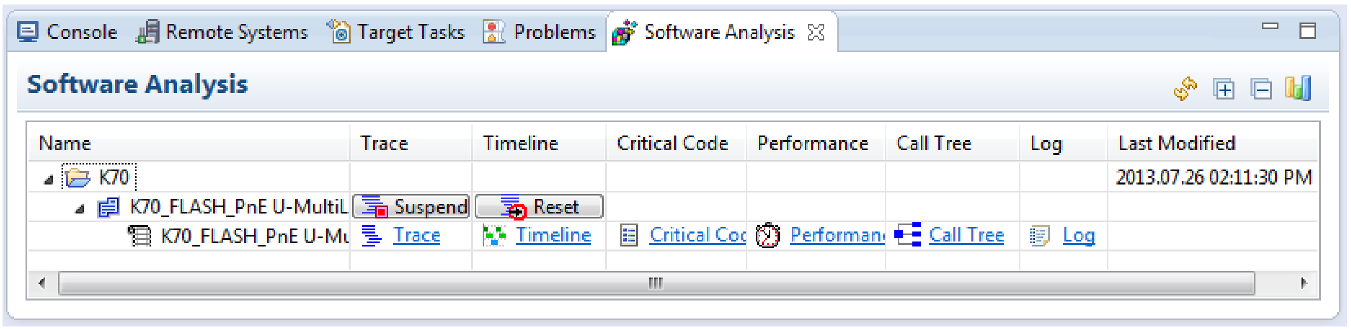
The e200z6 and e200z0 cores are not running the same code; Core 0 (e200z6) runs code from *Sources/main.c* and Core 1 (e200z0) runs from *PRC1_Sources/main_p1.c*. Therefore, the two cores do not stop at same entry point at first suspend after starting the debug session.

3.4 Viewing Data

When an application runs on any target, such as HCS08, ColdFire V1, Kinetis, MPC56xx, or DSC, the data files are generated and get displayed in the **Software Analysis** view. The **Software Analysis** view appears automatically while the application is running. To manually open it, select **Window > Show View > Software Analysis** from the menu bar.

1. Expand the project name in the **Software Analysis** view.

   The data source is listed under the project name along with the hyperlinks to the **Trace**, **Timeline**, **Critical Code**, **Performance**, and **Call Tree** results.



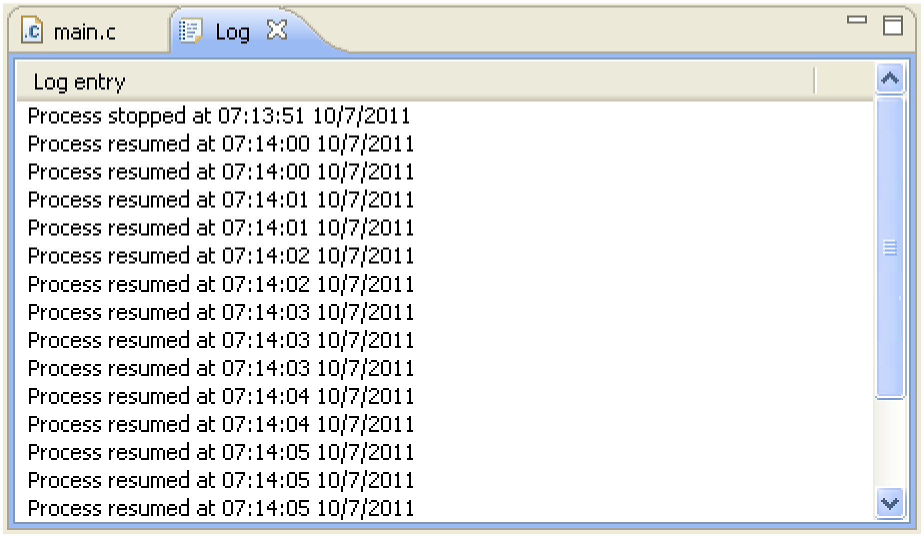
Name	Trace	Timeline	Critical Code	Performance	Call Tree	Log	Last Modified
K70							2013.07.26 02:11:30 PM
K70_FLASH_PnE U-MultiL	Suspend	Reset					
K70_FLASH_PnE U-Mu	Trace	Timeline	Critical Coc	Performan	Call Tree	Log	

Figure 3-26. software Analysis View - Kinetis Target

2. Click the **Trace**, **Timeline**, **Critical Code**, **Performance**, and **Call Tree** hyperlinks and view trace, timeline, critical code, and performance data results in the corresponding viewers.

You can perform the following actions in the **Software Analysis** view:

- Click the **Log** hyperlink in the **Log** column to view the log of actions that you performed while executing the target to collect trace data. The **Log** view appears displaying the performed actions along with the time when those actions were performed.



```
Log entry
Process stopped at 07:13:51 10/7/2011
Process resumed at 07:14:00 10/7/2011
Process resumed at 07:14:00 10/7/2011
Process resumed at 07:14:01 10/7/2011
Process resumed at 07:14:01 10/7/2011
Process resumed at 07:14:02 10/7/2011
Process resumed at 07:14:02 10/7/2011
Process resumed at 07:14:03 10/7/2011
Process resumed at 07:14:03 10/7/2011
Process resumed at 07:14:03 10/7/2011
Process resumed at 07:14:04 10/7/2011
Process resumed at 07:14:04 10/7/2011
Process resumed at 07:14:05 10/7/2011
Process resumed at 07:14:05 10/7/2011
Process resumed at 07:14:05 10/7/2011
```

Figure 3-27. Log View of Trace Collection

NOTE

The log entries appear if the **Enabled Logging** checkbox is checked in the **Trace and Profile** tab. If you choose to disable logging, no log appears in the **Log** view.

- Click the **Refresh the displayed data** button to refresh the displayed data. You no need to press this button manually at every trace collection. The trace results are refreshed automatically.
- Click the **Expands all nodes** button to expand all the nodes and the **Collapses all nodes** button to collapse all the nodes in the **Software Analysis** view.
- Click the **Import Data Trace from text file** button to import an external format of trace into your project. The **Data Trace Import** dialog box (Figure 3-28) appears. For more information, refer Data Trace Import Dialog Box.

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

To save trace, timeline, critical code, performance, and call tree results, select the **Save Results** option from the context menu. A data source with an `.0000` extension is added to the **Software Analysis** view containing the saved results. 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 **Software Analysis** view. For more information, refer Controlling Trace Generation.

3.4.1 Data Trace Import Dialog Box

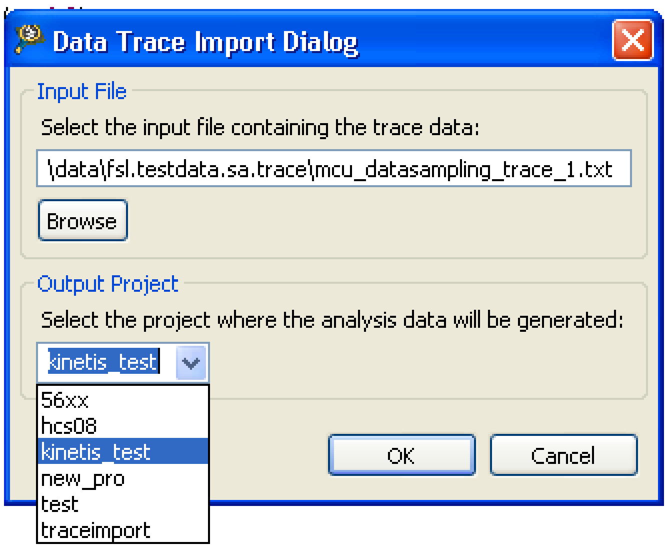
The **Data Trace Import** dialog box allows you to import an external format of trace which is different from the trace format of CodeWarrior Software Analysis. This trace format is generated from an external trace collection tool and can be saved in an input file in a text format. The trace data contained in this input file is converted into a format compatible with CodeWarrior. A sample input file, *dsc8257_Data_Visualization_1.txt*, is located at `<CWInstallDirectory>\MCU\morpho_sa\sasdk\data\fsl.testdata.sa.trace\` and `<CWInstallDirectory>\eclipse\plugins\com.freescale.morpho.sa.arm_*\data\fsl.testdata.sa.trace`.

NOTE

The **Data Trace Import** dialog box is used to import only data trace.

To import external trace format into your project:

1. Click the **Import Data Trace from text file** button in the **Software Analysis** view to display the **Data Trace Import** dialog box.
2. Locate the **Input File**.
3. From the **Output Project** drop-down list, select the project where you want to import the trace format.
4. Select the architecture for which the trace was collected from the **Select Platform** drop-down list.



Select the input file containing the trace data:

\data\fsl.testdata.sa.trace\mcu_datasampling_trace_1.txt

Select the project where the analysis data will be generated:

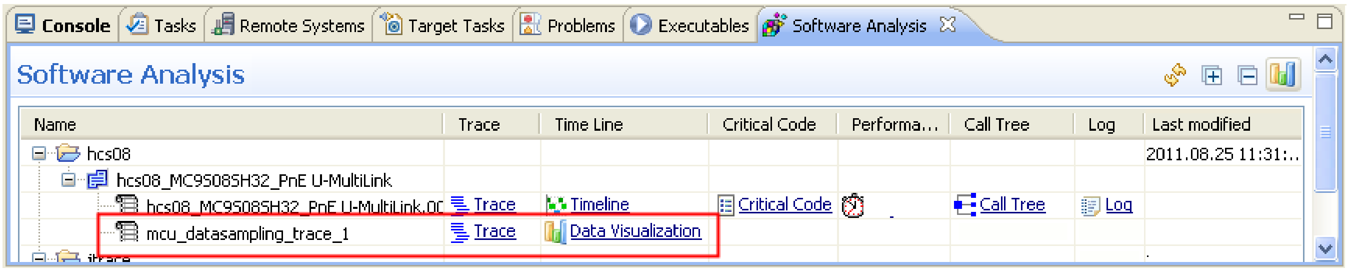
kinetis_test

56xx
hcs08
kinetis_test
new_pro
test
traceimport

Figure 3-28. Data Trace Import Dialog Box

5. Click **OK**.

   The results of the input file get displayed in the **Software Analysis** view. Two hyperlinks appear, **Trace** and **Data Visualization**.



Software Analysis

Name	Trace	Time Line	Critical Code	Performa...	Call Tree	Log	Last modified
hcs08							2011.08.25 11:31:..
hcs08_MC9S08SH32_PnE U-MultiLink							
hcs08_MC9S08SH32_PnE U-MultiLink.0C	Trace	Timeline	Critical Code		Call Tree	Log	
mcu_datasampling_trace_1	Trace	Data Visualization					

Figure 3-29. Software Analysis View After Importing Data Trace Format

6. Click the **Trace** hyperlink to view the imported trace results in CodeWarrior Software Analysis format.
7. Click the **Data Visualization** hyperlink to view the graphical representation of the imported trace results.

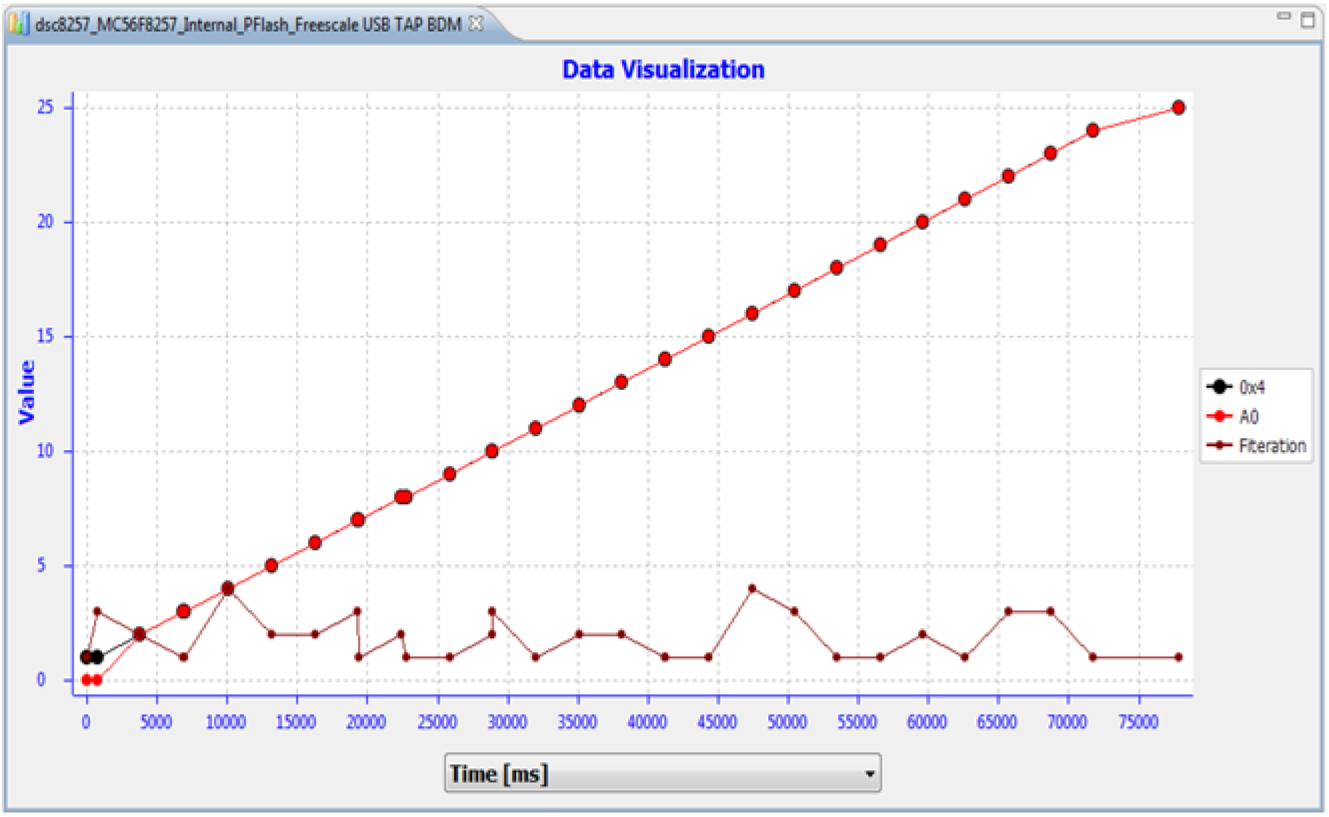


Figure 3-30. Data Visualization

The Data Visualization graph allows you to monitor variables of your application. You can identify and chart variables of the application against time. The chart displays tooltips when you place mouse cursor over the circular data dots.

The symbols have different sizes, so that multiple series points which coincide (ox/oy values) are visible. You can change the values displayed on the ox (time) axis between time and any of the series.

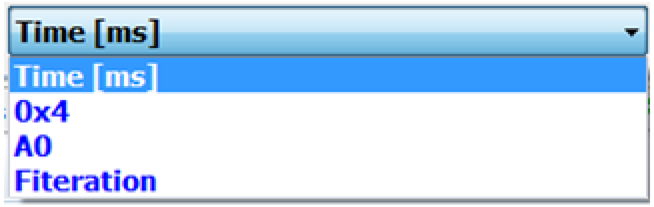


Figure 3-31. Changing Values Displayed on Time Axis

You can zoom-in and zoom-out the information in the graph using mouse scroll up and down. You can also select a particular area on the graph to zoom-in the information. Right-click in the graph to open a context menu, which allows you to:

- adjust X and Y axis range,
- show/hide legend,

- zoom-in/zoom-out the graph,
- zoom-in/zoom-out information on X axis or Y axis,
- save graph as picture,
- set various properties of graph such as color, legend, and grid style, and
- change series appearance, for example, show/hide, line color, symbol color/shape/ size, show/hide labels
- logarithmic scale.

For more information, refer Data Visualization.

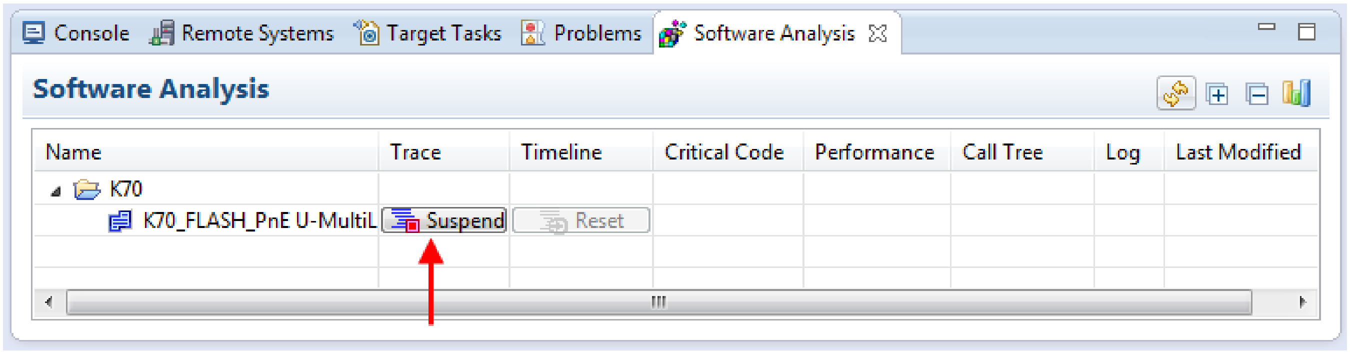
3.4.2 Controlling Trace Generation

Every project that has the Trace and Profile enabled will be listed in the **Software Analysis** view. You can control the generation of trace from the **Software Analysis** view using:

- Resume/Suspend Toggle Button
- Reset Button

3.4.2.1 Resume/Suspend Toggle Button

The **Resume/Suspend** toggle 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 **Suspend** toggle button appears next to the data source. When clicked, the button toggles to **Resume**.



Name	Trace	Timeline	Critical Code	Performance	Call Tree	Log	Last Modified
K70							
K70_FLASH_PnE U-MultiL	Suspend	Reset					

Figure 3-32. Stop Toggle Button

To stop trace collection:

1. Debug the application.

The **Suspend** toggle button appears next to the data source name in the **Software Analysis** view.

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

   The button toggles to **Resume**.

3. Click **Resume** in the **Debug** view.
4. Click **Suspend** in the **Debug** view after some time.
5. Look at the trace data in the **Software Analysis** view.

   The trace is not collected.

To start trace collection:

1. Click **Resume** toggle button.

   The button toggles to **Suspend**.

2. Click **Resume** in the **Debug** view.
3. Click **Suspend** in the **Debug** view after some time.
4. Look at the trace data in the **Software Analysis** view.

   The trace is collected.

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

3.4.2.2 Reset Button

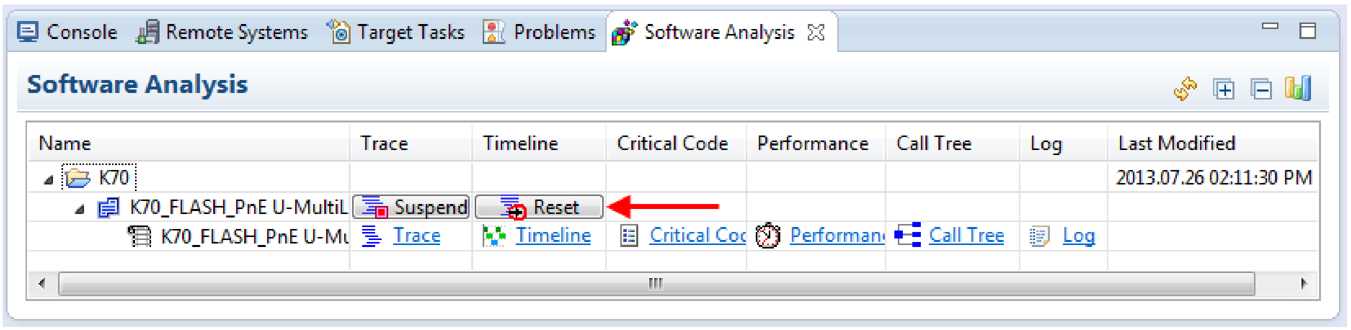
The **Reset** button lets you to reset trace during the debug session. The trace data can be reset only when the debug session is suspended. The **Reset** button is enabled only when debug session is suspended and there is trace available to be reset. The user interface provides three options that you can use to perform the rest action:

- The **Reset** button in the **Software Analysis** view
- The **Reset** button in the **Debug** view
- The **Reset** option in the context menu on right-clicking in the **Debug** view

When the reset action is performed, the trace data collected so far gets deleted and trace collection starts from the point where trace collection stopped. This feature is useful if you are collecting continuous trace and want to view trace data after a particular point of time. So you can perform the reset action which will erase the trace data collected so far and will start collecting trace from that point onwards.

To reset trace:

1. Collect trace data in Continuous mode.
2. View trace data.
3. Click **Reset** in the **Software Analysis** view.



Name	Trace	Timeline	Critical Code	Performance	Call Tree	Log	Last Modified
K70							2013.07.26 02:11:30 PM
K70_FLASH_PnE U-MultiL	Suspend	Reset					
K70_FLASH_PnE U-Mu	Trace	Timeline	Critical Coc	Performan	Call Tree	Log	

Figure 3-33. Reset Button in Software Analysis View

4. Click **Resume** in the **Debug** view.
5. Click **Suspend** in the **Debug** view.
6. View trace data.

   The old trace data will be removed and trace data collected from where it stopped is displayed.

NOTE

The **Reset** button is not available for Cfv2-v4 targets as trace on these targets is collected via Tracelink, which cannot collect trace continuously.

Chapter 4
Viewing Data

You can view the collected data for various targets such as HCS08, ColdFire V1, Kinetis, MPC56xx and DSC from the data file generated after running the application in the following viewers.

- Trace Data
- Timeline
- Critical Code Data
- Performance Data
- Call Tree

You can also view trace data of the other project by importing the trace data file of another project into your project using the **Import** wizard. For more information, refer Importing Trace Data Offline.

NOTE

This chapter demonstrates how to view collected data on the Kinetis target. The process of viewing data for all the targets is same.

4.1 Trace Data

To view trace data:

1. In the **Software Analysis** view, expand the project name.

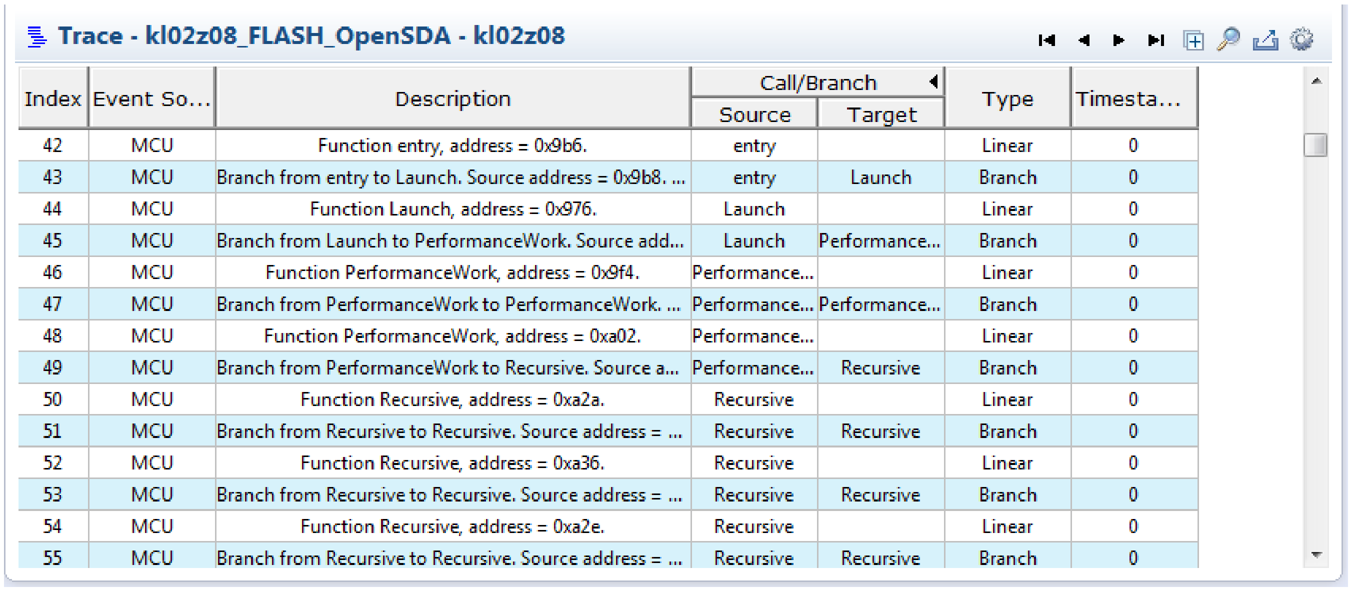
   The data source is listed under the project name.

NOTE

If the **Software Analysis** view is not open already, select **Window > Show View > Software Analysis** from the menu bar.

2. Click the **Trace** hyperlink.

   The **Trace Data** viewer appears.



Trace - kl02z08_FLASH_OpenSDA - kl02z08

Index	Event So...	Description	Call/Branch		Type	Timesta...
			Source	Target		
42	MCU	Function entry, address = 0x9b6.	entry		Linear	0
43	MCU	Branch from entry to Launch. Source address = 0x9b8. ...	entry	Launch	Branch	0
44	MCU	Function Launch, address = 0x976.	Launch		Linear	0
45	MCU	Branch from Launch to PerformanceWork. Source add...	Launch	Performance...	Branch	0
46	MCU	Function PerformanceWork, address = 0x9f4.	Performance...		Linear	0
47	MCU	Branch from PerformanceWork to PerformanceWork. ...	Performance...	Performance...	Branch	0
48	MCU	Function PerformanceWork, address = 0xa02.	Performance...		Linear	0
49	MCU	Branch from PerformanceWork to Recursive. Source a...	Performance...	Recursive	Branch	0
50	MCU	Function Recursive, address = 0xa2a.	Recursive		Linear	0
51	MCU	Branch from Recursive to Recursive. Source address = ...	Recursive	Recursive	Branch	0
52	MCU	Function Recursive, address = 0xa36.	Recursive		Linear	0
53	MCU	Branch from Recursive to Recursive. Source address = ...	Recursive	Recursive	Branch	0
54	MCU	Function Recursive, address = 0xa2e.	Recursive		Linear	0
55	MCU	Branch from Recursive to Recursive. Source address = ...	Recursive	Recursive	Branch	0

Figure 4-1. Trace Data Viewer

The **Trace Data** viewer displays the trace data collected by the Kinetis target in a tabular form. You can move the columns to the left or right of another column by dragging and dropping.

NOTE

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

The table below describes the fields of the **Trace Data** viewer.

Table 4-1. Trace Data - Description of Fields

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 continues on the next page...

Table 4-1. Trace Data - Description of Fields (continued)

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.

You can perform the following actions from the **Trace Data** viewer:

- Click the navigation buttons available to move between the trace data.
  - Click the **Go to Start of Trace** button to go to the beginning of trace data
  - Click the **Go to Previous Trace Entry** button to go to the previous trace entry
  - Click the **Go to Next Trace Entry** button to go to the next trace entry
  - Click the **Go to End of Trace** button to go to the end of trace data
- Click the **Expand All** button to display the source as well as assembly code of the instructions in the trace viewer.
- Click the **Collapse All** button to display the assembly code only.
- Click the **Search** button to find and filter trace data.
- Click the **Export to CSV** button to export trace data to a CSV file.
- Click the **Configure Table** button to configure time unit and time format.

This topic contains the following sub-topics:

- Exporting Trace Data
- Configuring Time Unit and Time Format
- Customizing Trace Data Viewer

4.1.1 Exporting Trace Data

To export trace data to a CSV file:

1. Click the **Export to CSV** button in the **Trace Data** viewer to display the **Exporting Trace Data** dialog box.

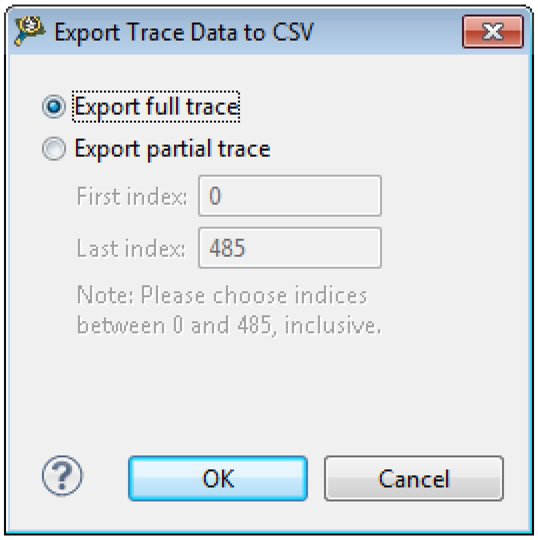


Figure 4-2. Exporting Trace Data Dialog Box

2. Select the **Export Full Trace** option to export complete trace data to a CSV file else select **Export Partial Trace**.
3. Specify the begin and end index of the trace data if you have selected the **Export Partial Trace** option.
4. Click **OK**.

   The **Export Trace Data to CSV** dialog box appears.

5. Browse to the location where you want to save the trace data.

This is how you export trace data to a CSV file.

4.1.2 Configuring Time Unit and Time Format

The **Configure Table** button in the **Trace Data** viewer allows you to perform two actions:

- **Configure Time Unit** - Lets you set CPU frequency and convert the clock cycles, displayed in the **Timestamp** column, into real time in milliseconds, microseconds, or nanoseconds. Click this button and select the **Configure Time Unit** option. To convert the clock cycles into milliseconds, microseconds, or nanoseconds, select the corresponding option.

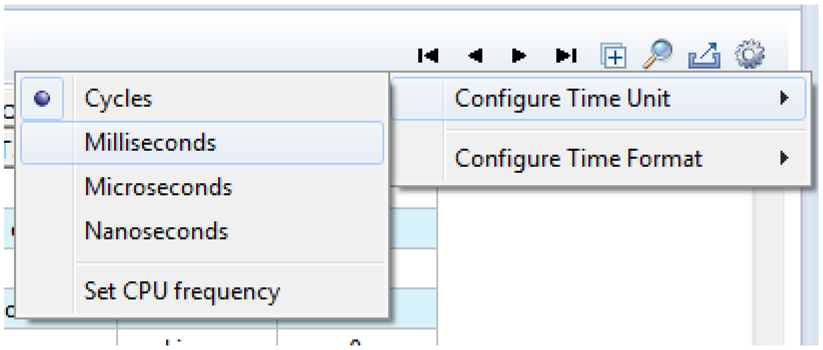


Figure 4-3. Configuring Time Unit

The **Set CPU Frequency** option allows you to set the CPU frequency needed to convert the clock cycles into real time. Select this option to display the **Set CPU Frequency** dialog box. Set the new CPU frequency according to requirements.

NOTE

The **Configure Time Unit** option is available in all other viewers also, that is **Critical Code Data** , **Timeline** , **Performance** , and **Call Tree** . All the viewers share the same time and frequency that you set for a particular viewer. For example, if you set time in microseconds for **Trace Data** viewer, all other viewers will display time in microseconds.

- **Configure Time Format** - Enables the **Timestamp** column of the **Trace Data** viewer for time formatting, **Radix** : *Hexadecimal* or *Decimal* and **Value** : *Absolute* or *Delta*. Time radix formatting is available only if time unit is set to **Time in cycles** . The default format of time is displayed in *Decimal* and *Absolute* value. *Delta* is calculated for same source event. The *Delta* value for a specific timestamp is the difference between current timestamp and previous timestamp with same source event as current one. If *Delta* value is negative, it will be displayed as zero.

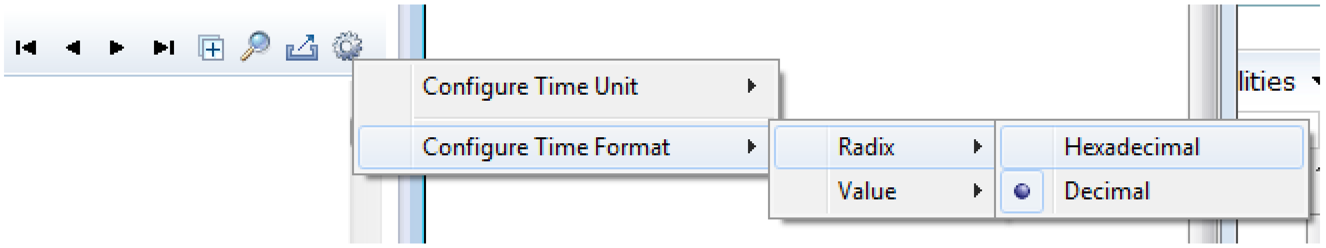


Figure 4-4. Configuring Time Format

NOTE

For HCS08 and ColdFire targets, the current value is a difference between current timestamp and the value from the previous row trace. For Kinetis (ITM and ETM source events), the difference is made with timestamp from the previous row only if it has the same source event as current, otherwise the difference is made with the previous timestamp that has the same source event as current.

4.1.3 Customizing Trace Data Viewer

You can modify the appearance or display of the trace results on the **Trace Data** viewer. Right-click 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 **Trace Data** viewer, 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.

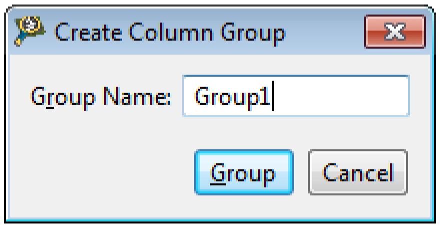
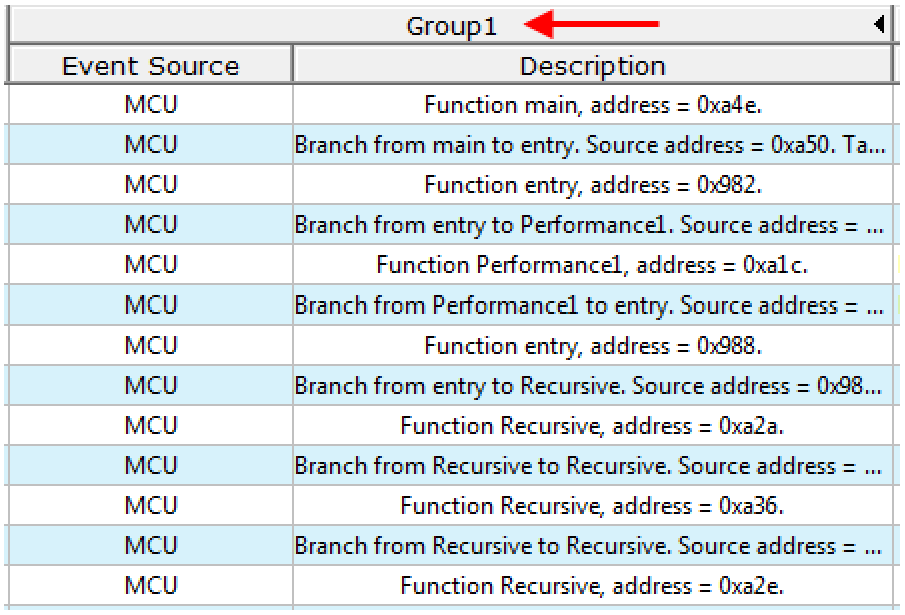


Figure 4-5. Create Column Group Dialog Box

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



Group1	
Event Source	Description
MCU	Function main, address = 0xa4e.
MCU	Branch from main to entry. Source address = 0xa50. Ta...
MCU	Function entry, address = 0x982.
MCU	Branch from entry to Performance1. Source address = ...
MCU	Function Performance1, address = 0xa1c.
MCU	Branch from Performance1 to entry. Source address = ...
MCU	Function entry, address = 0x988.
MCU	Branch from entry to Recursive. Source address = 0x98...
MCU	Function Recursive, address = 0xa2a.
MCU	Branch from Recursive to Recursive. Source address = ...
MCU	Function Recursive, address = 0xa36.
MCU	Branch from Recursive to Recursive. Source address = ...
MCU	Function Recursive, address = 0xa2e.

Figure 4-6. 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 **Trace Data** viewer. Right-click any column and select the **Choose columns** option. The **Column Chooser** dialog box appears.

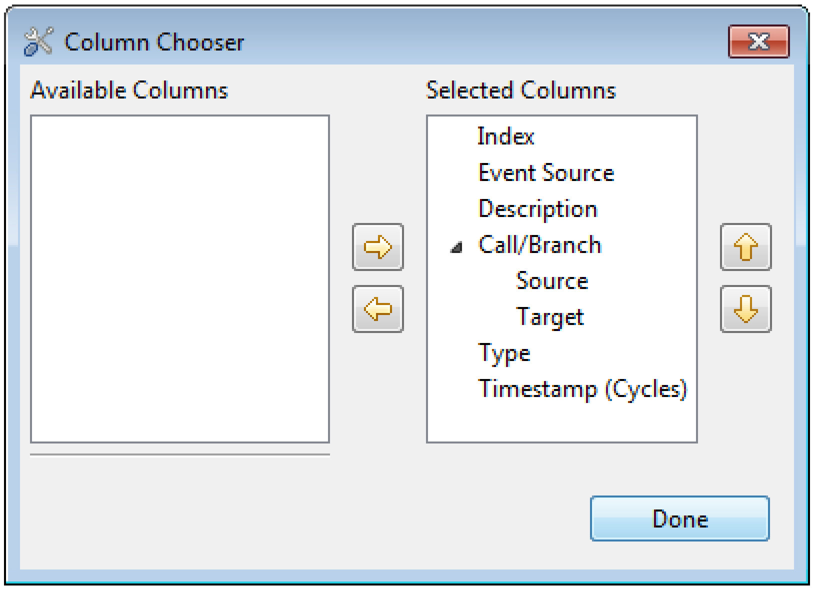
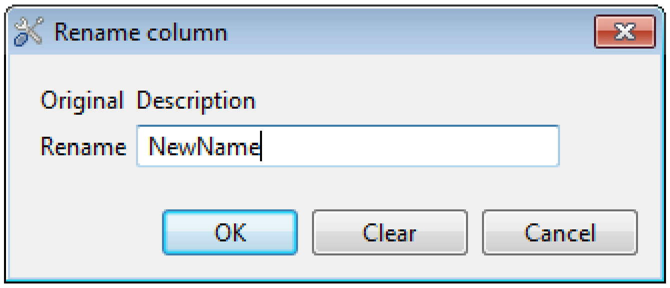


Figure 4-7. Column Chooser Dialog Box

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 **Trace Data** viewer. Click **Done** to save the settings.

- 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**.



Original Description

Rename NewName

Figure 4-8. Rename Column Dialog Box

4.2 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. In the **Software Analysis** view, expand the project name.

   The data source is listed under the project name.

2. Click the **Timeline** hyperlink.

   The **Timeline** viewer appears.

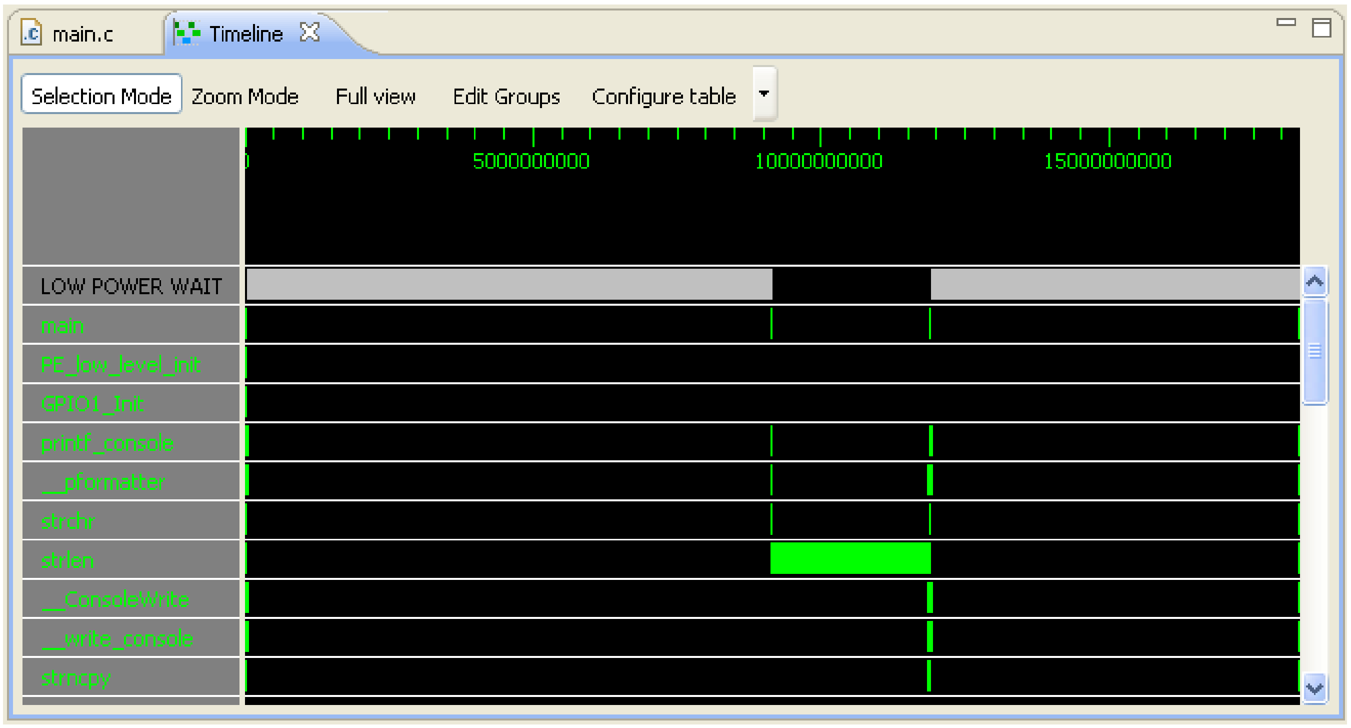


Figure 4-9. Timeline Viewer Displaying Timeline Data

The **Timeline** 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 of the application.

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

- Selection Mode
- Zoom Mode
- Full View
- Edit Groups
- Configure Table

4.2.1 Selection Mode

The **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:

1. Click **Selection Mode**.
2. Click on the bar where you want to mark the point.

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

3. Right-click 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.

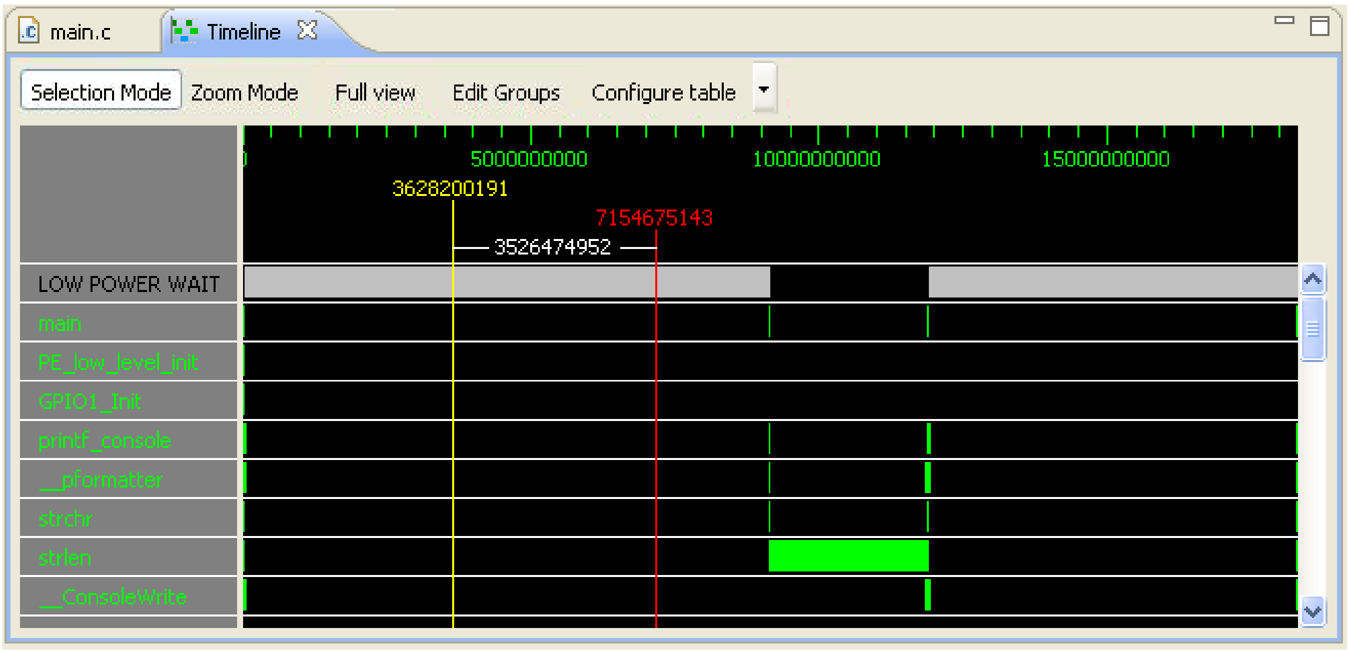


Figure 4-10. Selection Mode to Measure Difference of Cycles Between Functions

You might view a difference in the time cycles displayed in the **Timeline** and the **Critical Code** viewer. The difference is caused by the events in the functions (in your source code) that have no new timestamp. For timeline, any instruction that has no timestamp information is considered to take one CPU cycle.

4.2.2 Zoom Mode

The **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 in the timeline graph. You can also move the mouse wheel up and down to zoom-in and zoom-out.

4.2.3 Full View

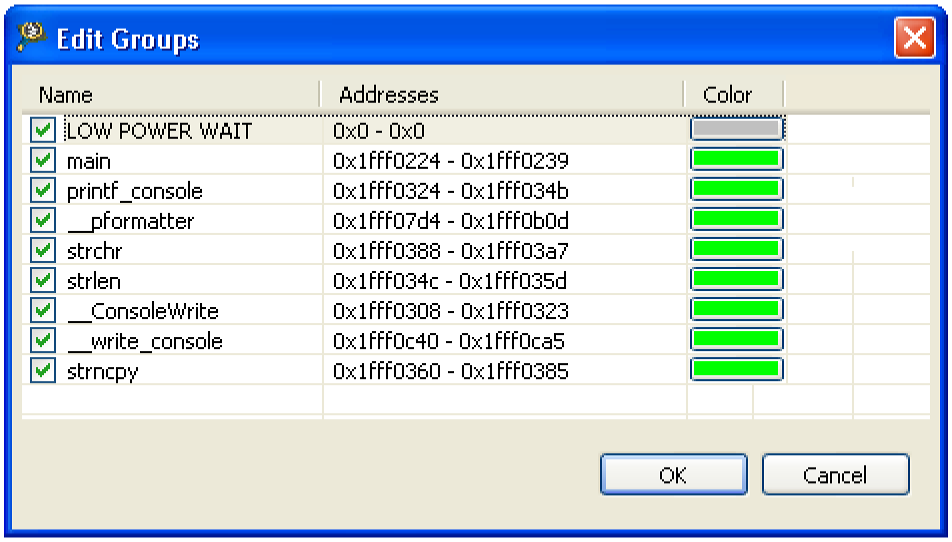
The **Full View** allows you to get back to the original view if you selected the zoom mode.

NOTE

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

4.2.4 Edit Groups

The **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, select **Edit Groups**. The **Edit Groups** dialog box appears.



	Name	Addresses	Color
✓	LOW POWER WAIT	0x0 - 0x0	⬜
✓	main	0x1fff0224 - 0x1fff0239	🟩
✓	printf_console	0x1fff0324 - 0x1fff034b	🟩
✓	__pformatter	0x1fff07d4 - 0x1fff0b0d	🟩
✓	strchr	0x1fff0388 - 0x1fff03a7	🟩
✓	strlen	0x1fff034c - 0x1fff035d	🟩
✓	__ConsoleWrite	0x1fff0308 - 0x1fff0323	🟩
✓	__write_console	0x1fff0c40 - 0x1fff0ca5	🟩
✓	strncpy	0x1fff0360 - 0x1fff0385	🟩

Figure 4-11. 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

4.2.4.1 Add/Remove Function

Right-click the function name in the **Name** column, and select **Insert Function** or press *Ctrl+F* to add a function. Select **Delete Selected** from the context menu to delete the function from the graph. You can disable a function from the graph by clearing the corresponding checkbox in the **Name** column. Check it again to include it in the graph.

4.2.4.2 Edit Address Range of Function

1. Select the function of which you want to change the address range.
2. Double-click 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.

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

4.2.4.3 Change Color

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

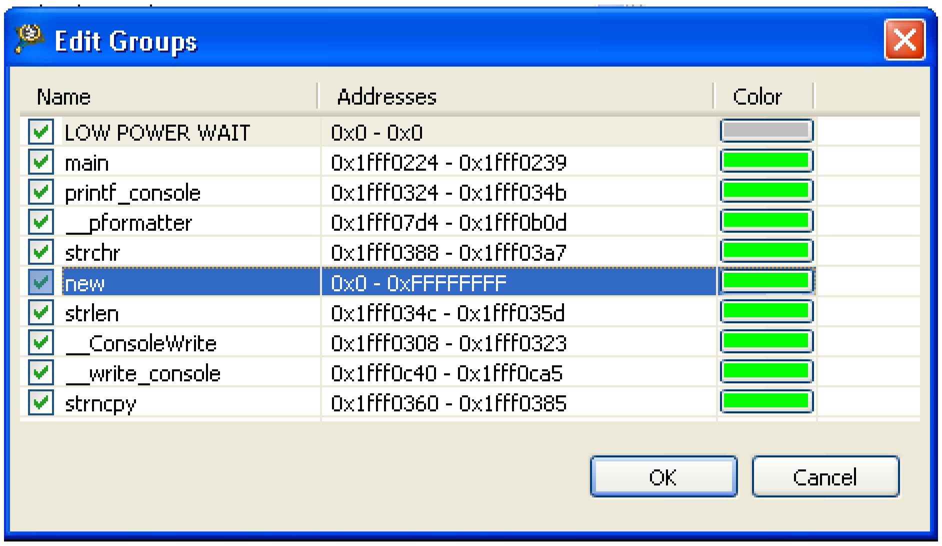
4.2.4.4 Add/Remove Group

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 the row, in the **Edit Groups** dialog box, where you want to insert a group, and select **Insert Group** from the context menu. Alternatively, press the *Ctrl +G* key.

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



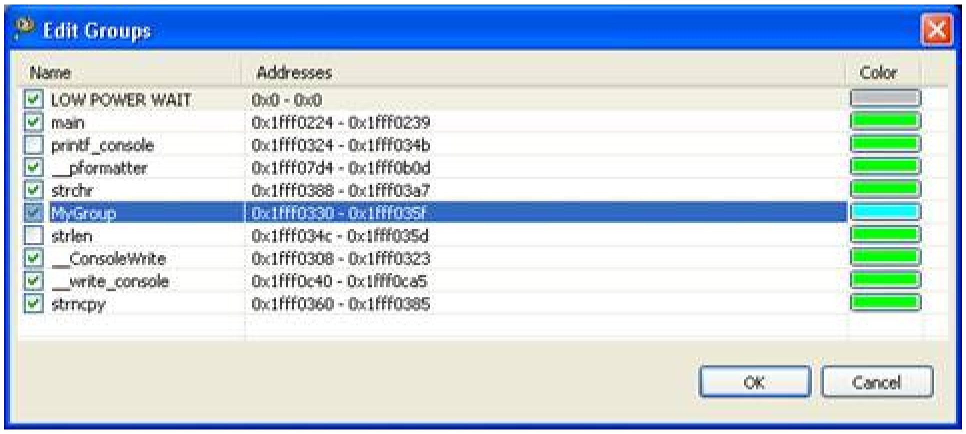
Name	Addresses	Color
✓ LOW POWER WAIT	0x0 - 0x0	⬜
✓ main	0x1fff0224 - 0x1fff0239	🟩
✓ printf_console	0x1fff0324 - 0x1fff034b	🟩
✓ __pformatter	0x1fff07d4 - 0x1fff0b0d	🟩
✓ strchr	0x1fff0388 - 0x1fff03a7	🟩
✓ new	0x0 - 0xFFFFFFFF	🟩
✓ strlen	0x1fff034c - 0x1fff035d	🟩
✓ __ConsoleWrite	0x1fff0308 - 0x1fff0323	🟩
✓ __write_console	0x1fff0c40 - 0x1fff0ca5	🟩
✓ strncpy	0x1fff0360 - 0x1fff0385	🟩

Figure 4-12. Adding Group

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

   The cell becomes editable.

3. Type a name for the group, for example, *MyGroup*.
4. Double-click the cell of the corresponding **Addresses** column, and edit the address range according to requirements, for example, `0x1fff0330 - 0x1fff035f`.
5. Change the color of the group.



Name	Addresses	Color
✓ LOW POWER WAIT	0x0 - 0x0	⬜
✓ main	0x1fff0224 - 0x1fff0239	🟩
☐ printf_console	0x1fff0324 - 0x1fff034b	🟩
✓ __pformatter	0x1fff07d4 - 0x1fff0b0d	🟩
✓ strchr	0x1fff0388 - 0x1fff03a7	🟩
✓ MyGroup	0x1fff0330 - 0x1fff035f	🟦
☐ strlen	0x1fff034c - 0x1fff035d	🟩
✓ __ConsoleWrite	0x1fff0308 - 0x1fff0323	🟩
✓ __write_console	0x1fff0c40 - 0x1fff0ca5	🟩
✓ strncpy	0x1fff0360 - 0x1fff0385	🟩

Figure 4-13. Edit Groups Dialog Box After Editing Address Range and Color of Group

6. Click **OK**.

   The *MyGroup* group is added to the timeline. The *MyGroup* group covers a bit of the `printf_console` group and full address range of the `strlen` group. The uncovered address range will become part of the <*other*> group, as shown in the figure below.

NOTE

In case the address range you entered overlaps with the address range of any existing function, you get the

Overlapped Groups message after clicking **OK**. You can disable the functions that are overlapped with the address range of the added group.The `printf_console` and `strlen` functions have been disabled, as shown in the figure above, to prevent overlapping. Disabling a function does not delete the function.

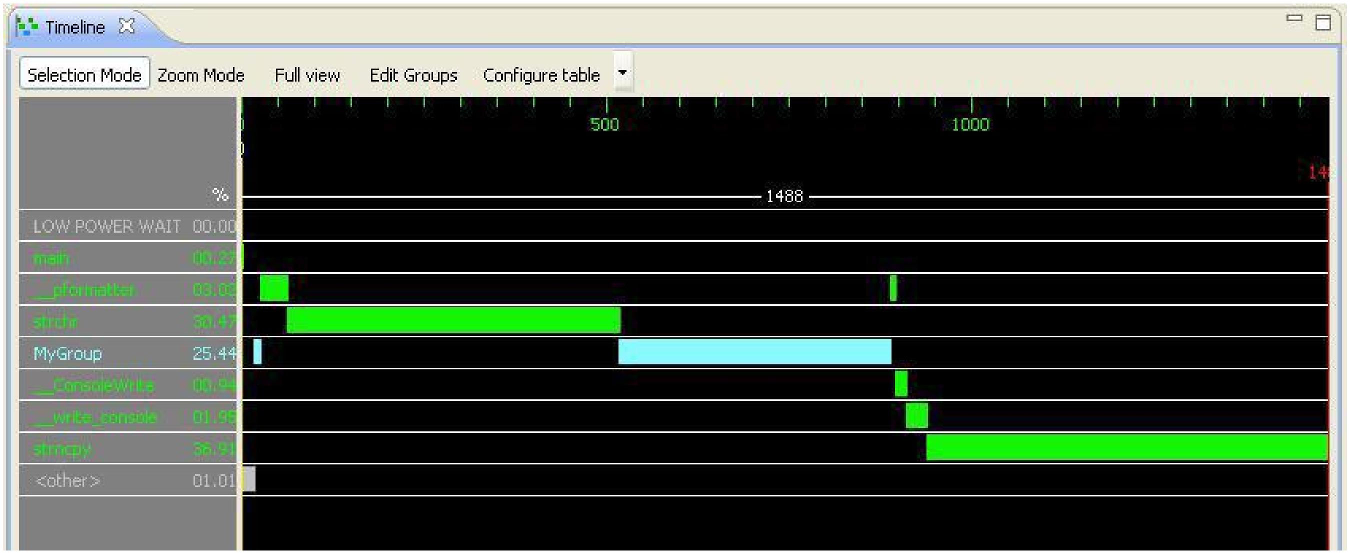


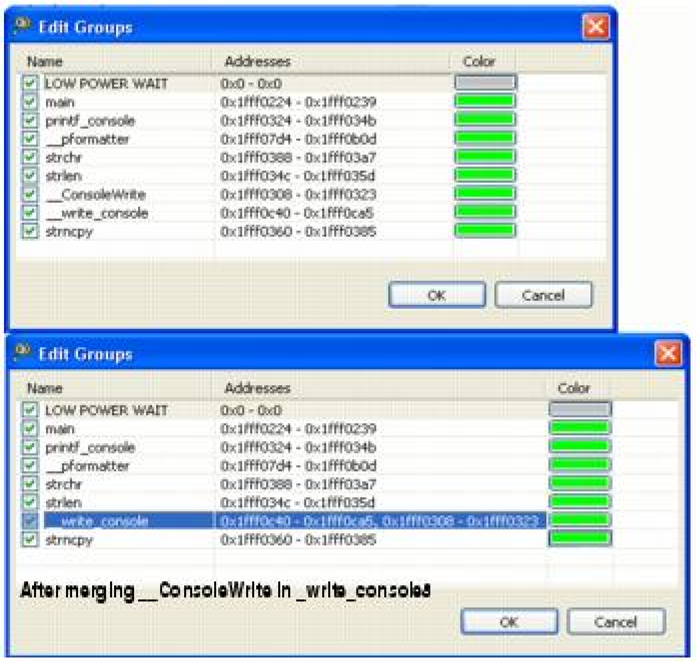
Figure 4-14. Timeline After Adding Group

To delete a group, select it, right-click 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.

4.2.4.5 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 below, where the `__ConsoleWrite` function is merged into another function `_write_console`.


Edit Groups

Name	Addresses	Color
✓ LOW POWER WAIT	0x0 - 0x0	⬜
✓ main	0x1fff0224 - 0x1fff0239	🟩
✓ printf_console	0x1fff0324 - 0x1fff034b	🟩
✓ __pformatter	0x1fff07d4 - 0x1fff0b0d	🟩
✓ strchr	0x1fff0388 - 0x1fff03a7	🟩
✓ strlen	0x1fff034c - 0x1fff035d	🟩
✓ __ConsoleWrite	0x1fff0308 - 0x1fff0323	🟩
✓ __write_console	0x1fff0c40 - 0x1fff0ca5	🟩
✓ strncpy	0x1fff0360 - 0x1fff0385	🟩

Edit Groups

Name	Addresses	Color
✓ LOW POWER WAIT	0x0 - 0x0	⬜
✓ main	0x1fff0224 - 0x1fff0239	🟩
✓ printf_console	0x1fff0324 - 0x1fff034b	🟩
✓ __pformatter	0x1fff07d4 - 0x1fff0b0d	🟩
✓ strchr	0x1fff0388 - 0x1fff03a7	🟩
✓ strlen	0x1fff034c - 0x1fff035d	🟩
✓ __write_console	0x1fff0c40 - 0x1fff0ca5, 0x1fff0308 - 0x1fff0323	🟩
✓ strncpy	0x1fff0360 - 0x1fff0385	🟩

After merging __ConsoleWrite in _write_consoles

Figure 4-15. Merging Functions/Groups

3. Click **OK**.

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.

4.2.5 Configure Table

The **Configure Table** button lets you configure time unit in cycles, milliseconds, microseconds, and nanoseconds. For example, to display time displayed in the timeline graph in milliseconds, select **Configure Table > Configure Time Unit > Milliseconds** . It also allows you to set CPU frequency. For more information, refer Configuring Time Unit and Time Format.

4.3 Critical Code Data

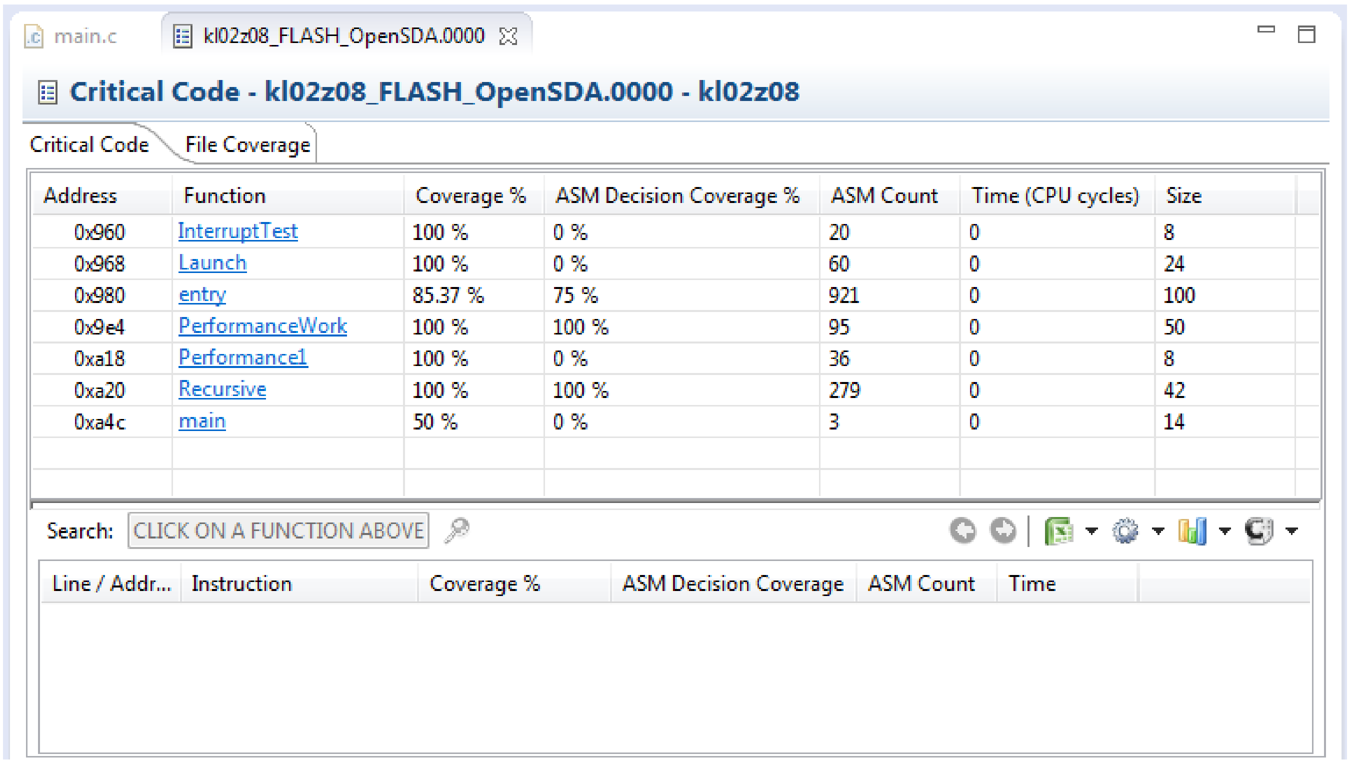
To view critical code data:

1. In the **Software Analysis** view, expand the project name.

   The data source is listed under the project name.

2. Click the **Critical Code** hyperlink.

   The **Critical Code** viewer appears.


main.c
kl02z08_FLASH_OpenSDA.0000
Critical Code - kl02z08_FLASH_OpenSDA.0000 - kl02z08
Critical Code
File Coverage
Search:
CLICK ON A FUNCTION ABOVE

Address	Function	Coverage %	ASM Decision Coverage %	ASM Count	Time (CPU cycles)	Size
0x960	InterruptTest	100 %	0 %	20	0	8
0x968	Launch	100 %	0 %	60	0	24
0x980	entry	85.37 %	75 %	921	0	100
0x9e4	PerformanceWork	100 %	100 %	95	0	50
0xa18	Performance1	100 %	0 %	36	0	8
0xa20	Recursive	100 %	100 %	279	0	42
0xa4c	main	50 %	0 %	3	0	14

Line / Addr...	Instruction	Coverage %	ASM Decision Coverage	ASM Count	Time

Figure 4-16. Critical Code Viewer

The critical code data displays the summarized data of a function in a tabular form. The table below describes the fields of the critical code data. The columns are movable; you can drag and drop the columns to move them according to your requirements.

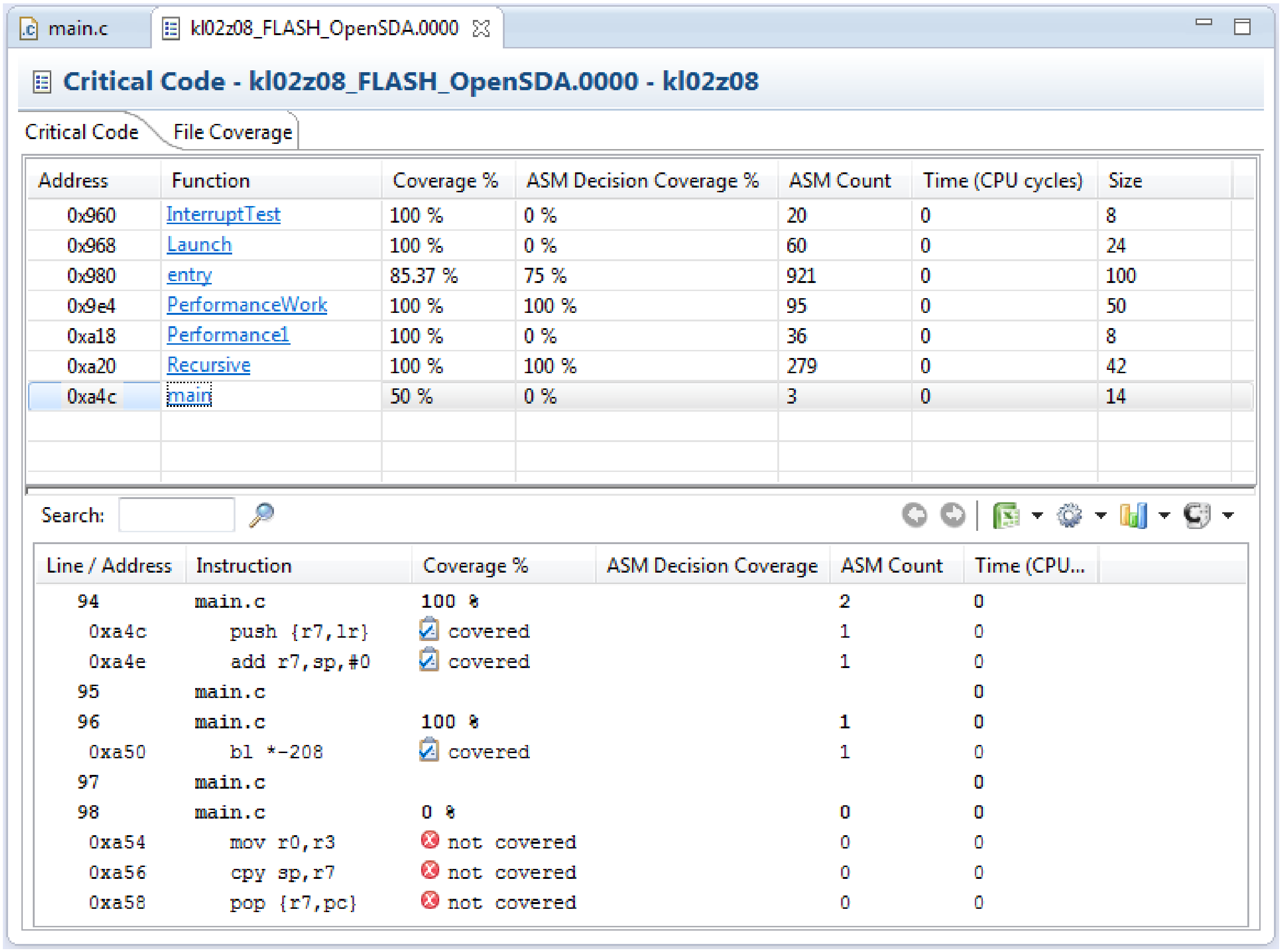
Table 4-2. Critical Code Data - Description of Fields

Name	Description
Address	Displays the start address of the function.
Function	Displays the name of the function that has executed.
Coverage %	Displays the percentage of number of assembly instructions executed from the total number of assembly instructions in a function.
ASM Decision Coverage %	Displays the percentage of decision coverage computed for direct and indirect conditional branches.
ASM Count	Displays the number of lines executed in the function.
Time (CPU Cycles)	Displays the total number of clock cycles that the function takes.
Size	Displays the number of bytes required by each function.

The **Critical Code** viewer divides the critical code data in two tabs: **Critical Code** and **File Coverage**.

4.3.1 Critical Code Tab

The **Critical Code** tab displays data into 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 a hyperlinked function in the top view of the **Critical Code** viewer to view the corresponding statistics for the instructions executed in that function. For example, the statistics of the `main()` function are shown in the figure below.



main.c | kl02z08_FLASH_OpenSDA.0000

Critical Code - kl02z08_FLASH_OpenSDA.0000 - kl02z08

Critical Code | File Coverage

Address	Function	Coverage %	ASM Decision Coverage %	ASM Count	Time (CPU cycles)	Size
0x960	InterruptTest	100 %	0 %	20	0	8
0x968	Launch	100 %	0 %	60	0	24
0x980	entry	85.37 %	75 %	921	0	100
0x9e4	PerformanceWork	100 %	100 %	95	0	50
0xa18	Performance1	100 %	0 %	36	0	8
0xa20	Recursive	100 %	100 %	279	0	42
0xa4c	main	50 %	0 %	3	0	14

Search:

Line / Address	Instruction	Coverage %	ASM Decision Coverage	ASM Count	Time (CPU...
94	main.c	100 %		2	0
0xa4c	push {r7,lr}	✓ covered		1	0
0xa4e	add r7,sp,#0	✓ covered		1	0
95	main.c				0
96	main.c	100 %		1	0
0xa50	bl *-208	✓ covered		1	0
97	main.c				0
98	main.c	0 %		0	0
0xa54	mov r0,r3	✗ not covered		0	0
0xa56	cpy sp,r7	✗ not covered		0	0
0xa58	pop {r7,pc}	✗ not covered		0	0

Figure 4-17. Statistics of Critical Code Data

The table below describes the fields of the statistics of the critical code data.

Table 4-3. Description of Statistics of Critical Code Data

Name	Description
Line/Address	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.
Coverage	For C source lines, displays the percentage of number of assembly instructions executed from the total number of assembly instructions corresponding to the source line. For assembly source lines, it shows if the instructions were executed or not.
ASM Decision Coverage	Displays the decision coverage computed for direct and indirect conditional branches. It is the mean value of the individual decision coverages. So if a function has two conditional instructions, one with 100% and another with 50% decision coverage, the decision coverage would be (100 + 50) / 2 = 75% . It is calculated only for assembly instructions and not for C source code.
ASM Count	Displays the number of times each instruction is executed.
Time (CPU Cycles)	Displays the total number of clock cycles that each instruction in the function takes.

Click the column header to sort the critical code data by that column. However, you can only sort the critical code data available on the top view. The table below lists the buttons available in the statistics view of the **Critical Code** tab.

Table 4-4. Buttons Available in Statistics View of Critical Code Tab

Name	Button	Description
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 **Search** button. The first instance of the data is selected in the statistics view. Click the button 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 it 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 it to view the details of the previous function.
Export	📊▾	Lets you export the critical code data of both top and bottom views in a CSV file. Click the button 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.

Table continues on the next page...

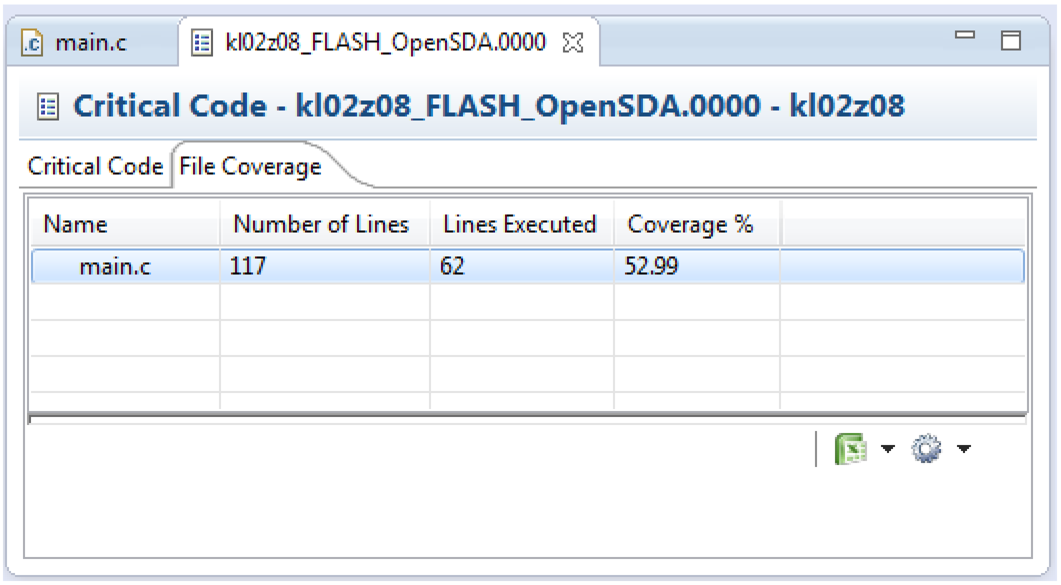
Table 4-4. Buttons Available in Statistics View of Critical Code Tab (continued)

Name	Button	Description
Configure table	⚙ ▾	Lets you show and hide column(s) of the critical code data. Click the button 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 **Critical Code** viewer. The option also allows you to set CPU frequency and set time in cycles, milliseconds, microseconds, and nanoseconds.
Graphics	📊 ▾	Lets you display the histograms in two colors for the **CPU Cycles** and **ASM Count** columns in the bottom view of the critical code data. Click the button and select the **Assembly/Source > ASM Count or Assembly/Source > CPU Cycles** option to display histograms in the **ASM Count** or **CPU Cycles** 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 critical code data. Click the button and select the **Assembly** option to display the assembly code or the **Mixed** option to display the mixed code in the statistical details.

4.3.2 File Coverage Tab

The **File Coverage** tab displays information about the lines executed from each file or the functions executed form each library in the application.

Click the **File Coverage** tab to display the file coverage data.


main.c
kl02z08_FLASH_OpenSDA.0000
Critical Code - kl02z08_FLASH_OpenSDA.0000 - kl02z08
Critical Code
File Coverage

Name	Number of Lines	Lines Executed	Coverage %
main.c	117	62	52.99

Figure 4-18. File Coverage Tab

The table below describes the fields of the file coverage data.

Table 4-5. Description of File Coverage Data

Name	Description
Name	Displays the name of the file.
Number of Lines	Displays the size of the file in number of lines present in the file.
Lines Executed	Displays the number of lines executed in the file.
Coverage %	Displays the percentage of number of lines executed compared to the total number of lines, that is *(lines executed / number of lines) * 100*.

You can perform the Export and Configure table actions on the file coverage data.

4.4 Performance Data

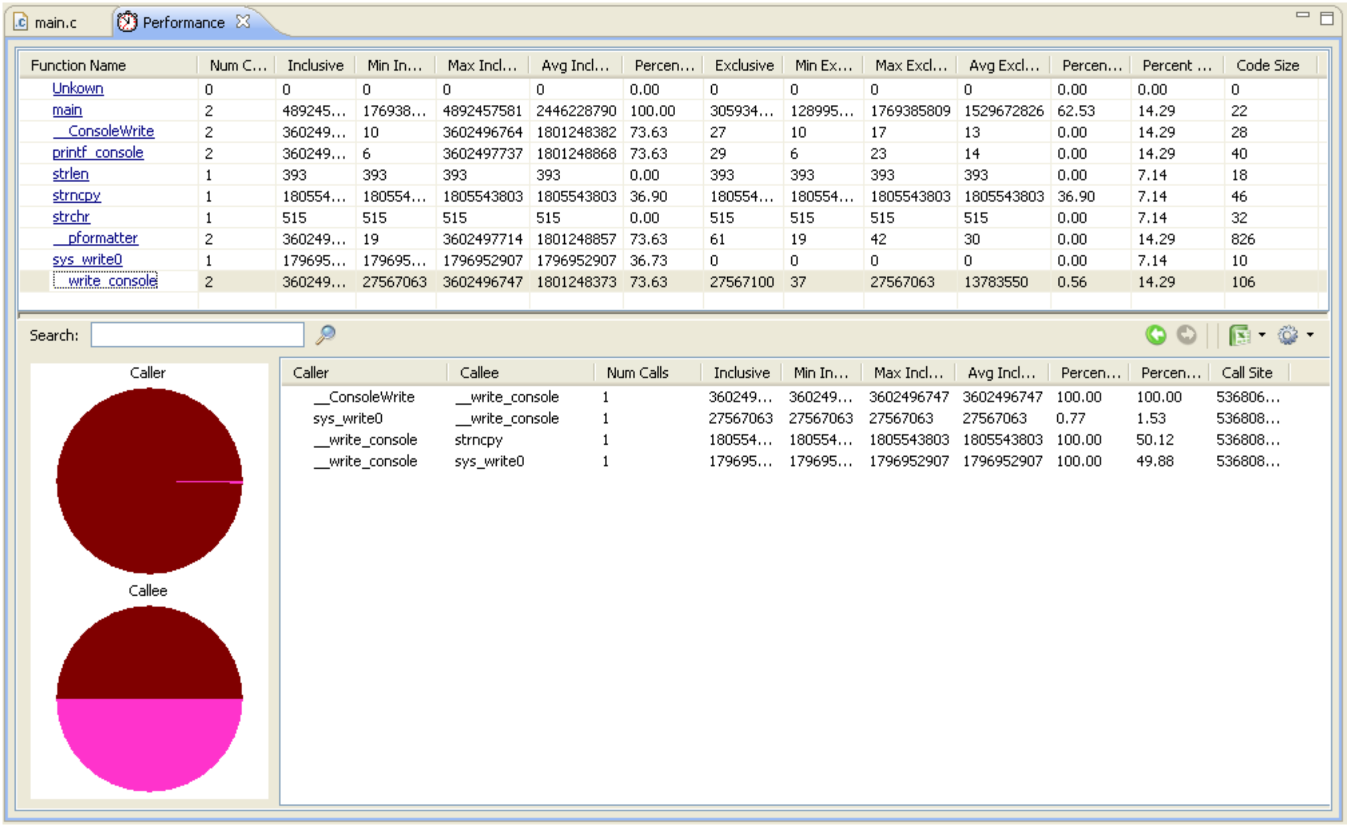
To view performance data:

1. In the **Software Analysis** view, expand the project name.

   The data source is listed under the project name.

2. Click the **Performance** hyperlink.

   The **Performance** viewer appears.



Function Name	Num C...	Inclusive	Min In...	Max Incl...	Avg Incl...	Percen...	Exclusive	Min Ex...	Max Excl...	Avg Excl...	Percen...	Percent ...	Code Size
Unkown	0	0	0	0	0	0.00	0	0	0	0	0.00	0.00	0
main	2	489245...	176938...	4892457581	2446228790	100.00	305934...	128995...	1769385809	1529672826	62.53	14.29	22
__ConsoleWrite	2	360249...	10	3602496764	1801248382	73.63	27	10	17	13	0.00	14.29	28
printf_console	2	360249...	6	3602497737	1801248868	73.63	29	6	23	14	0.00	14.29	40
strlen	1	393	393	393	393	0.00	393	393	393	393	0.00	7.14	18
strncpy	1	180554...	180554...	1805543803	1805543803	36.90	180554...	180554...	1805543803	1805543803	36.90	7.14	46
strchr	1	515	515	515	515	0.00	515	515	515	515	0.00	7.14	32
__pformatter	2	360249...	19	3602497714	1801248857	73.63	61	19	42	30	0.00	14.29	826
sys_write0	1	179695...	179695...	1796952907	1796952907	36.73	0	0	0	0	0.00	7.14	10
__write_console	2	360249...	27567063	3602496747	1801248373	73.63	27567100	37	27567063	13783550	0.56	14.29	106

Caller	Callee	Num Calls	Inclusive	Min In...	Max Incl...	Avg Incl...	Percen...	Percen...	Call Site
__ConsoleWrite	__write_console	1	360249...	360249...	3602496747	3602496747	100.00	100.00	536806...
sys_write0	__write_console	1	27567063	27567063	27567063	27567063	0.77	1.53	536808...
__write_console	strncpy	1	180554...	180554...	1805543803	1805543803	100.00	50.12	536808...
__write_console	sys_write0	1	179695...	179695...	1796952907	1796952907	100.00	49.88	536808...

Figure 4-19. Performance Viewer

The **Performance** viewer is divided into two views:

- The top view presents function performance data in the *Function Performance* table. It displays the count and invocation information for each function that executes during the measurement, enabling you to compare the relative data for various portions of your target program. The information in the Function Performance table can be sorted by column in ascending or descending order. Click the column header to sort the corresponding data. The table below describes the fields of the *Function Performance* table.
- The bottom view or the *Call Site Performance* table presents call pair data for the function selected in the Function Performance table. It displays call pair relationships for the selected function, that is which function called which function. Each function pair consists of a caller and a callee. The percent caller and percent callee data is also displayed graphically. The functions are represented in different colors in the pie chart, you can move the mouse cursor over the color to see the corresponding

function. The next following table describes the fields of the *Call Site Performance* table. You cannot sort the columns of this table.

Table 4-6. Field Description of Function Performance Table

Name	Description
Function Name	Name of the function that has executed.
Num Calls	Number of times the function has executed.
Inclusive	Cumulative metric count during execution time spent from function entry to exit.
Min Inclusive	Minimum metric count during execution time spent from function entry to exit.
Max Inclusive	Maximum metric count during execution time spent from function entry to exit.
Avg Inclusive	Average metric count during execution time spent from function entry to exit.
Percent Inclusive	Percentage of total metric count spent from function entry to exit.
Exclusive	Cumulative metric count during execution time spent within function.
Min Exclusive	Minimum metric count during execution time spent within function.
Max Exclusive	Maximum metric count during execution time spent within function.
Avg Exclusive	Average metric count during execution time spent within function.
Percent Exclusive	Percentage of total metric count spent within function.
Percent Total Calls	Percentage of the calls to the function compared to the total calls.
Code Size	Number of bytes required by each function.

Table 4-7. Field Description of Call Site Performance Table

Name	Description
Caller	Name of the calling function.
Callee	Name of the function that is called by the calling function.
Num Calls	Number of times the caller called the callee.
Inclusive	Cumulative metric count during execution time spent from function entry to exit.
Min Inclusive	Minimum metric count during execution time spent from function entry to exit.
Max Inclusive	Maximum metric count during execution time spent from function entry to exit.

Table continues on the next page...

Table 4-7. Field Description of Call Site Performance Table (continued)

Name	Description
Avg Inclusive	Average metric count during execution time spent from function entry to exit.
Percent Callee	Percent of total metric count during the time the selected function is the caller of a specific callee. The data is also shown in the Caller pie chart.
Percent Caller	Percent of total metric count during the time the selected function is the callee of a specific caller. The data is also shown in the Callee pie chart.
Call Site	Address from where the function was called.

You can move the columns to the left or right of another column depending on your requirements by dragging and dropping. You can perform the Export and Configure table actions on the performance data similar to critical code data. You can also view the previous and next functions of the performance data using the icons available in the lower section of the **Performance** viewer.

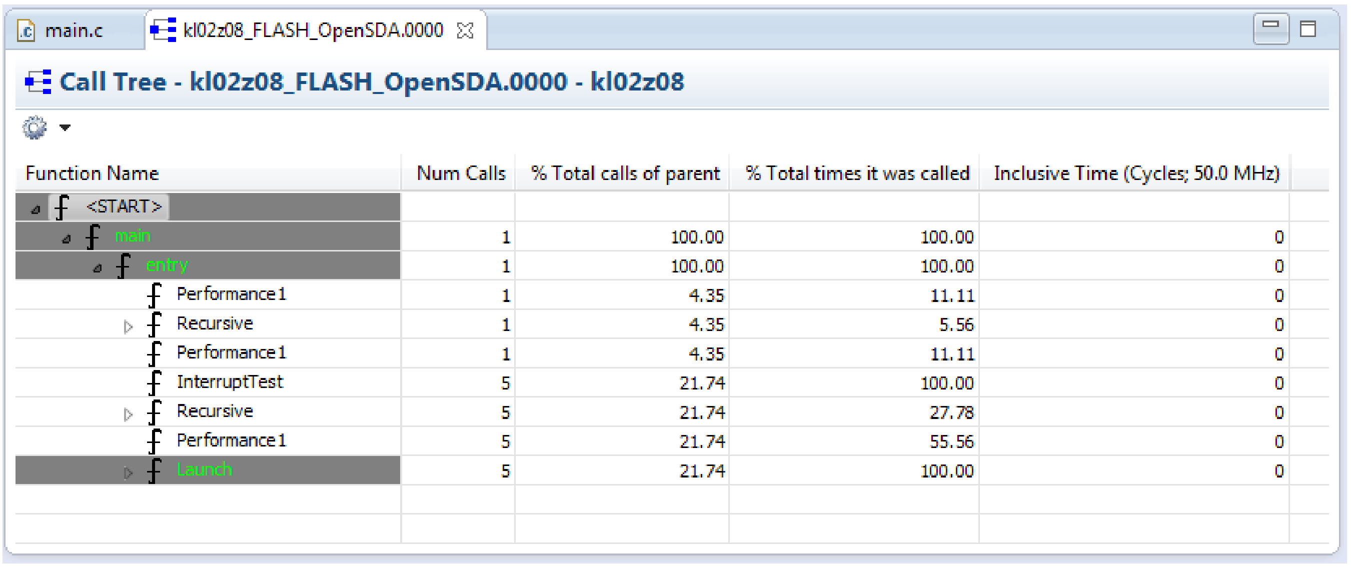
4.5 Call Tree

To view call tree data:

1. In the **Software Analysis** view, expand the project name.

   The data source is listed under the project name.
2. Click the **Call Tree** hyperlink.

   The **Call Tree** viewer appears.



Call Tree - kl02z08_FLASH_OpenSDA.0000 - kl02z08

Function Name	Num Calls	% Total calls of parent	% Total times it was called	Inclusive Time (Cycles; 50.0 MHz)
<START>				
main	1	100.00	100.00	0
entry	1	100.00	100.00	0
Performance1	1	4.35	11.11	0
Recursive	1	4.35	5.56	0
Performance1	1	4.35	11.11	0
InterruptTest	5	21.74	100.00	0
Recursive	5	21.74	27.78	0
Performance1	5	21.74	55.56	0
Launch	5	21.74	100.00	0

Figure 4-20. Call Tree Viewer

In the Call Tree viewer, START is the root of the tree. You can click "+" to expand the tree and "-" to collapse the tree. It shows the biggest depth for stack utilization in **Call Tree** and the functions on this call path are displayed in green color.

The Call Tree nodes are synchronized with the source code. You can double-click the node to view the source code.

The table below describe the fields of **Call Tree** data. The columns are movable; you can move the columns to the left or right of another column depending on your requirements by dragging and dropping.

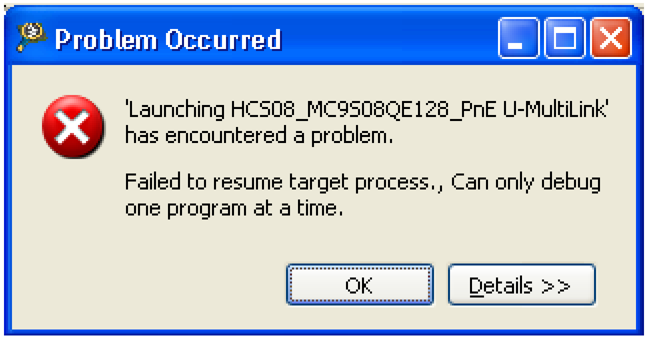
Table 4-8. Call Tree Viewer Fields

Name	Description
Function Name	Name of function that has executed.
Num Calls	Number of times function has executed.
% Total calls of parent	Percent of number of function calls from total number of calls in the application.
% Total times it was called	Percent of number of times a function was called.
Inclusive Time	Cumulative count during execution time spent from function entry to exit.

You can set CPU frequency and set Inclusive Time displayed in the **Call Tree** viewer in cycles, milliseconds, microseconds, and nanoseconds. Click the **Configure Table** button to configure time unit and set CPU frequency. For more information, refer Configuring Time Unit and Time Format.

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 the figure below. On clicking **OK**, the second debug session will terminate and trace will not collect.To collect trace, terminate the initial debug session and start it again.


Problem Occurred

'Launching HCS08_MC9S08QE128_PnE U-MultiLink' has encountered a problem.

Failed to resume target process., Can only debug one program at a time.

Figure 4-21. Error Message

4.6 Importing Trace Data Offline

You can import a trace data file offline from one project to another. Trace data can be imported into an existing project or a new project using the Import wizard. Importing a trace data file or raw trace in an existing project and existing launch config needs to be performed when you want to utilize that project's resources, that is the elf and the source mappings.

Before importing a raw trace into an existing project, replace the executable file (elf) of the existing project with the executable file of the project associated with the trace being imported. Make sure that you keep the original name of the executable file (of the existing project).

The executable is also required when you are importing raw trace data into a new project.

To import a trace data file offline to a new project:

1. Select **File > Import** to open the **Import** wizard.
2. Expand the **Software Analysis** node and select **Trace**.

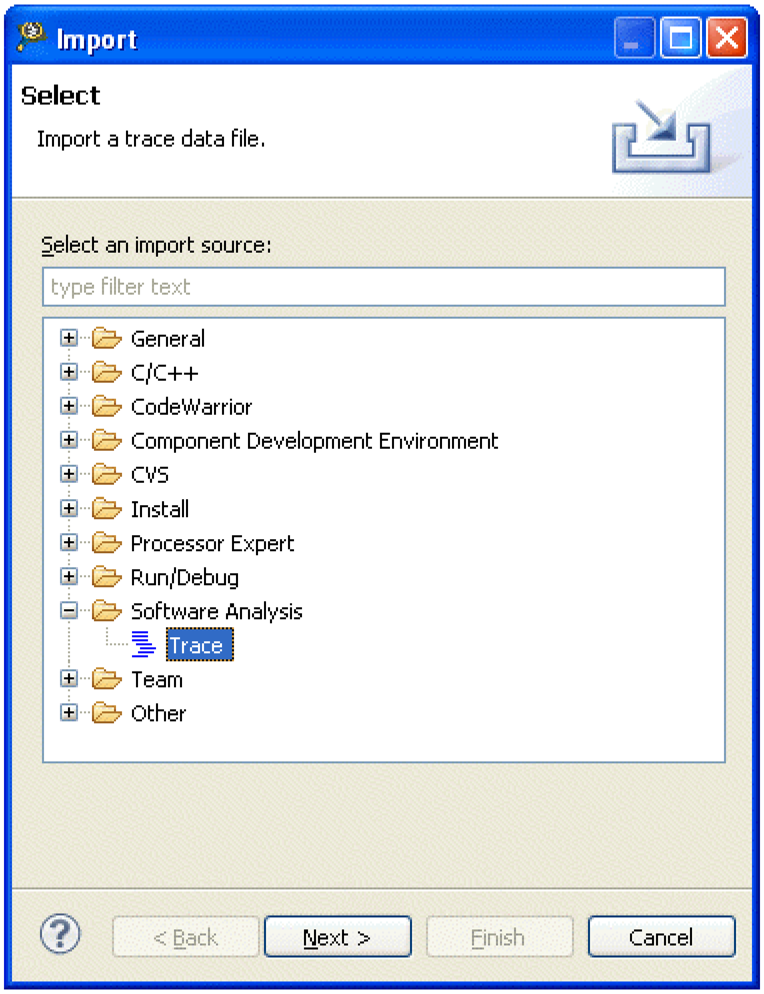
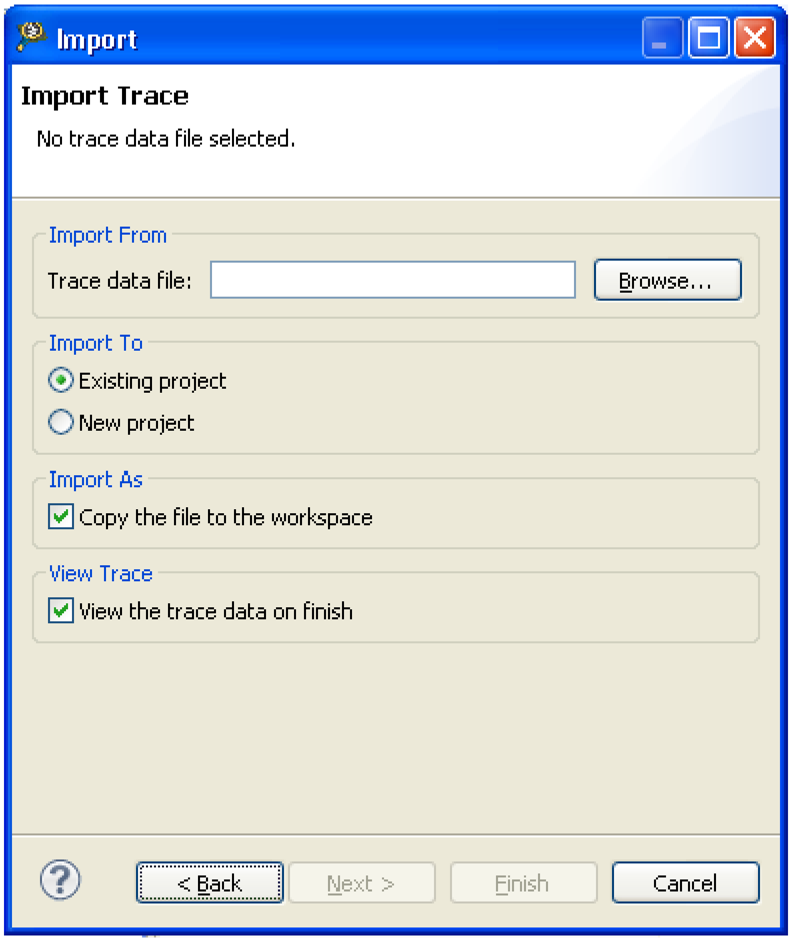


Figure 4-22. Select Page - Import Wizard

3. Click **Next** to display the **Import Trace** page of the **Import** wizard.



Import Trace

No trace data file selected.

Import From

Trace data file: Browse...

Import To

[✓ Existing project]

New project

Import As

[✓ Copy the file to the workspace]

View Trace

[✓ View the trace data on finish]

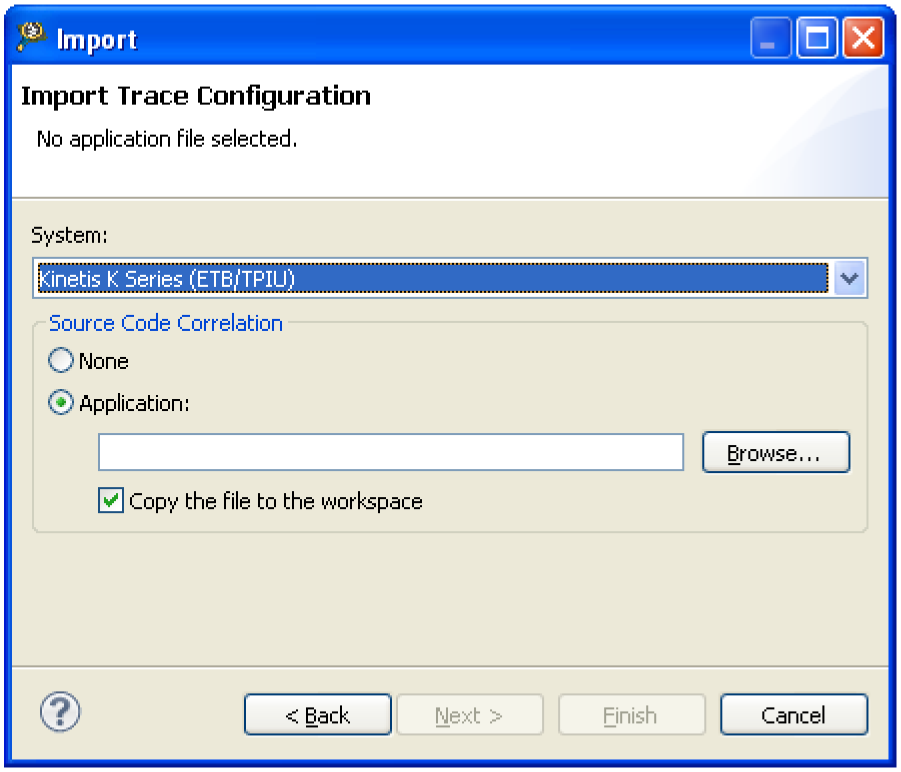
Figure 4-23. Import Trace Page

4. Click **Browse** and locate the trace data file of the project that you want to import into your project. The trace data file is located in the `.Analysis Data` folder of the project in the workspace.
5. Select the **New project** option to import trace data into a new project, which is created with a dedicated launch configuration. You can use this launch configuration to update the source path mappings in case sources have changed from the time of raw trace data collection.

NOTE

If you select the **Existing project** option in the **Import To** group in Figure 4-23 and click **Next**, the **Import Trace to Existing Project** page appears. In this page, type or select the project in which you want to import the trace data. Type a new file name for the trace data file in the **Enter new file name** text box. This field is optional.

6. Click **Next** to display the **Import Trace Configuration** page. This page allows you to specify the trace configuration details of the project to display imported trace data based on the executable file or launch configuration used during trace collection.



Import Trace Configuration

No application file selected.

System: Kinetis K Series (ETB/TPIU)

Source Code Correlation

None [✓ Application:]

Browse...

✓ Copy the file to the workspace

Figure 4-24. Import Trace Configuration Page

7. Specify the target of your project in the **System** drop-down box.
8. In the **Source Code Correlation** group, you can select any of the following options:
   - **None** if you do not want to correlate trace data with the source of your project, that is when the trace being imported is not program trace. The trace data will appear without displaying the source lines corresponding to the trace events.
   - **Application** if you want to correlate trace data with the source. Click **Browse** to locate the executable file (`.abs`/ `.elf`/ `.afx`) of the application used during trace collection.

NOTE

In case you are importing trace into an existing project, you can select the **Application** option if the trace data you are importing was not collected with any of the project's executable (or it has changed in the meantime). The imported raw trace will be (re)decoded with the new executable. Click **Browse** to locate the executable file.

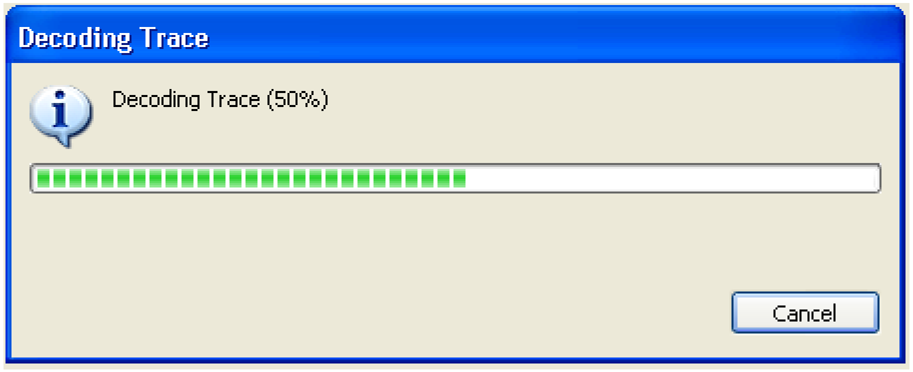
NOTE

The **Launch Configuration** option is also displayed when you choose to import trace data into an existing project. Use this option when the raw trace was collected with one of the project's launch configuration. You will reuse the project's resources specified in the selected launch configuration (elf, source path mappings). Make sure that the elf did not change from the time when the imported raw trace was collected.

9. Click **Finish**.

If you kept the **View trace data on finish** checkbox checked in the Figure 4-23, the imported trace data will automatically open in the **Trace Data** viewer of your project. If you uncheck this option, the imported data will be visible from the **Software Analysis** view. The imported trace data is saved in the `.Analysis Data` folder of your project also.

After importing trace data, the **Decoding Trace** dialog box appears displaying the progress of the trace to be decoded. If the trace data file to be imported is relatively small in size, the **Decoding Trace** dialog box disappears after a few seconds. However, when you import a heavy trace file (around 300 - 400 KB), the **Decoding Trace** dialog box displays the percentage of trace to be decoded.



Decoding Trace (50%)

Figure 4-25. Decoding Trace

If you do not want to see the complete contents of the trace file, you can cancel the trace in between by clicking **Cancel** . For example, if you cancel the trace decoding at 20%, only that much amount of trace will be displayed in the **Trace Data** viewer. After cancelling, you get a message box displaying that decoding is not finished and trace displayed will be incomplete. Click **OK** in the message box.

Chapter 5
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. Tracepoints are used for optimizing trace collection by tracing only on the code of interest and discarding the rest of the code.

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
- Trace Modes
- Enabling and Disabling the Tracepoints

5.1 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

For more information on trigger modes, refer Table 3-1.

Warning

If you are setting breakpoints with triggers for trace control in the HCS08 target, ensure that you do not to use more than one breakpoint. This is because on HCS08, two hardware debug modules, BDC (Background debug controller) and DBG (debug module) are used. Both debug modules can be used for setting hardware breakpoints, while only DBG can be used for setting triggers. The first breakpoint is set using BDC, any other breakpoints will use DBG and will conflict with the triggers. Therefore, it is mandatory not to use more than one breakpoint when setting triggers on the HCS08 platform.

5.2 Trace Modes

The triggers can be set in the following trace modes:

- Collect Program Trace
  - Continuously

    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

    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

  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
- Expert

NOTE

The HCS08 target does not generate trace data on the default stationery project. This is because the default source code does not contain any branch required for an HCS08 application to generate trace. To generate trace, you need to add a subroutine to the default code and call it from the `main()` function.

This topic contains the following sub-topics:

- Setting Triggers in Continuously Mode
- Setting Triggers in Automatically Mode
- Setting Triggers in Collect Data Trace Mode

- Collecting Trace in Profile-Only Mode
- Collecting Trace in Expert Mode

5.2.1 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, refer Collecting Data chapter for information about how to collect the trace and profiling data.

5.2.1.1 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 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 below.
   Listing: Source Code 1

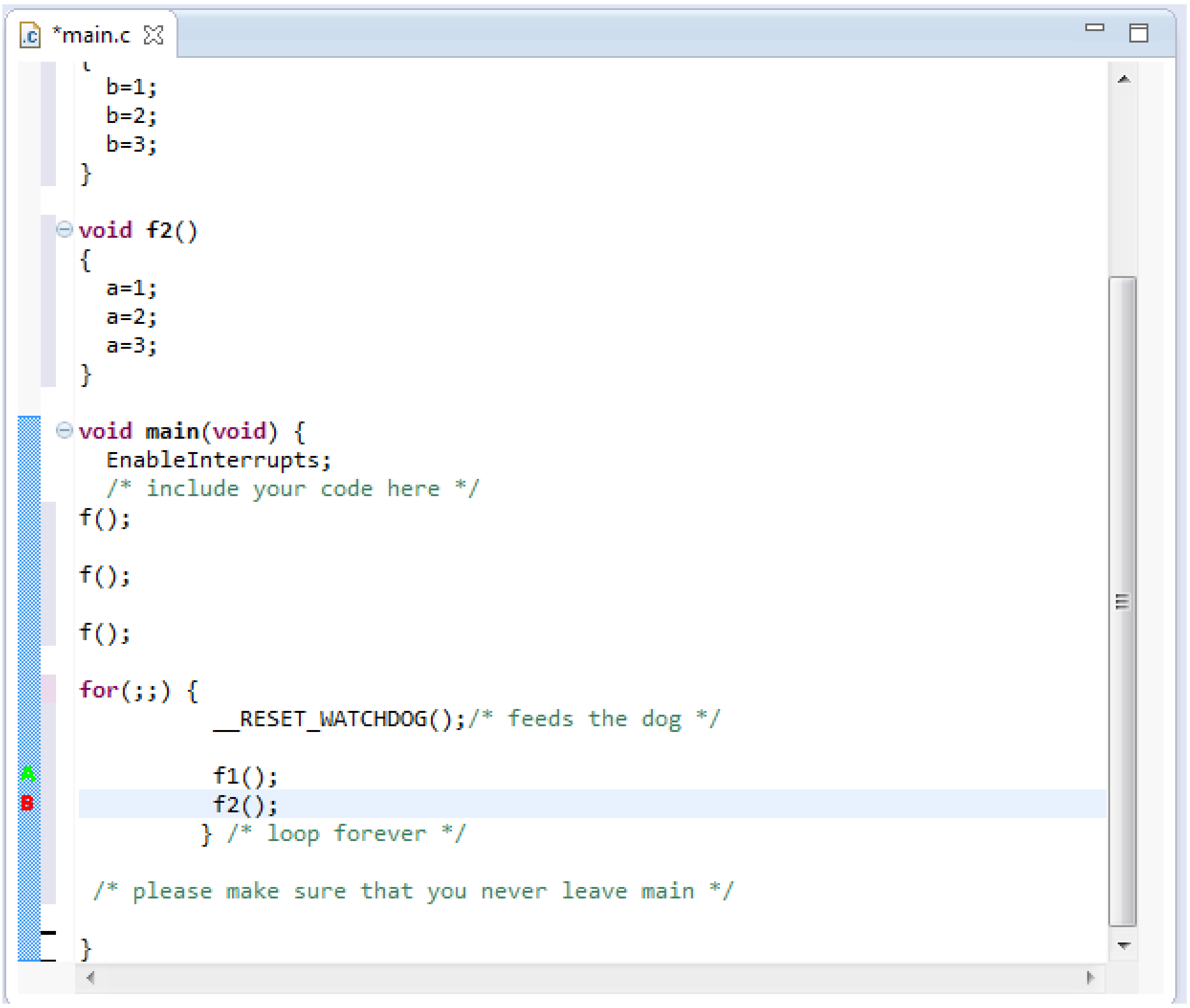
```
#include <hidef.h> /* for EnableInterrupts macro */

#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 the marker bar corresponding to the statement, `f1();`.
8. Select **Trace Triggers > Toggle Trace Trigger A** from the context menu. The same option is also used to remove trigger A from the marker bar.
9. Right-click 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.



```
  {
    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 */

A         f1();
B         f2();
        } /* loop forever */

 /* please make sure that you never leave main */

}
```

Figure 5-1. Trigger A and Trigger B Set in the Editor Area

NOTE

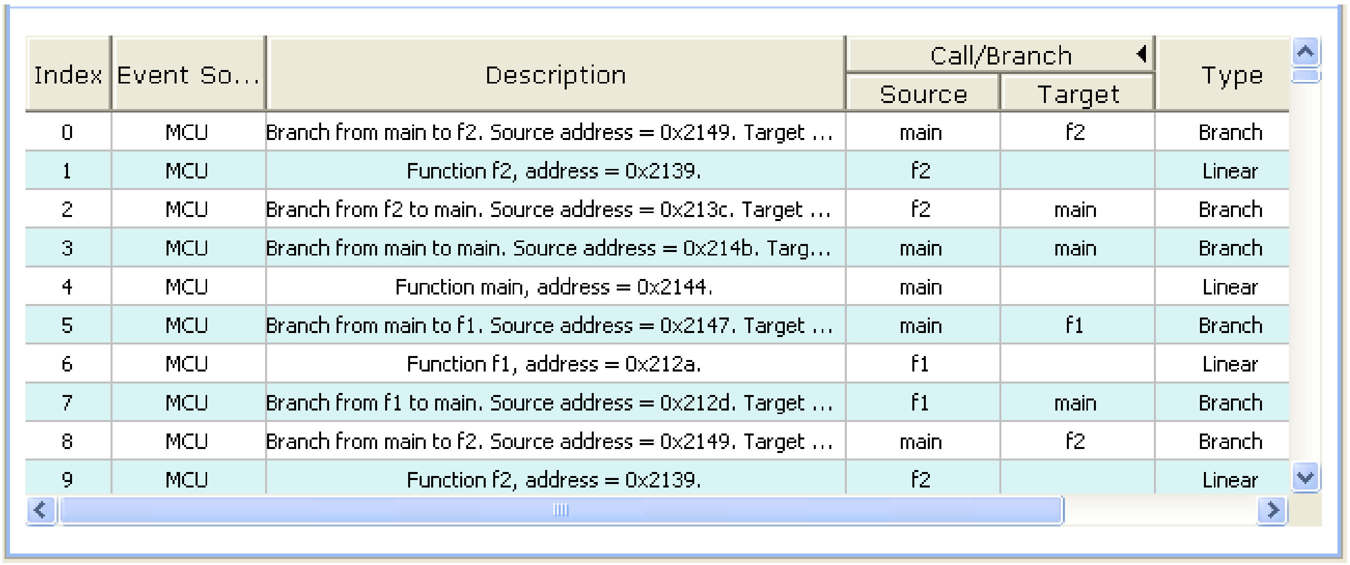
Do not set tracepoints on the statements containing only comments, brackets, and variable declaration with no value. If tracepoints are set on invalid lines, they are automatically disabled when the application is debugged.

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. Select the **Collect Trace From Trigger** option in the **Trace Start/Stop Conditions** group.
16. Select the **Instruction at Address A, Then Instruction at Address B are Executed** option from the **Trigger Type** drop-down list.

17. Clear the **Keep Last Buffer Before Trigger** checkbox.
18. Ensure that the **Instruction Execute** option is selected in the **Trigger Selection** group.
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 to view the collected data.

The figure below 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 the figure, 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.



Index	Event So...	Description	Call/Branch		Type
			Source	Target	
0	MCU	Branch from main to f2. Source address = 0x2149. Target ...	main	f2	Branch
1	MCU	Function f2, address = 0x2139.	f2		Linear
2	MCU	Branch from f2 to main. Source address = 0x213c. Target ...	f2	main	Branch
3	MCU	Branch from main to main. Source address = 0x214b. Targ...	main	main	Branch
4	MCU	Function main, address = 0x2144.	main		Linear
5	MCU	Branch from main to f1. Source address = 0x2147. Target ...	main	f1	Branch
6	MCU	Function f1, address = 0x212a.	f1		Linear
7	MCU	Branch from f1 to main. Source address = 0x212d. Target ...	f1	main	Branch
8	MCU	Branch from main to f2. Source address = 0x2149. Target ...	main	f2	Branch
9	MCU	Function f2, address = 0x2139.	f2		Linear

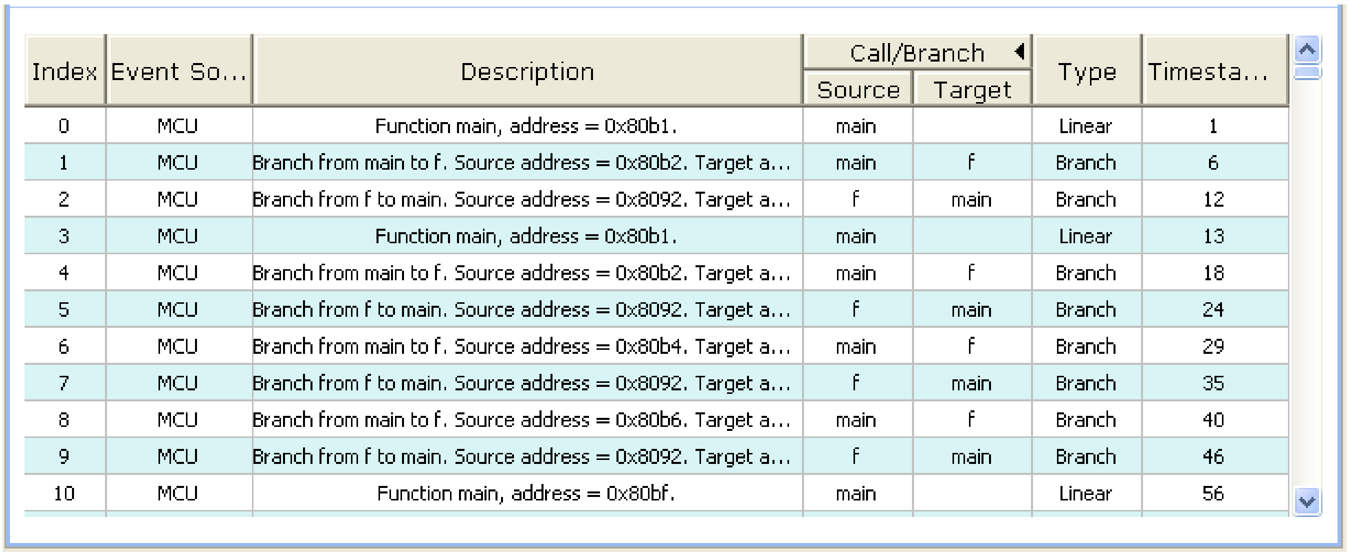
Figure 5-2. Trace Data After Setting *From* Trigger in Continuously Mode

The figure below 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 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, as shown in the above figure, is different from the data file generated without the tracepoints, as shown in the figure below.

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.



Index	Event So...	Description	Call/Branch		Type	Timesta...
			Source	Target		
0	MCU	Function main, address = 0x80b1.	main		Linear	1
1	MCU	Branch from main to f. Source address = 0x80b2. Target a...	main	f	Branch	6
2	MCU	Branch from f to main. Source address = 0x8092. Target a...	f	main	Branch	12
3	MCU	Function main, address = 0x80b1.	main		Linear	13
4	MCU	Branch from main to f. Source address = 0x80b2. Target a...	main	f	Branch	18
5	MCU	Branch from f to main. Source address = 0x8092. Target a...	f	main	Branch	24
6	MCU	Branch from main to f. Source address = 0x80b4. Target a...	main	f	Branch	29
7	MCU	Branch from f to main. Source address = 0x8092. Target a...	f	main	Branch	35
8	MCU	Branch from main to f. Source address = 0x80b6. Target a...	main	f	Branch	40
9	MCU	Branch from f to main. Source address = 0x8092. Target a...	f	main	Branch	46
10	MCU	Function main, address = 0x80bf.	main		Linear	56

Figure 5-3. 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 below 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.

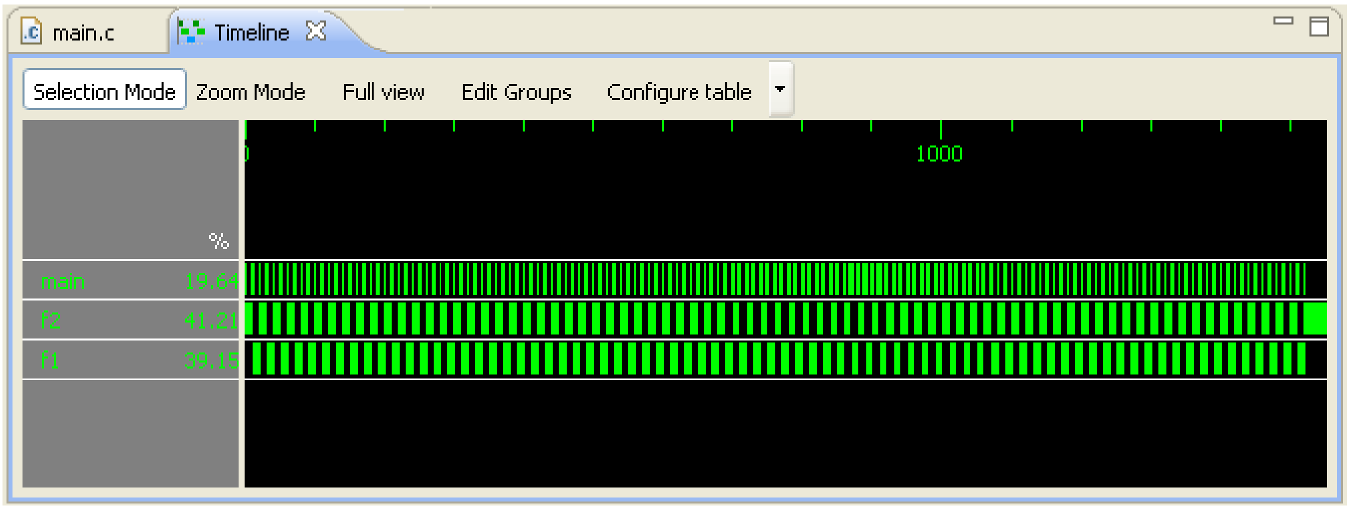


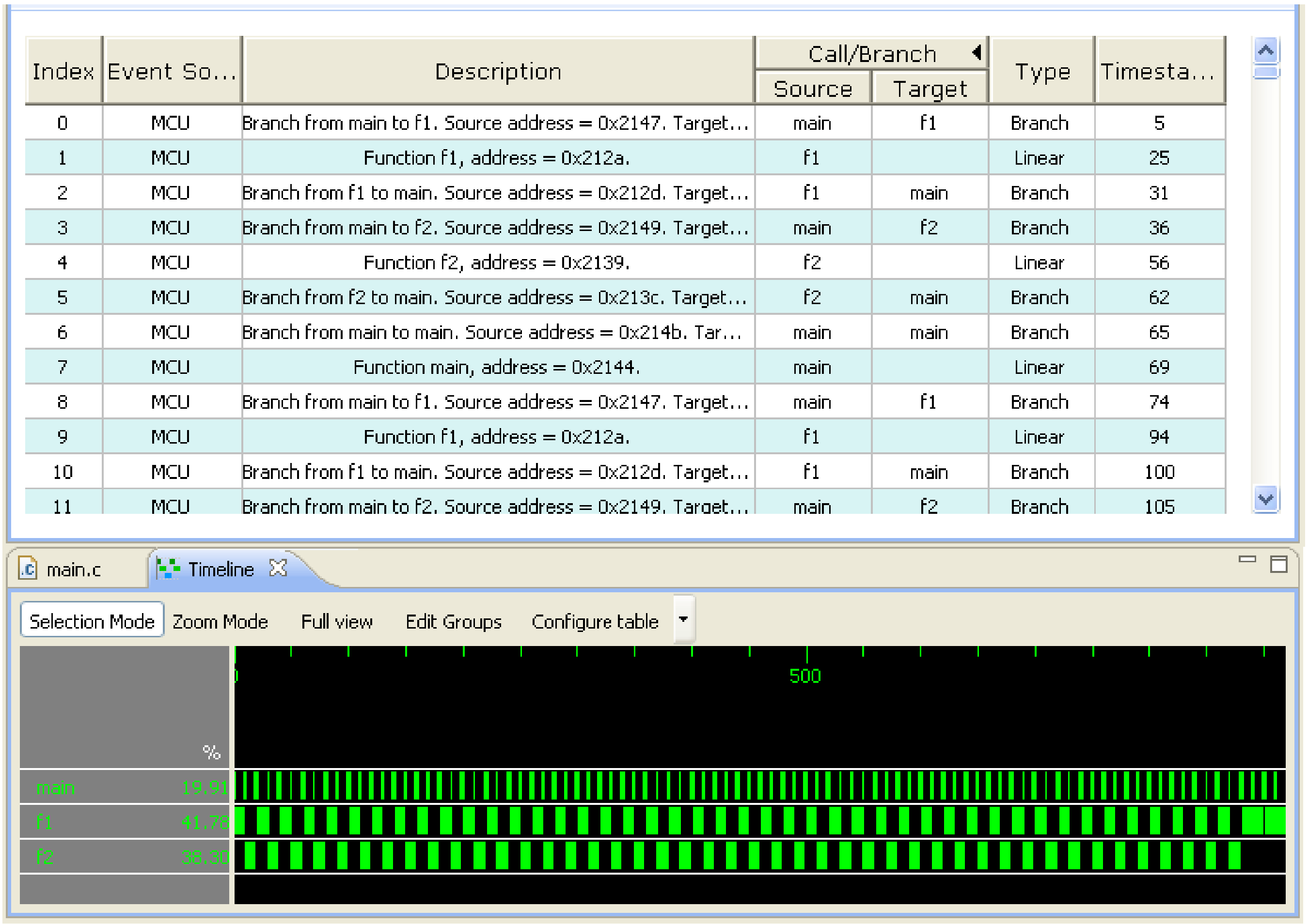
Figure 5-4. Graph Displaying Timeline of Trace Data

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.

5.2.1.2 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.

The figure below 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.



Index	Event So...	Description	Call/Branch		Type	Timesta...
			Source	Target		
0	MCU	Branch from main to f1. Source address = 0x2147. Target...	main	f1	Branch	5
1	MCU	Function f1, address = 0x212a.	f1		Linear	25
2	MCU	Branch from f1 to main. Source address = 0x212d. Target...	f1	main	Branch	31
3	MCU	Branch from main to f2. Source address = 0x2149. Target...	main	f2	Branch	36
4	MCU	Function f2, address = 0x2139.	f2		Linear	56
5	MCU	Branch from f2 to main. Source address = 0x213c. Target...	f2	main	Branch	62
6	MCU	Branch from main to main. Source address = 0x214b. Tar...	main	main	Branch	65
7	MCU	Function main, address = 0x2144.	main		Linear	69
8	MCU	Branch from main to f1. Source address = 0x2147. Target...	main	f1	Branch	74
9	MCU	Function f1, address = 0x212a.	f1		Linear	94
10	MCU	Branch from f1 to main. Source address = 0x212d. Target...	f1	main	Branch	100
11	MCU	Branch from main to f2. Source address = 0x2149. Target...	main	f2	Branch	105

Figure 5-5. Trace Data - Instruction at Address A or Address B is Executed

5.2.1.3 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.

NOTE

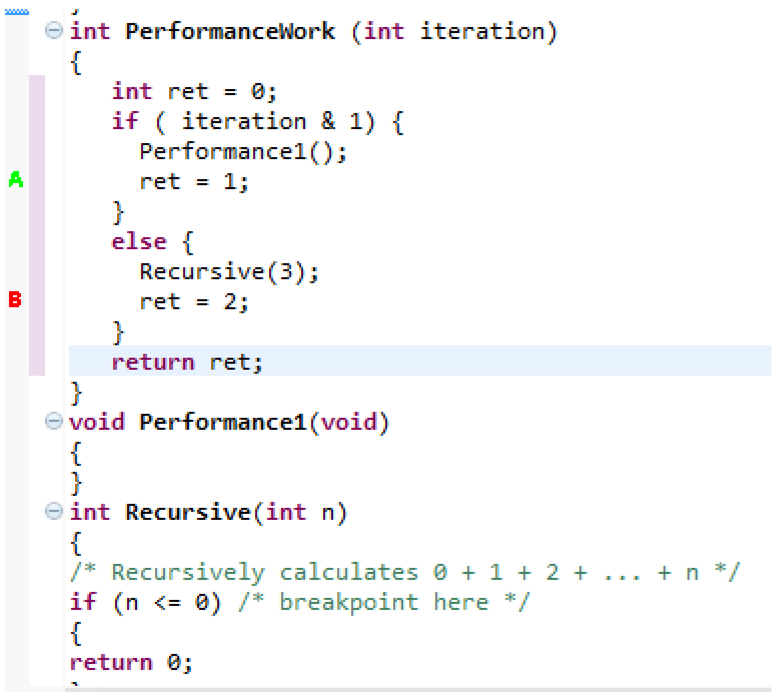
For the MC9S08PT60 target specifically, the **Instruction Inside Range from Address A to Address B is Executed** trigger will hit when any instruction inside the range between trigger address A and trigger address B matches with the data on the bus or program address inside the range.

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 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 below.
   Listing: 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.



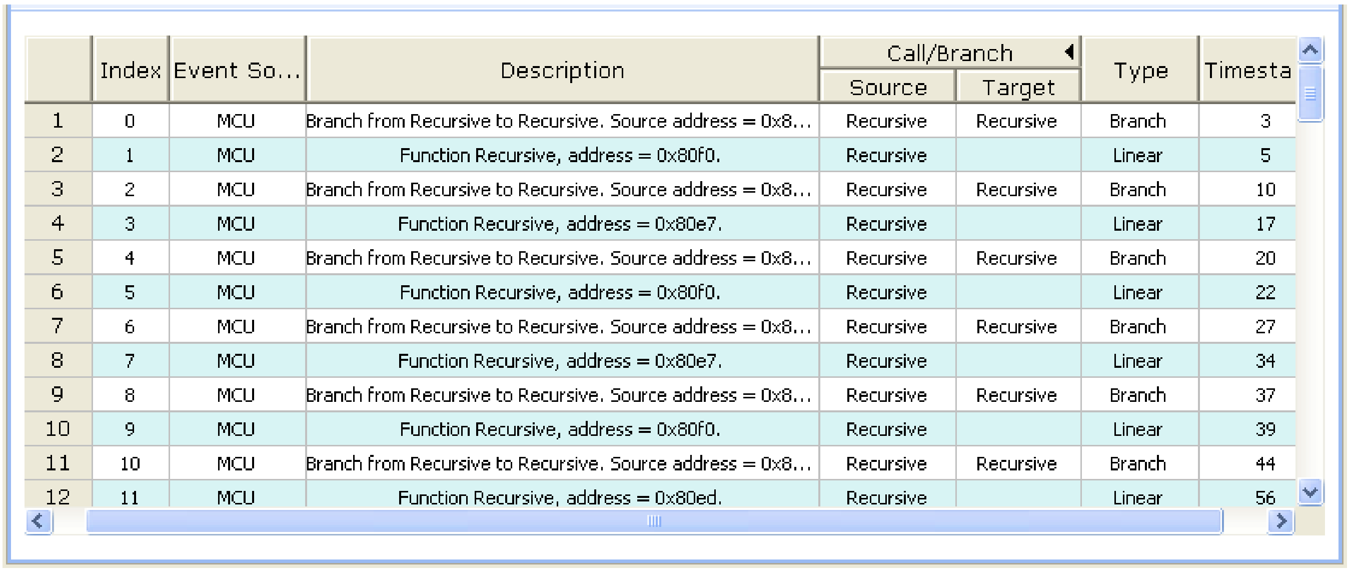
```
int PerformanceWork (int iteration)
{
   int ret = 0;
   if ( iteration & 1) {
     Performance1();
A    ret = 1;
   }
   else {
     Recursive(3);
B    ret = 2;
   }
   return ret;
}
void Performance1(void)
{
}
int Recursive(int n)
{
/* Recursively calculates 0 + 1 + 2 + ... + n */
if (n <= 0) /* breakpoint here */
{
return 0;
```

Figure 5-6. 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 to view the collected data.

    The figure below 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.



	Index	Event So...	Description	Call/Branch		Type	Timesta
				Source	Target		
1	0	MCU	Branch from Recursive to Recursive. Source address = 0x8...	Recursive	Recursive	Branch	3
2	1	MCU	Function Recursive, address = 0x80f0.	Recursive		Linear	5
3	2	MCU	Branch from Recursive to Recursive. Source address = 0x8...	Recursive	Recursive	Branch	10
4	3	MCU	Function Recursive, address = 0x80e7.	Recursive		Linear	17
5	4	MCU	Branch from Recursive to Recursive. Source address = 0x8...	Recursive	Recursive	Branch	20
6	5	MCU	Function Recursive, address = 0x80f0.	Recursive		Linear	22
7	6	MCU	Branch from Recursive to Recursive. Source address = 0x8...	Recursive	Recursive	Branch	27
8	7	MCU	Function Recursive, address = 0x80e7.	Recursive		Linear	34
9	8	MCU	Branch from Recursive to Recursive. Source address = 0x8...	Recursive	Recursive	Branch	37
10	9	MCU	Function Recursive, address = 0x80f0.	Recursive		Linear	39
11	10	MCU	Branch from Recursive to Recursive. Source address = 0x8...	Recursive	Recursive	Branch	44
12	11	MCU	Function Recursive, address = 0x80ed.	Recursive		Linear	56

Figure 5-7. 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.

This is how you can collect trace using the **Instruction Inside Range from Address A to Address B is Executed** trigger type.

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

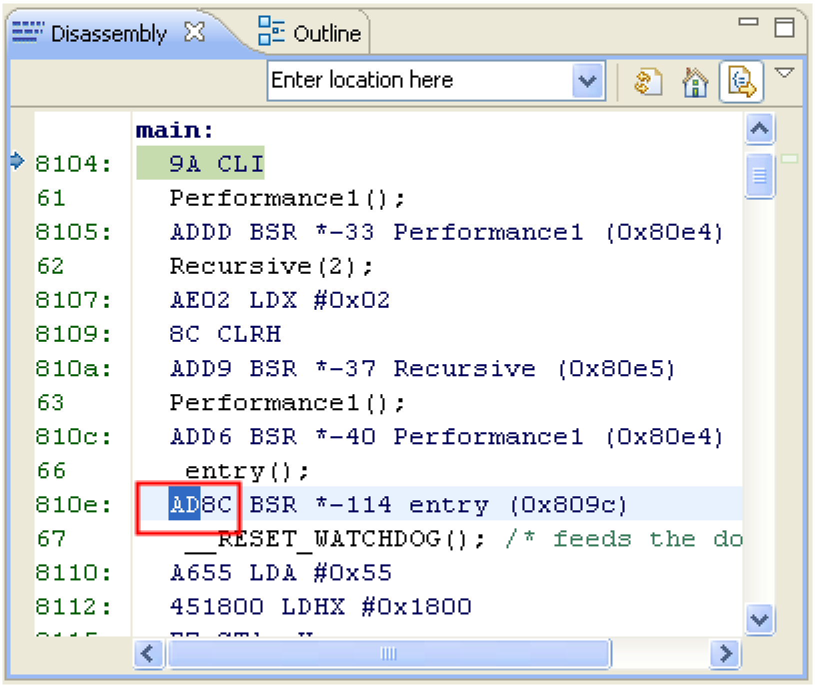
The **Instruction at Address A is Executed and Value on Data Bus Match** trigger type is used to trigger on a program instruction execution at trigger A address when the opcode of that instruction matches a specific byte value. This trigger type is useful in detecting the self-modifying instructions 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 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: Source code 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.

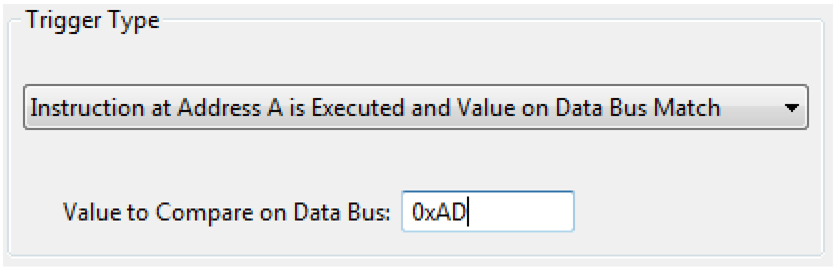
14. After the application is debugged, set trigger A at `entry();` in the `main()` function as shown in figure below.
15. In the **Disassembly** view, copy first two hexadecimal digits at call to `entry()` from `main()` as shown below.



```
main:
8104:   9A CLI
61      Performance1();
8105:   ADDD BSR *-33 Performance1 (0x80e4)
62      Recursive(2);
8107:   AE02 LDX #0x02
8109:   8C CLRH
810a:   ADD9 BSR *-37 Recursive (0x80e5)
63      Performance1();
810c:   ADD6 BSR *-40 Performance1 (0x80e4)
66       entry();
810e:   AD8C BSR *-114 entry (0x809c)
67       __RESET_WATCHDOG(); /* feeds the do
8110:   A655 LDA #0x55
8112:   451800 LDHX #0x1800
```

Figure 5-8. 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.



Trigger Type

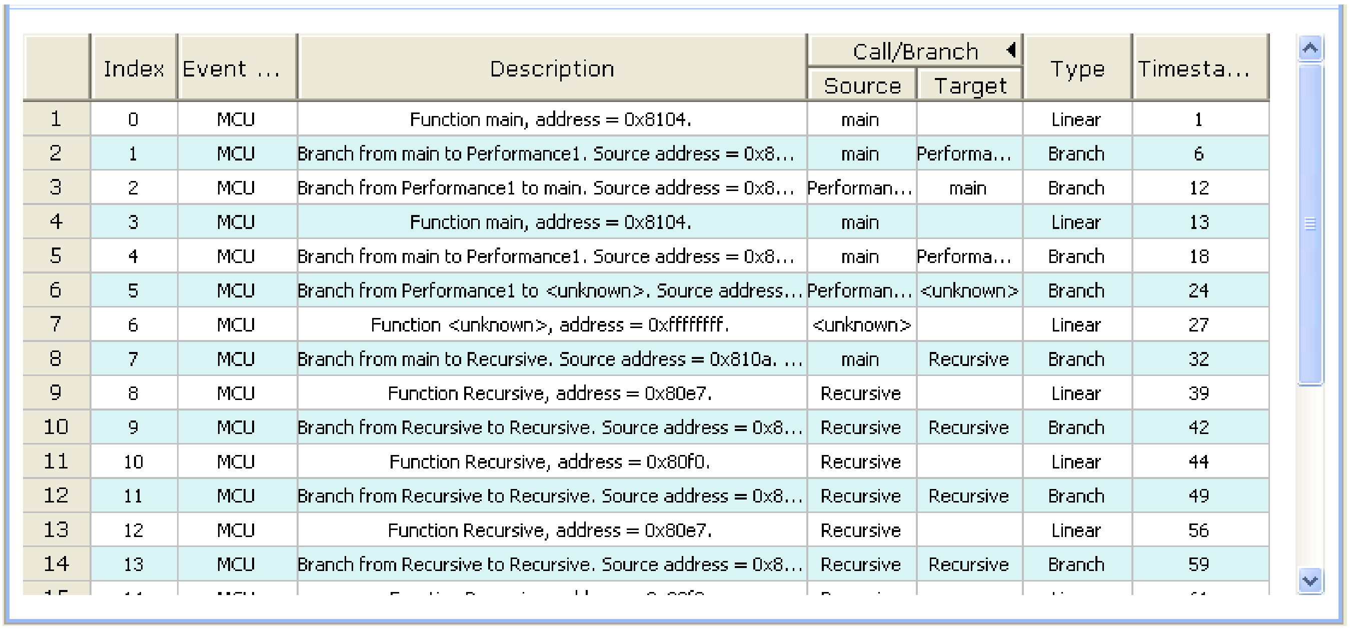
Instruction at Address A is Executed and Value on Data Bus Match

Value to Compare on Data Bus: 0xAD

Figure 5-9. Value to Compare on Data Bus Text Box

18. Click **Apply** and close the **Debug Configurations** dialog box.
19. Click the **Terminate** button 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 Viewing Data to view the collected data.

The figure below 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.



	Index	Event ...	Description	Call/Branch		Type	Timesta...
				Source	Target		
1	0	MCU	Function main, address = 0x8104.	main		Linear	1
2	1	MCU	Branch from main to Performance1. Source address = 0x8...	main	Performa...	Branch	6
3	2	MCU	Branch from Performance1 to main. Source address = 0x8...	Performan...	main	Branch	12
4	3	MCU	Function main, address = 0x8104.	main		Linear	13
5	4	MCU	Branch from main to Performance1. Source address = 0x8...	main	Performa...	Branch	18
6	5	MCU	Branch from Performance1 to <unknown>. Source address...	Performan...	<unknown>	Branch	24
7	6	MCU	Function <unknown>, address = 0xffffffff.	<unknown>		Linear	27
8	7	MCU	Branch from main to Recursive. Source address = 0x810a. ...	main	Recursive	Branch	32
9	8	MCU	Function Recursive, address = 0x80e7.	Recursive		Linear	39
10	9	MCU	Branch from Recursive to Recursive. Source address = 0x8...	Recursive	Recursive	Branch	42
11	10	MCU	Function Recursive, address = 0x80f0.	Recursive		Linear	44
12	11	MCU	Branch from Recursive to Recursive. Source address = 0x8...	Recursive	Recursive	Branch	49
13	12	MCU	Function Recursive, address = 0x80e7.	Recursive		Linear	56
14	13	MCU	Branch from Recursive to Recursive. Source address = 0x8...	Recursive	Recursive	Branch	59

Figure 5-10. Trace Data - Instruction at Address A is Executed and Value on Data Bus Match

5.2.1.5 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.

5.2.1.6 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.

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 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: Source code 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. In the `main ()` function of the source code, before `for(;;)`, type the statement, `a = 1;`.
6. Save and build the project.
7. Open the **Debug Configurations** dialog box, and select your project in the tree structure.
8. Click the **Trace and Profile** tab, and check the **Enable Trace and Profile** checkbox.
9. Select the **Collect Program Trace** option in the **Trace Mode Options** group.
10. Select the **Continuously** option.
11. Select the **Collect Trace From Trigger** option in the **Trace Start/Stop Conditions** group.
12. Clear the **Keep Last Buffer Before Trigger** checkbox.
13. Select the **Memory Access** option in the **Trigger Selection** group.
14. Select the **Memory at Address A is Accessed** option from the **Trigger Type** drop-down list.
15. Click **Apply** to save the settings.
16. Click **Debug** to debug the application.
17. After the application is debugged, right-click in the **Variables** view, and select the **Add Global Variables** option from the context menu.

    The **Add Globals** dialog box appears.

18. Scroll down and select the variable `a`.
19. Click **OK**.

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

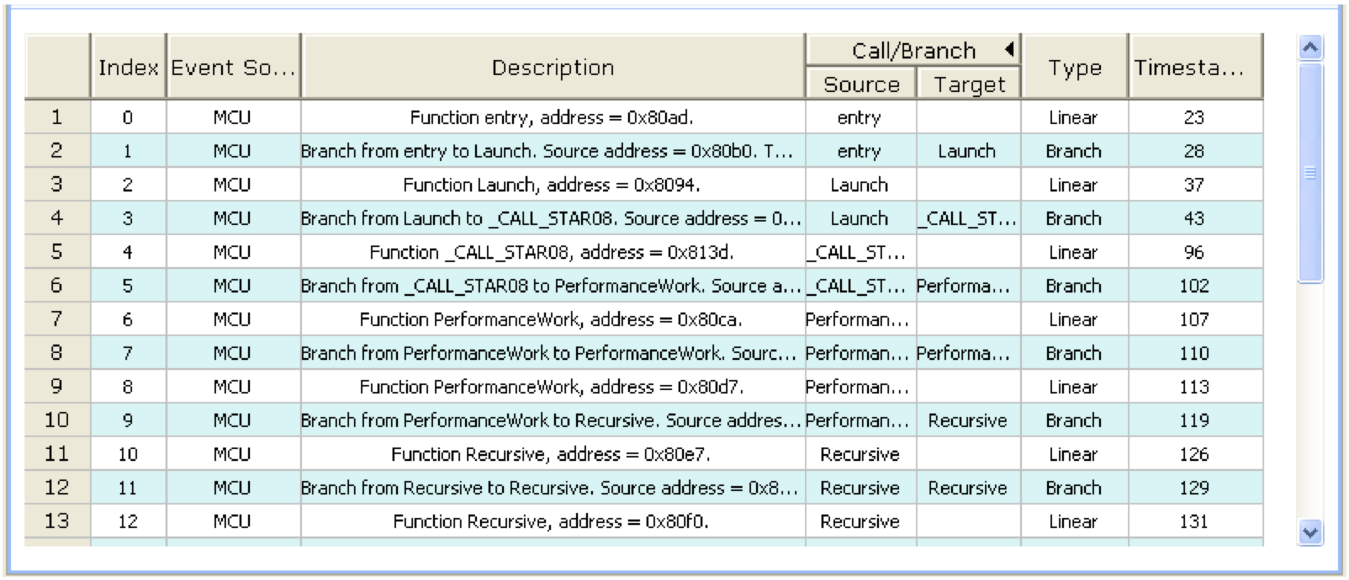
20. Right-click the variable `a` and select **Toggle Triggers > Toggle HCS08 Trace Trigger A** from the context menu.

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

21. Click **OK**.
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 to view the trace results. The figure below 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.



	Index	Event So...	Description	Call/Branch		Type	Timesta...
				Source	Target		
1	0	MCU	Function entry, address = 0x80ad.	entry		Linear	23
2	1	MCU	Branch from entry to Launch. Source address = 0x80b0. T...	entry	Launch	Branch	28
3	2	MCU	Function Launch, address = 0x8094.	Launch		Linear	37
4	3	MCU	Branch from Launch to _CALL_STAR08. Source address = 0...	Launch	_CALL_ST...	Branch	43
5	4	MCU	Function _CALL_STAR08, address = 0x813d.	_CALL_ST...		Linear	96
6	5	MCU	Branch from _CALL_STAR08 to PerformanceWork. Source a...	_CALL_ST...	Performa...	Branch	102
7	6	MCU	Function PerformanceWork, address = 0x80ca.	Performan...		Linear	107
8	7	MCU	Branch from PerformanceWork to PerformanceWork. Sourc...	Performan...	Performa...	Branch	110
9	8	MCU	Function PerformanceWork, address = 0x80d7.	Performan...		Linear	113
10	9	MCU	Branch from PerformanceWork to Recursive. Source addres...	Performan...	Recursive	Branch	119
11	10	MCU	Function Recursive, address = 0x80e7.	Recursive		Linear	126
12	11	MCU	Branch from Recursive to Recursive. Source address = 0x8...	Recursive	Recursive	Branch	129
13	12	MCU	Function Recursive, address = 0x80f0.	Recursive		Linear	131

Figure 5-11. Trace Data After Setting Memory Access Trigger

This is how you can collect trace using the **Memory at Address A is Accessed** trigger type.

NOTE

Similarly, you can set memory access triggers in the **Automatically** mode. Also, you can set other memory access triggers on variables or instructions, such as **Memory at Address A or Address B is Accessed, Memory Inside Range from Address A to Address B is Accessed** , **Memory Outside Range from Address A to Address B is Accessed** , **Memory at Address A, Then Memory at Address B are Accessed** , **Memory access at Address A and Value on Data Bus**

Match , **Memory access at Address A and Value on Data Bus Mismatch** , and obtain trace results accordingly.

5.2.2 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.

5.2.2.1 From the Disassembly View

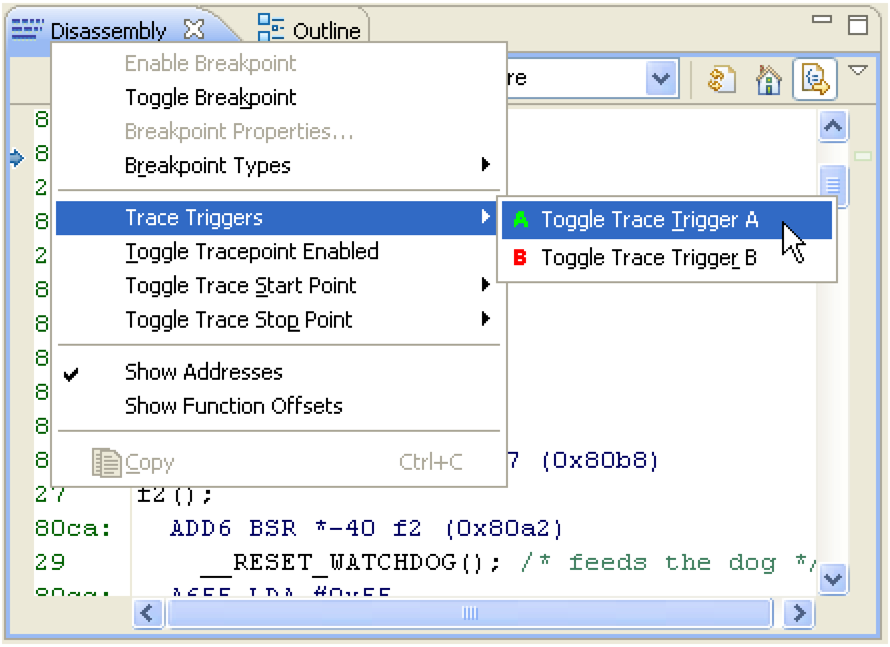
To set a trigger in the **Disassembly** view:

1. In the **CodeWarrior Projects** view, expand the `Sources` folder of your project.
2. Double-click 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 below.
   Listing: 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 the marker bar corresponding to the address line of the function, `f2()` before `_RESET_WATCHDOG()`.



```
7 (0x80b8)
27      f2();
80ca:     ADD6 BSR *-40 f2 (0x80a2)
29          __RESET_WATCHDOG(); /* feeds the dog */
```

Figure 5-12. 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.

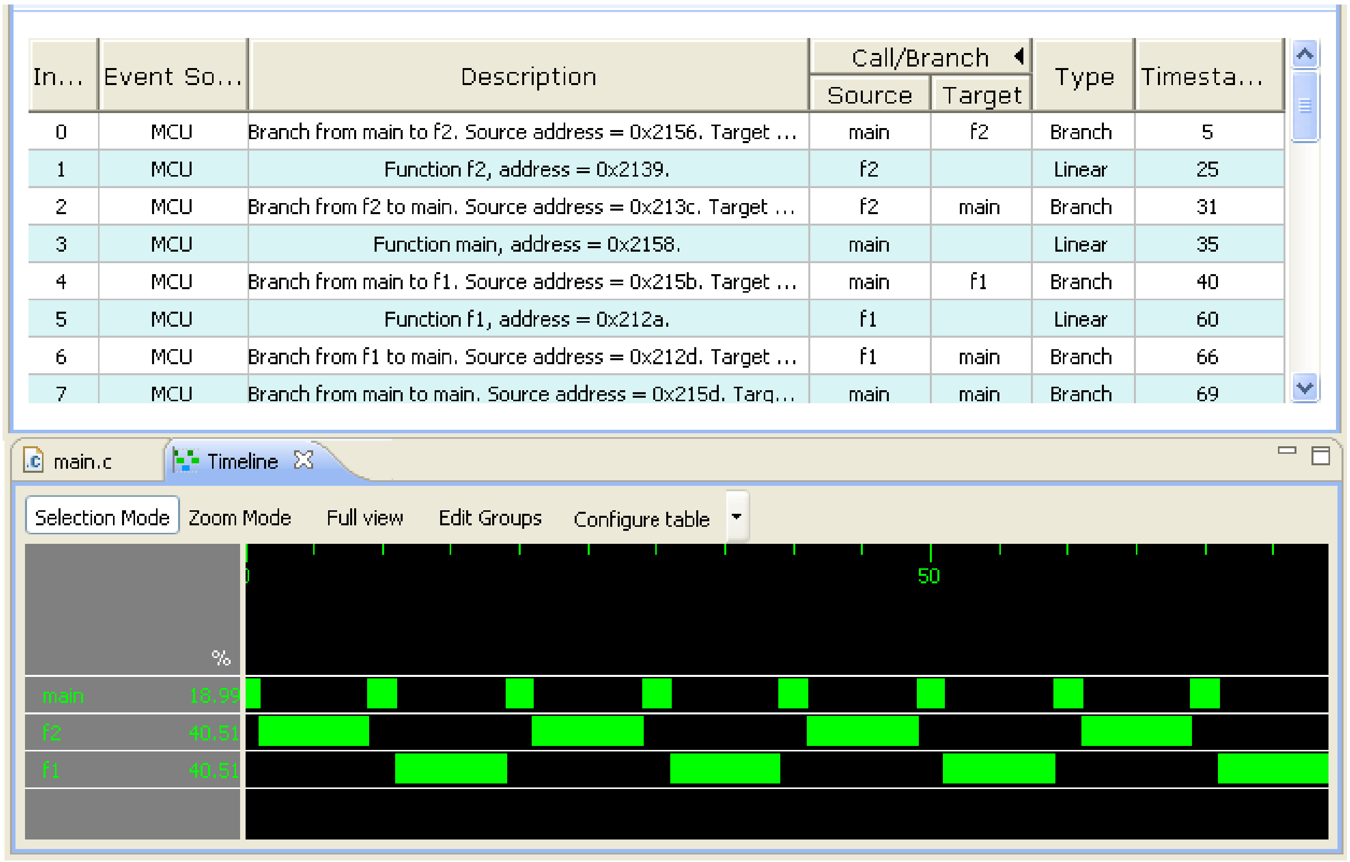
15. Click **Resume** to continue collecting trace.

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 to view the collected data.

The figure below 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 5-3 to view the data files that are generated before setting tracepoints. The **Trace Data** viewer in the figure below 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.



In...	Event So...	Description	Call/Branch		Type	Timesta...
			Source	Target		
0	MCU	Branch from main to f2. Source address = 0x2156. Target ...	main	f2	Branch	5
1	MCU	Function f2, address = 0x2139.	f2		Linear	25
2	MCU	Branch from f2 to main. Source address = 0x213c. Target ...	f2	main	Branch	31
3	MCU	Function main, address = 0x2158.	main		Linear	35
4	MCU	Branch from main to f1. Source address = 0x215b. Target ...	main	f1	Branch	40
5	MCU	Function f1, address = 0x212a.	f1		Linear	60
6	MCU	Branch from f1 to main. Source address = 0x212d. Target ...	f1	main	Branch	66
7	MCU	Branch from main to main. Source address = 0x215d. Targ...	main	main	Branch	69

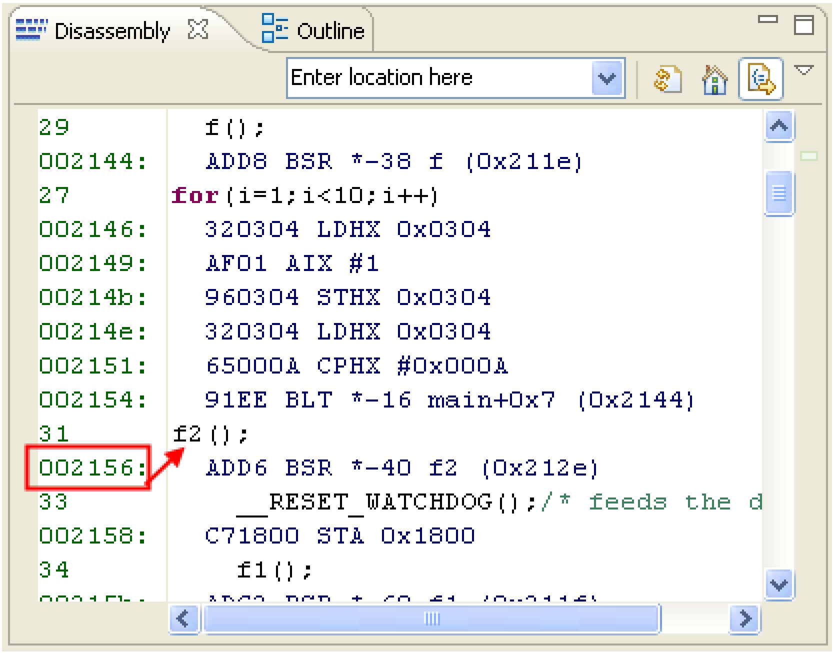
main	18.99
f2	40.51
f1	40.51

Figure 5-13. Trace Data After Setting *From* Trigger in Automatically Mode

5.2.2.2 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 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: Source code 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, view the address corresponding to the call to `f2()` before `__RESET_WATCHDOG()`.



```
29          f();
002144:     ADD8 BSR *-38 f (0x211e)
27        for(i=1;i<10;i++)
002146:     320304 LDHX 0x0304
002149:     AF01 AIX #1
00214b:     960304 STHX 0x0304
00214e:     320304 LDHX 0x0304
002151:     65000A CPHX #0x000A
002154:     91EE BLT *-16 main+0x7 (0x2144)
31        f2();
002156:     ADD6 BSR *-40 f2 (0x212e)
33            __RESET_WATCHDOG();/* feeds the d
002158:     C71800 STA 0x1800
34            f1();
```

Figure 5-14. Disassembly View - Selecting Function Address

12. Select **Window > Show View > Other > Debug > Memory** to open the **Memory** view.

13. Click the **Add Memory Monitor** button in the **Memory** view to open the **Monitor Memory** dialog box.
14. Enter the memory address in hexadecimal form in the **Enter address or expression to monitor** text box.

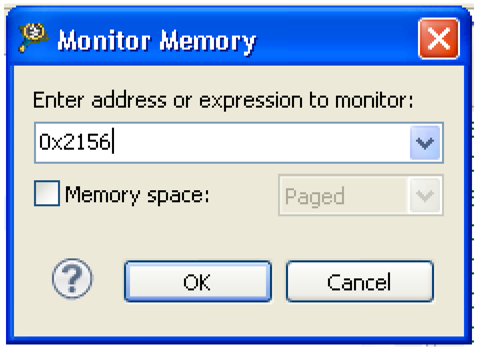


Figure 5-15. Monitor Memory Dialog Box

15. Click **OK**.

    The memory address appears in the **Memory** view.

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

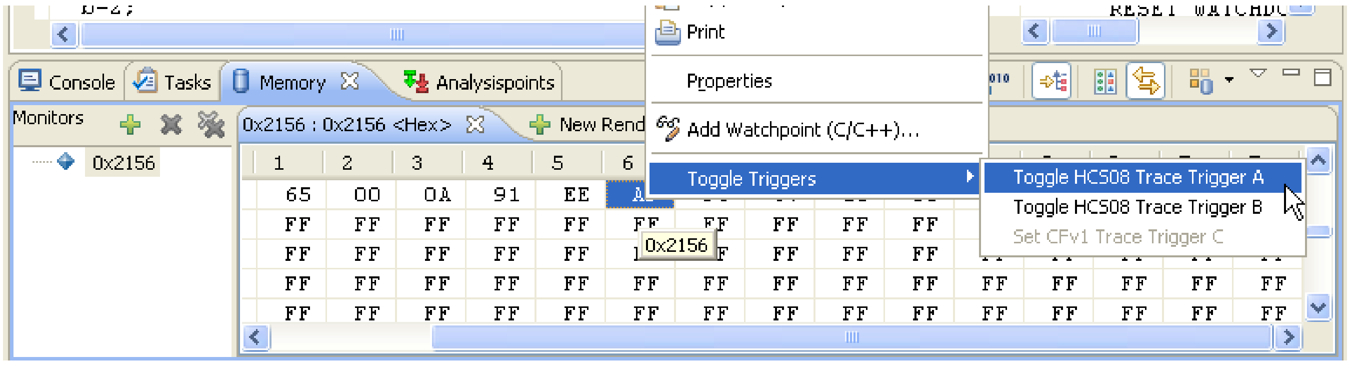
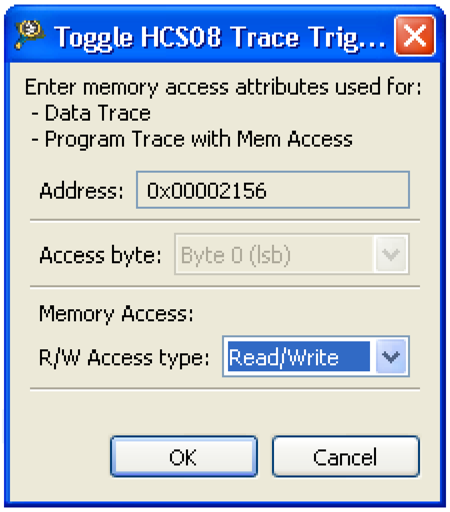


Figure 5-16. Memory View

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



Enter memory access attributes used for:
- Data Trace
- Program Trace with Mem Access

Address: 0x00002156

Access byte: Byte 0 (lsb)

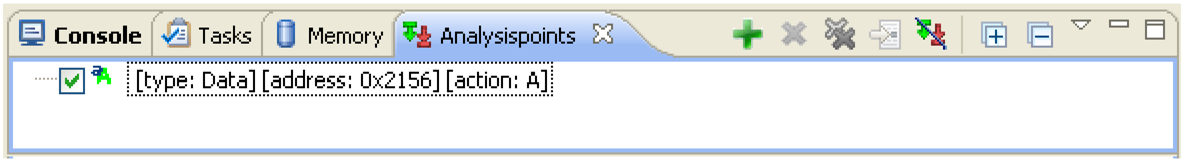
Memory Access:

R/W Access type: Read/Write

Figure 5-17. Toggle HCS08 Trace Trigger Dialog Box

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

    The **Analysispoints** view displays the trigger that you set on the memory address.



[type: Data] [address: 0x2156] [action: A]

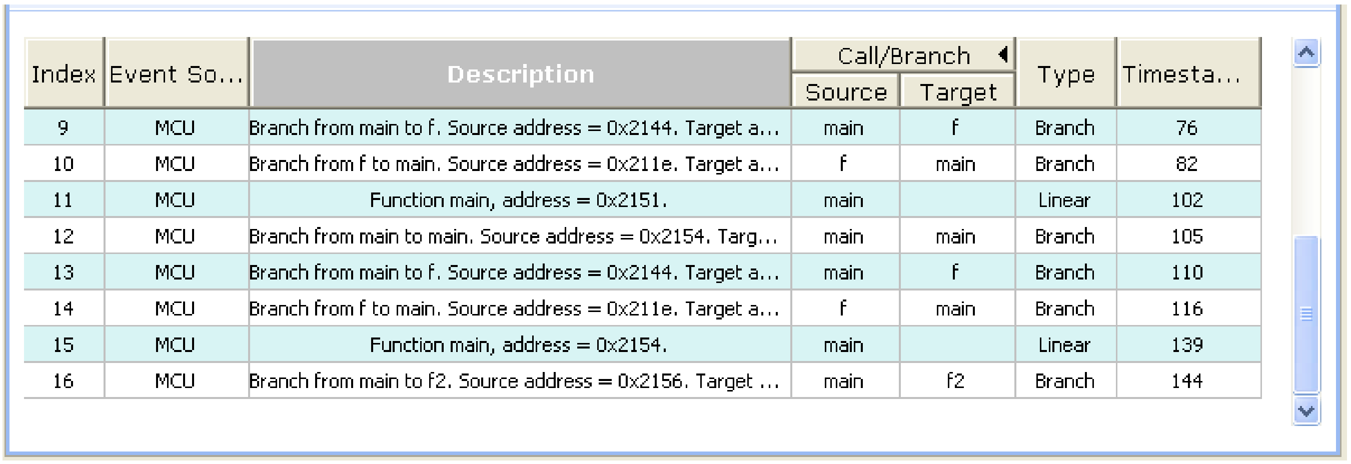
Figure 5-18. Analysispoints View

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 to view the trace results.

The figure below shows the data files generated by the application after setting trigger A in the **Automatically** mode on memory.



Index	Event So...	Description	Call/Branch		Type	Timesta...
			Source	Target		
9	MCU	Branch from main to f. Source address = 0x2144. Target a...	main	f	Branch	76
10	MCU	Branch from f to main. Source address = 0x211e. Target a...	f	main	Branch	82
11	MCU	Function main, address = 0x2151.	main		Linear	102
12	MCU	Branch from main to main. Source address = 0x2154. Targ...	main	main	Branch	105
13	MCU	Branch from main to f. Source address = 0x2144. Target a...	main	f	Branch	110
14	MCU	Branch from f to main. Source address = 0x211e. Target a...	f	main	Branch	116
15	MCU	Function main, address = 0x2154.	main		Linear	139
16	MCU	Branch from main to f2. Source address = 0x2156. Target ...	main	f2	Branch	144

Figure 5-19. 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()`.

5.2.2.3 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.

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.

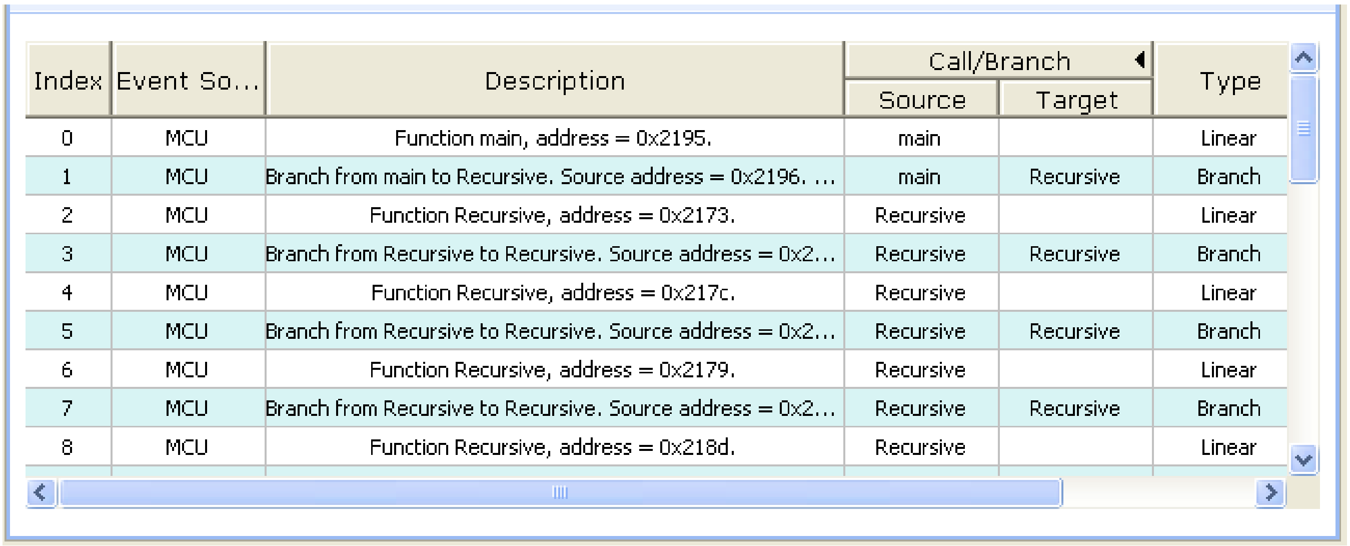
To set a trigger in LOOP1 Mode:

1. In the **CodeWarrior Projects** view, select the `Sources` folder of your project.

2. Double-click 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: Source code 4
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 the marker bar corresponding to the statement, `Recursive(3);` in the `main()` function.
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 Program Trace** option in the **Trace Mode Options** group.
11. Select the **Automatically** option.
12. Select the **LOOP1 Mode** check box.
13. Select **Collect Trace From Trigger** in the **Trace Start/Stop Conditions** group.
14. Check the **Break on FIFO Full** checkbox.
15. Select **Instruction at Address A is Executed** from the **Trigger Type** drop-down list.
16. Click **Apply** to save the settings.
17. Click **Debug** to debug the application.
18. Click **Resume**.

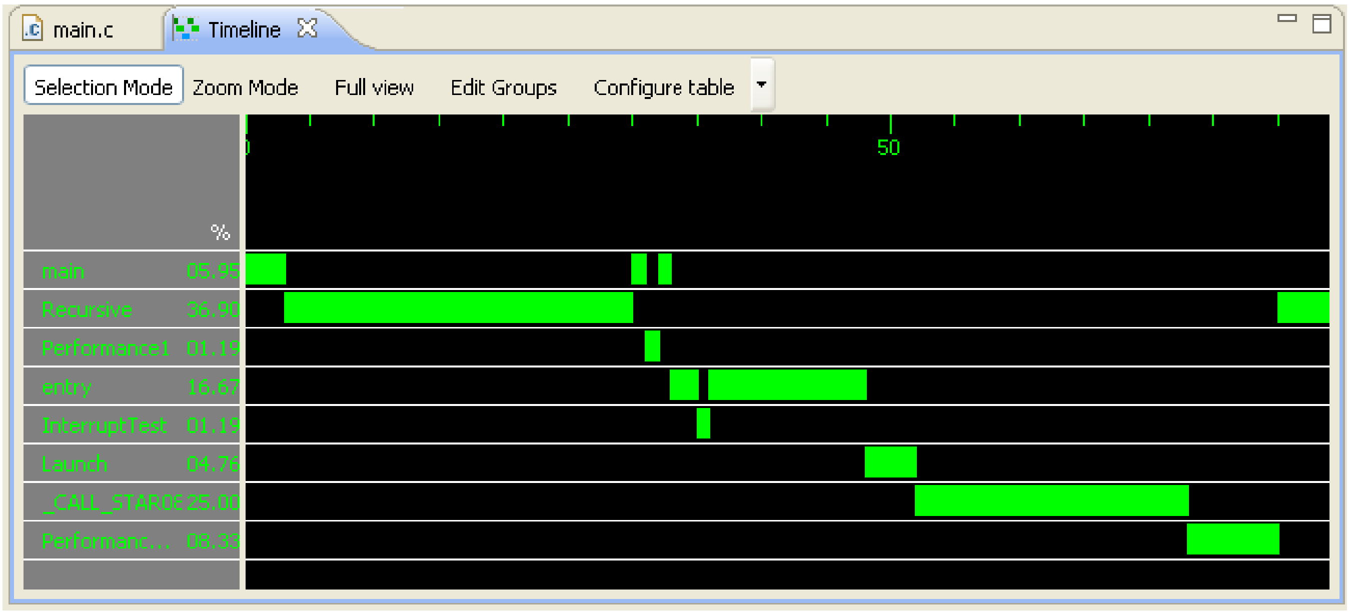
    The application stops automatically and collects data in data file.

19. Open the **Trace Data** viewer following the steps explained in the Viewing Data topic to see the trace results. The following figures show the data files generated by the application after setting trigger A in the LOOP1 Mode.



Index	Event So...	Description	Call/Branch		Type
			Source	Target	
0	MCU	Function main, address = 0x2195.	main		Linear
1	MCU	Branch from main to Recursive. Source address = 0x2196. ...	main	Recursive	Branch
2	MCU	Function Recursive, address = 0x2173.	Recursive		Linear
3	MCU	Branch from Recursive to Recursive. Source address = 0x2...	Recursive	Recursive	Branch
4	MCU	Function Recursive, address = 0x217c.	Recursive		Linear
5	MCU	Branch from Recursive to Recursive. Source address = 0x2...	Recursive	Recursive	Branch
6	MCU	Function Recursive, address = 0x2179.	Recursive		Linear
7	MCU	Branch from Recursive to Recursive. Source address = 0x2...	Recursive	Recursive	Branch
8	MCU	Function Recursive, address = 0x218d.	Recursive		Linear

Figure 5-20. Trace Data After Setting Trigger A in LOOP1 Mode - Automatically



	%
main	05.95
Recursive	36.90
Performance1	01.19
entry	16.67
InterruptTest	01.19
Launch	04.76
_CALL_STAROE	25.00
Performanc...	08.33

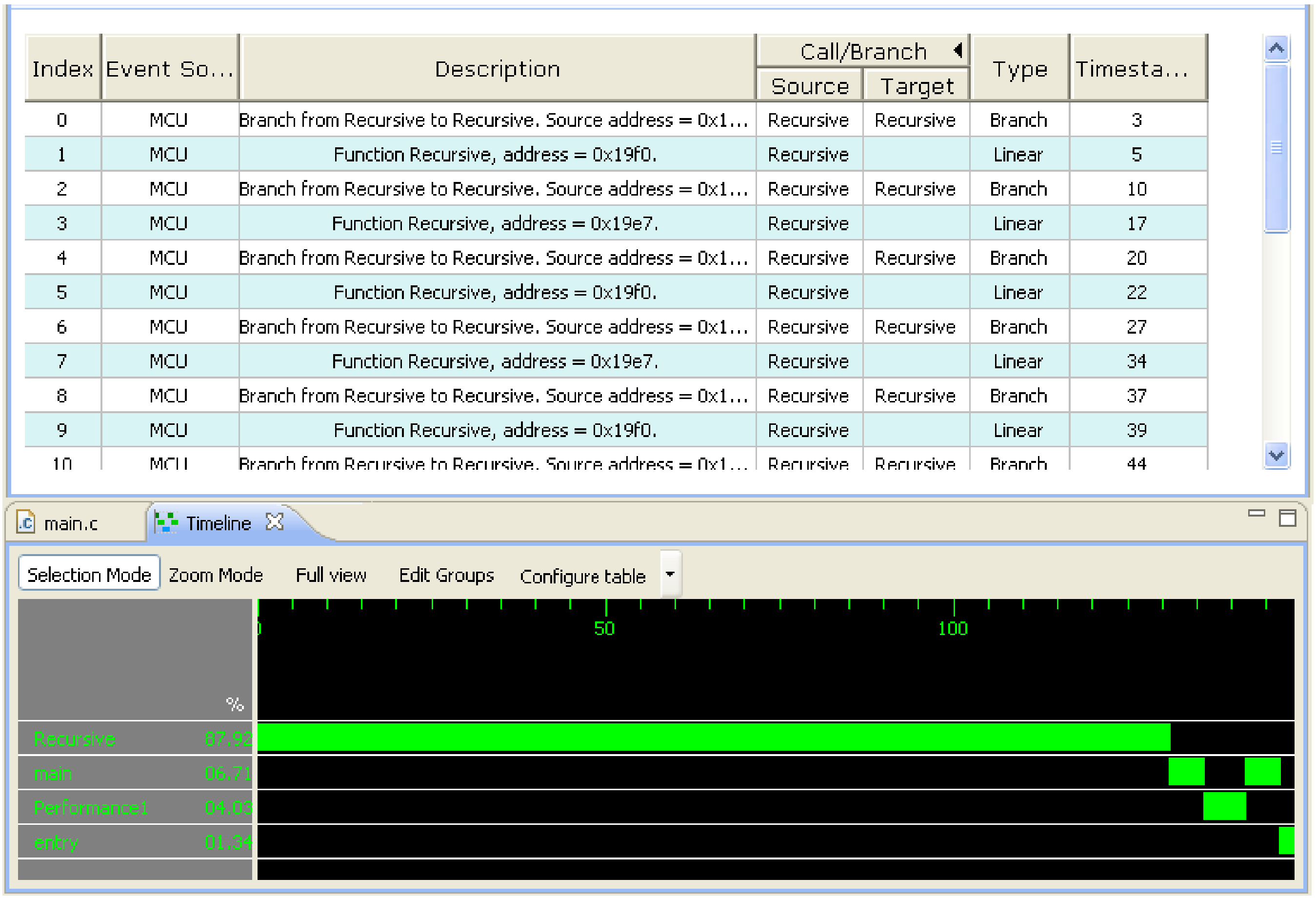
Figure 5-21. Timeline Data After Setting Trigger A in LOOP1 Mode - Automatically

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. The figure below shows the trace and timeline 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)`.



Index	Event So...	Description	Call/Branch		Type	Timesta...
			Source	Target		
0	MCU	Branch from Recursive to Recursive. Source address = 0x1...	Recursive	Recursive	Branch	3
1	MCU	Function Recursive, address = 0x19f0.	Recursive		Linear	5
2	MCU	Branch from Recursive to Recursive. Source address = 0x1...	Recursive	Recursive	Branch	10
3	MCU	Function Recursive, address = 0x19e7.	Recursive		Linear	17
4	MCU	Branch from Recursive to Recursive. Source address = 0x1...	Recursive	Recursive	Branch	20
5	MCU	Function Recursive, address = 0x19f0.	Recursive		Linear	22
6	MCU	Branch from Recursive to Recursive. Source address = 0x1...	Recursive	Recursive	Branch	27
7	MCU	Function Recursive, address = 0x19e7.	Recursive		Linear	34
8	MCU	Branch from Recursive to Recursive. Source address = 0x1...	Recursive	Recursive	Branch	37
9	MCU	Function Recursive, address = 0x19f0.	Recursive		Linear	39
10	MCU	Branch from Recursive to Recursive. Source address = 0x1...	Recursive	Recursive	Branch	44

Recursive	87.92
main	06.71
Performance1	04.03
entry	01.34

Figure 5-22. Trace Data After Setting Trigger A in Normal Mode - Automatically

5.2.2.4 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.

NOTE

For the MC9S08PT60 target specifically, the **Instruction Outside Range from Address A to Address B is Executed** trigger will hit when any instruction outside the range between trigger address A and trigger address B matches with the data on the bus or program address outside the range.

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 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: Source code 2.
3. Remove the entries of `Performance1();` from the `main();` function. The `main()` function now should look as shown below:
   Listing: main() function of source code 2

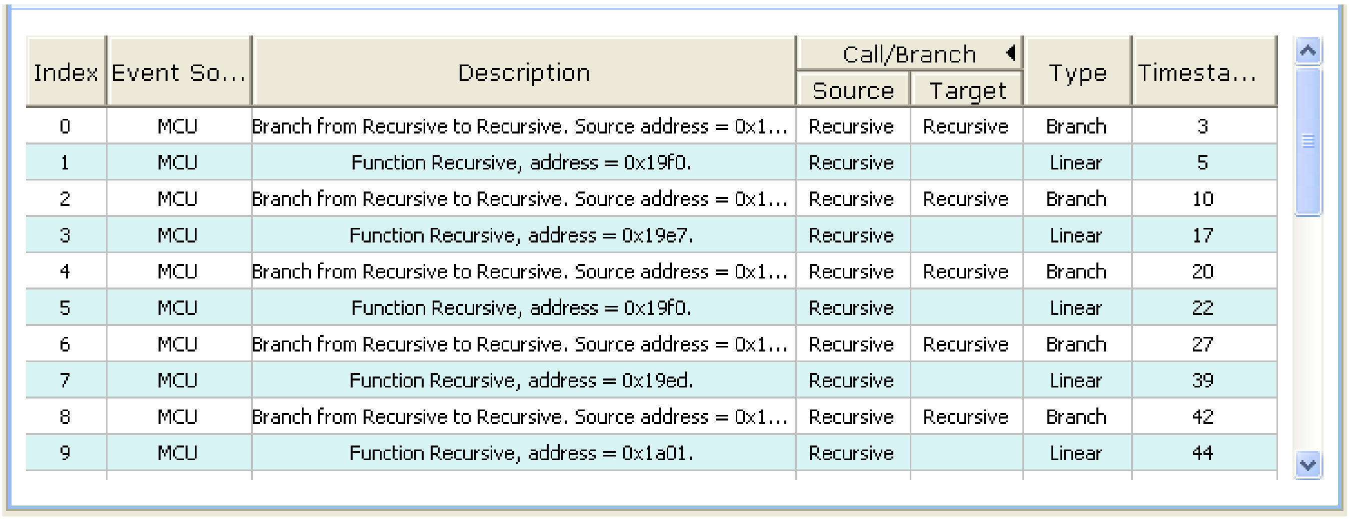
```
void main(void)
{
  EnableInterrupts; /* enable interrupts */
      /* include your code here */
  Recursive(2);
  for(;;)
  {
     entry();
     __RESET_WATCHDOG(); /* feeds the dog */
  } /* loop forever */
}
```

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 below.
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.

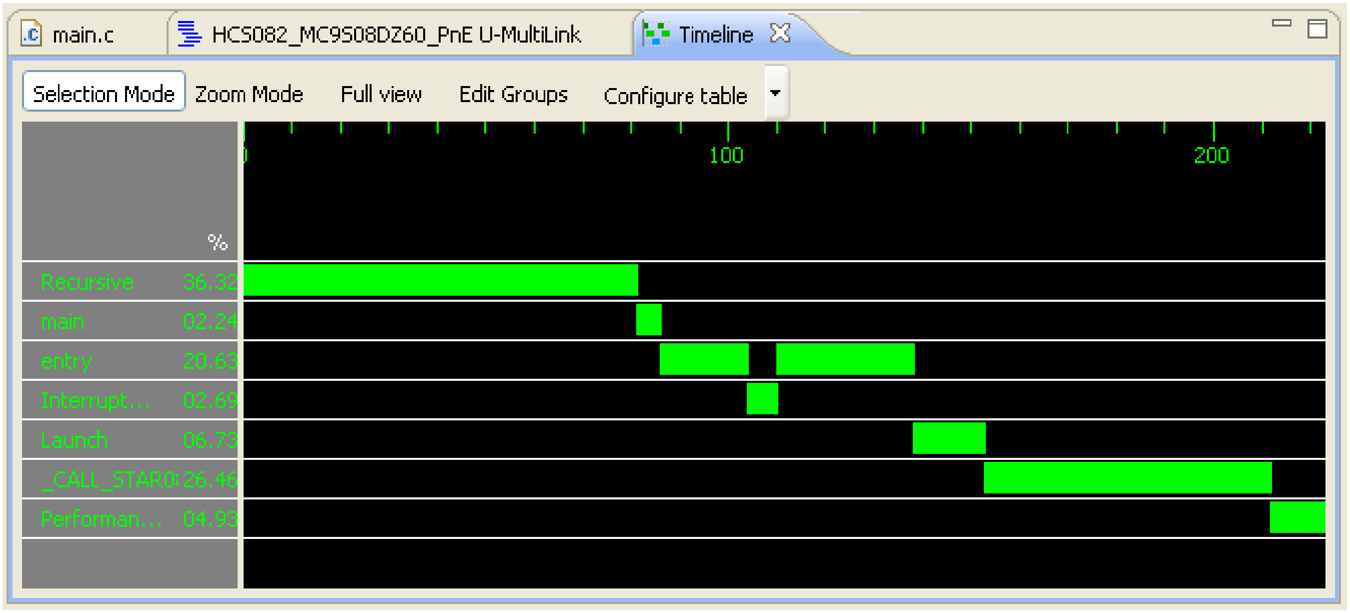
20. Open the **Trace Data** viewer following the steps explained in the Viewing Data topic to view the collected data.

The following figures show the trace and timeline data 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.



Index	Event So...	Description	Call/Branch		Type	Timesta...
			Source	Target		
0	MCU	Branch from Recursive to Recursive. Source address = 0x1...	Recursive	Recursive	Branch	3
1	MCU	Function Recursive, address = 0x19f0.	Recursive		Linear	5
2	MCU	Branch from Recursive to Recursive. Source address = 0x1...	Recursive	Recursive	Branch	10
3	MCU	Function Recursive, address = 0x19e7.	Recursive		Linear	17
4	MCU	Branch from Recursive to Recursive. Source address = 0x1...	Recursive	Recursive	Branch	20
5	MCU	Function Recursive, address = 0x19f0.	Recursive		Linear	22
6	MCU	Branch from Recursive to Recursive. Source address = 0x1...	Recursive	Recursive	Branch	27
7	MCU	Function Recursive, address = 0x19ed.	Recursive		Linear	39
8	MCU	Branch from Recursive to Recursive. Source address = 0x1...	Recursive	Recursive	Branch	42
9	MCU	Function Recursive, address = 0x1a01.	Recursive		Linear	44

Figure 5-23. Trace Data - Instruction Outside Range



	%
Recursive	36.32
main	02.24
entry	20.63
Interrupt...	02.69
Launch	06.73
_CALL_STAR0	26.46
Performan...	04.93

Figure 5-24. Timeline - Instruction Outside Range

5.2.3 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.

5.2.3.1 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 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 below.
   Listing: 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.
13. Select the **Add Global Variables** option from the context menu.

    The **Add Globals** dialog box appears.

14. Select `iteration` from the list of available variables.
15. Click **OK**.

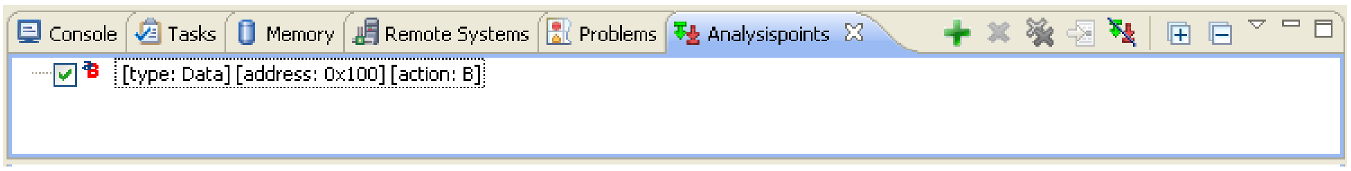
    The entry for the `iteration` variable gets added to the **Variables** view.

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

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

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

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



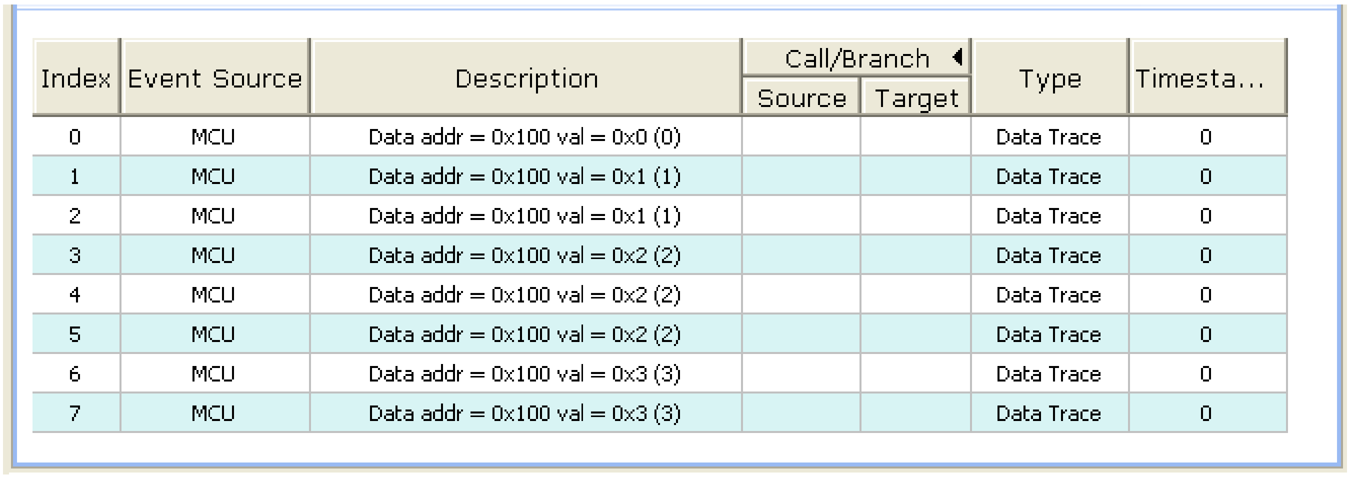
[type: Data] [address: 0x100] [action: B]

Figure 5-25. Analysispoints 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 topic to view the trace results. The figure below 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*.



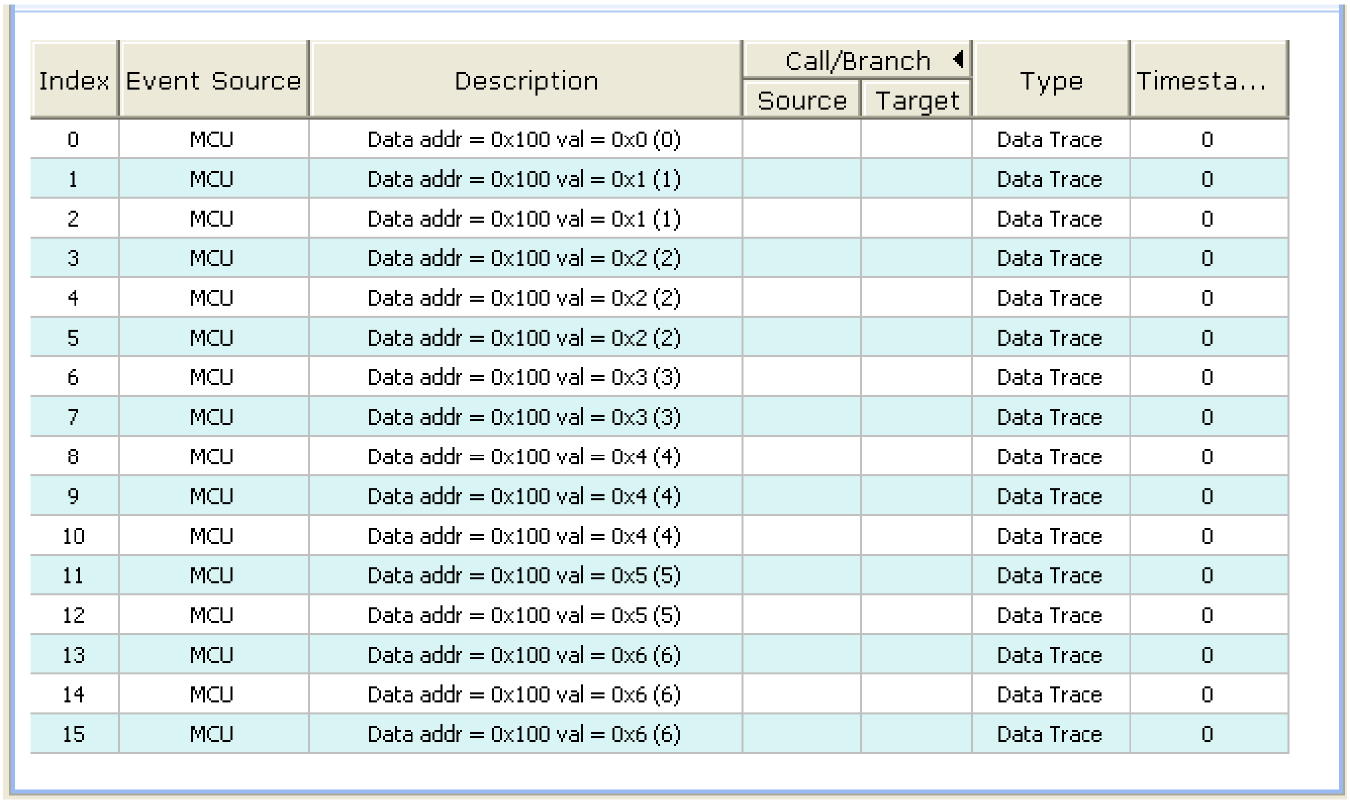
Index	Event Source	Description	Call/Branch		Type	Timesta...
			Source	Target		
0	MCU	Data addr = 0x100 val = 0x0 (0)			Data Trace	0
1	MCU	Data addr = 0x100 val = 0x1 (1)			Data Trace	0
2	MCU	Data addr = 0x100 val = 0x1 (1)			Data Trace	0
3	MCU	Data addr = 0x100 val = 0x2 (2)			Data Trace	0
4	MCU	Data addr = 0x100 val = 0x2 (2)			Data Trace	0
5	MCU	Data addr = 0x100 val = 0x2 (2)			Data Trace	0
6	MCU	Data addr = 0x100 val = 0x3 (3)			Data Trace	0
7	MCU	Data addr = 0x100 val = 0x3 (3)			Data Trace	0

Figure 5-26. 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.



Index	Event Source	Description	Call/Branch		Type	Timesta...
			Source	Target		
0	MCU	Data addr = 0x100 val = 0x0 (0)			Data Trace	0
1	MCU	Data addr = 0x100 val = 0x1 (1)			Data Trace	0
2	MCU	Data addr = 0x100 val = 0x1 (1)			Data Trace	0
3	MCU	Data addr = 0x100 val = 0x2 (2)			Data Trace	0
4	MCU	Data addr = 0x100 val = 0x2 (2)			Data Trace	0
5	MCU	Data addr = 0x100 val = 0x2 (2)			Data Trace	0
6	MCU	Data addr = 0x100 val = 0x3 (3)			Data Trace	0
7	MCU	Data addr = 0x100 val = 0x3 (3)			Data Trace	0
8	MCU	Data addr = 0x100 val = 0x4 (4)			Data Trace	0
9	MCU	Data addr = 0x100 val = 0x4 (4)			Data Trace	0
10	MCU	Data addr = 0x100 val = 0x4 (4)			Data Trace	0
11	MCU	Data addr = 0x100 val = 0x5 (5)			Data Trace	0
12	MCU	Data addr = 0x100 val = 0x5 (5)			Data Trace	0
13	MCU	Data addr = 0x100 val = 0x6 (6)			Data Trace	0
14	MCU	Data addr = 0x100 val = 0x6 (6)			Data Trace	0
15	MCU	Data addr = 0x100 val = 0x6 (6)			Data Trace	0

Figure 5-27. Trace Data Viewer - New Data Appended

This is how you set trigger B in the **Collect Data Trace** mode and collect trace using the **Capture Read/Write Values at Address B** trigger type.

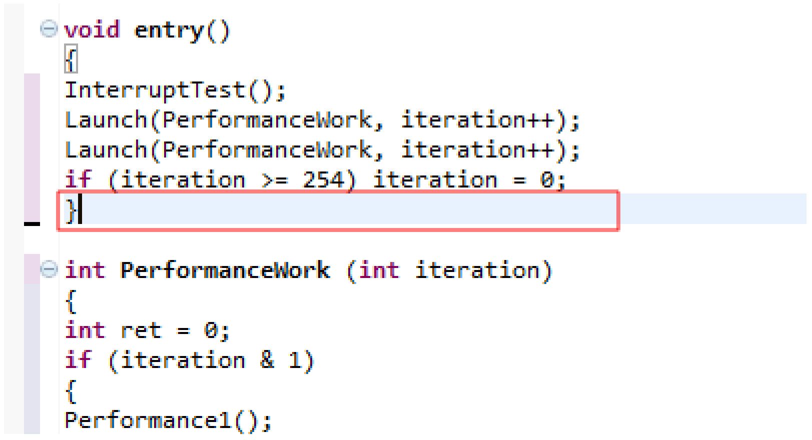
NOTE

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.

5.2.3.2 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 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: Source code 4.
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 the marker bar corresponding to the statement as highlighted in the figure below.

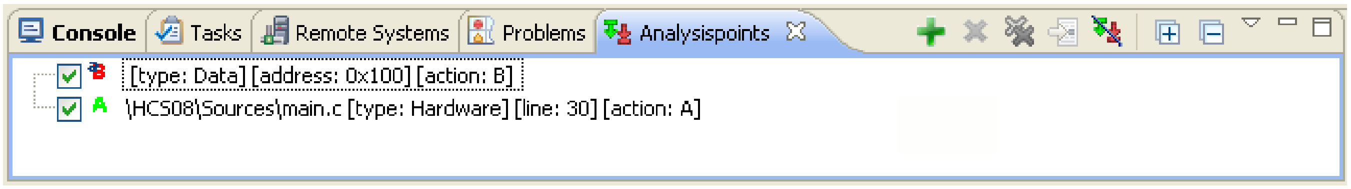


```
void entry()
{
InterruptTest();
Launch(PerformanceWork, iteration++);
Launch(PerformanceWork, iteration++);
if (iteration >= 254) iteration = 0;
}

int PerformanceWork (int iteration)
{
int ret = 0;
if (iteration & 1)
{
Performance1();
```

Figure 5-28. 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 Capture Read/Write Values at Address B topic. After setting triggers, A and B, the **Analysispoints** view contains the following entries.



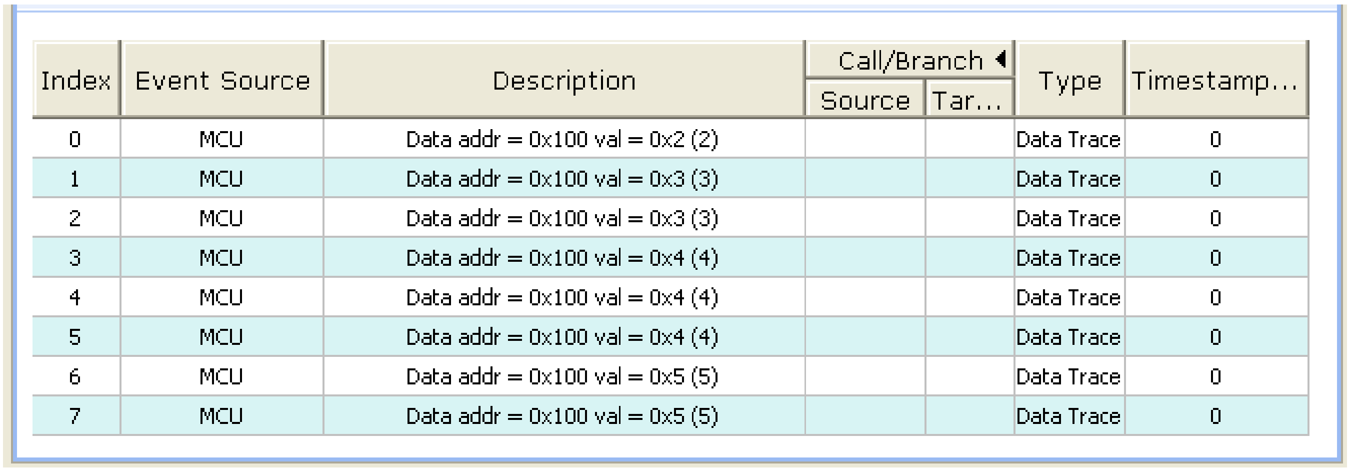
B [type: Data] [address: 0x100] [action: B]
A \HCS08\Sources\main.c [type: Hardware] [line: 30] [action: A]

Figure 5-29. Analysispoints 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 to view the trace results. The figure below 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.



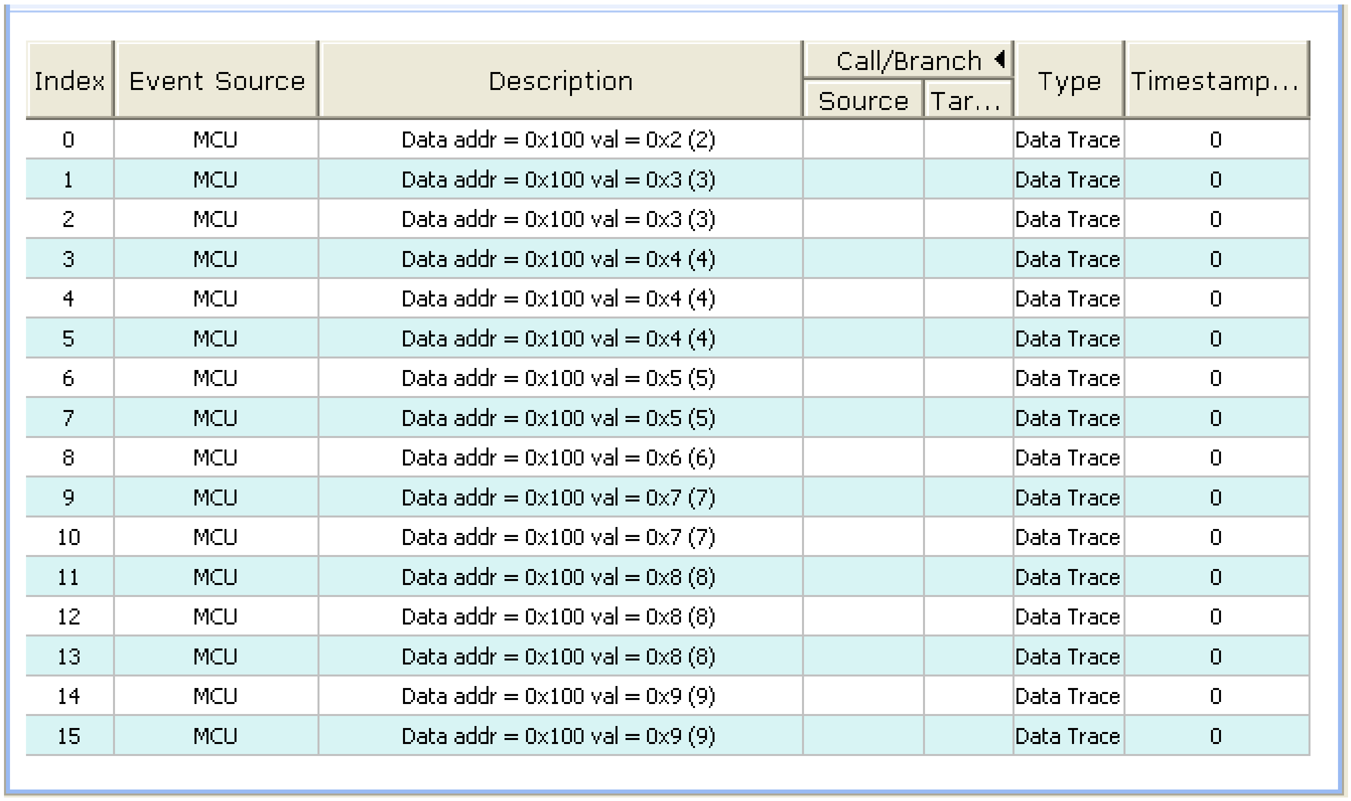
Index	Event Source	Description	Call/Branch		Type	Timestamp...
			Source	Tar...		
0	MCU	Data addr = 0x100 val = 0x2 (2)			Data Trace	0
1	MCU	Data addr = 0x100 val = 0x3 (3)			Data Trace	0
2	MCU	Data addr = 0x100 val = 0x3 (3)			Data Trace	0
3	MCU	Data addr = 0x100 val = 0x4 (4)			Data Trace	0
4	MCU	Data addr = 0x100 val = 0x4 (4)			Data Trace	0
5	MCU	Data addr = 0x100 val = 0x4 (4)			Data Trace	0
6	MCU	Data addr = 0x100 val = 0x5 (5)			Data Trace	0
7	MCU	Data addr = 0x100 val = 0x5 (5)			Data Trace	0

Figure 5-30. 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.



Index	Event Source	Description	Call/Branch		Type	Timestamp...
			Source	Tar...		
0	MCU	Data addr = 0x100 val = 0x2 (2)			Data Trace	0
1	MCU	Data addr = 0x100 val = 0x3 (3)			Data Trace	0
2	MCU	Data addr = 0x100 val = 0x3 (3)			Data Trace	0
3	MCU	Data addr = 0x100 val = 0x4 (4)			Data Trace	0
4	MCU	Data addr = 0x100 val = 0x4 (4)			Data Trace	0
5	MCU	Data addr = 0x100 val = 0x4 (4)			Data Trace	0
6	MCU	Data addr = 0x100 val = 0x5 (5)			Data Trace	0
7	MCU	Data addr = 0x100 val = 0x5 (5)			Data Trace	0
8	MCU	Data addr = 0x100 val = 0x6 (6)			Data Trace	0
9	MCU	Data addr = 0x100 val = 0x7 (7)			Data Trace	0
10	MCU	Data addr = 0x100 val = 0x7 (7)			Data Trace	0
11	MCU	Data addr = 0x100 val = 0x8 (8)			Data Trace	0
12	MCU	Data addr = 0x100 val = 0x8 (8)			Data Trace	0
13	MCU	Data addr = 0x100 val = 0x8 (8)			Data Trace	0
14	MCU	Data addr = 0x100 val = 0x9 (9)			Data Trace	0
15	MCU	Data addr = 0x100 val = 0x9 (9)			Data Trace	0

Figure 5-31. Trace Collected at Address B, After Access at Address A - New Data Appended

This is how you collect trace using the **Capture Read/Write Values at Address B, After Access at Address A** trigger type.

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.

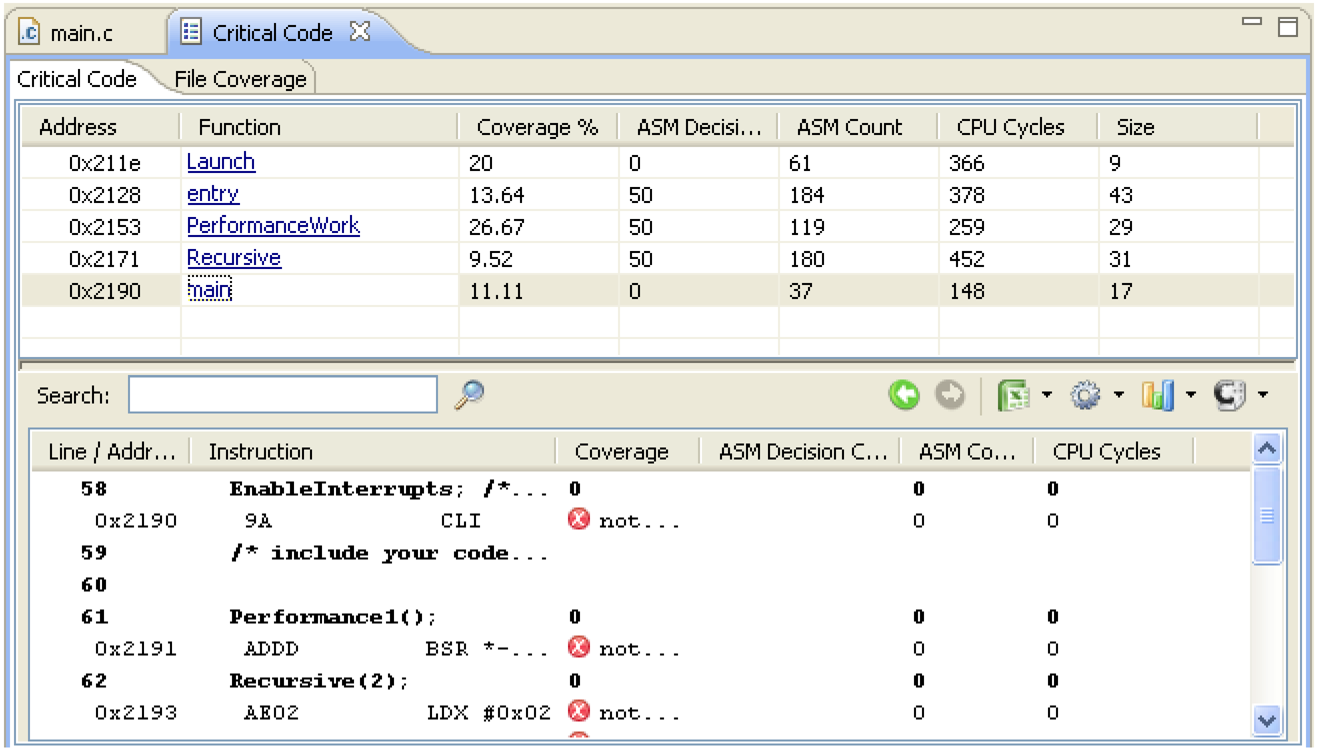
5.2.4 Collecting Trace in Profile-Only Mode

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 critical code data in the **Critical Code** viewer. To collect trace in Profile-Only mode:

1. In the **CodeWarrior Projects** view, expand the `Sources` folder of your project.
2. Double-click 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: Source code 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** viewer following the steps explained in the Viewing Data topic to view the critical code data results.



Address	Function	Coverage %	ASM Decisi...	ASM Count	CPU Cycles	Size
0x211e	Launch	20	0	61	366	9
0x2128	entry	13.64	50	184	378	43
0x2153	PerformanceWork	26.67	50	119	259	29
0x2171	Recursive	9.52	50	180	452	31
0x2190	main	11.11	0	37	148	17

Line / Addr...	Instruction	Coverage	ASM Decision C...	ASM Co...	CPU Cycles
58	EnableInterrupts; /*...	0		0	0
0x2190	9A CLI	❌ not...		0	0
59	/* include your code...				
60					
61	Performance1();	0		0	0
0x2191	ADDD BSR *-...	❌ not...		0	0
62	Recursive(2);	0		0	0
0x2193	AE02 LDX #0x02	❌ not...		0	0

Figure 5-32. HCS08 Critical Code Viewer - Profile-Only

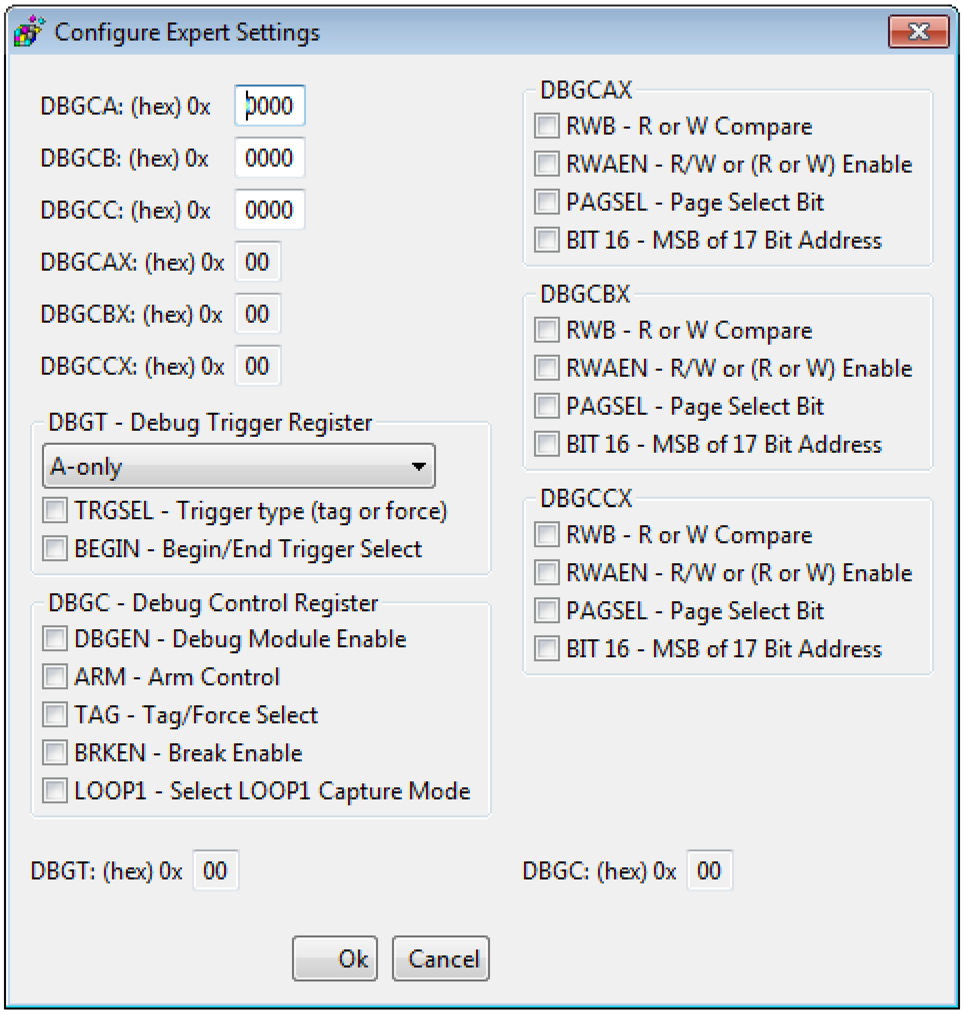
This is how you collect trace data in the Profile-Only mode of the HCS08 target.

5.2.5 Collecting Trace 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.


Configure Expert Settings

DBGCA: (hex) 0x 0000
DBGCB: (hex) 0x 0000
DBGCC: (hex) 0x 0000
DBGCAX: (hex) 0x 00
DBGCBX: (hex) 0x 00
DBGCCX: (hex) 0x 00

DBGT - Debug Trigger Register
A-only
☐ TRGSEL - Trigger type (tag or force)
☐ BEGIN - Begin/End Trigger Select

DBGC - Debug Control Register
☐ DBGEN - Debug Module Enable
☐ ARM - Arm Control
☐ TAG - Tag/Force Select
☐ BRKEN - Break Enable
☐ LOOP1 - Select LOOP1 Capture Mode

DBGCAX
☐ RWB - R or W Compare
☐ RWAEN - R/W or (R or W) Enable
☐ PAGSEL - Page Select Bit
☐ BIT 16 - MSB of 17 Bit Address

DBGCBX
☐ RWB - R or W Compare
☐ RWAEN - R/W or (R or W) Enable
☐ PAGSEL - Page Select Bit
☐ BIT 16 - MSB of 17 Bit Address

DBGCCX
☐ RWB - R or W Compare
☐ RWAEN - R/W or (R or W) Enable
☐ PAGSEL - Page Select Bit
☐ BIT 16 - MSB of 17 Bit Address

DBGT: (hex) 0x 00
DBGC: (hex) 0x 00

Figure 5-33. Configure Expert Settings Dialog Box

5. Specify the settings according to requirements.

NOTE

The register layout in the **Configure Expert Settings** dialog box may vary depending on the debug version target. For example, the register layout of the DbgVer 3 target might be different from the register layout of the DbgVer 2 target.

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

NOTE

In **Expert** mode, only Trace and Timeline data is collected.

This is how you collect trace data in the **Expert** mode of the HCS08 target.

NOTE

For HCS08 PT16 and PA4 families (PT/PL/PA16, PT/PL/PA8, PA4, PA2), the trace triggers set on the **Variables** or **Memory** views, with **Memory Access** set to “Write” or “Read/Write”, will not work correctly due to a hardware issue of the DBG module when writing to SRAM memory. When using these type of triggers:

- Data Trace will not work as expected. Only the read accesses will be traced or the hardware trace will be triggered.
- Program trace start/stop conditions involving data bus match/mismatch will not work. The hardware trace will not be triggered.
- Advanced developers using the **Configuration set in User code** option or the **Expert** trace option must avoid using trigger conditions (writing trigger registers manually) which involve tracing or triggering on write accesses, such as “Event B only”, “A then Event only B”, “A and (not) B” hardware trigger modes.

This is applicable for some early revisions of the PT60 family (PT/PL/PA60 and PT/PL/PA32) also.

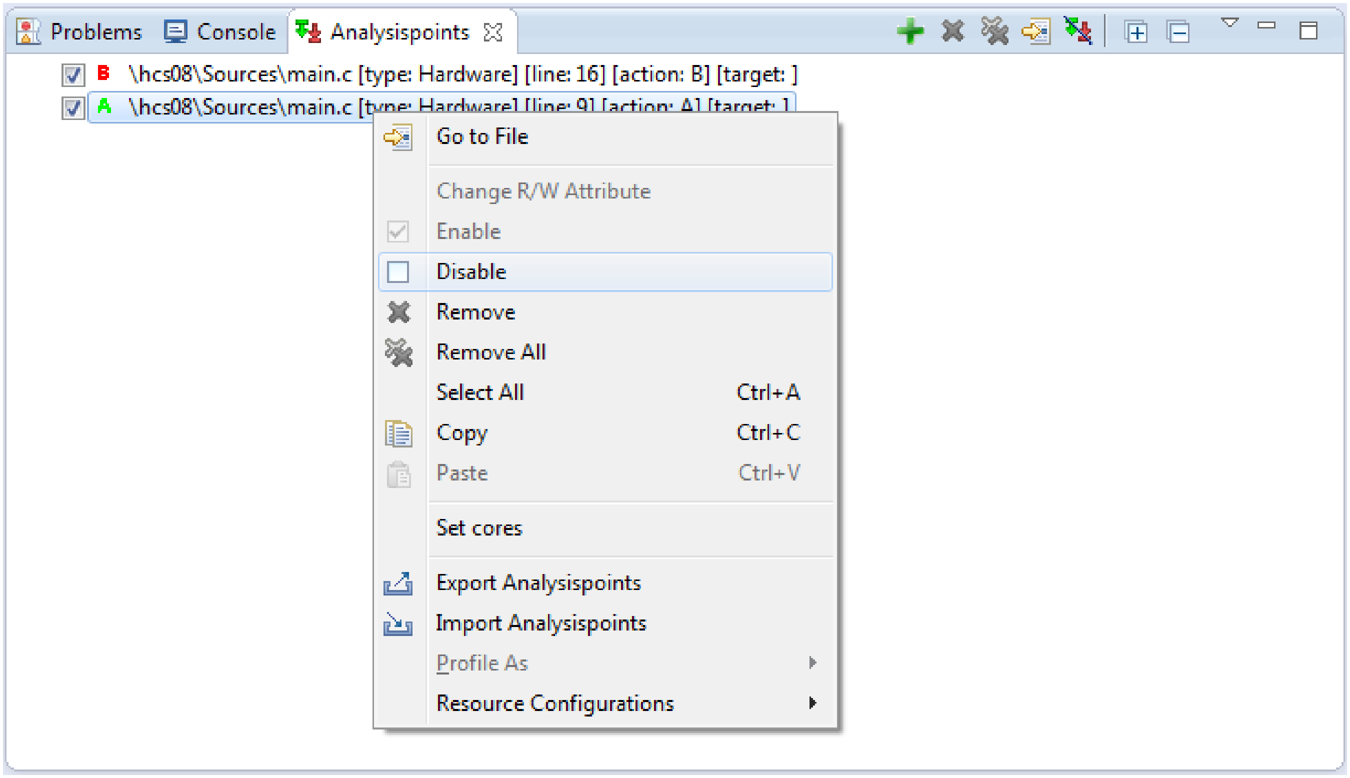
5.3 Enabling and Disabling the Tracepoints

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

If you want to disable the tracepoints, right-click the marker bar where triggers are already set and enabled, select the **Disable Tracepoint** 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 **Analysispoints** view. Select **Window > Show View > Other > Software Analysis > Analysispoints** to open the **Analysispoints** view.

Right-click the selected attribute and select **Disable/Enable** option. The unchecked attribute indicates the disabled tracepoint.

You can also use the **Ignore all Analysispoints** button to disable all the tracepoints without manually selecting them in the **Analysispoints** view. You can click **Ignore All** again to enable the tracepoints.



B \hcs08\Sources\main.c [type: Hardware] [line: 16] [action: B] [target:]
A \hcs08\Sources\main.c [type: Hardware] [line: 9] [action: A] [target:]

Figure 5-34. Disabling/Enabling the Trigger from Analysispoints View

For detailed information on the **Analysispoints** view, refer Viewing Tracepoints.

Chapter 6
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:

6.1 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.

Warning

Do not use breakpoints with triggers when collecting trace on the ColdFire V1 target. This is because there is only one hardware debug module on ColdFire V1 which is shared for setting hardware breakpoints or trace triggers. So you can either set breakpoint or trace trigger using this debug module.

6.2 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

6.2.1 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.

6.2.1.1 Trace From Trigger A Onward

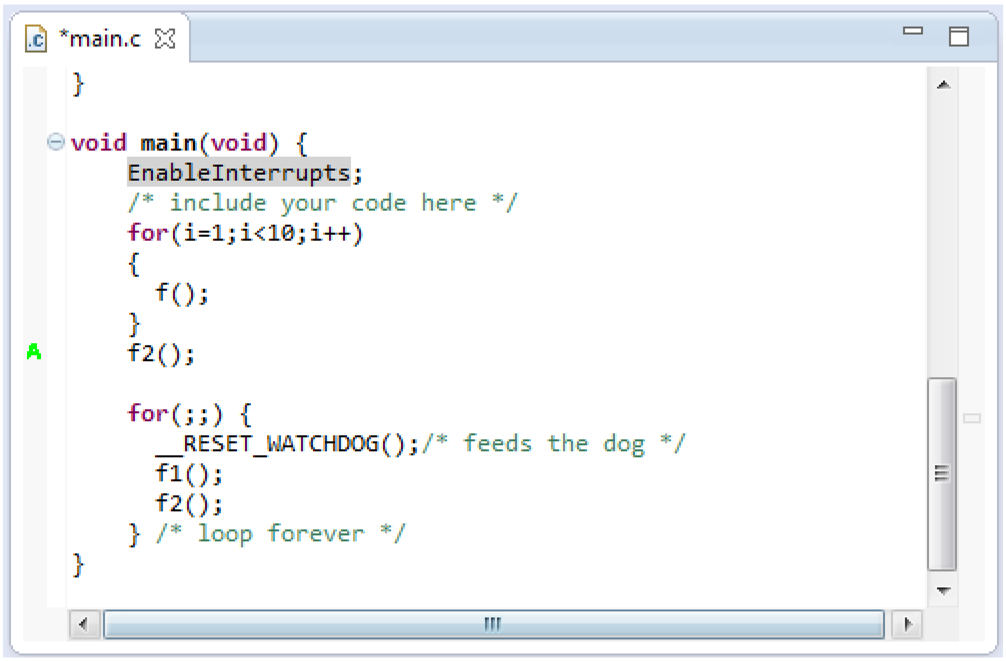
To set trigger A in the editor area for the ColdFire V1 target:

1. In the **CodeWarrior Projects** view, select the `Sources` folder of your project.

2. Double-click 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 below.
   Listing: 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();` (before the `for (;;)` statement).
8. Right-click 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. The same option is also used to remove trigger A from the marker bar.



```
}

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 */
}
```

Figure 6-1. Trigger A Set in Editor Area

NOTE

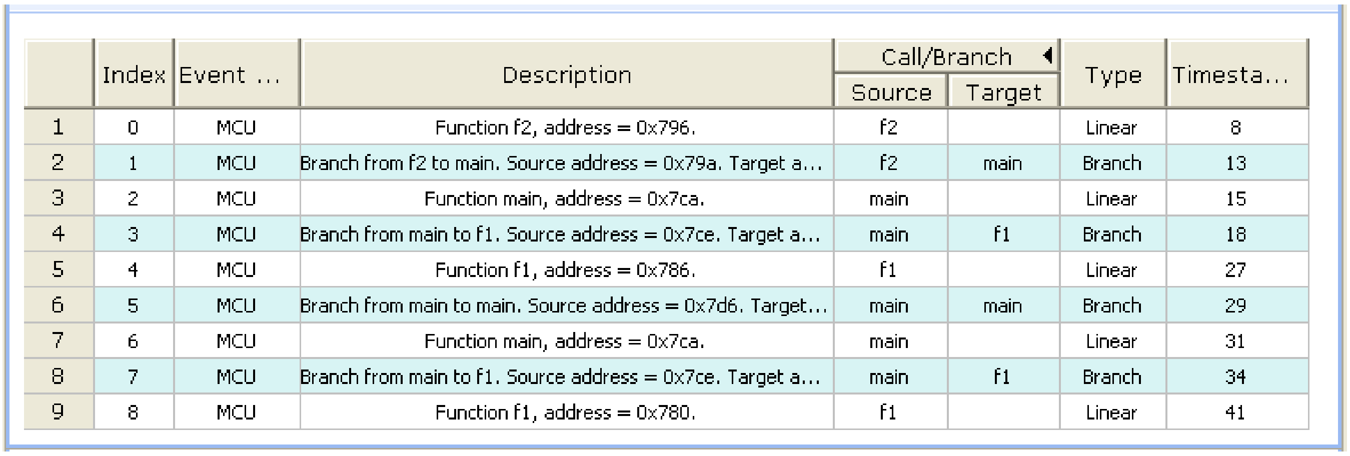
Do not set tracepoints on the statements containing only comments, brackets, and variable declaration with no value. If tracepoints are set on invalid statements, they are automatically disabled when the application is debugged.

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.
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 to view the trace results.

    The figure below 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 **Description** column 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. The following figure shows the timeline data.



	Index	Event ...	Description	Call/Branch Source	Call/Branch Target	Type	Timesta...
1	0	MCU	Function f2, address = 0x796.	f2		Linear	8
2	1	MCU	Branch from f2 to main. Source address = 0x79a. Target a...	f2	main	Branch	13
3	2	MCU	Function main, address = 0x7ca.	main		Linear	15
4	3	MCU	Branch from main to f1. Source address = 0x7ce. Target a...	main	f1	Branch	18
5	4	MCU	Function f1, address = 0x786.	f1		Linear	27
6	5	MCU	Branch from main to main. Source address = 0x7d6. Target...	main	main	Branch	29
7	6	MCU	Function main, address = 0x7ca.	main		Linear	31
8	7	MCU	Branch from main to f1. Source address = 0x7ce. Target a...	main	f1	Branch	34
9	8	MCU	Function f1, address = 0x780.	f1		Linear	41

Figure 6-2. Trace Data After Setting *Trace From Trigger A Onward* in Continuous Mode

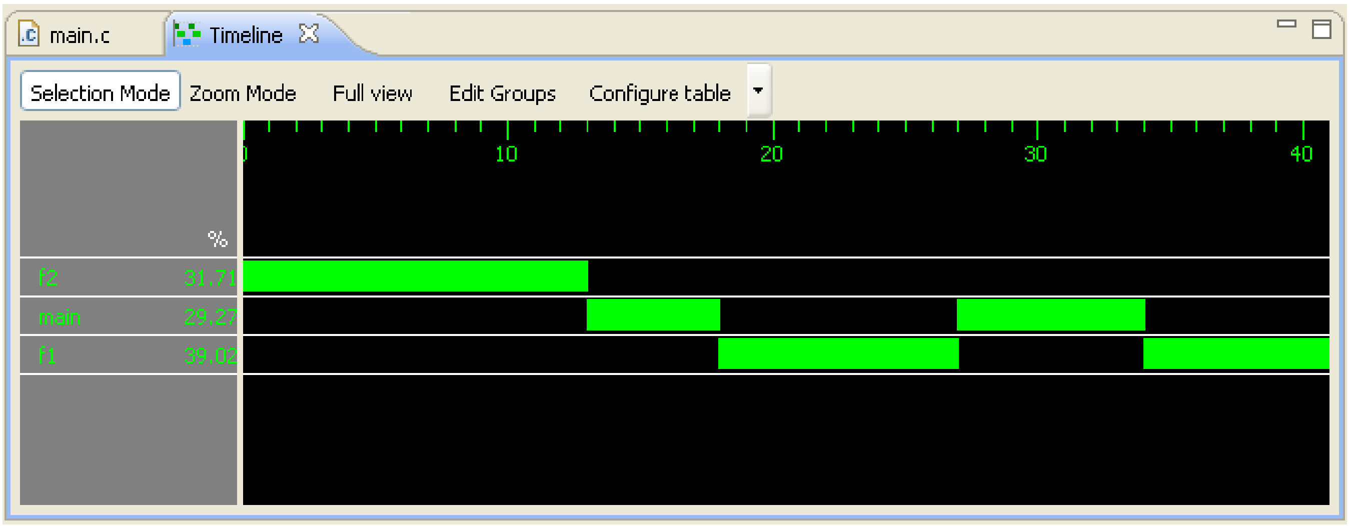


Figure 6-3. Timeline Data After Setting *Trace From Trigger A Onward* in Continuous Mode

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

This is how you set the **Trace From Trigger A Onward** trace conditions in the Continuous mode of the ColdFire V1 target and collect trace data.

6.2.1.2 Trace From Trigger A to Trigger B

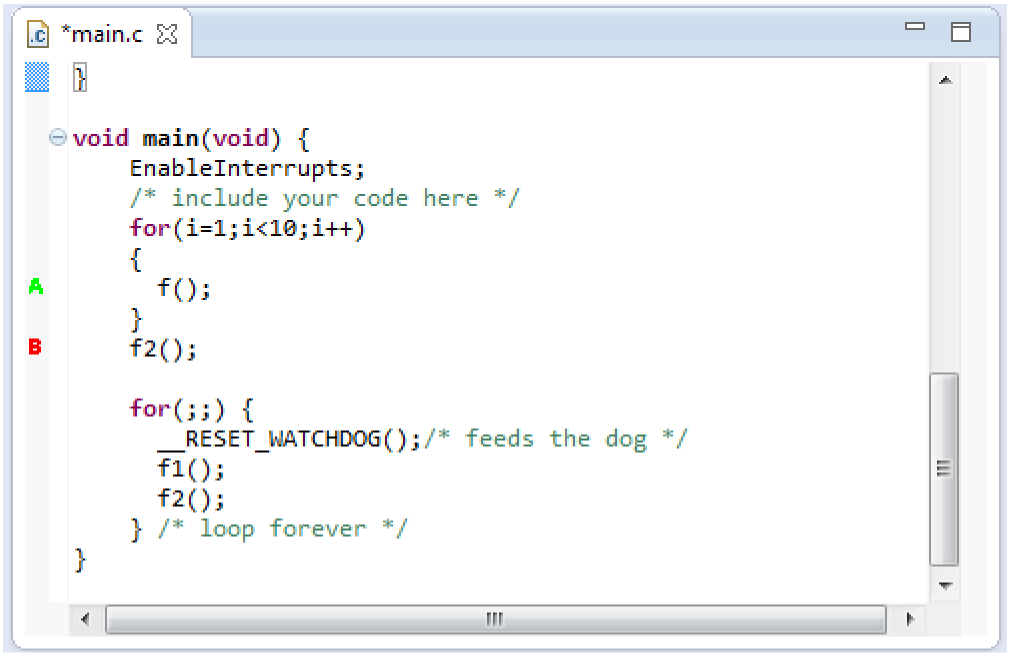
To set trigger A and trigger B in the editor area in the Continuous mode:

1. In the **CodeWarrior Projects** view, select the `Sources` folder of your project.
2. Double-click 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: Source code for trace collection.

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();`.
8. Right-click the marker bar, select the **Trace Triggers > Toggle Trace Trigger A** option from the context menu.
9. Right-click 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 as shown in figure below. .

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



```
}

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 */
}
```

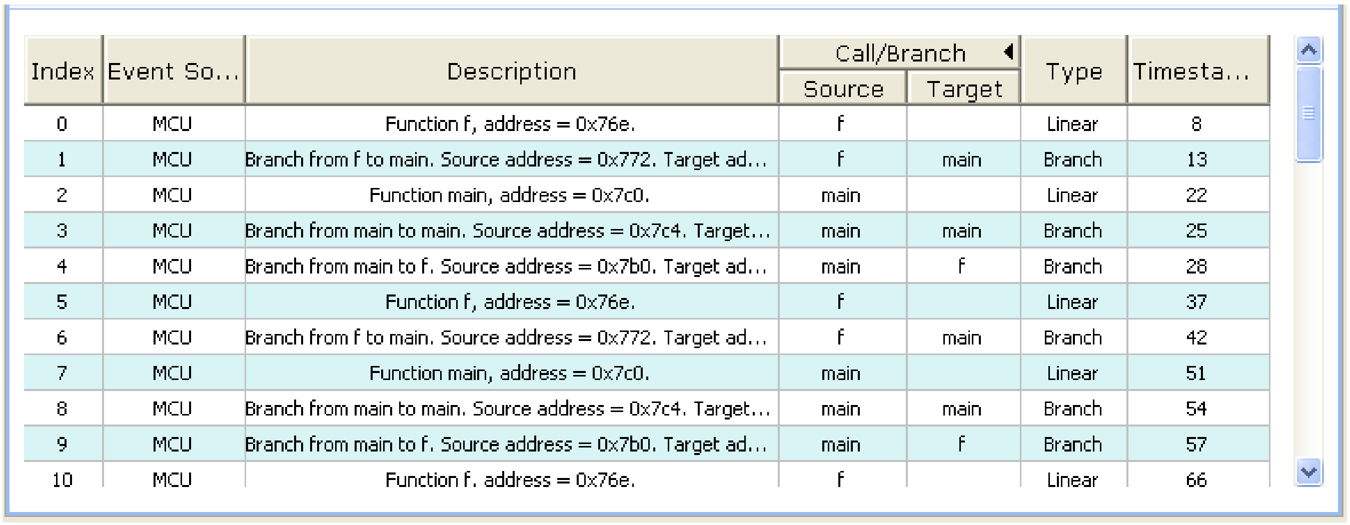
Figure 6-4. Setting Triggers A and B

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 to view the trace results.

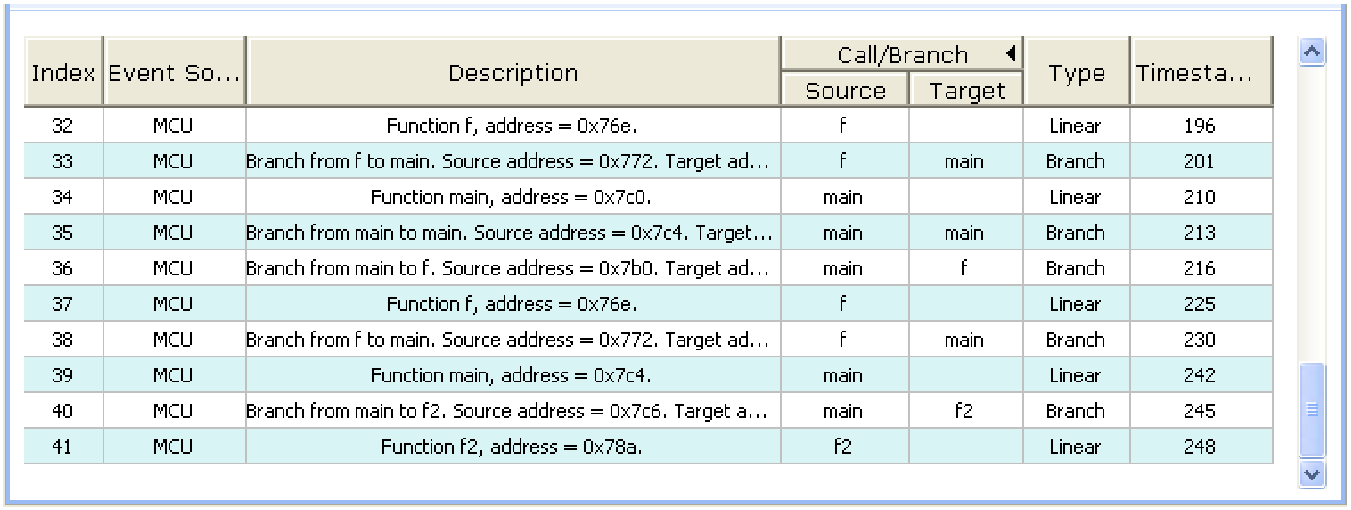
    The figure below shows the data files and the timeline graph that is generated by the application in which the data has been collected after setting triggers, A and B. The **Trace Data** viewer in the figure shows that trace starts collecting from the `f()` function where you set trigger A.



Index	Event So...	Description	Call/Branch		Type	Timesta...
			Source	Target		
0	MCU	Function f, address = 0x76e.	f		Linear	8
1	MCU	Branch from f to main. Source address = 0x772. Target ad...	f	main	Branch	13
2	MCU	Function main, address = 0x7c0.	main		Linear	22
3	MCU	Branch from main to main. Source address = 0x7c4. Target...	main	main	Branch	25
4	MCU	Branch from main to f. Source address = 0x7b0. Target ad...	main	f	Branch	28
5	MCU	Function f, address = 0x76e.	f		Linear	37
6	MCU	Branch from f to main. Source address = 0x772. Target ad...	f	main	Branch	42
7	MCU	Function main, address = 0x7c0.	main		Linear	51
8	MCU	Branch from main to main. Source address = 0x7c4. Target...	main	main	Branch	54
9	MCU	Branch from main to f. Source address = 0x7b0. Target ad...	main	f	Branch	57
10	MCU	Function f. address = 0x76e.	f		Linear	66

Figure 6-5. Trace Data View After Setting *Trace From Trigger A to Trigger B* in Continuous Mode - Begin

The figure below shows that trace stops at the `f2()` function where you set trigger B.



Index	Event So...	Description	Call/Branch		Type	Timesta...
			Source	Target		
32	MCU	Function f, address = 0x76e.	f		Linear	196
33	MCU	Branch from f to main. Source address = 0x772. Target ad...	f	main	Branch	201
34	MCU	Function main, address = 0x7c0.	main		Linear	210
35	MCU	Branch from main to main. Source address = 0x7c4. Target...	main	main	Branch	213
36	MCU	Branch from main to f. Source address = 0x7b0. Target ad...	main	f	Branch	216
37	MCU	Function f, address = 0x76e.	f		Linear	225
38	MCU	Branch from f to main. Source address = 0x772. Target ad...	f	main	Branch	230
39	MCU	Function main, address = 0x7c4.	main		Linear	242
40	MCU	Branch from main to f2. Source address = 0x7c6. Target a...	main	f2	Branch	245
41	MCU	Function f2, address = 0x78a.	f2		Linear	248

Figure 6-6. 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.

The graph in figure below 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.

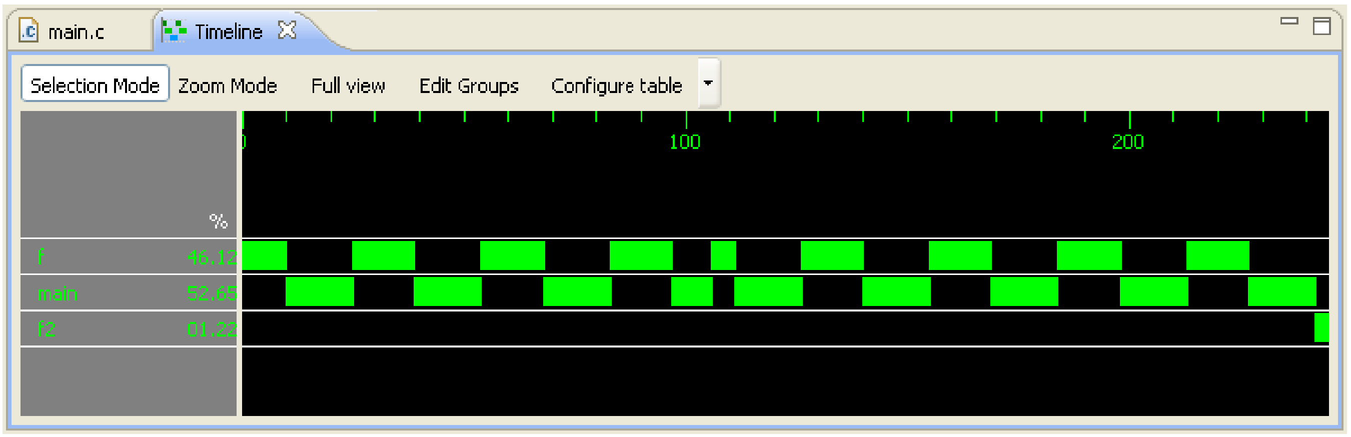


Figure 6-7. Graph Displaying Timeline of Trace Data

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

This is how you set the **Trace From Trigger A to Trigger B** trace conditions in the Continuous mode of the ColdFire V1 target and collect trace data.

6.2.2 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:

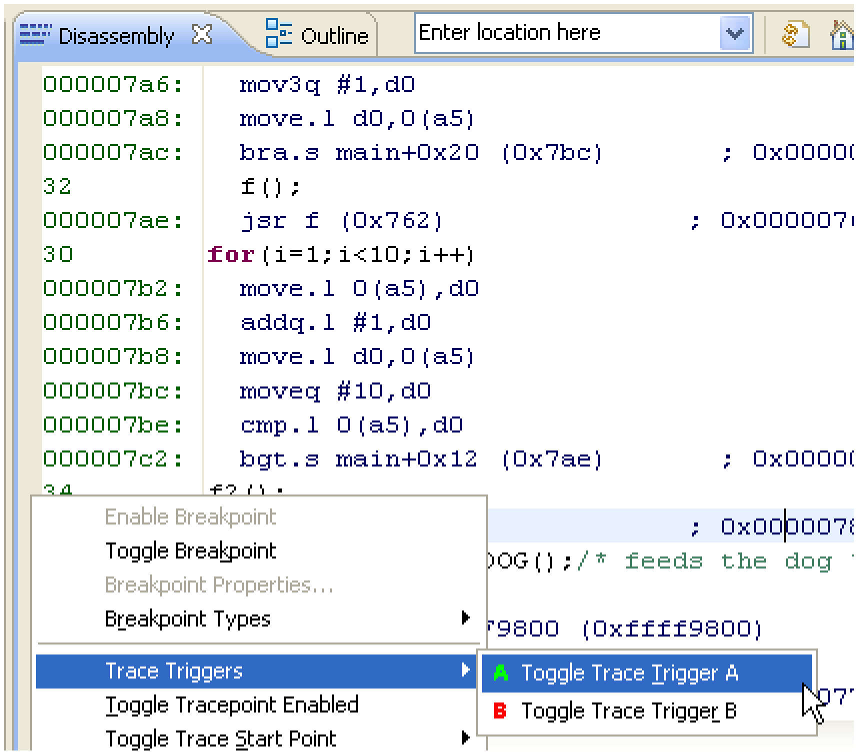
- Trace From Trigger A Onward
- Trace From Trigger A to Trigger B

6.2.2.1 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 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: Source code for trace collection.
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 the marker bar corresponding to the statement, `f2();` before `for(;;)`.



```
000007a6:   mov3q #1,d0
000007a8:   move.l d0,0(a5)
000007ac:   bra.s main+0x20 (0x7bc)        ; 0x0000(
32          f();
000007ae:   jsr f (0x762)                 ; 0x000007(
30        for(i=1;i<10;i++)
000007b2:   move.l 0(a5),d0
000007b6:   addq.l #1,d0
000007b8:   move.l d0,0(a5)
000007bc:   moveq #10,d0
000007be:   cmp.l 0(a5),d0
000007c2:   bgt.s main+0x12 (0x7ae)        ; 0x0000(
34        f2();
                                          ; 0x000007(
                    )OG();/* feeds the dog
                    '9800 (0xffff9800)
```

Figure 6-8. 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**.

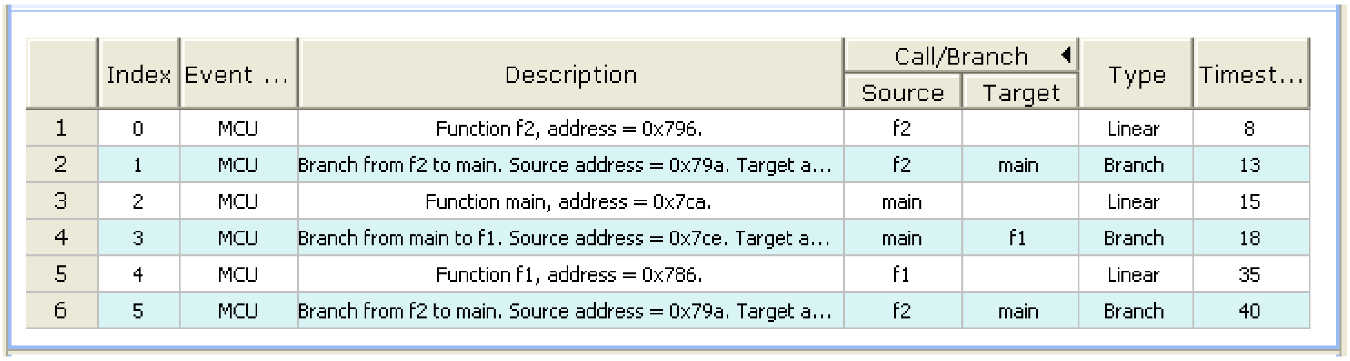
14. Open the **Trace Data** viewer following the steps explained in the topic Viewing Data to view the trace results.

The figure below 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. The figure 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 the following figure 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.



	Index	Event ...	Description	Call/Branch		Type	Timest...
				Source	Target		
1	0	MCU	Function f2, address = 0x796.	f2		Linear	8
2	1	MCU	Branch from f2 to main. Source address = 0x79a. Target a...	f2	main	Branch	13
3	2	MCU	Function main, address = 0x7ca.	main		Linear	15
4	3	MCU	Branch from main to f1. Source address = 0x7ce. Target a...	main	f1	Branch	18
5	4	MCU	Function f1, address = 0x786.	f1		Linear	35
6	5	MCU	Branch from f2 to main. Source address = 0x79a. Target a...	f2	main	Branch	40

Figure 6-9. Trace Data After Setting *Trace From Trigger A Onward* in Automatic Mode

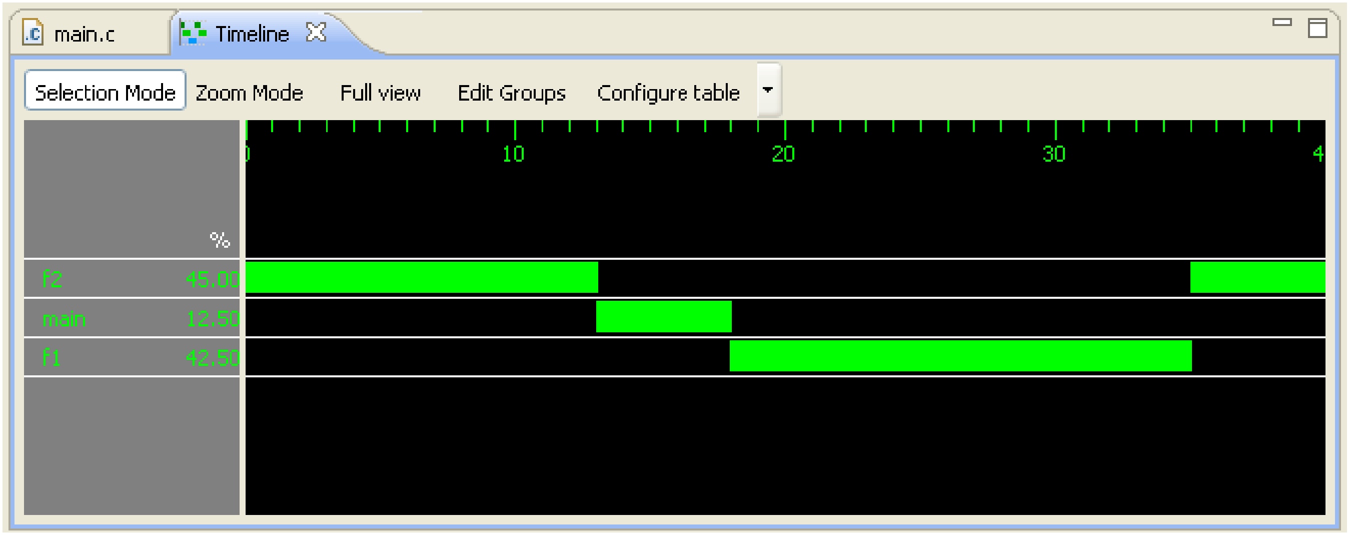


Figure 6-10. Timeline Data After Setting *Trace From Trigger A Onward* in Automatic Mode

This is how you collect trace data in the Automatic (One-buffer) mode of the ColdFire V1 target.

NOTE

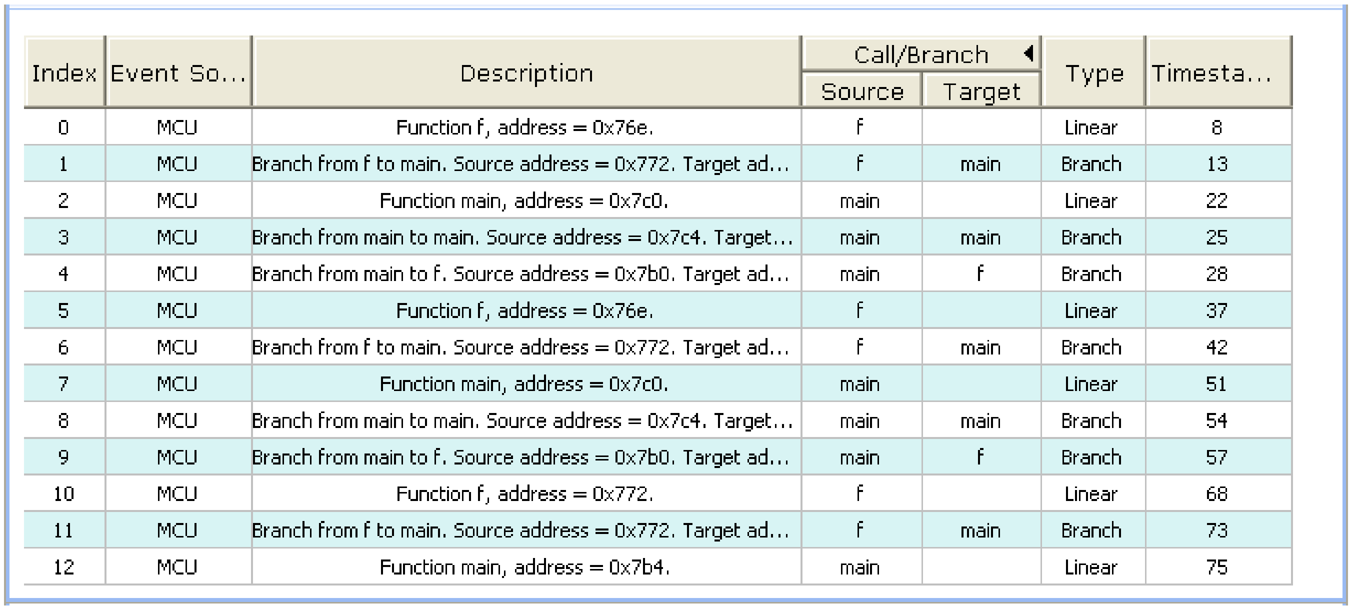
Similarly, you can set the **Trace from Trigger B Onward** trace condition in both Continuous and Automatic modes and collect the trace data.

6.2.2.2 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 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: Source code for trace collection.
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 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.
14. Right-click 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.
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 to view the trace results.

The figure below 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 the figure 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.



Index	Event So...	Description	Call/Branch		Type	Timesta...
			Source	Target		
0	MCU	Function f, address = 0x76e.	f		Linear	8
1	MCU	Branch from f to main. Source address = 0x772. Target ad...	f	main	Branch	13
2	MCU	Function main, address = 0x7c0.	main		Linear	22
3	MCU	Branch from main to main. Source address = 0x7c4. Target...	main	main	Branch	25
4	MCU	Branch from main to f. Source address = 0x7b0. Target ad...	main	f	Branch	28
5	MCU	Function f, address = 0x76e.	f		Linear	37
6	MCU	Branch from f to main. Source address = 0x772. Target ad...	f	main	Branch	42
7	MCU	Function main, address = 0x7c0.	main		Linear	51
8	MCU	Branch from main to main. Source address = 0x7c4. Target...	main	main	Branch	54
9	MCU	Branch from main to f. Source address = 0x7b0. Target ad...	main	f	Branch	57
10	MCU	Function f, address = 0x772.	f		Linear	68
11	MCU	Branch from f to main. Source address = 0x772. Target ad...	f	main	Branch	73
12	MCU	Function main, address = 0x7b4.	main		Linear	75

Figure 6-11. Trace Data View After Setting *Trace From Trigger A to Trigger B* in Automatic Mode

In this example, the trace buffer got full before trigger B was executed, therefore, the **Trace Data** viewer, as shown in the above figure, 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.

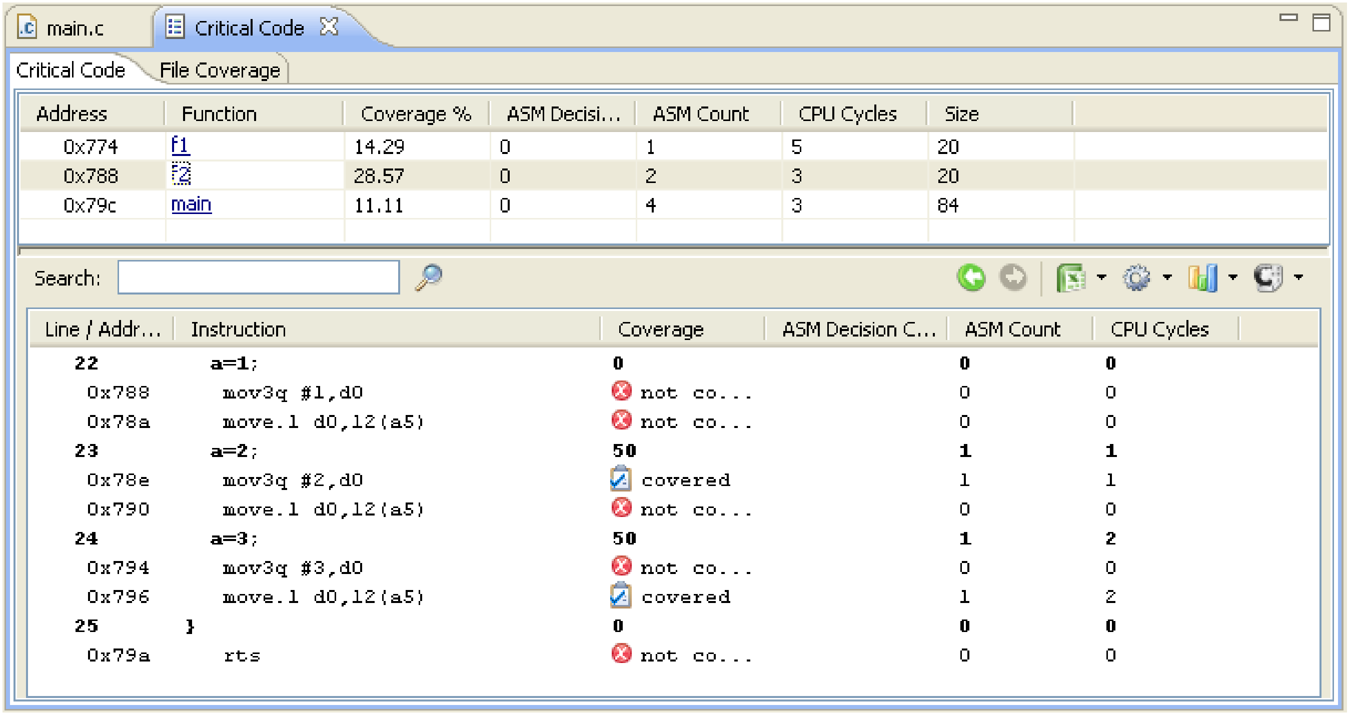
Similarly, you can set the trace conditions, **Trace from Trigger B to Trigger A** , **Trace from Trigger A to Trigger C** , **Trace from Trigger C to Trigger A** , **Trace from Trigger B to Trigger C** , **Trace from Trigger C to trigger B** in both Continuous and Automatic modes and collect the trace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

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

6.2.3 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 to view the critical code data results.



Address	Function	Coverage %	ASM Decisi...	ASM Count	CPU Cycles	Size
0x774	f1	14.29	0	1	5	20
0x788	f2	28.57	0	2	3	20
0x79c	main	11.11	0	4	3	84

Line / Addr...	Instruction	Coverage	ASM Decision C...	ASM Count	CPU Cycles
22	a=1;	0		0	0
0x788	mov3q #1,d0	✗ not co...		0	0
0x78a	move.l d0,12(a5)	✗ not co...		0	0
23	a=2;	50		1	1
0x78e	mov3q #2,d0	✓ covered		1	1
0x790	move.l d0,12(a5)	✗ not co...		0	0
24	a=3;	50		1	2
0x794	mov3q #3,d0	✗ not co...		0	0
0x796	move.l d0,12(a5)	✓ covered		1	2
25	}	0		0	0
0x79a	rts	✗ not co...		0	0

Figure 6-12. Critical Code Data - Profile-Only Mode of ColdFire V1

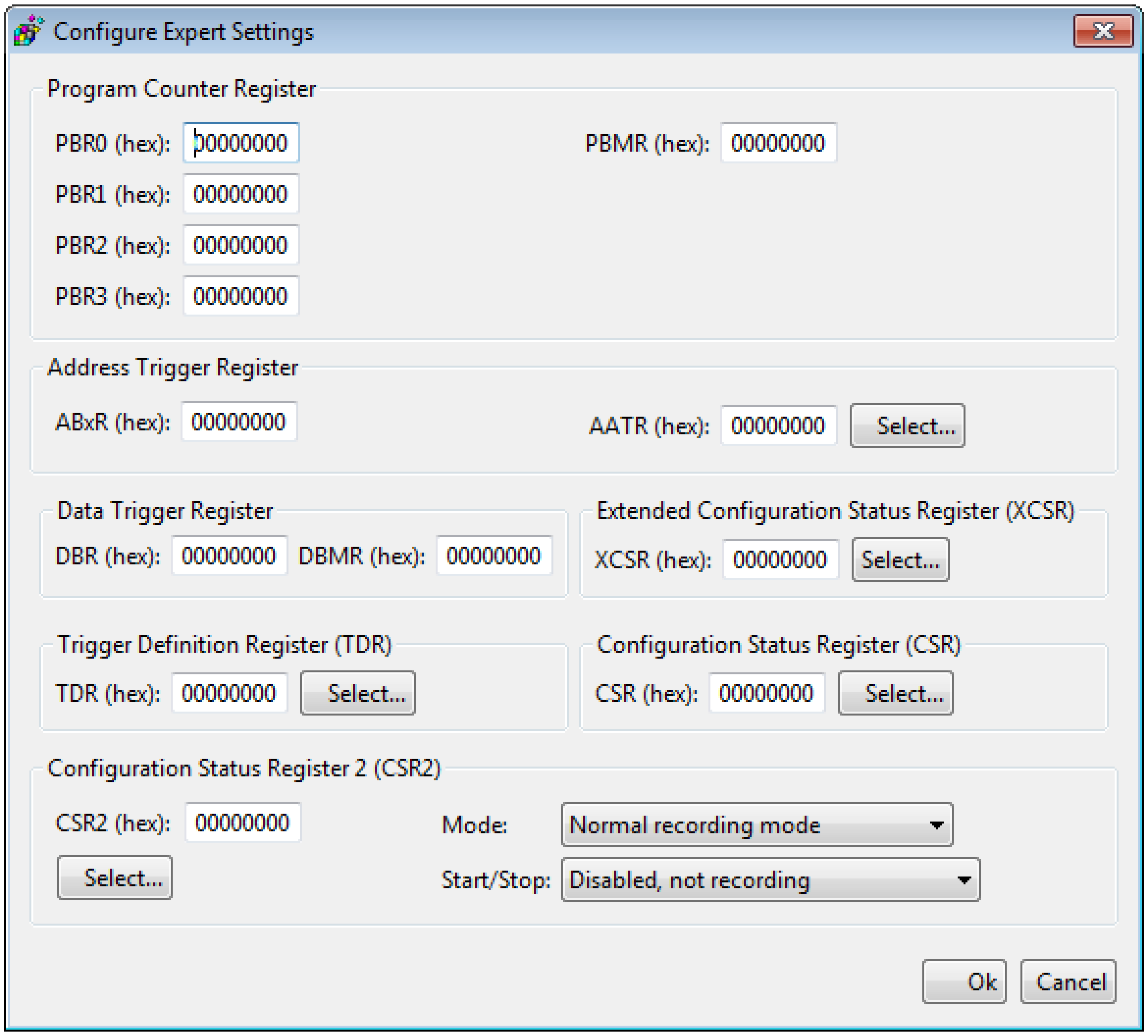
This is how you collect trace data in the Profile-Only mode of the ColdFire V1 target.

6.2.4 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.


Configure Expert Settings

Program Counter Register
PBR0 (hex): 00000000
PBMR (hex): 00000000
PBR1 (hex): 00000000
PBR2 (hex): 00000000
PBR3 (hex): 00000000

Address Trigger Register
ABxR (hex): 00000000
AATR (hex): 00000000

Data Trigger Register
DBR (hex): 00000000
DBMR (hex): 00000000

Extended Configuration Status Register (XCSR)
XCSR (hex): 00000000

Trigger Definition Register (TDR)
TDR (hex): 00000000

Configuration Status Register (CSR)
CSR (hex): 00000000

Configuration Status Register 2 (CSR2)
CSR2 (hex): 00000000
Mode: Normal recording mode
Start/Stop: Disabled, not recording

Figure 6-13. 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.

NOTE

In **Expert** mode, only Trace and Timeline data is collected.

This is how you collect trace data in the **Expert** mode of the ColdFire V1 target.

6.3 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: Source code for trace collection.

- From Variables View
- From Memory View

6.3.1 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 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: Source code for trace collection.
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 a cell in the **Name** column.
11. Select the **Add Global Variables** option from the context menu. The **Add Globals** dialog box appears.

12. Select all the variables with *Shift* key pressed and click **OK** . All the variables of the program get added to the **Variables** view.
13. Right-click 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.

15. Select the **Read/Write** option from the **Access** drop-down list.

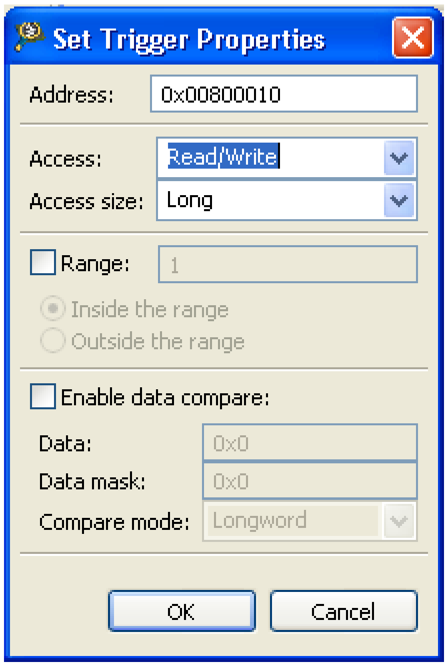
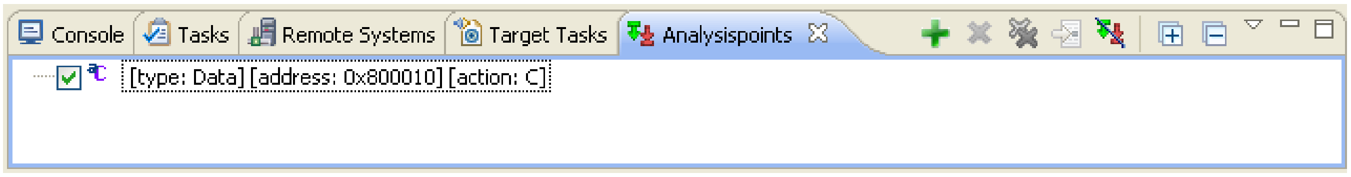


Figure 6-14. Set Trigger Properties Dialog Box

16. Click **OK**.
17. Select **Window > Show View > Other > Analysis > Analysispoints** to open the **Analysispoints** view. The **Analysis points** view displays the trigger C set on the variable address.

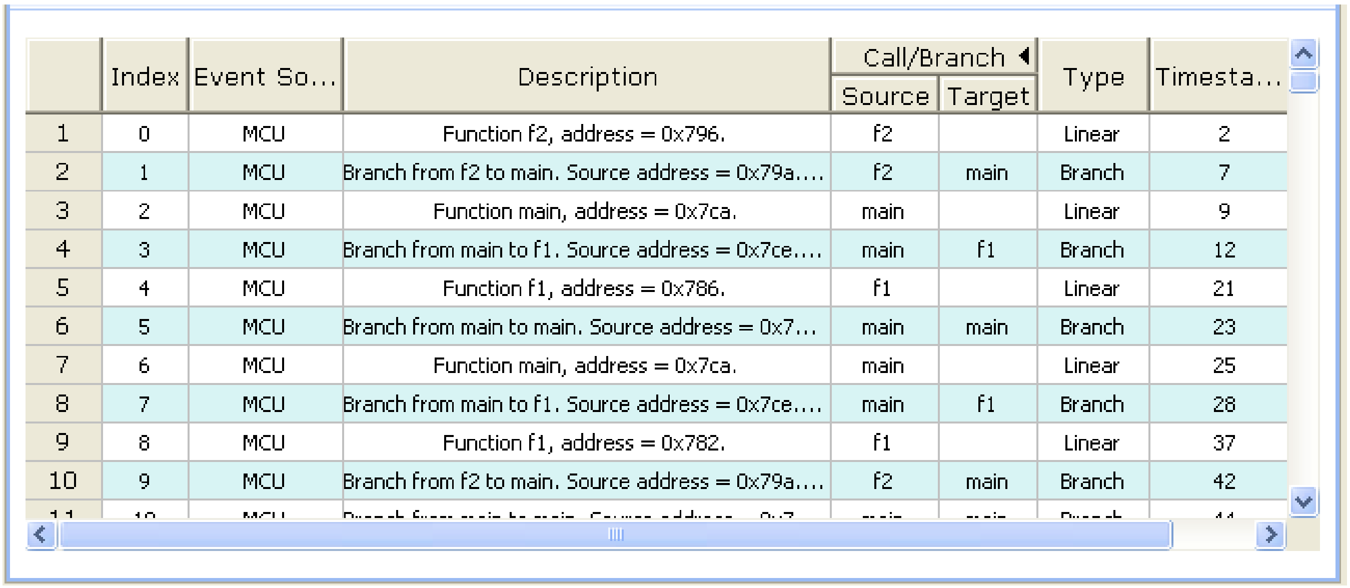


[type: Data] [address: 0x800010] [action: C]

Figure 6-15. Analysispoints 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 to view the trace results.

The figure below shows the data files generated by the application after setting trigger C in the Continuous mode on the variable address.



	Index	Event So...	Description	Call/Branch		Type	Timesta...
				Source	Target		
1	0	MCU	Function f2, address = 0x796.	f2		Linear	2
2	1	MCU	Branch from f2 to main. Source address = 0x79a....	f2	main	Branch	7
3	2	MCU	Function main, address = 0x7ca.	main		Linear	9
4	3	MCU	Branch from main to f1. Source address = 0x7ce....	main	f1	Branch	12
5	4	MCU	Function f1, address = 0x786.	f1		Linear	21
6	5	MCU	Branch from main to main. Source address = 0x7...	main	main	Branch	23
7	6	MCU	Function main, address = 0x7ca.	main		Linear	25
8	7	MCU	Branch from main to f1. Source address = 0x7ce....	main	f1	Branch	28
9	8	MCU	Function f1, address = 0x782.	f1		Linear	37
10	9	MCU	Branch from f2 to main. Source address = 0x79a....	f2	main	Branch	42

Figure 6-16. 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.

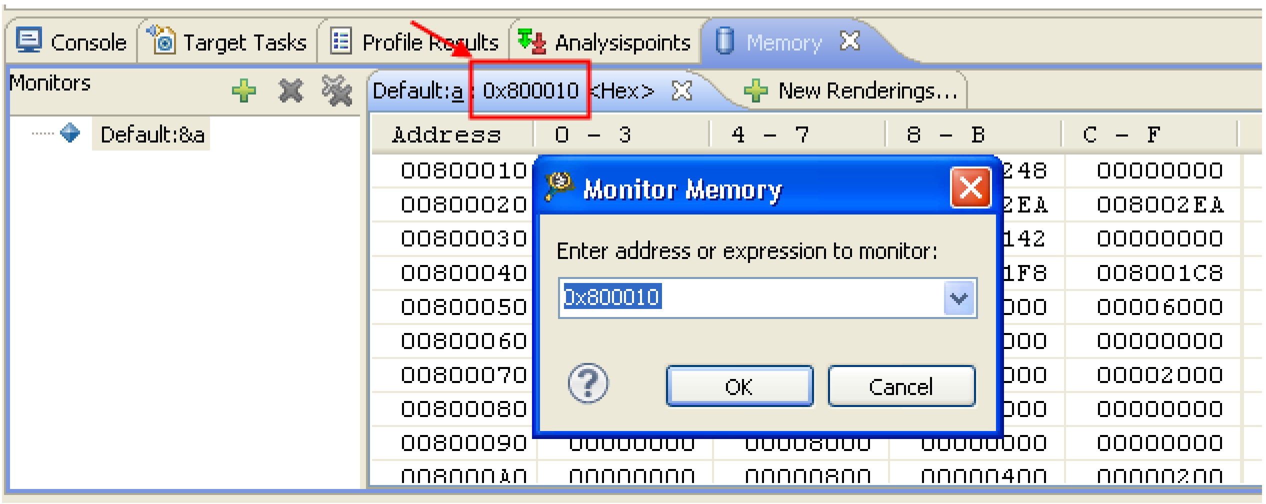
6.3.2 From Memory View

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

1. Perform steps 1 - 12 as explained in From Variables View
2. Right-click the variable `a`, in the **Name** column of the **Variables** view.
3. Select the **View Memory** option from the context menu.

   The **Memory** view appears.

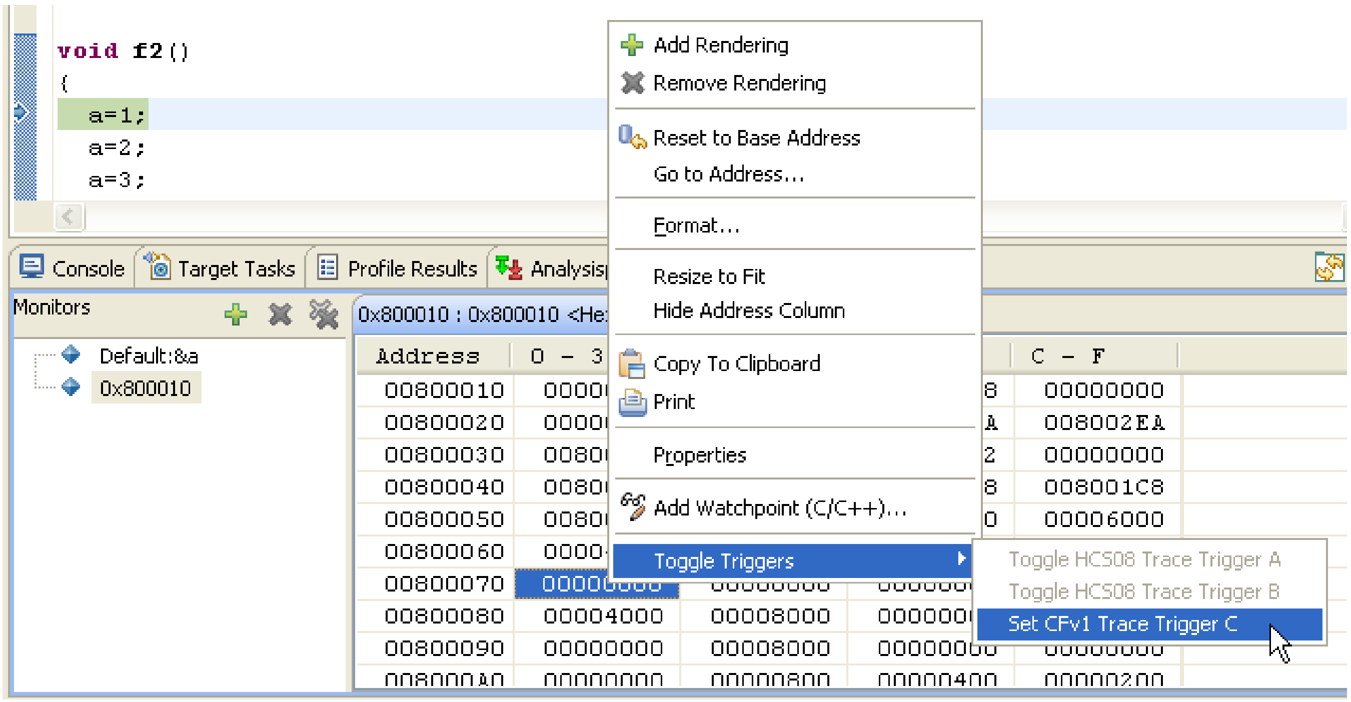
4. Click the **Add Memory Monitor** button in the **Memory** view to open the **Monitor Memory** dialog box.



```
Address    0 - 3     4 - 7     8 - B     C - F
00800010                            248  00000000
00800020                            2EA  008002EA
00800030                            142  00000000
00800040                            1F8  008001C8
00800050                            000  00006000
00800060                            000  00000000
00800070                            000  00002000
00800080                            000  00000000
00800090   00000000  00008000  00000000  00000000
008000A0   00000000  00000800  00000400  00000200
```

Figure 6-17. Monitor Memory Dialog Box

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



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

Figure 6-18. Setting Trigger C From Memory View

The **Set Trigger Properties** dialog box appears displaying 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 6-16, after setting trigger C in the Continuous mode on the variable address.

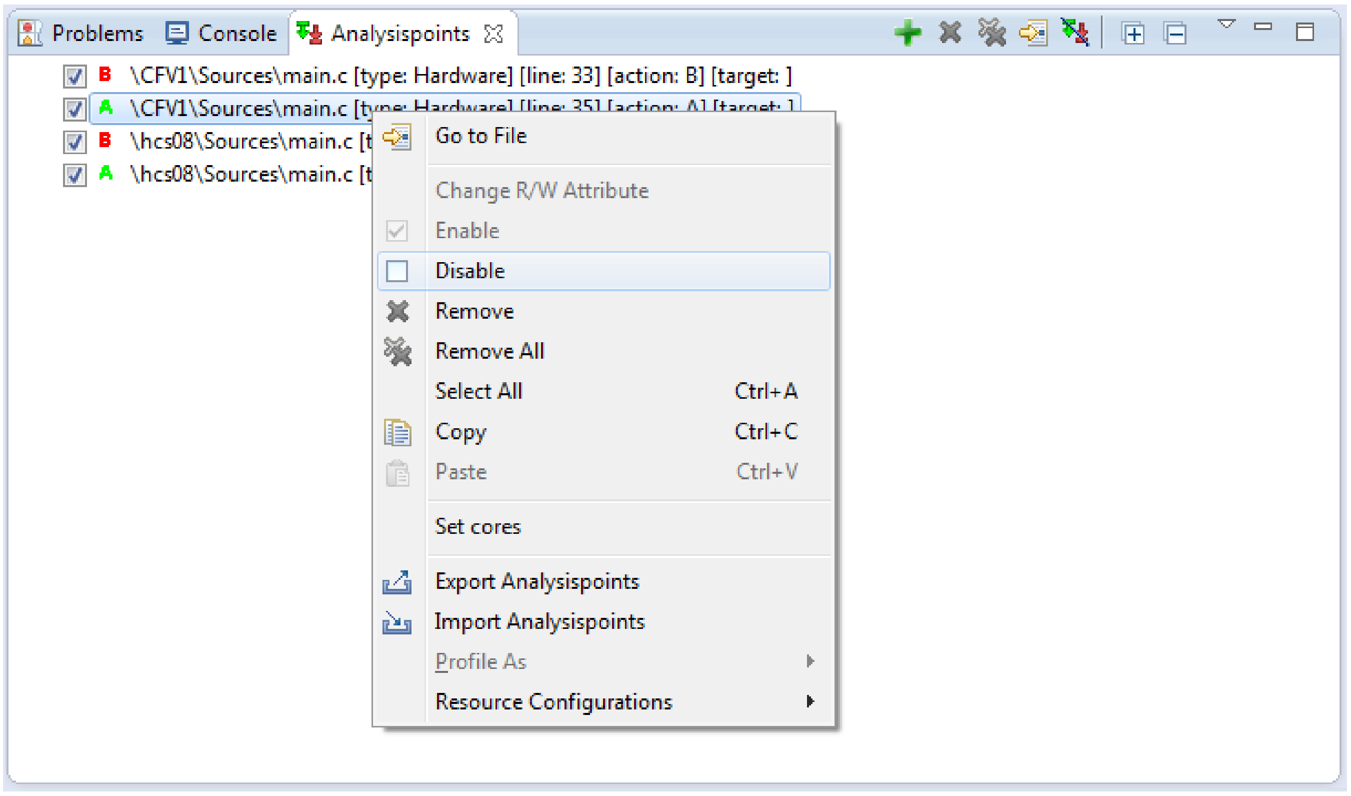
6.4 Enable and Disable Tracepoints

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

If you want to disable the tracepoints, right-click the marker bar where triggers are already set and enabled, select the **Disable Tracepoint** 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 **Analysispoints** view. Select **Window > Show View > Other > Software Analysis > Analysispoints** to open the **Analysispoints** view.

Right-click the selected attribute and select **Disable/Enable** option. The unchecked attribute indicates the disabled tracepoint.

You can also use the **Ignore all Analysispoints** button to disable all the tracepoints without manually selecting them in the **Analysispoints** view. You can click **Ignore All** again to enable the tracepoints.



B \CFV1\Sources\main.c [type: Hardware] [line: 33] [action: B] [target:]
A \CFV1\Sources\main.c [type: Hardware] [line: 35] [action: A] [target:]
B \hcs08\Sources\main.c [t
A \hcs08\Sources\main.c [t

Figure 6-19. Disabling/Enabling the Trigger from Analysispoints View

For detailed information on the **Analysispoints** view, refer Viewing Tracepoints.

Chapter 7
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 a limited number of comparators to be set for trace collection because they use DWT comparators to start or stop the trace collection. There are four comparators for Kinetis Cortex M4 core and two comparators for Kinetis Cortex M0+ core.

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, limited only by the memory available for storing tracepoint data. Software tracepoints are more intrusive than hardware tracepoints because of the overhead added by the interrupts being used. Kinetis Cortex M0+ core does not support software tracepoints.

This chapter consists of the following topics:

- Setting Hardware Tracepoints
- Setting Software Tracepoints
- Viewing Tracepoints

7.1 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 - Kinetis Cortex M4 Core

- From Source Code - Kinetis Cortex M0+ Core
- From Trace and Profile Tab - Kinetis Cortex M4 Core
- From Trace and Profile Tab - Kinetis Cortex M0+ Core

NOTE

The e200 hardware does not provide support for hardware tracepoints.

NOTE

Do not use hardware tracepoints and software tracepoints together as they do not work simultaneously. This is because hardware tracepoints require ETM enabled, which only happens when a start software tracepoint is hit. And ETM logic has to be disabled by default when using a software tracepoint since ETM is enabled from the software interrupt code.

7.1.1 From Source Code - Kinetis Cortex M4 Core

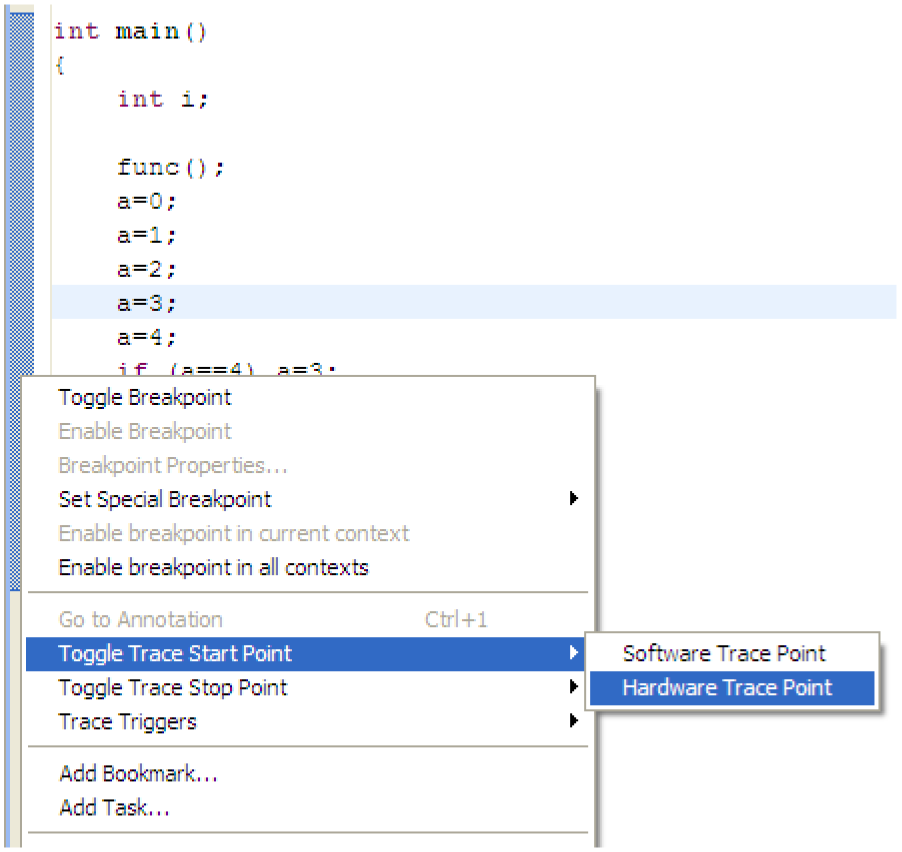
To set hardware tracepoints in the source code and collect trace data on the Kinetis Cortex M4 project:

1. In the **CodeWarrior Projects** view, select the `Sources` folder of your project.
2. Double-click 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 below.
   Listing: 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 statement: `if (a==4) a=3;`.
8. Right-click the marker bar, select the **Toggle Trace Start Point > Hardware Trace Point** option from the context menu. The same option is also used to remove the start trigger from the marker bar.



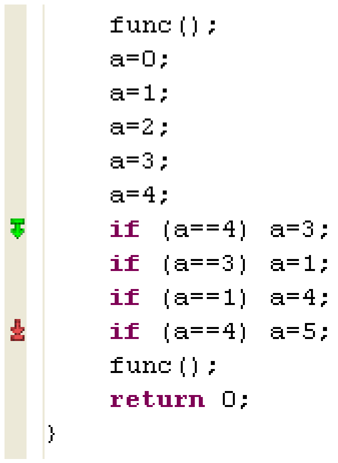
```
int main()
{
    int i;

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

Figure 7-1. 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 statement: `if (a==4) a=5;`.
10. Right-click the marker bar, and 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. The same option is also used to remove the stop trigger from the marker bar.



```
    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;
}
```

Figure 7-2. Hardware Start and Stop Tracepoints set in Source Code

11. Debug your application.

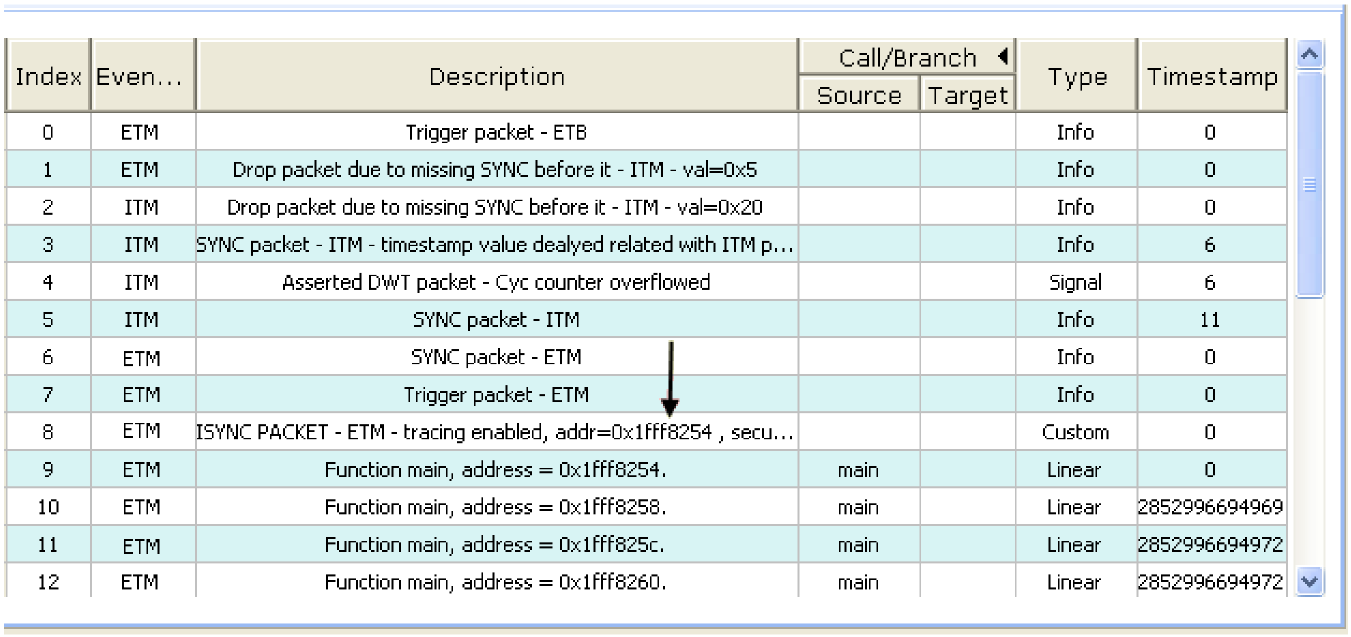
    The application halts at the beginning of the `main()` function.

12. Click **Resume** to resume the program execution.
13. Wait for some time to let the application terminate.

    The **Software Analysis** view appears.

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

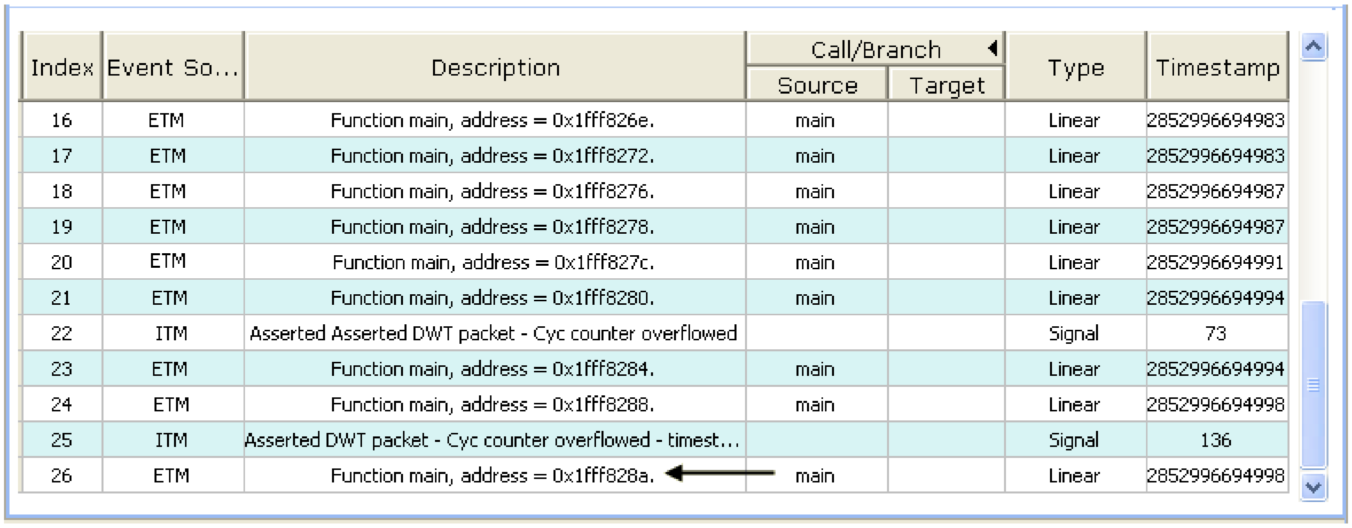
    The **Trace Data** viewer appears as shown in the following two figures.



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

Figure 7-3. Trace Results After Setting Hardware Tracepoints in Source Code - Start Address

These figures display 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.



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

Figure 7-4. Trace Results After Setting Hardware Tracepoints in Source Code - Stop Address

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 data.

7.1.2 From Source Code - Kinetis Cortex M0+ Core

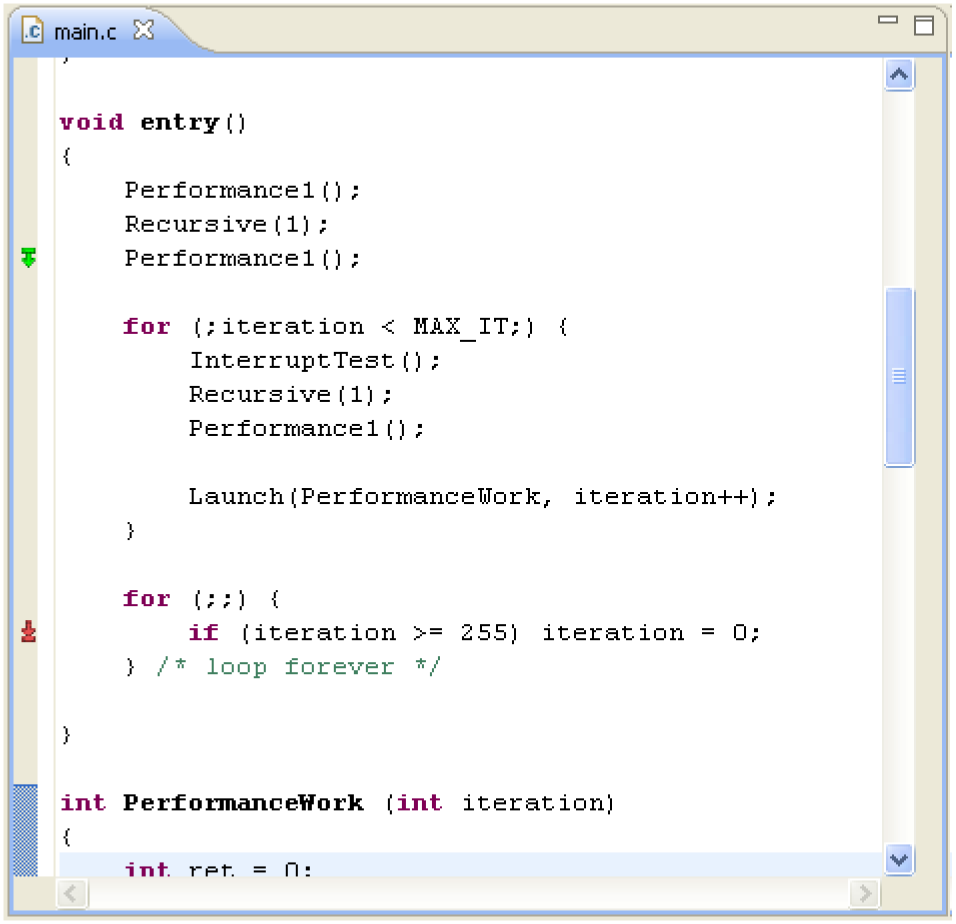
To set hardware tracepoints in the source code and collect trace data on the Kinetis Cortex M0+ project:

1. In the **CodeWarrior Projects** view, select the `Sources` folder of your project.
2. Double-click 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 below.
   Listing: Source code for trace collection

```
#include "derivative.h" /* include peripheral declarations */

#define MAX_IT 5

#define SIMPLE


typedef int(*FUNC_TYPE)(int);


void Launch(FUNC_TYPE, int);

void entry();
void InterruptTest();

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

volatile char iteration = 0;

void InterruptTest()
{
}

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

void entry()
{
     Performance1();
     Recursive(1);
     Performance1();

     for (;iteration < MAX_IT;) {
         InterruptTest();
         Recursive(1);
         Performance1();
         Launch(PerformanceWork, iteration++);
     }
     for (;;) {
         if (iteration >= 255) iteration = 0;
     } /* loop forever */
}

int PerformanceWork (int iteration)
```

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

void Performance1(void)
{
#ifndef SIMPLE
     int vecSize = 10;
     int vec[]={10,9,8,7,6,5,4,3,2,1};

     int i,aux;

     for (i=0; i<(vecSize/2) ;i++)
     {
         aux = vec[i];
         vec[i] = vec[vecSize-i-1];
         vec[vecSize-i-1] = aux;
     }
#endif
}

int Recursive(int n)
{
      /* Recursively calculates 0 + 1 + 2 + ... + n */
      if (n <= 0)
      {
          return 0;
      }
      else
      {
          return (n + Recursive(n-1));
      }
}

int main(void) {
      entry();
/* please make sure that you never leave main */
}
```

3. Save the project.
4. Open the **Debug Configurations** dialog box.
5. Enable tracing and profiling using steps 3 - 11 of Configuring Kinetis Cortex M0+ Core.
6. Select the **Continuous** option, click **Apply**, and close the dialog box..
7. Set start and stop hardware tracepoints in `main.c` as shown below:



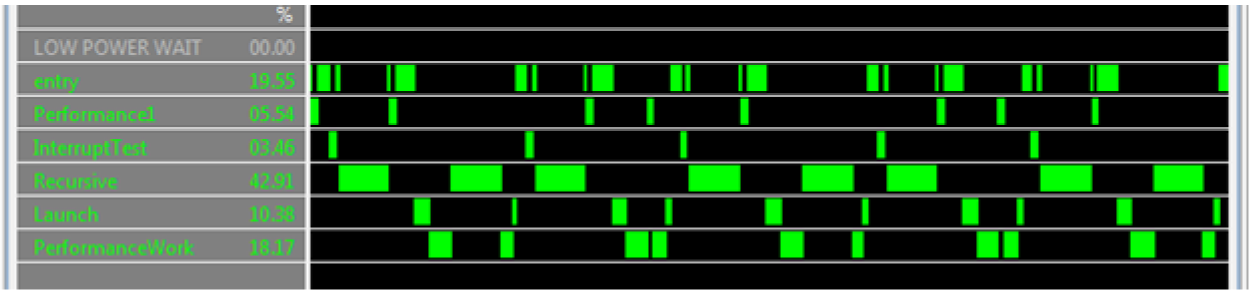
```
void entry()
{
    Performance1();
    Recursive(1);
    Performance1();

    for (;iteration < MAX_IT;) {
        InterruptTest();
        Recursive(1);
        Performance1();

        Launch(PerformanceWork, iteration++);
    }

    for (;;) {
        if (iteration >= 255) iteration = 0;
    } /* loop forever */

}

int PerformanceWork (int iteration)
{
    int ret = 0;
```

Figure 7-5. Hardware Start and Stop Tracepoints

8. Debug your project.
9. Click **Resume**. After a few seconds, click **Suspend**.

   The **Software Analysis** view appears.

10. Expand the project name and click the **Timeline** hyperlink to see the results.



	%
LOW POWER WAIT	00.00
entry	19.55
Performance1	05.54
InterruptTest	03.46
Recursive	42.91
Launch	10.38
PerformanceWork	18.17

Figure 7-6. Timeline Results After Setting HW Tracepoints on Kinetis Cortex M0+ Project

7.1.3 From Trace and Profile Tab - Kinetis Cortex M4 Core

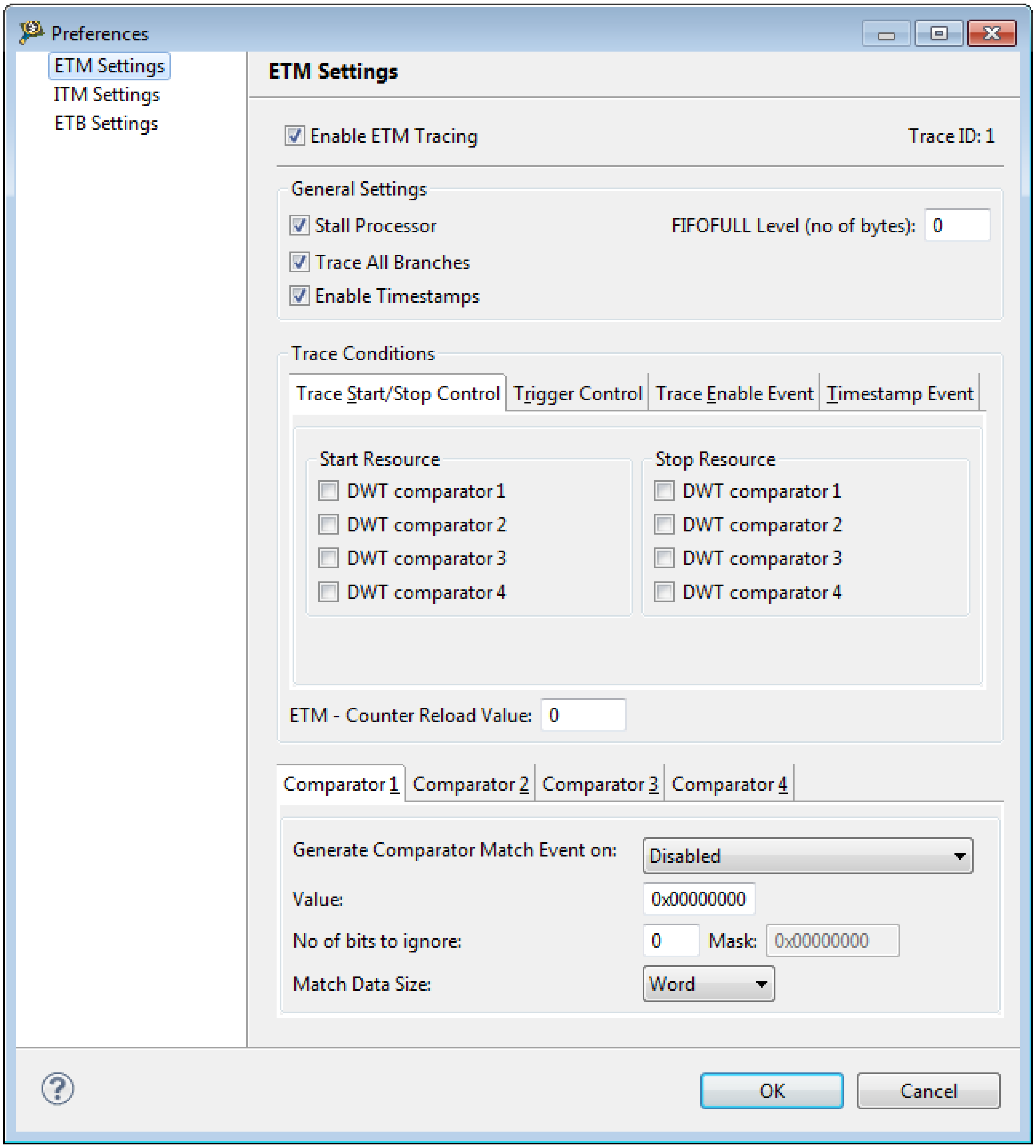
This topic explains how to set comparator 1 and comparator 2 as hardware tracepoints for trace collection. You can take any combination of comparators to be used as hardware tracepoints. You simply need to specify the correct corresponding reference value against them.

This topic uses the addresses of the instructions, of Listing: Source code for trace collection, 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 on a Kinetis Cortex M4 project:

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.



ETM Settings

☑ Enable ETM Tracing   Trace ID: 1

General Settings

☑ Stall Processor   FIFOFULL Level (no of bytes): 0
☑ Trace All Branches
☑ Enable Timestamps

Trace Conditions

Trace Start/Stop Control | Trigger Control | Trace Enable Event | Timestamp Event

Start Resource	Stop Resource
☐ DWT comparator 1	☐ DWT comparator 1
☐ DWT comparator 2	☐ DWT comparator 2
☐ DWT comparator 3	☐ DWT comparator 3
☐ DWT comparator 4	☐ DWT comparator 4

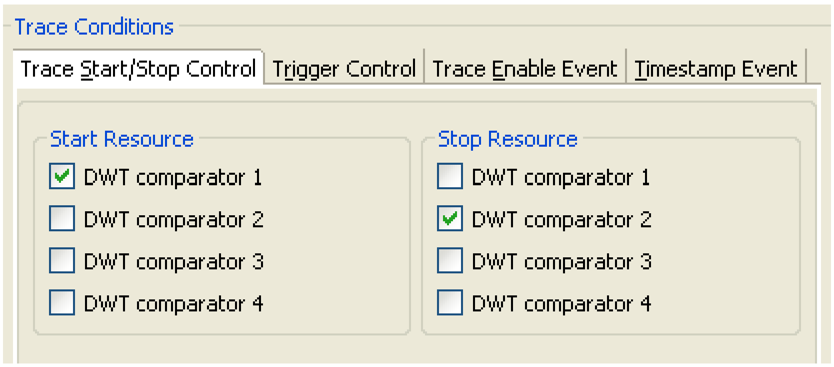
ETM - Counter Reload Value: 0

Comparator 1 | Comparator 2 | Comparator 3 | Comparator 4

Generate Comparator Match Event on: Disabled
Value: 0x00000000
No of bits to ignore: 0   Mask: 0x00000000
Match Data Size: Word

Figure 7-7. Preferences Dialog Box

5. In the **Trace Start/Stop Control** group, check the **DWT comparator 1** checkbox in the **Start Resource** group and the **DWT comparator 2** checkbox in the **Stop Resource** group.


Trace Conditions

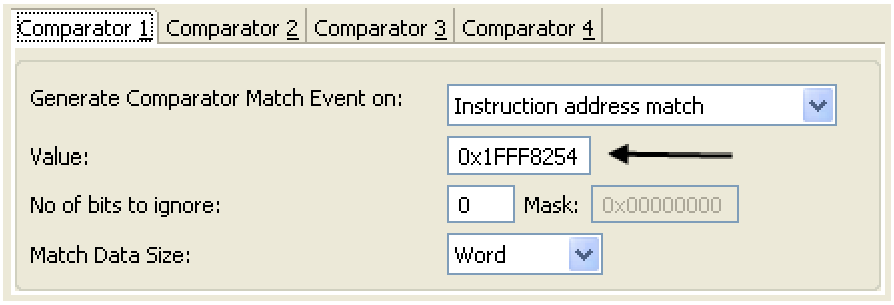
Trace Start/Stop Control | Trigger Control | Trace Enable Event | Timestamp Event

Start Resource
- [✓] DWT comparator 1
- [ ] DWT comparator 2
- [ ] DWT comparator 3
- [ ] DWT comparator 4

Stop Resource
- [ ] DWT comparator 1
- [✓] DWT comparator 2
- [ ] DWT comparator 3
- [ ] DWT comparator 4

Figure 7-8. Trace Start/Stop Control Settings

6. In the **Comparator 1** tab, specify the following settings: Select the **Instruction address match** option in the **Generate Comparator Match Event on** drop-down list. 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: Source code for trace collection.


Comparator 1 | Comparator 2 | Comparator 3 | Comparator 4

Generate Comparator Match Event on: Instruction address match

Value: 0x1FFF8254

No of bits to ignore: 0 Mask: 0x00000000

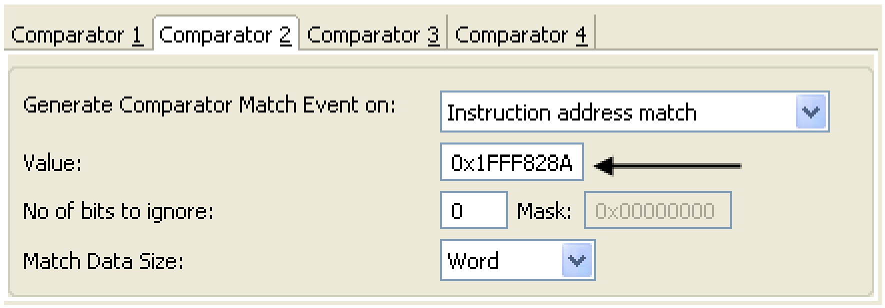
Match Data Size: Word

Figure 7-9. 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: Select the **Instruction address match** option in the **Generate Comparator Match Event on** drop-down list. 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: Source code for trace collection.



Generate Comparator Match Event on: Instruction address match

Value: 0x1FFF828A

No of bits to ignore: 0 Mask: 0x00000000

Match Data Size: Word

Figure 7-10. 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. In the **Software Analysis** view, expand the project name and click the **Trace** hyperlink.

The **Trace Data** viewer appears displaying the same results as shown in Figure 7-3 and Figure 7-4.

7.1.4 From Trace and Profile Tab - Kinetis Cortex M0+ Core

This topic explains how to set comparator 1 and comparator 2 as hardware tracepoints for collecting trace on a Kinetis Cortex M0+ project.

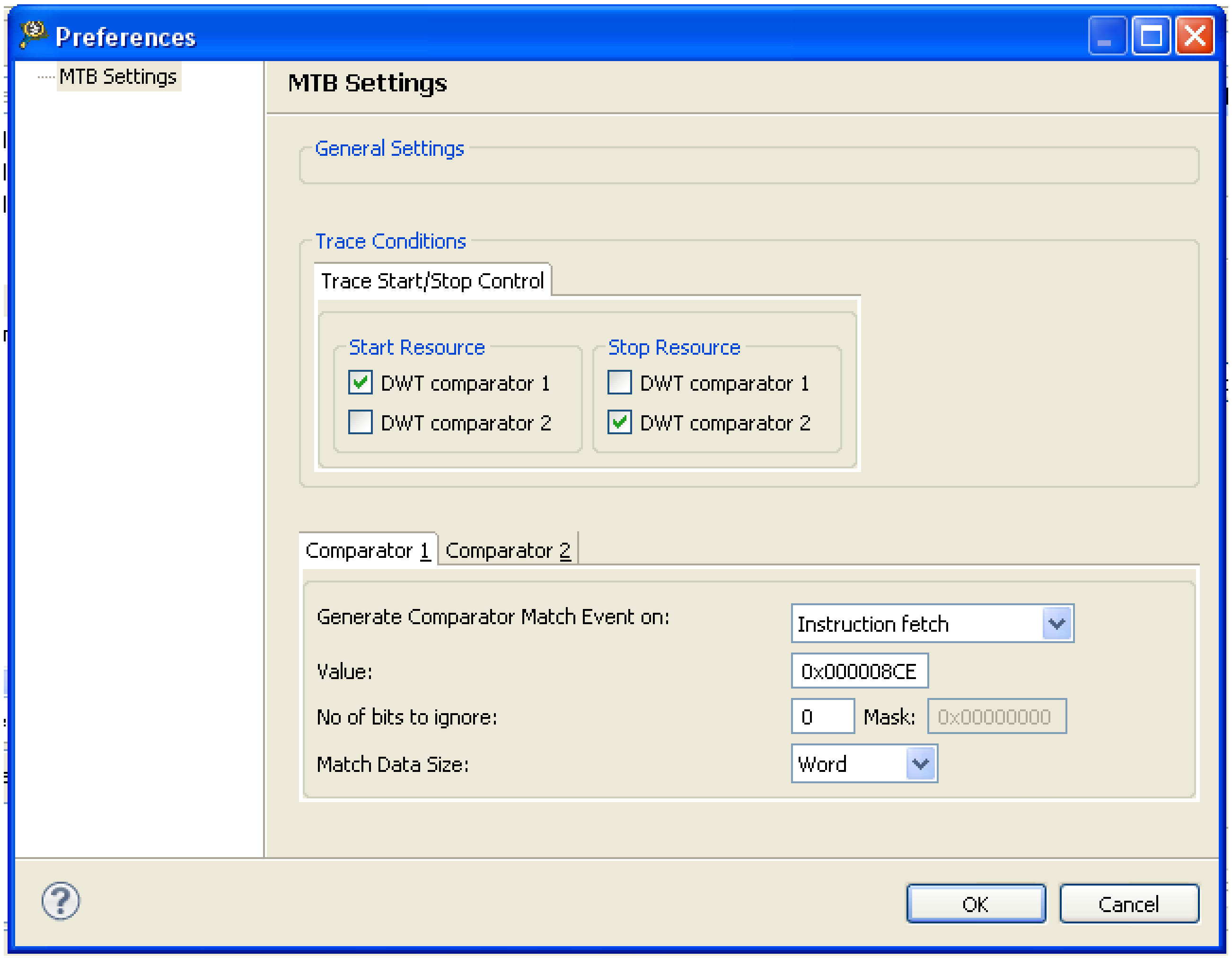
This topic uses the addresses of the instructions, of Listing: Source code for trace collection, at which you set start and stop hardware tracepoint. These addresses will be set against comparator 1 and comparator 2 to collect the same trace results that appear on setting hardware tracepoints from the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

To set hardware tracepoints using the **Trace and Profile** tab and collect trace on a Kinetis Cortex M0+ project:

1. Select your project and open the **Debug Configurations** dialog box.
2. Enable tracing and profiling using steps 3 - 11 of Configuring Kinetis Cortex M0+ Core.
3. Close the **Debug Configurations** dialog box.
4. In the **CodeWarrior Projects** view, right-click the .elf file of your project, and select the **Disassemble** option.

   The disassembly file of the project opens automatically.

5. Search for the string `entry()`.
6. Look for the address of the second call to `Performance1()`. For example, `0x8ce`.
7. Look for the address of the first assembly instruction of the line `if (iteration >= 255) iteration = 0;`. For example, `0x90a`.
8. Open the **Debug Configurations** dialog box, and select the **Trace and Profile** tab.
9. Click **Advanced Settings** to open the **Preferences** dialog box displaying the MTB settings.
10. Select **DWT comparator 1** as **Start Resource** and **DWT comparator 2** as **Stop Resource**.
11. In the **Comparator 1** tab, select **Instruction Fetch from the Generate Comparator Match Event On** drop-down box.
12. In the **Value** text box, specify the address of the instruction on which you want to set comparator 1. For example, 0x000008CE, which is the address of the second call to `Performance1()`.



MTB Settings

Start Resource: [✓ DWT comparator 1] DWT comparator 2

Stop Resource: DWT comparator 1 [✓ DWT comparator 2]

Generate Comparator Match Event on: Instruction fetch

Value: 0x000008CE

No of bits to ignore: 0 Mask: 0x00000000

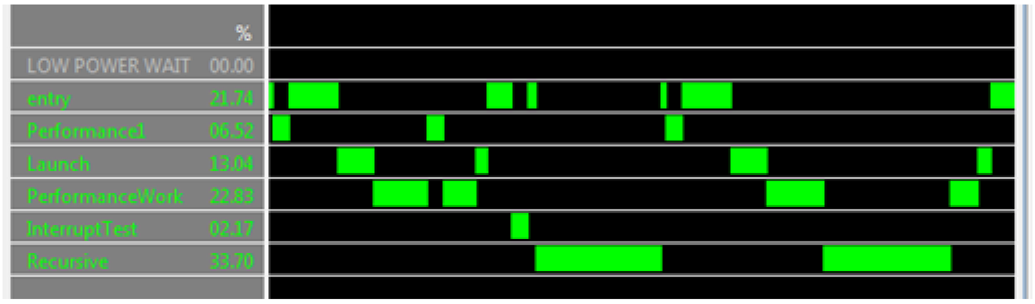
Match Data Size: Word

Figure 7-11. Setting Comparator 1 — MTB Settings

13. In the **Comparator 2** tab, select **Instruction Fetch from the Generate Comparator Match Event On** drop-down box.

14. In the **Value** text box, specify the address of the instruction on which you want to set comparator 2. For example, 0x0000090A, which is the address of the first assembly instruction of the line `if (iteration >= 255) iteration = 0;`.
15. Click **OK** and close the **Preferences** dialog box.
16. Debug the application.
17. Click **Resume** and after a few seconds, click **Suspend**.
18. In the **Software Analysis** view, expand the project name and click the **Timeline** hyperlink.

    The **Timeline** viewer appears displaying the results shown below.



	%
LOW POWER WAIT	00.00
entry	21.74
Performance1	06.52
Launch	13.04
PerformanceWork	22.83
InterruptTest	02.17
Recursive	33.70

Figure 7-12. Timeline Results After Setting DWT Triggers on Kinetis Cortex M0+ Project

7.2 Setting Software Tracepoints

Software tracepoints use an interrupt to start and stop trace. You can update your project either manually or automatically to add support of software tracepoints for collecting trace data on the Kinetis target.

- Setting Software Tracepoints Manually
- Setting Software Tracepoints Automatically

NOTE

Do not use hardware tracepoints and software tracepoints together as they do not work simultaneously. This is because hardware tracepoints require ETM enabled, which only happens when a start software tracepoint is hit. And ETM logic has to be disabled by default when using a software tracepoint since ETM is enabled from the software interrupt code.

7.2.1 Setting Software Tracepoints Manually

To enable software tracepoints manually, you need to add the sa handler files to your project and modify the `kinetis_sysinit.c` file. The sa handler 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.

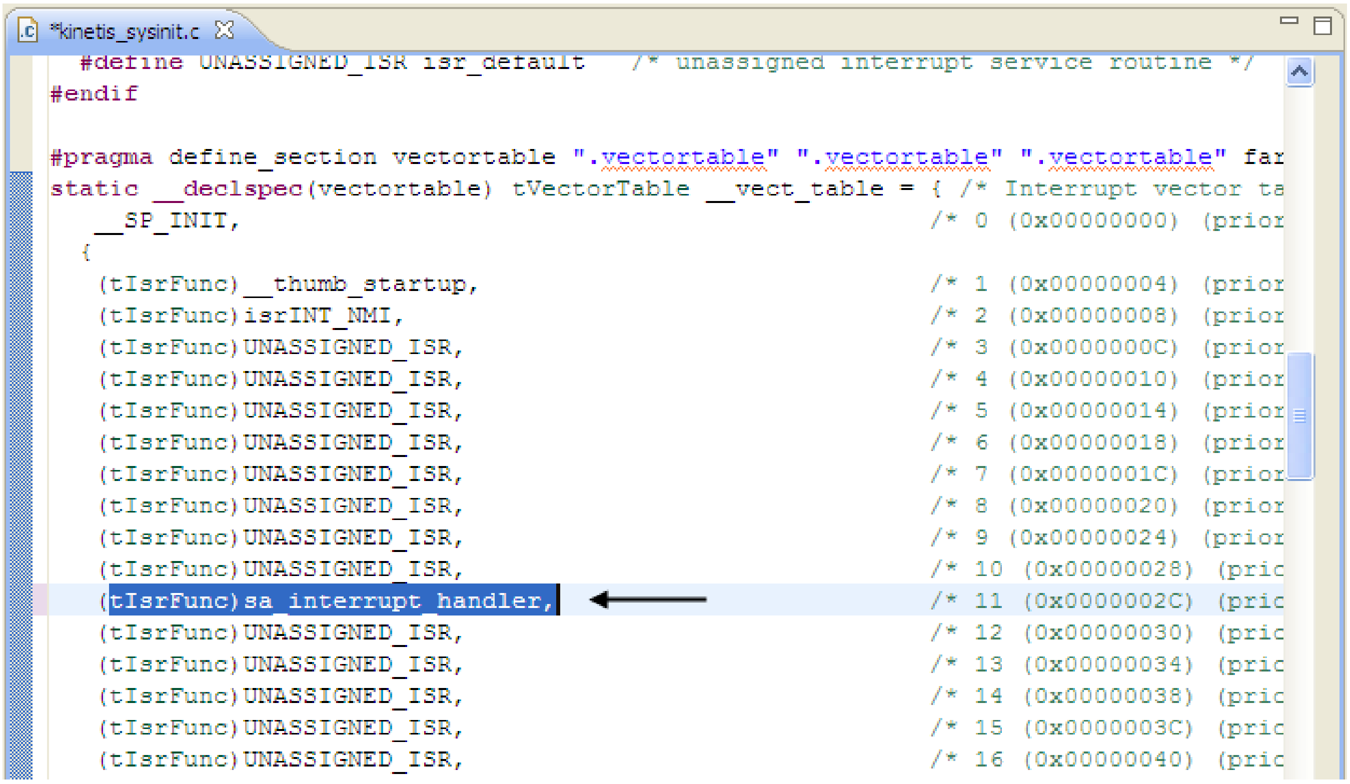
Two toolchains, Freescale and GCC, are available for enabling and installing software tracepoints manually. You need to select the required option in the **Language and Build Tools Options** page while creating a Kinetis project.

- Using Freescale Toolchain
- Using GCC Toolchain

7.2.1.1 Using Freescale Toolchain

To enable software tracepoints manually and collect trace data on the Kinetis target using the Freescale toolchain:

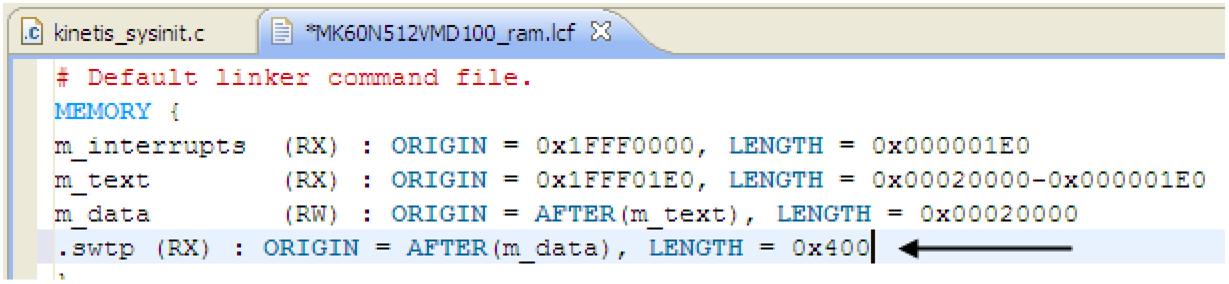
1. Create a stationary Kinetis project with **Freescale** option selected in the **Language and Build Tools Options** page.
2. Add the `sa_handle.c` file to your project using the following steps:
   a. In the **CodeWarrior Projects** view, select the `Sources` folder of your project.
   b. Right-click 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**.
   e. In the **File Operation** dialog box, select the **Copy Files** option.
3. Expand **Project_Settings > Startup_Code** in the **CodeWarrior Projects** view, and open the `kinetis_sysinit.c` file in the editor area.
4. Edit the `kinetis_sysinit.c` file. Include this header file in the source code: `#include "sa_handler.h"`. And add this line in the interrupt vector section: `(tIsrFunc)sa_interrupt_handler,.`



```
  #define UNASSIGNED_ISR isr_default   /* unassigned interrupt service routine */
#endif

#pragma define_section vectortable ".vectortable" ".vectortable" ".vectortable" far
static __declspec(vectortable) tVectorTable __vect_table = { /* Interrupt vector ta
   __SP_INIT,                                                /* 0 (0x00000000) (prior
  {
   (tIsrFunc)__thumb_startup,                                /* 1 (0x00000004) (prior
   (tIsrFunc)isrINT_NMI,                                     /* 2 (0x00000008) (prior
   (tIsrFunc)UNASSIGNED_ISR,                                 /* 3 (0x0000000C) (prior
   (tIsrFunc)UNASSIGNED_ISR,                                 /* 4 (0x00000010) (prior
   (tIsrFunc)UNASSIGNED_ISR,                                 /* 5 (0x00000014) (prior
   (tIsrFunc)UNASSIGNED_ISR,                                 /* 6 (0x00000018) (prior
   (tIsrFunc)UNASSIGNED_ISR,                                 /* 7 (0x0000001C) (prior
   (tIsrFunc)UNASSIGNED_ISR,                                 /* 8 (0x00000020) (prior
   (tIsrFunc)UNASSIGNED_ISR,                                 /* 9 (0x00000024) (prior
   (tIsrFunc)UNASSIGNED_ISR,                                 /* 10 (0x00000028) (pric
   (tIsrFunc)sa_interrupt_handler,                           /* 11 (0x0000002C) (pric
   (tIsrFunc)UNASSIGNED_ISR,                                 /* 12 (0x00000030) (pric
   (tIsrFunc)UNASSIGNED_ISR,                                 /* 13 (0x00000034) (pric
   (tIsrFunc)UNASSIGNED_ISR,                                 /* 14 (0x00000038) (pric
   (tIsrFunc)UNASSIGNED_ISR,                                 /* 15 (0x0000003C) (pric
   (tIsrFunc)UNASSIGNED_ISR,                                 /* 16 (0x00000040) (pric
```

Figure 7-13. Interrupt Vector Section

5. Save the `kinetis_sysinit.c` file.
6. Edit the linker control file of your project.
   a. Expand **Project_Settings > Linker_Files** in the **CodeWarrior Projects** view.
   b. Open the `<project>_ram.lcf` file in the editor area.
   c. Add this statement in the memory section: `.swtp (RX) : ORIGIN = AFTER(m_data), LENGTH = 0x400`



```
# Default linker command file.
MEMORY {
m_interrupts   (RX) : ORIGIN = 0x1FFF0000, LENGTH = 0x000001E0
m_text         (RX) : ORIGIN = 0x1FFF01E0, LENGTH = 0x00020000-0x000001E0
m_data         (RW) : ORIGIN = AFTER(m_text), LENGTH = 0x00020000
.swtp  (RX)  : ORIGIN = AFTER(m_data), LENGTH = 0x400
```

Figure 7-14. Memory Section of Linker File

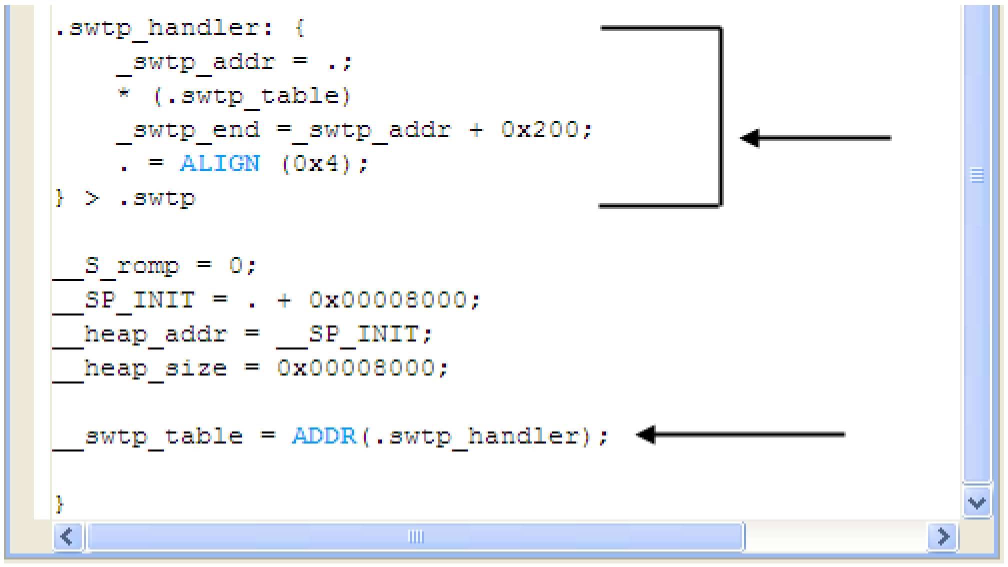
   d. Add the `.swtp` section containing the following statements:

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

NOTE

Make sure that you do not change the section name because the Software Analysis tool looks for this exact name when software tracepoints are installed.

e. Add this statement at the end of the file: `__swtp_table = ADDR(.swtp_handler);`. This will store the address of the software tracepoint handler.



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

__S_romp = 0;
__SP_INIT = . + 0x00008000;
__heap_addr = __SP_INIT;
__heap_size = 0x00008000;

__swtp_table = ADDR(.swtp_handler);

}
```

Figure 7-15. `.swtp` Section of Linker File When Freescale is Selected

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

7. Save and build the project.
8. 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.
9. Set start and stop software tracepoints.
   a. In the editor area, select this statement: `int counter = 0;`
   b. Right-click 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 tracepoint from the marker bar.

The **Software Tracepoints Support** dialog appears. For more information on this dialog box, refer Setting Software Tracepoints Automatically.

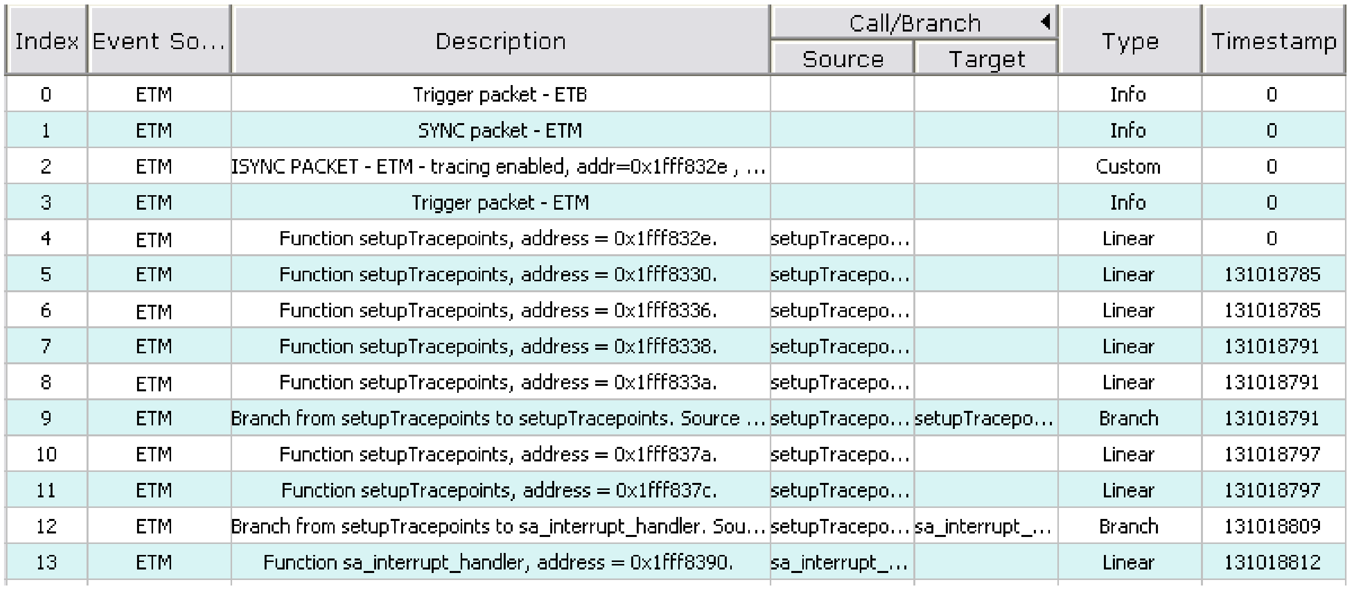
c. Click **Cancel**.
d. In the editor area, select this statement: `for(;;)`
e. Right-click the marker bar, select the **Toggle Trace Stop Point > Software Trace Point** option from the context menu.

NOTE

It is recommended to set Inlining as **Off** before you debug the project. To set Inlining as **Off** , right-click 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 list on the right-side of the tab.

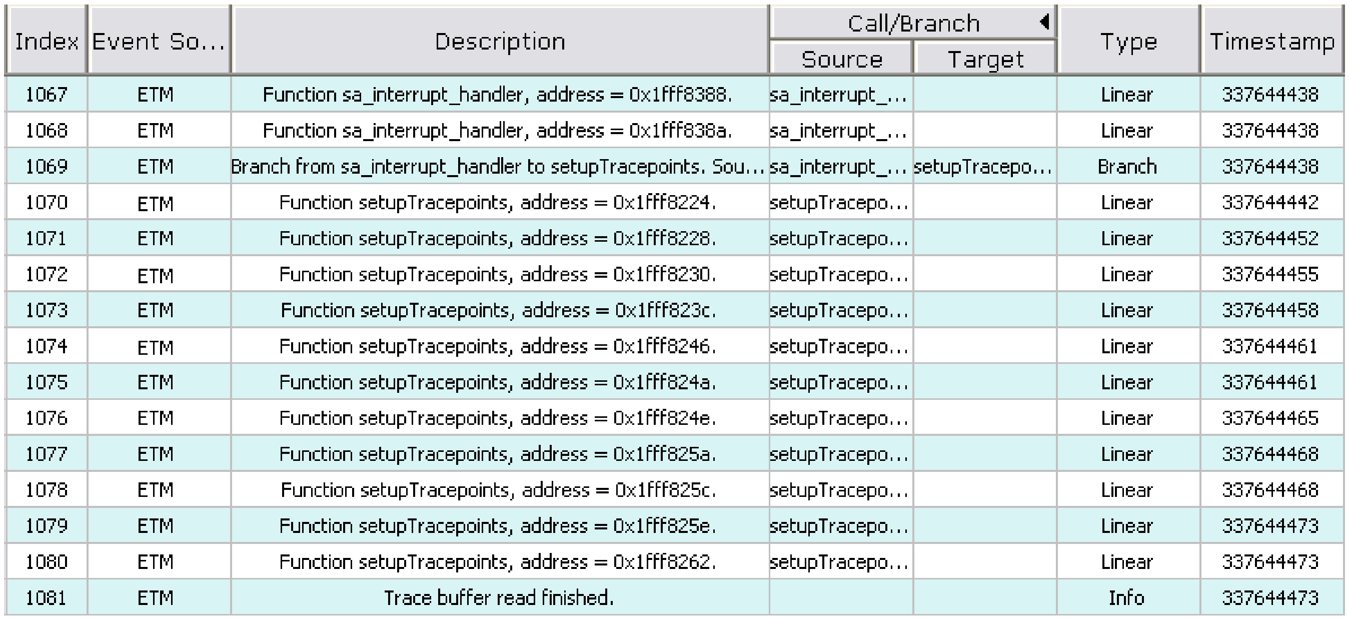
10. Debug the application.
11. Collect trace and profile results.
    a. Click **Resume**.
    b. Click **Suspend** after a few seconds.
12. In the **Software Analysis** view, expand the project name and click the **Trace** hyperlink.

    The **Trace Data** viewer appears. The following two figures display 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.



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

Figure 7-16. Trace Results After Setting Software Tracepoints - Beginning of Trace



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

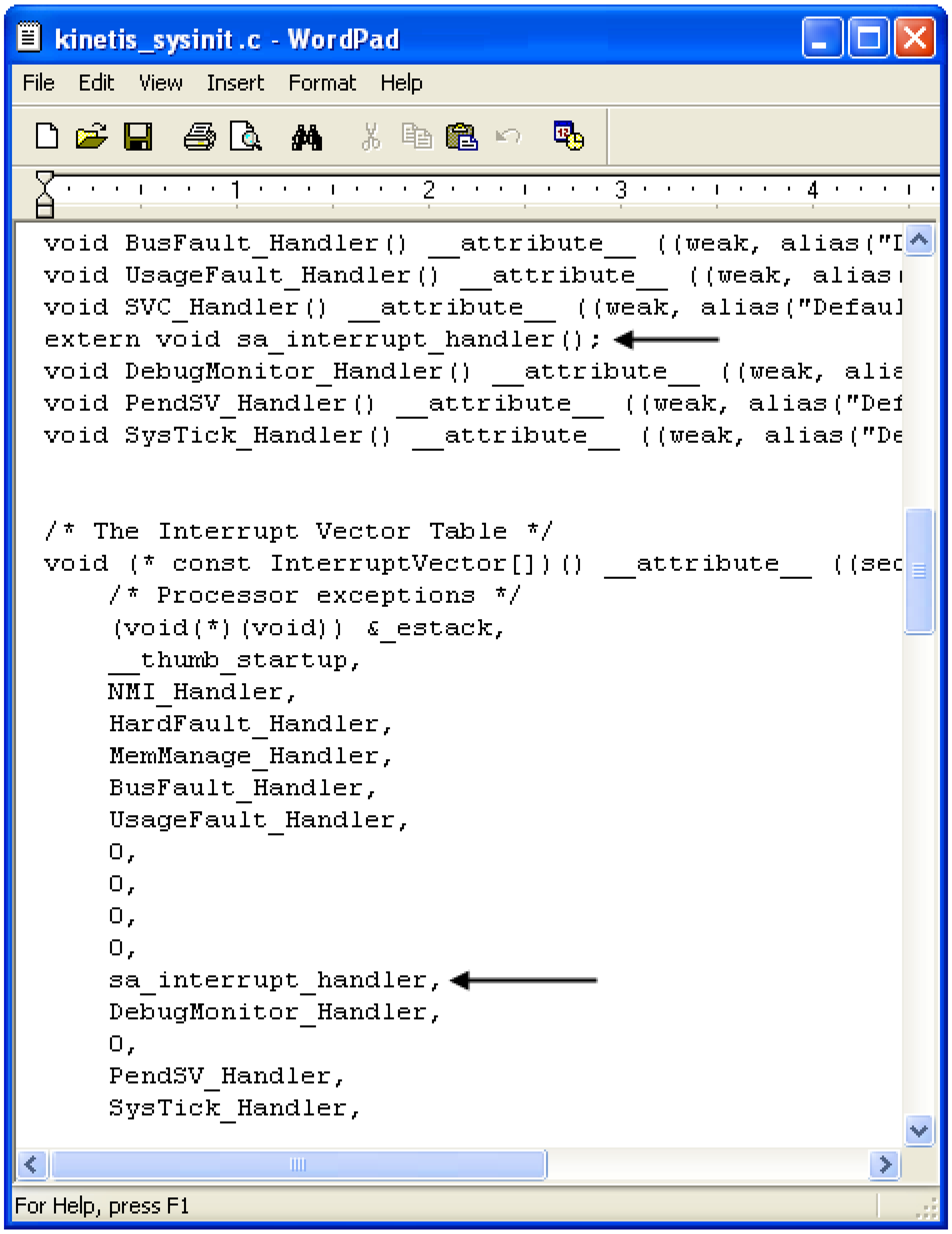
Figure 7-17. Trace Results After Setting Software Tracepoints - End of Trace

This is how you set software tracepoints manually on the Kinetis architecture using the Freescale toolchain.

7.2.1.2 Using GCC Toolchain

To enable software tracepoints manually and collect trace data on the Kinetis target using the GCC toolchain:

1. Create a stationary Kinetis project with the **GCC** option selected in the **Language and Build Tools Options** page..
2. Add the `sa_handler_gcc.c` and `sa_asm_handler_gcc.s` files to your project as explained in Step 2 of Using Freescale Toolchain.
3. Expand **Project_Settings > Startup_Code** in the **CodeWarrior Projects** view, and open the `kinetis_sysinit.c` file in the editor area.
4. Edit the `kinetis_sysinit.c` file. Include this header file in the source code: `#include "sa_handler.h"`. Include the definition of sa interrupt handler: `extern void sa_interrupt_handler();`. And add this line in the interrupt vector table: `sa_interrupt_handler,`.

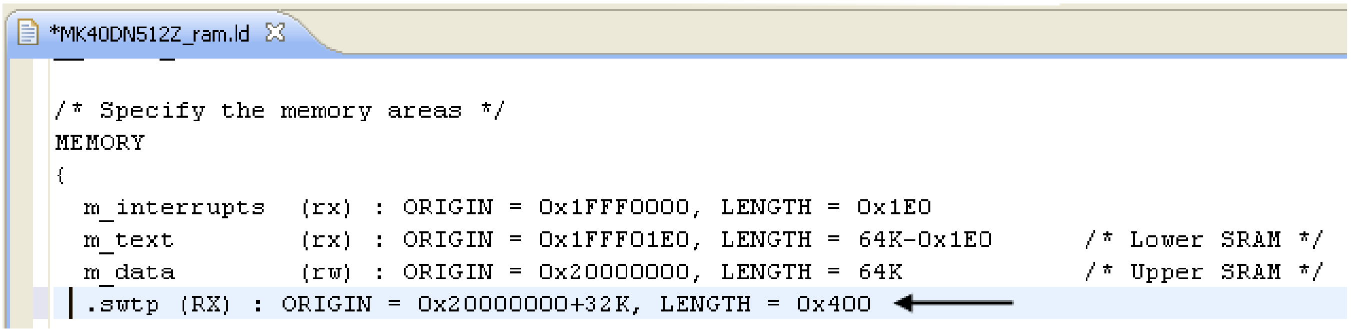

kinetis_sysinit.c - WordPad
File Edit View Insert Format Help
For Help, press F1

```
void BusFault_Handler() __attribute__ ((weak, alias("I
void UsageFault_Handler() __attribute__ ((weak, alias
void SVC_Handler() __attribute__ ((weak, alias("Defaul
extern void sa_interrupt_handler();
void DebugMonitor_Handler() __attribute__ ((weak, alia
void PendSV_Handler() __attribute__ ((weak, alias("Def
void SysTick_Handler() __attribute__ ((weak, alias("De


/* The Interrupt Vector Table */
void (* const InterruptVector[])() __attribute__ ((sec
    /* Processor exceptions */
    (void(*)(void)) &_estack,
    __thumb_startup,
    NMI_Handler,
    HardFault_Handler,
    MemManage_Handler,
    BusFault_Handler,
    UsageFault_Handler,
    0,
    0,
    0,
    0,
    sa_interrupt_handler,
    DebugMonitor_Handler,
    0,
    PendSV_Handler,
    SysTick_Handler,
```

Figure 7-18. Interrupt Vector Table

5. Save the `kinetis_sysinit.c` file.

6. Edit the linker control file of your project.
   a. Expand **Project_Settings > Linker_Files** in the **CodeWarrior Projects** view.
   b. Open the `<project>_ram.ld` file in the editor area.
   c. Add this statement in the memory section: `.swtp (RX) : ORIGIN = 0x20000000+32K, LENGTH = 0x400`



*MK40DN512Z_ram.ld

```
/* Specify the memory areas */
MEMORY
{
  m_interrupts   (rx)  : ORIGIN = 0x1FFF0000, LENGTH = 0x1E0
  m_text         (rx)  : ORIGIN = 0x1FFF01E0, LENGTH = 64K-0x1E0      /* Lower SRAM */
  m_data         (rw)  : ORIGIN = 0x20000000, LENGTH = 64K            /* Upper SRAM */
 .swtp (RX)  : ORIGIN = 0x20000000+32K, LENGTH = 0x400
```

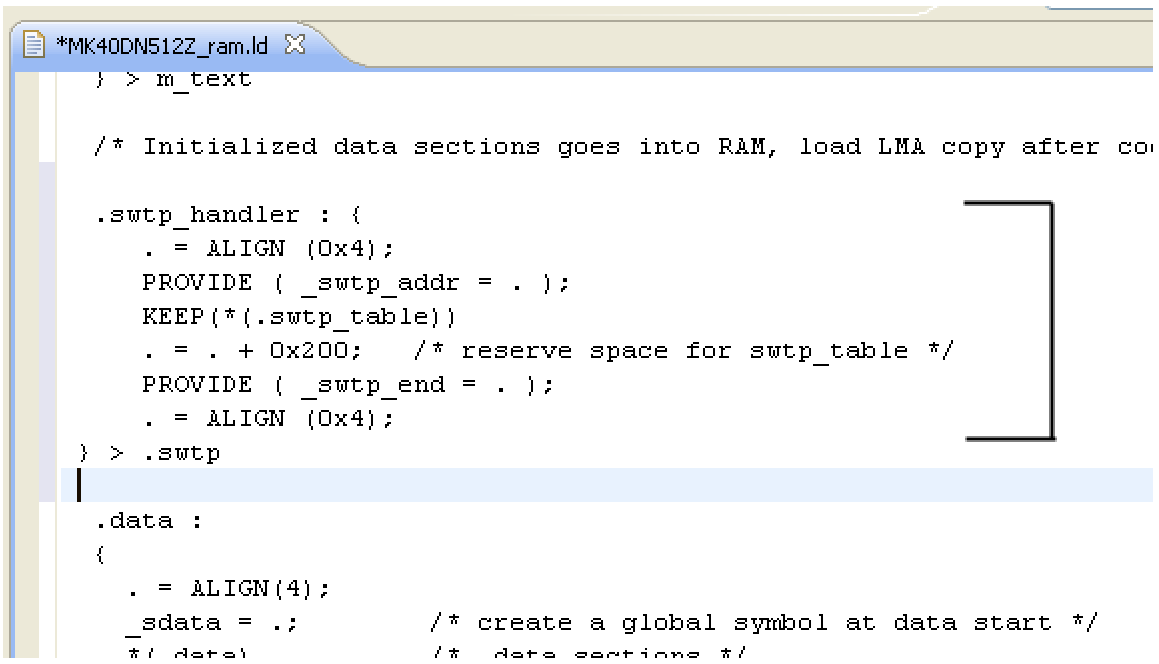
Figure 7-19. Memory Section of Linker File

NOTE

Make sure that you do not change the section name because the Software Analysis tool looks for this exact name when software tracepoints are installed.

   d. Add the `.swtp` section containing the following statements:

```
.swtp_handler: {
   . = ALIGN (0x4);
  PROVIDE ( _swtp_addr = . );
  KEEP(*(.swtp_table))
  . = . + 0x200;   /* reserve space for swtp_table */
  PROVIDE ( _swtp_end = . );
  . = ALIGN (0x4);
} > .swtp
```


*MK40DN512Z_ram.ld

```
  } > m_text

  /* Initialized data sections goes into RAM, load LMA copy after co

  .swtp_handler : {
     . = ALIGN (0x4);
     PROVIDE ( _swtp_addr = . );
     KEEP(*(.swtp_table))
     . = . + 0x200;    /* reserve space for swtp_table */
     PROVIDE ( _swtp_end = . );
     . = ALIGN (0x4);
 } > .swtp

  .data :
  {
    . = ALIGN(4);
    _sdata = .;          /* create a global symbol at data start */
```

Figure 7-20. `.swtp` Section of Linker File When GCC is Selected

e. Save `<project>_ram.ld`.
f. Save and build the project.
g. Follow steps 9 -13 of Using Freescale Toolchain to collect trace.

This is how you set software tracepoints manually on the Kinetis architecture using the GCC toolchain.

7.2.2 Setting Software Tracepoints Automatically

To enable software tracepoints automatically, you can use the **Software Tracepoints Support** dialog box, which appears when you add first start software tracepoint on a project in the source code. You can also invoke the dialog box by selecting the project in **CodeWarrior Projects** view, right-click it, and select the **Profiler > Add software tracepoint support** option.

This dialog box:

- creates a memory zone for the software tracepoints table marked with `.swtp_table` symbol
- updates the interrupt vector to add sa handler for svc interrupt, and
- adds the source file with sa handler which makes the trace settings
- creates the `sa_backup` folder which contains the original LCF file along with the log files displaying the details of the operation performed. The `sa_backup` folder is created inside the project at the same level as the `Source` folder.

NOTE

If you remove the `sa_backup` folder, the LCF file will not be restored to the initial form before adding software tracepoint support.

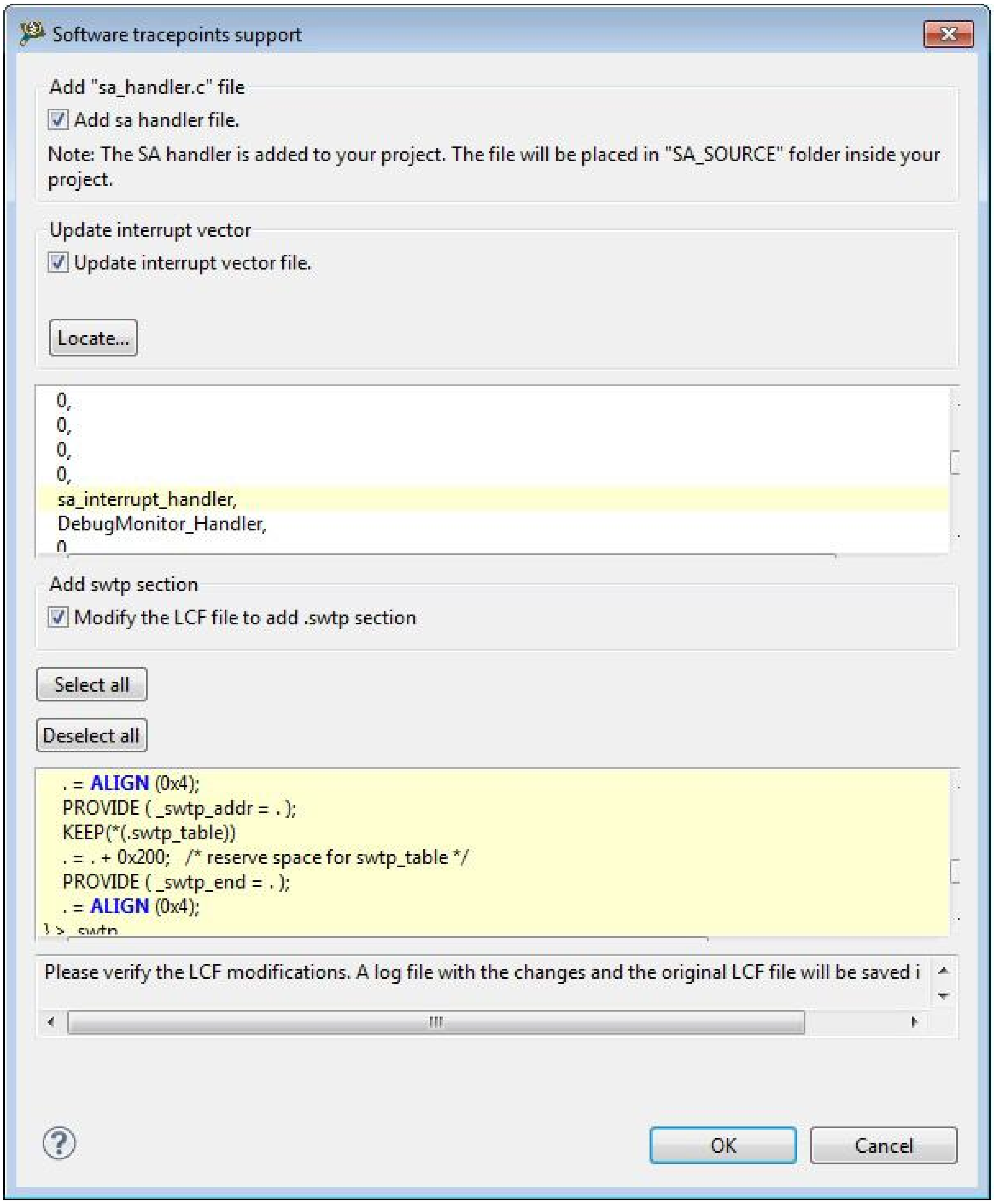
Perform the following steps to enable software tracepoints automatically and collect trace data. The steps are same for Freescale as well as GCC toolchain.

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. Check **Continuous Trace collection** and uncheck all other checkboxes.
4. Click **Apply** to save the settings, and close the **Debug Configurations** dialog box.
5. In the editor area, select this statement: `int counter = 0;`
6. Right-click 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 tracepoint from the marker bar.

   The **Software Tracepoints Support** dialog box appears.

NOTE

If you click **Cancel** in the **Software Tracepoints Support** dialog box, the start tracepoint is set in the source code, but you are required to enable the software tracepoints functionality manually as discussed in Setting Software Tracepoints Manually.



Note: The SA handler is added to your project. The file will be placed in "SA_SOURCE" folder inside your project.

```
0,
0,
0,
0,
sa_interrupt_handler,
DebugMonitor_Handler,
0
```

```
  . = ALIGN (0x4);
  PROVIDE ( _swtp_addr = . );
  KEEP(*(.swtp_table))
  . = . + 0x200;   /* reserve space for swtp_table */
  PROVIDE ( _swtp_end = . );
  . = ALIGN (0x4);
} > .swtp
```

Please verify the LCF modifications. A log file with the changes and the original LCF file will be saved i

Figure 7-21. Software Tracepoints Support Dialog Box

7. Click **Locate** to search for the source file where the interrupt vector is declared.
8. Select `kinetis_sysinit.c` file in the dialog box that appears.
9. Click **OK** in the **Software Tracepoints Support** dialog box.
10. In the editor area, select this statement: `for(;;)`
11. Right-click the marker bar, select the **Toggle Trace Stop Point > Software Trace Point** option from the context menu.

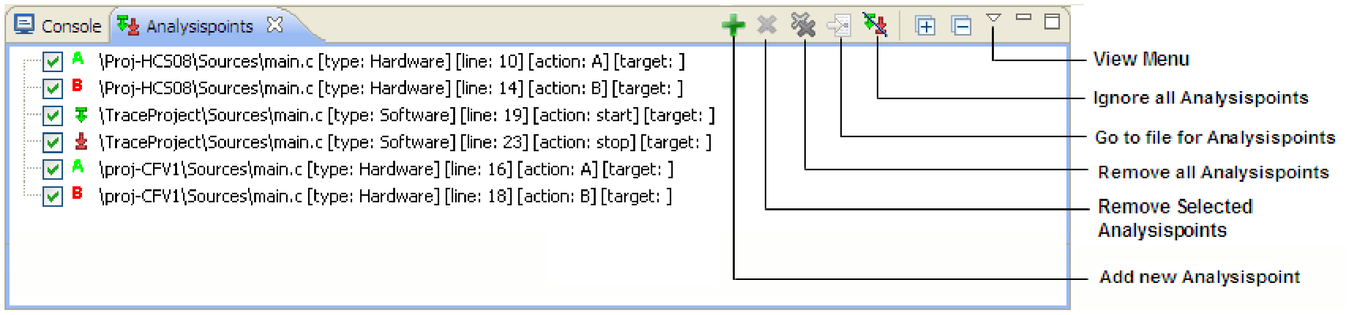
12. Debug the application.
13. Collect trace and profile results.
    a. Click **Resume**.
    b. Click **Suspend** after a few seconds.
14. In the **Software Analysis** view, expand the project name and click the **Trace** hyperlink.

    The **Trace Data** viewer appears.

This is how you set software tracepoints automatically on the Kinetis architecture.

7.3 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.


Console
Analysispoints
View Menu
Ignore all Analysispoints
Go to file for Analysispoints
Remove all Analysispoints
Remove Selected Analysispoints
Add new Analysispoint

```
A \Proj-HCS08\Sources\main.c [type: Hardware] [line: 10] [action: A] [target: ]
B \Proj-HCS08\Sources\main.c [type: Hardware] [line: 14] [action: B] [target: ]
  \TraceProject\Sources\main.c [type: Software] [line: 19] [action: start] [target: ]
  \TraceProject\Sources\main.c [type: Software] [line: 23] [action: stop] [target: ]
A \proj-CFV1\Sources\main.c [type: Hardware] [line: 16] [action: A] [target: ]
B \proj-CFV1\Sources\main.c [type: Hardware] [line: 18] [action: B] [target: ]
```

Figure 7-22. Analysispoints View

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

- Name and path 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 and path 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:

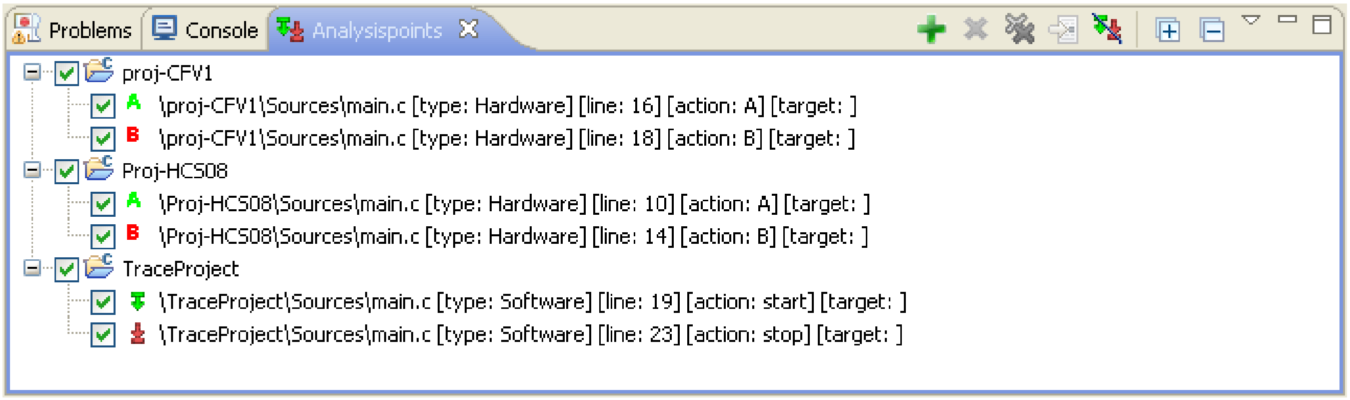
- View Full Path of Tracepoint Attribute
- Group Tracepoints
- Define Working Sets
- Add New Analysispoint
- Enable/Disable Tracepoints
- Navigate to Tracepoint Line
- Remove Tracepoints
- Context Menu

7.3.1 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.

7.3.2 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. The following figure shows the tracepoint attributes in the **Analysispoints** view which are grouped by projects. You can ungroup the attributes by selecting the same option again.



proj-CFV1
A \proj-CFV1\Sources\main.c [type: Hardware] [line: 16] [action: A] [target:]
B \proj-CFV1\Sources\main.c [type: Hardware] [line: 18] [action: B] [target:]
Proj-HCS08
A \Proj-HCS08\Sources\main.c [type: Hardware] [line: 10] [action: A] [target:]
B \Proj-HCS08\Sources\main.c [type: Hardware] [line: 14] [action: B] [target:]
TraceProject
\TraceProject\Sources\main.c [type: Software] [line: 19] [action: start] [target:]
\TraceProject\Sources\main.c [type: Software] [line: 23] [action: stop] [target:]

Figure 7-23. Analysispoints View - Group by Projects

7.3.3 Define Working Sets

Working sets allow you to create named groups for the tracepoints. You can define working sets for the selected tracepoints and then group the tracepoints by working sets. To create a working set:

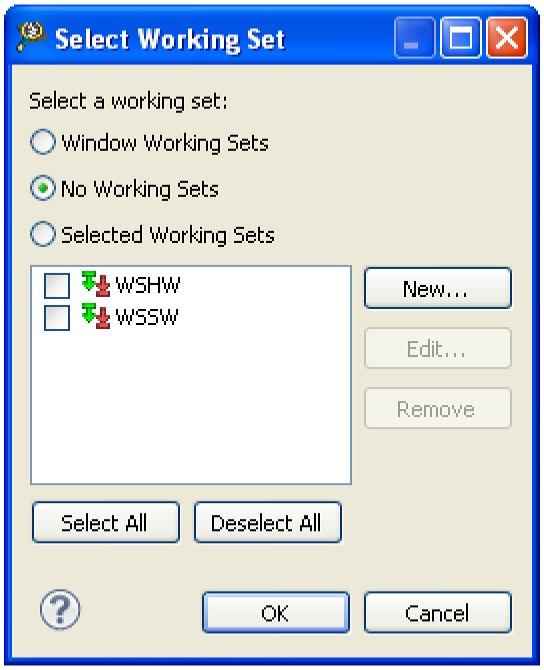
1. Click the drop-down in the **Analysispoints** view and select the **Working Sets** option.

   The **Select Working Set** dialog box appears.

2. Click **New** to create a new working set.

   The **New Working Set** dialog box appears.

3. Type a name for the working set in the **Working Set Name** textbox.
4. Select the tracepoints in the **Working Sets** list that you want to add to the new working set. For example, you can select hardware tracepoints for one working set and software tracepoints for another working set.
5. Click **Finish**.
   The new working set appears in the **Select Working Set** dialog box.

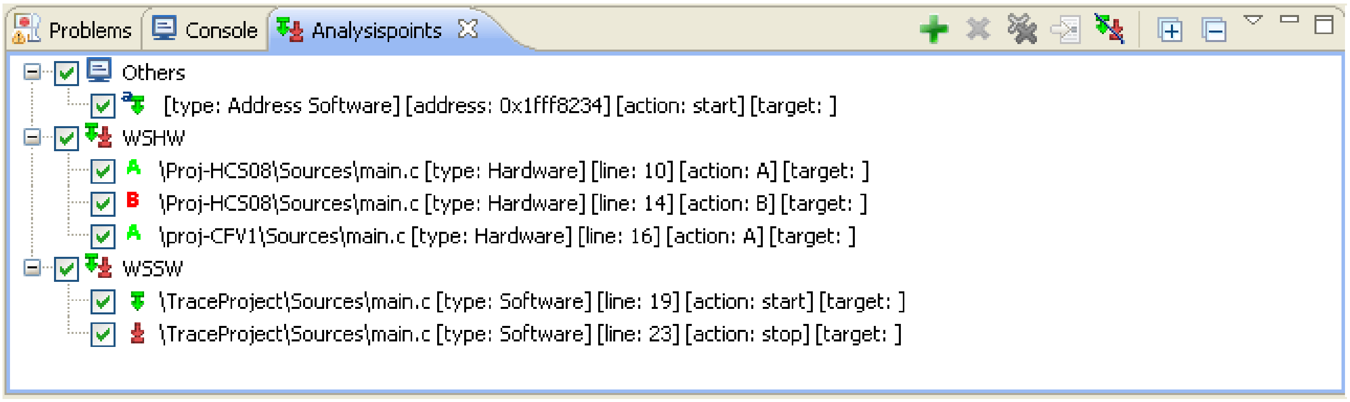


Select a working set:
Window Working Sets
[✓ No Working Sets]
Selected Working Sets
☐ WSHW
☐ WSSW

Figure 7-24. Select Working Set Dialog Box

6. Click **OK** to close the dialog box.

To group tracepoints by working sets, click the drop-down in the **Analysispoints** view, and select **Group By > AnalysispointWorking Sets**. The tracepoints grouped by the working sets appear in the **Analysispoints** view.



```
Others
    [type: Address Software] [address: 0x1fff8234] [action: start] [target: ]
WSHW
    \Proj-HCS08\Sources\main.c [type: Hardware] [line: 10] [action: A] [target: ]
    \Proj-HCS08\Sources\main.c [type: Hardware] [line: 14] [action: B] [target: ]
    \proj-CFV1\Sources\main.c [type: Hardware] [line: 16] [action: A] [target: ]
WSSW
    \TraceProject\Sources\main.c [type: Software] [line: 19] [action: start] [target: ]
    \TraceProject\Sources\main.c [type: Software] [line: 23] [action: stop] [target: ]
```

Figure 7-25. Analysispoints Grouped by Working Sets

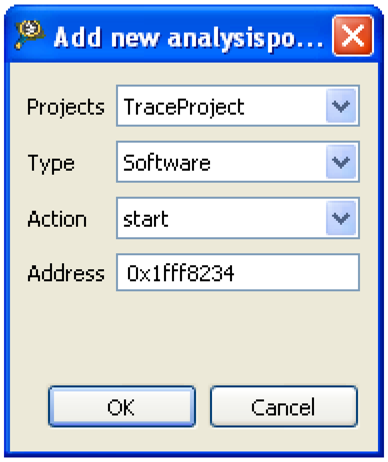
You can also set a working set as the default one so that whenever you add any stop or start tracepoint to your application, it gets automatically added to the default working set. In this way, working sets help you work on a particular set of tracepoints at a particular time. Click the drop-down in the **Analysispoints** view and select the **Select Default Working Set** option. It opens the **Select Default Working Set** dialog box where you can set one of the available tracepoints working sets as the default one. Alternatively, right-click the working set, and select the **ToggleDefault.label** option from the context menu to set the working set as default.

You can deselect the working set which you set as default, using the **Deselect Default Working Set** option from the drop-down or selecting the **ToggleDefault.label option** from the context menu.

7.3.4 Add New Analysispoint

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

1. Click the **Add new Analysispoint** icon to display the **Add New Analysispoint** dialog box.

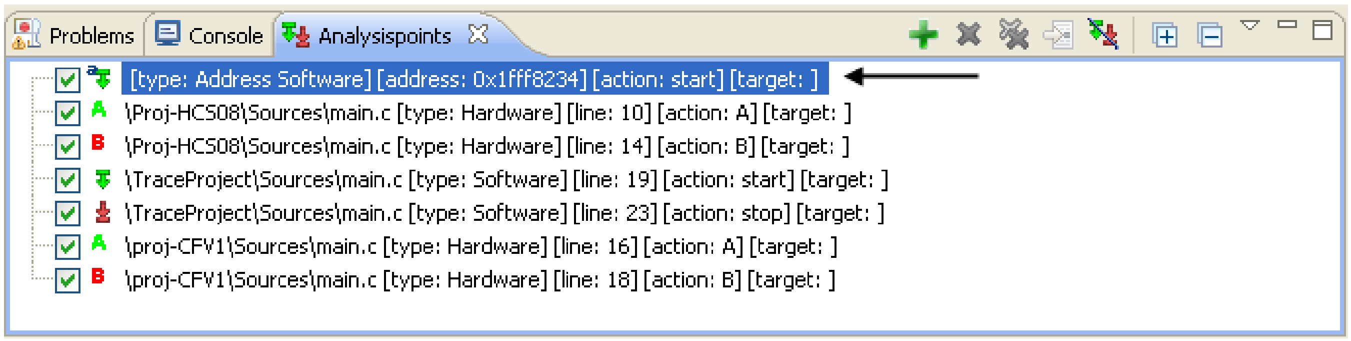

Add new analysispo...

Projects: TraceProject
Type: Software
Action: start
Address: 0x1fff8234

Figure 7-26. Add New Analysispoint Dialog Box

2. Select the required project from the **Projects** text box.
3. Select the type of the tracepoint from the **Type** text box..
4. Select the action of the tracepoint, that is whether start or stop, from the **Action** text box.
5. Enter the address where you want to set the tracepoint in the **Address** text box
6. Click **OK**.

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



```
[type: Address Software] [address: 0x1fff8234] [action: start] [target: ]
\Proj-HCS08\Sources\main.c [type: Hardware] [line: 10] [action: A] [target: ]
\Proj-HCS08\Sources\main.c [type: Hardware] [line: 14] [action: B] [target: ]
\TraceProject\Sources\main.c [type: Software] [line: 19] [action: start] [target: ]
\TraceProject\Sources\main.c [type: Software] [line: 23] [action: stop] [target: ]
\proj-CFV1\Sources\main.c [type: Hardware] [line: 16] [action: A] [target: ]
\proj-CFV1\Sources\main.c [type: Hardware] [line: 18] [action: B] [target: ]
```

Figure 7-27. Address Tracepoint Set in Analysispoints View

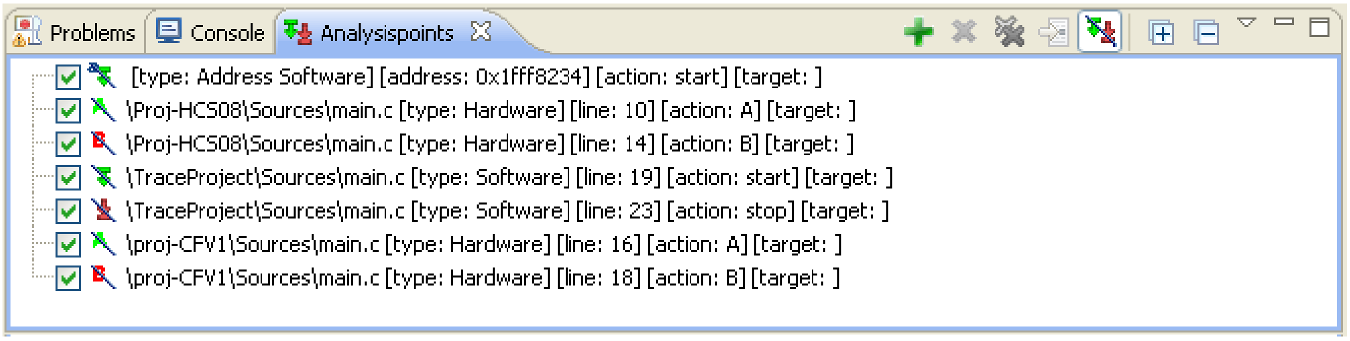
NOTE

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.

7.3.5 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. Click **Ignore All Analysispoints** again to enable the tracepoints.



```
[type: Address Software] [address: 0x1fff8234] [action: start] [target: ]
\Proj-HCS08\Sources\main.c [type: Hardware] [line: 10] [action: A] [target: ]
\Proj-HCS08\Sources\main.c [type: Hardware] [line: 14] [action: B] [target: ]
\TraceProject\Sources\main.c [type: Software] [line: 19] [action: start] [target: ]
\TraceProject\Sources\main.c [type: Software] [line: 23] [action: stop] [target: ]
\proj-CFV1\Sources\main.c [type: Hardware] [line: 16] [action: A] [target: ]
\proj-CFV1\Sources\main.c [type: Hardware] [line: 18] [action: B] [target: ]
```

Figure 7-28. Disabling Tracepoints from Analysispoints View

7.3.6 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.

7.3.7 Remove Tracepoints

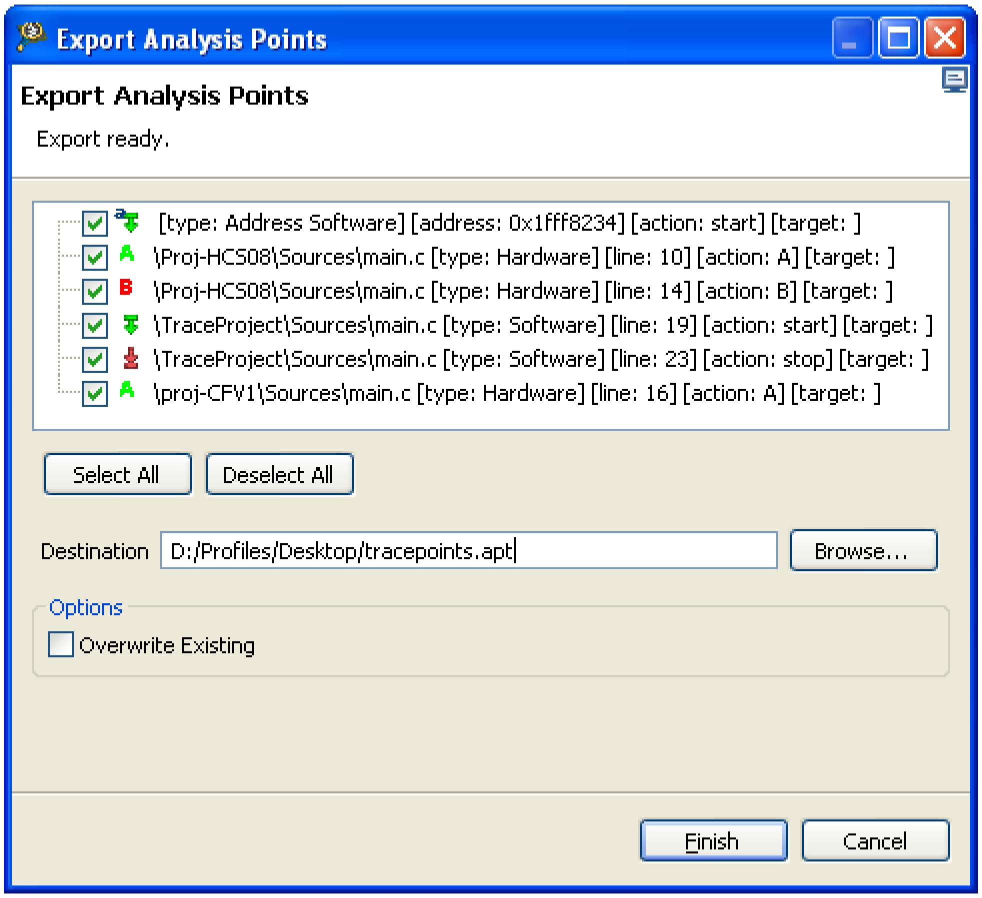
To remove a tracepoint, select the tracepoint attribute and click the **Remove Selected Analysispoint** icon. To remove all the tracepoints, click the **Remove All Analysispoints** icon.

7.3.8 Context Menu

Right-click in the **Analysispoints** view or on the tracepoint to display a context menu, which allows you to perform the following actions:

- Go to File — Navigates you to the specific line of the source file where you have set the selected tracepoint.

- Enable/Disable — Allows you to enable/disable tracepoints. A disabled tracepoint will have no effect during the collection of trace data.
- Remove — Removes the selected tracepoint.
- Remove All — Removes all tracepoints that are displayed in the **Analysispoints** view.
- Select All — Selects all tracepoints in the **Analysispoints** view.
- Copy — Copies the selected tracepoint on the clipboard.
- Set Cores — Lets you select the cores for which you want to set tracepoints. This feature is applicable for multicore tracing, which is not yet supported in CodeWarrior Microcontrollers.
- Export Analysispoints — Lets you save the tracepoints in a .apt file at a desired location. Select this option to display the **Export Analysis Points** dialog box. Select the tracepoint(s) that you want to save, and click **Browse** to navigate to a location where you want to save the tracepoint(s), in the **Save Analysis Points As** dialog box. Specify a filename in which the tracepoints will be exported. The full path will appear in the **Destination** text box. Click **Finish**.


Export Analysis Points

Export ready.

- [✓] [type: Address Software] [address: 0x1fff8234] [action: start] [target:]
- [✓] \Proj-HCS08\Sources\main.c [type: Hardware] [line: 10] [action: A] [target:]
- [✓] \Proj-HCS08\Sources\main.c [type: Hardware] [line: 14] [action: B] [target:]
- [✓] \TraceProject\Sources\main.c [type: Software] [line: 19] [action: start] [target:]
- [✓] \TraceProject\Sources\main.c [type: Software] [line: 23] [action: stop] [target:]
- [✓] \proj-CFV1\Sources\main.c [type: Hardware] [line: 16] [action: A] [target:]

Destination: D:/Profiles/Desktop/tracepoints.apt

Options

- [ ] Overwrite Existing

Figure 7-29. Export Analysis Points Dialog Box

- Import Analysispoints — Lets you import analysispoints from another location in the **Analysispoints** view. Select this option to display the **Import Analysis Points** dialog box. Click **Browse** to locate the .apt file where you saved the analysispoints. Click **Finish**. The analysispoints will be imported in the **Analysispoints** view.

Chapter 8
Data Visualization

Data visualization allows you to visualize during runtime the evolution of certain application data, such as variables, registers, memory, which represent parameters defining the functionality of certain equipment, for example, a motor.

Data Visualization is available for DSC architectures only. Data visualization helps DSC users identify and chart variables against time and other variables. Data Visualization:

- samples the values of registers, variables and/or raw memory data as the application is running
- displays the collected data in a chart as the application is running
- samples the required data without stopping the core if the target allows or allows you to set visualization points (special type of breakpoints) which would automatically stop the target, read the required data and resume the target immediately.

NOTE

The JTAG interface on the DSC is powerful enough to extract the value of variables. To do this, a special Visualization Breakpoint is set. When this breakpoint is reached, the JTAG interface relays the information about the variables being monitored so that they can be graphed.

This chapter consists of the following topics:

- Creating DSC Project
- Configuring for Data Visualization
- Setting Analysispoints for Data Visualization
- Collecting and Viewing Data

8.1 Creating DSC Project

To create a DSC project:

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

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

2. Enter the name of your project in the **Project Name** text box and click **Next**.

   The **Devices** page appears.

3. Select the target device or board for your project from the DSC family. For example, select **56800/E (DSC) > MC56F825X > MC56F8257**.
4. Click **Next**.

   The **Connections** page appears.

5. Select the available connection.
6. Click **Next** two times and then click **Finish**.

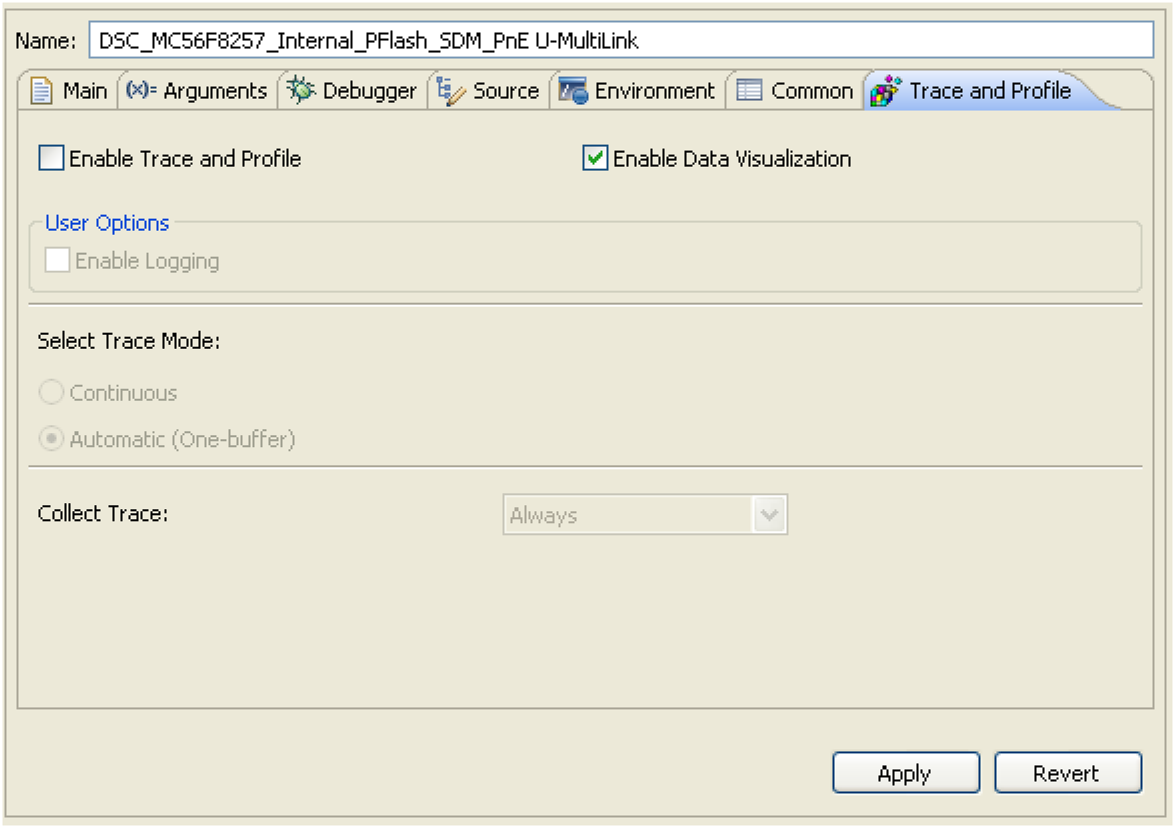
The project is created and appears in the **CodeWarrior Projects** view. You can now build your project.

8.2 Configuring for Data Visualization

To configure your DSC project for data visualization:

1. Open the **Debug Configurations** dialog box.
2. Click the **Trace and Profile** tab.
3. Check the **Enable Data Visualization** checkbox.

The options corresponding to Trace and Profile get disabled. This is because Data Visualization and tracing cannot be used simultaneously. Trace collection and Data Visualization sampling may be used together on the platforms which support reading target memory and registers while running. In this case, both checkboxes will be enabled and may be selected at the same time.



Name: DSC_MC56F8257_Internal_PFlash_SDM_PnE U-MultiLink

Main | Arguments | Debugger | Source | Environment | Common | Trace and Profile

☐ Enable Trace and Profile ☑ Enable Data Visualization

User Options

☐ Enable Logging

Select Trace Mode:

○ Continuous

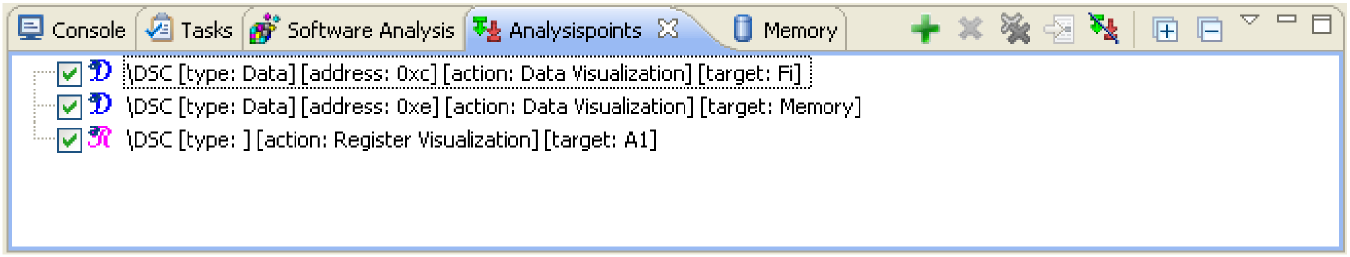
● Automatic (One-buffer)

Collect Trace: Always

Figure 8-1. Trace and Profile Tab - DSC Architecture

8.3 Setting Analysispoints for Data Visualization

To select the data to be visualized, you need to set the analysispoints. Two types of analysispoints are set, Data analysispoints and Register analysispoints, which appear in the **Analysispoints** view. You can set data analysispoints on **Memory** or **Variables** view, while the register analysispoints are set on the **Registers** view.



```
☑ \DSC [type: Data] [address: 0xc] [action: Data Visualization] [target: Fi]
☑ \DSC [type: Data] [address: 0xe] [action: Data Visualization] [target: Memory]
☑ \DSC [type: ] [action: Register Visualization] [target: A1]
```

Figure 8-2. Analysispoints View with Data and Register Analysispoints

This topic contains the following sub-topics:

- Setting Data Analysispoints on Memory View

- Setting Data Analysispoints on Variables View
- Setting Register Analysispoints on Registers View

8.3.1 Setting Data Analysispoints on Memory View

To set data analysispoints for visualization on the **Memory** view:

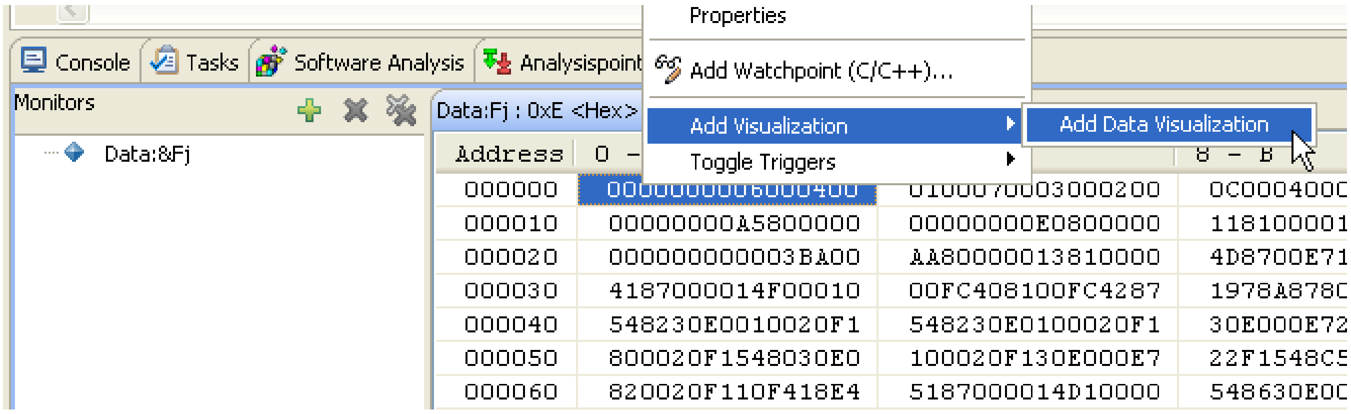
1. Debug your application.
2. Open the **Variables** view and select the required variable on which you want to set the data analysispoint.
3. Right-click and select the **View Memory** option from the context menu.

   The **Memory** view appears with memory renderings opened.

4. Look for the required address among the cells displayed in the **Memory** view. The required address is the address of the selected variable in the **Variables** view.

NOTE

It is recommended to change the memory rendering cell formatting to a smaller value, for example, 2 bytes, if the required address cannot be found among the cells being displayed by default. To change the format value, right-click the selected cell, select **Format** and specify **Column Size** as 2. For details, see Figure 8-4.



```
Address   0 -                                  8 - B
000000    0000000006000400   0100070003000200   0C000400C
000010    00000000A5800000   00000000E0800000   118100001
000020    000000000003BA00   AA80000013810000   4D8700E71
000030    4187000014F00010   00FC408100FC4287   1978A878C
000040    548230E0010020F1   548230E0100020F1   30E000E72
000050    800020F1548030E0   100020F130E000E7   22F1548C5
000060    820020F110F418E4   5187000014D10000   548630E0C
```

Figure 8-3. Memory View

5. Right-click the cell which has the required address, and select **Add Visualization > Add Data Visualization** from the context menu.

   The **Add Data Visualization** dialog box appears.

6. Select the size of the memory cell to be visualized from the **Size** drop-down list.

7. Click **OK** to add a memory data analysispoint to the **Analysispoints** view.

This sets data analysispoints for visualization on the **Memory** view.

8.3.2 Setting Data Analysispoints on Variables View

To set data analysispoints for visualization on the **Variables** view:

1. Debug your application.
2. Open the **Variables** view.
3. Select the required variable on which you want to set the data analysispoint.

NOTE
By default, the global variables are not displayed in the **Variables** view. To view global variables, right-click and select **Add Global Variables** from the context menu. In the **Add Globals** dialog box that appears, select the variables that you want to add and click **OK**.

4. Right-click and select **Add Visualization > Add Data Visualization** from the context menu.

   The **Add Data Visualization** dialog box appears.

5. Select the size of the memory cell to be visualized from the **Size** drop-down list.
6. Click **OK** to add a variable data analysispoint to the **Analysispoints** view.

This sets data analysispoints for visualization on the **Variables** view.

8.3.3 Setting Register Analysispoints on Registers View

To set register analysispoints for visualization on the **Variables** view:

1. Debug your application.
2. Open the **Registers** view.
3. Select the required register on which you want to set the register analysispoint.
4. Right-click and select **Add Visualization > Add Data Visualization** from the context menu.

   The **Add Data Visualization** dialog box appears.

5. Select the size of the memory cell to be visualized from the **Size** drop-down list.
6. Click **OK** to add a register analysispoint to the **Analysispoints** view.

This sets data analysispoints for visualization on the **Registers** view.

8.4 Collecting and Viewing Data

This section demonstrates data visualization by using an example.

To collect and view data for data visualization:

1. Create a DSC project with the source code shown below.
   Listing: Sample source code used for data visualization

```
#include <stdio.h>

#include <stdlib.h>


#define SIZE 10


#define NO_PRINT


// prototypes


void swap(int *a, int *b);


void print_array(int arr[], int length);


int i = 0;


long j = 0;


int arr[SIZE] = { 4, 6, 7, 1, 2, 3, 4, 12, 4, 5 };


int main(void) {


    #ifndef NO_PRINT


       printf("\n\n\n======================================\n");
```

```
        printf("======================================\n\n");

        print_array(arr, SIZE);

    #endif

for (i = 0; i < SIZE - 1; i++)

    for (j = i; j < SIZE; j++)
        if (arr[i] > arr[j])
            swap(&arr[i], &arr[j]);

#ifndef NO_PRINT

    print_array(arr, SIZE);

    printf("\n\n... program done.\n");

#endif


  return (0);


}


void print_array(int arr[], int length) {


    int i;


    printf("Array = ");


    for (i = 0; i < length; i++) {


        printf("%d ", arr[i]);


    }

    printf("\n");


}
```

```
void swap(int *a, int *b) {

   int c = *a;

   *a = *b;

   *b = c;

}
```

2. Save and build your project.
3. Debug the project.
4. Set data analysispoints.
   a. Open the **Variables** view. Right-click and select the **Add Global Variables** option from the context menu.

      The **Add Globals** dialog box appears.

   b. Select all variables `Farr`, `Fi`, `Fj` and click **OK**.
   c. Select `Fi` and select **Add Visualization > Add Data Visualization**.

      The **Add Data Visualization** dialog box appears.

   d. Accept the default settings and click **OK**.
   e. Select `Fj`, right-click and select **View Memory** from the context menu.

      The **Memory** view appears with memory renderings opened.

   f. Right-click the selected cell in the **Memory** view, and select the **Format** option from the context menu.

      The **Format** dialog box appears.

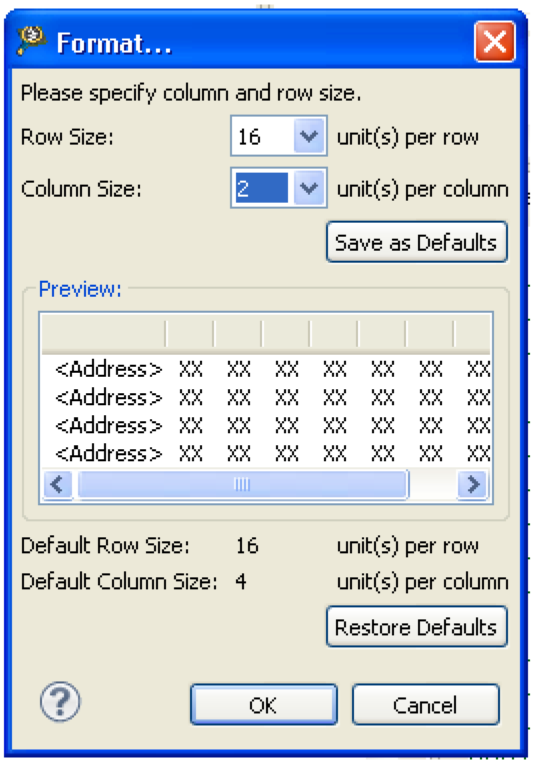

Format...
Please specify column and row size.
Row Size: 16 unit(s) per row
Column Size: 2 unit(s) per column
Save as Defaults
Preview:
<Address> XX XX XX XX XX XX XX
<Address> XX XX XX XX XX XX XX
<Address> XX XX XX XX XX XX XX
<Address> XX XX XX XX XX XX XX
Default Row Size: 16 unit(s) per row
Default Column Size: 4 unit(s) per column
Restore Defaults
OK
Cancel

Figure 8-4. Format Dialog Box

g. Select **Column Size** as **2** and click **OK**.
h. Right-click again in the **Memory** view on the cell that displays the required address (that is the address of the `Fj` variable in the **Variables** view) and select **Add Visualization > Add Data Visualization** from the context menu.

   The **Add Data Visualization** dialog box appears.

i. Select the size of the memory cell as 4 bytes in the **Size** drop-down list.
j. Click **OK** to add the memory data analysispoint to the **Analysispoints** view.

5. Set the register analysispoint.
   a. Open the **Registers** view.
   b. Expand **Core Registers > A1** , right-click, and select **Add Visualization > Add Data Visualization**.

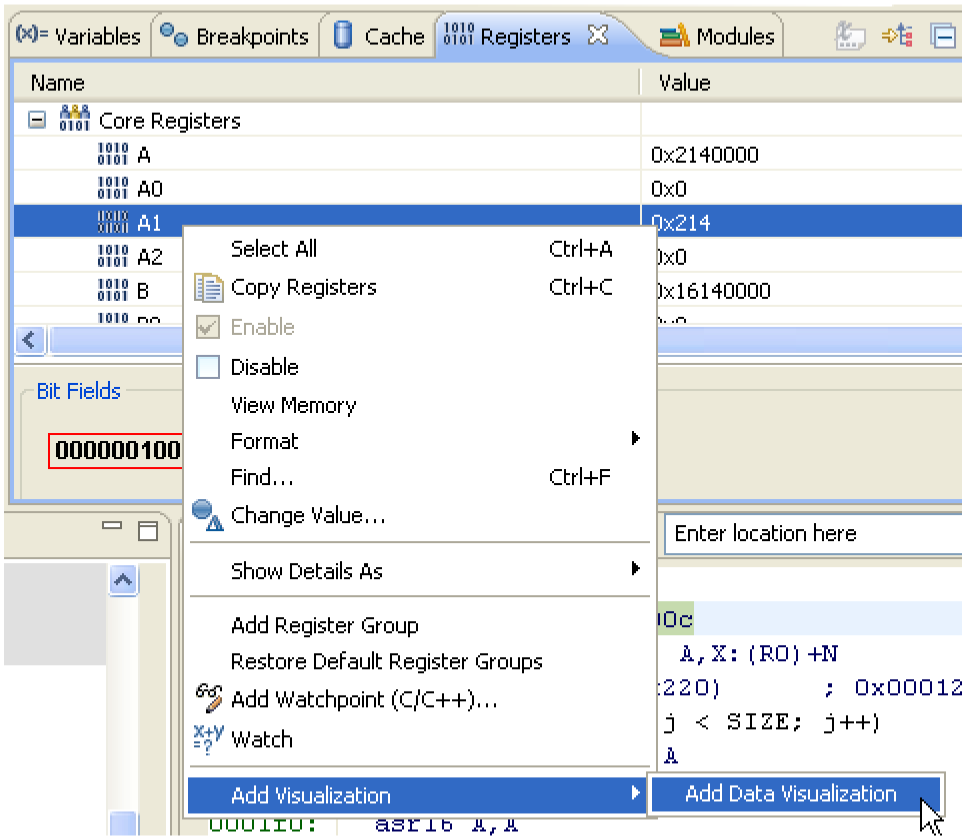


Figure 8-5. Registers View

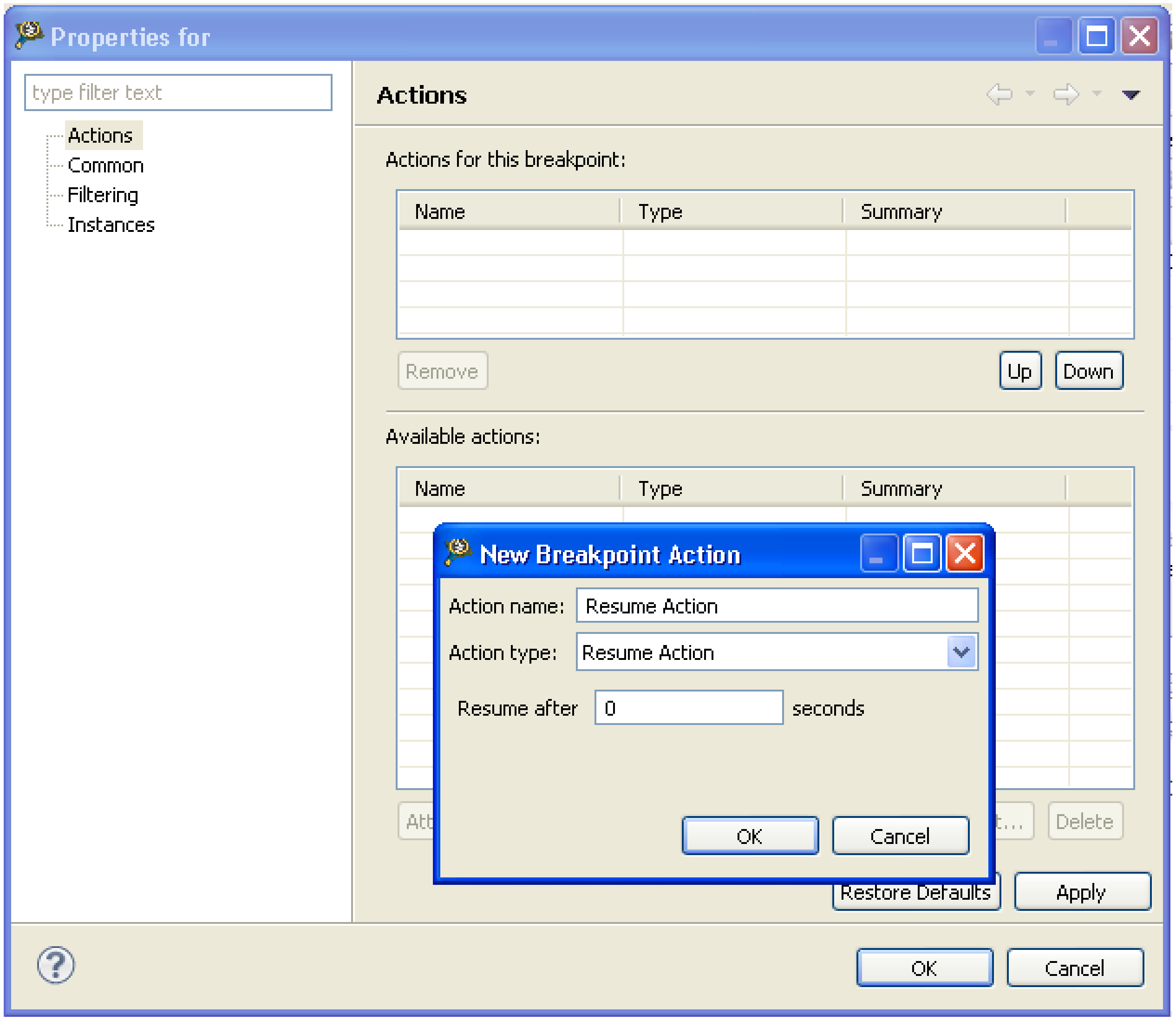
The **Add Data Visualization** dialog box appears.

c. Accept the default settings and click **OK**.

6. Open the **Analysispoints** view and see data and register analysis points that you have set.
7. Set data visualization breakpoints.
   a. Set a breakpoint on the line `swap(&arr[i], &arr[j]);`.
   b. Right-click the breakpoint and select **Breakpoint Properties** from the context menu.

      The **Properties** dialog box appears.
   c. Click **New** under **Available Actions** group to add a new resume action.

      The **New Breakpoint Action** dialog box appears.



Action name: Resume Action

Action type: Resume Action

Resume after 0 seconds

Figure 8-6. New Breakpoint Action Dialog Box

d. Type a name for the new resume action, and select **Action type** as **Resume Action**.

e. Keep 0 seconds in the **Resume after** textbox and click **OK**.

The resume action appears in the **Available actions** table in the **Properties** dialog box.

f. Click **Attach** to attach the resume action with the breakpoint.

g. Set another breakpoint on the line `return 0;` with no action attached to the breakpoint.

8. Resume the application.

As the application runs and reaches the second breakpoint, the **Data Visualization** page opens automatically. It displays the last values in a SWT chart read for each visualization analysispoint.

9. Terminate the application.

   The **Software Analysis** view opens and populates with a new result set. Only **Trace** , **Data Visualization** and **Log** are visible. The **Data Visualization** chart populates with the full set of collected values.

   NOTE

   The sampled data can also be seen as a data trace, with Data Read events for memory data and Profile Counters events for variable and register values.

10. Open the **Trace Data** viewer.

    The **Trace Data** viewer contains the same data.

11. Select another metric for the x-axis, for example, `Fj` and view results related to it.

    NOTE

    The metric versus time is displayed as lines with symbols, while metric versus metric is displayed as symbols only.

For more information, refer Data Trace Import Dialog Box.

Chapter 9
Launching Scripts

The Profiling and Analysis tools help you launch scripts written in Python language. These scripts allow you to collect and export trace data automatically using remote launch. This feature is integrated with CodeWarrior IDE scripting. The scripts are launched in remote launch using the Jython console. The remote launch feature of CodeWarrior allows launch configurations to be executed remotely.

NOTE

The Profiling and Analysis tools specific scripting APIs are available in the `SA_Scripting.doc` document, which is located at `<CW Installation Directory>\MCU\morpho_sa\sasdk\support.`

You can export the data in a CSV file or set the tracepoints in the source code by launching a Python script. You can write the script to contain all details required to configure tracing and profiling. For example, enabling trace and profile, setting breakpoints, launching the application, resuming, suspending, and terminating. You can also set the trace mode, for example, automatic or continuous for trace collection in the script itself. You can also modify the sample Python scripts according to your needs to extend the functionality to other platforms by changing the configuration attributes. After writing these scripts, you run them in the Jython console and collect trace.

This chapter consists of the following topics:

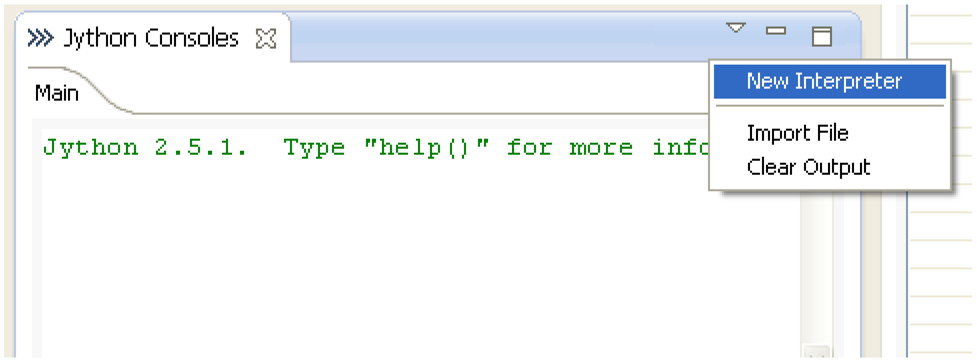
- Run Sample Python Script
- Collect Trace Using Jython
- Export Trace to CSV File
- Modify Sample Python Script

9.1 Run Sample Python Script

The steps in this topic demonstrate the procedure of running a sample Python script from Jython console, that is running the script that you might have written for exporting trace data or collecting trace data.

1. Copy the Python scripts, which you want to execute, at `<CW installation directory>\eclipse\plugins\com.freescale.sa.mcu.scripting_*\remote\scripting_sa_mcu`.
2. Select **Window > Show View > Other > Debug > Remote Launch to open the Remote Launch** view.
3. Click the **Enable Remote Launch** button displayed on the top right of the view to enable remote launching.

   The color of the button changes to red.
4. Select **Window > Show View > Other > Debug > Jython Consoles** to open the **Jython Consoles** view.
5. Open the **View** menu in the **Jython Consoles** view, and select the **New Interpreter** option.



```
Jython 2.5.1.  Type "help()" for more info
```

Figure 9-1. Select New Interpreter Option in Jython Consoles View

The **Input Name** dialog box appears.

6. Specify a name for the new interpreter and click **OK**.
7. In the Jython console command prompt (">>>"), input code in Python syntax. You may access any classes in your test scripts, foe example, `test_file.py`, by inputting the following commands in the Jython command prompt:

```
>>> from scripting_sa_mcu import test_file

>>> test = test_file.test_class()

>>> test.a_function(args)
```

NOTE

The advantage of using Jython is that besides Python modules, it can also access Java modules in the similar manner.

This is how you run a sample Python script from Jython console.

9.2 Collect Trace Using Jython

To collect trace using remote launch through Jython console, perform the following steps. These steps demonstrate the procedure of launching configuration and collecting both continuous and automatic trace on the HCS08 MC9SO8QE128 target using the sample scripts, `TestCollectContinuousHcs08.py` and `TestCollectAutomaticHcs08.py`.

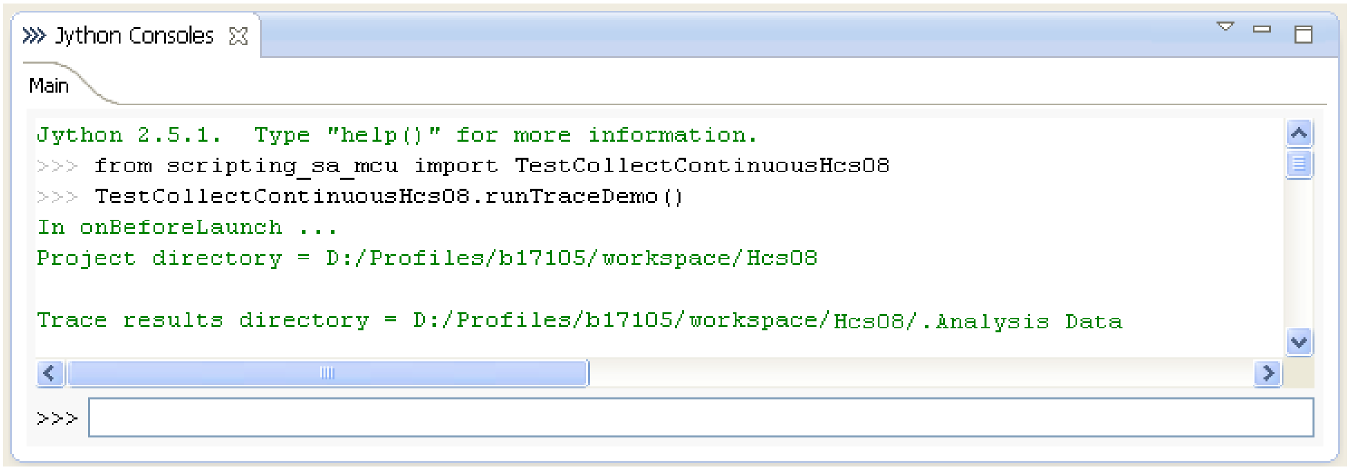
These scripts are available at `<CW Installation Directory>\eclipse\plugins\com.freescale.sa.mcu.scripting_*\remote\scripting_sa_mcu`.

1. Create a new HCS08 project with the name `hcs08`.
2. Replace the source code of `main.c` with the code shown below.
   Listing: HCS08 Source Code

```
#include <hidef.h> /* for EnableInterrupts macro */

#include "derivative.h" /* include peripheral declarations */

volatile i = 0;

void f() {

    if (i < 100) {

      i++;

      return;

    }

    i = 0;

}

void main(void) {
    EnableInterrupts;


   /* include your code here */


   for(;;) {

     f();

     __RESET_WATCHDOG();/* feeds the dog */


   } /* loop forever */


   /* please make sure that you never leave main */

}
```

3. Save and build your project.
4. Open the **Remote Launch** and **Jython Consoles** views.
5. Activate remote launch by clicking the **Enable Remote Launch** button.
6. Execute the `TestCollectContinuousHcs08.py` script by running the following commands at the Jython command prompt:

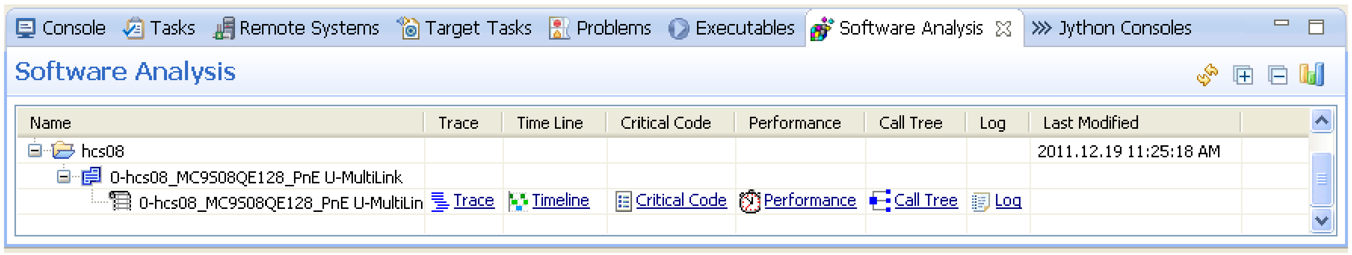
```
>>> from scripting_sa_mcu import TestCollectContinuousHcs08

>>> TestCollectContinuousHcs08.runTraceDemo()
```



```
Jython 2.5.1.  Type "help()" for more information.
>>> from scripting_sa_mcu import TestCollectContinuousHcs08
>>> TestCollectContinuousHcs08.runTraceDemo()
In onBeforeLaunch ...
Project directory = D:/Profiles/b17105/workspace/Hcs08

Trace results directory = D:/Profiles/b17105/workspace/Hcs08/.Analysis Data
>>>
```

Figure 9-2. Running Commands at Jython Command Prompt for Collecting Trace

The trace collection starts and the **Software Analysis** view appears with the trace results.



Name	Trace	Time Line	Critical Code	Performance	Call Tree	Log	Last Modified
hcs08							2011.12.19 11:25:18 AM
0-hcs08_MC9S08QE128_PnE U-MultiLink							
0-hcs08_MC9S08QE128_PnE U-MultiLin	Trace	Timeline	Critical Code	Performance	Call Tree	Log	

Figure 9-3. Software Analysis View Containing Trace Results Collected Using Remote Launch

7. Start a new Jython interpreter by selecting the **New Interpreter** option from the **View** menu of the **Jython Consoles** view.
8. Execute the `TestCollectAutomaticHcs08.py` script by running the following commands at the Jython command prompt:

```
>>> from scripting_sa_mcu import TestCollectAutomaticHcs08

>>> TestCollectAutomaticHcs08.runTraceDemo()
```

9. Open the **Software Analysis** view and check the collected results.

NOTE

The sample scripts used for collecting continuous and automatic trace on the ColdFire V1 and Kinetis targets are also located at `<CW Installation Directory>\eclipse\plugins\com.freescale.sa.mcu.scripting_*\remote\scripting_sa_mcu`.

This is how you collect trace using the Jython console.

9.3 Export Trace to CSV File

You can export trace data collected on a project to a CSV file using Jython console.

NOTE

Another way of exporting trace data to a CSV file is through buttons available on the **Trace Data** , **Critical Code** , and **Performance** viewers.

To export trace data using Jython console, perform the following steps. Make sure that you have some trace data collected that you want to export to CSV files.

1. Open the **Remote Launch** and **Jython Consoles** views.
2. Activate remote launch by clicking the **Enable Remote Launch** button.

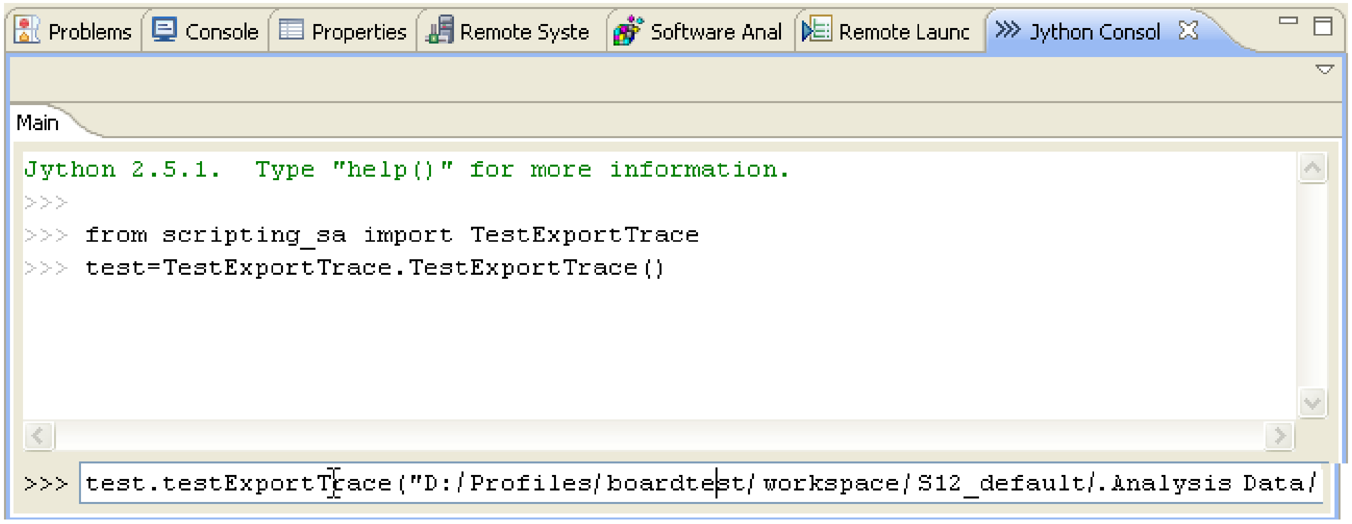
3. Open the **View** menu in the **Jython Consoles** view, and select the **New Interpreter** option. This step is not required if you have opened **Jython Consoles** for the first time.
4. Type a name and click **OK** in the **Input Name** dialog box.
5. Type the following commands one by one in the Jython command prompt:

```
>>> from scripting_sa import TestExportTrace

>>> test = TestExportTrace.TestExportTrace()

>>> test.testExportTrace(config_path)
```

where `config_path` is a string containing the full path (in double quotes) of the `.config` file present in the **.Analysis Data** directory (in your project workspace) with path separators '/' or '\\'. The `.config` file contains the configuration details of the project the trace data of which you are exporting to CSV.

NOTE

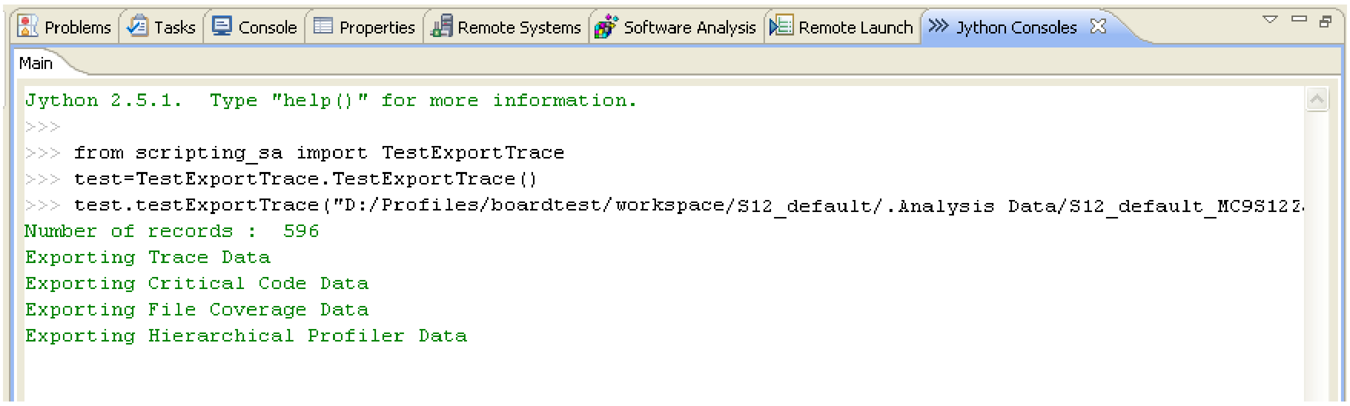
The `from scripting_sa import TestExportTrace` command imports the Python script `TestExportTrace.py` written for exporting trace results in CSV files. This script is located at `<CW Installation Directory>\eclipse\plugins \com.freescale.sa.scripting_*\remote\scripting_sa.`



```
Jython 2.5.1.  Type "help()" for more information.
>>>
>>> from scripting_sa import TestExportTrace
>>> test=TestExportTrace.TestExportTrace()

>>> test.testExportTrace("D:/Profiles/boardtest/workspace/S12_default/.Analysis Data/
```

Figure 9-4. Running Commands in Jython Console

The following output is shown in the Jython console and four `.csv` files are created in the `.Analysis Data` directory of your workspace.



```
Jython 2.5.1.  Type "help()" for more information.
>>>
>>> from scripting_sa import TestExportTrace
>>> test=TestExportTrace.TestExportTrace()
>>> test.testExportTrace("D:/Profiles/boardtest/workspace/S12_default/.Analysis Data/S12_default_MC9S12Z
Number of records :  596
Exporting Trace Data
Exporting Critical Code Data
Exporting File Coverage Data
Exporting Hierarchical Profiler Data
```

Figure 9-5. Output in Jython Console After Running Export Commands

6. Browse to the `.Analysis Data` directory and view the `.csv` files containing the exported data.

This is how you export trace data to a CSV file using the Jython console.

9.4 Modify Sample Python Script

You can modify the sample Python scripts `TestCollectContinuousHcs08.py` and `TestCollectAutomaticHcs08.py` according to your needs to extend the functionality to other platforms. You can use these scripts as base for creating test scripts.

To create your scripts, you can modify the sample script as follows:

1. Rename the script according to platform name.
2. Rename the two classes inside the scripts.
3. Update the references to these classes in the `onBeforeLaunchStart(self)` and `runTraceDemo()` functions. For example, in the `onBeforeLaunchStart(self)` function, replace `TestConfigHcs08()` with `TestConfigABC()` as follows, where ABC is the platform you are working on.

   ```
   if ( self.__enableTrace ):

                self.__traceSession = TestConfigABC()
   ```
4. Similarly, in the `runTraceDemo()` function, replace `TestCollectContinuousHcs08()` with `TestCollectContinuousABC()`, where ABC is the platform you are working on.
5. Modify the configuration details in the `Config data for the script` section (lines 20 to 23) as follows:
   a. Update platform name in `test_platform`. The platform type string is case-sensitive and can take any one of the following values: "HCS08", "Kinetis", "CFv1", "CFv2", "CFv3", "CFv4", "56xx", "DSC", "S12Z".
   b. Update project name in `test_project`.

c. Update launch configuration name in `test_launchconfig`.
d. Update the breakpoint line in `test_bp_line`, that is the line where you want to set a breakpoint. For example, for the Kinetis stationery project, you may use line 21 for the breakpoint.

6. Update other configuration details after line 57 according to your needs.

NOTE

Besides checking that trace is collected and a new result set named *0-{launch_config_name}* is created in the **Software Analysis** view, you may edit the file `0-{launch_config_name}.traceConfig` available in the project root. You can see that the attributes corresponding to the `self.__analysisConfig.setXXX` functions called in the sample scripts are changed in the `.traceConfig` file. For example, if `self.__analysisConfig.setContinuousTrace(True)` is called from the scripts (see line 59 in the sample scripts) then in the `.traceConfig` file, you should notice that the "Continuous" attribute is changed to true. The remote launch creates a copy of the launch configuration created by the **New Project Wizard** and deletes it after the debug session terminates successfully, so checking the trace and profile tab controls for changes is not possible.

This is how you can modify a sample Python script.

Chapter 10
Simple Instrumentation Profiling on ColdFire V2 - V4e Targets

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
- Configure your project to turn on profiling
- Modify the application source code to make use of the profiler API
- Debug the application and collect profiling information
- Open the Simple Profiler Viewer to view the 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

This chapter explains profiling on ColdFire V4e target, the process of profiling is same for ColdFire V2 - V4 targets.

This chapter contains the following topics:

- Include Profiler Library and Files

- Configure Project for Profiling
- Modify Source Code
- Debug Application and Collect Profiling Information
- View Profiling Results

10.1 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.

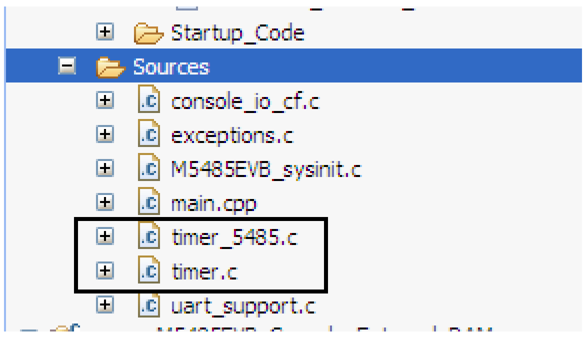


Figure 10-1. Profiler Files Added to Sources Folder

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.

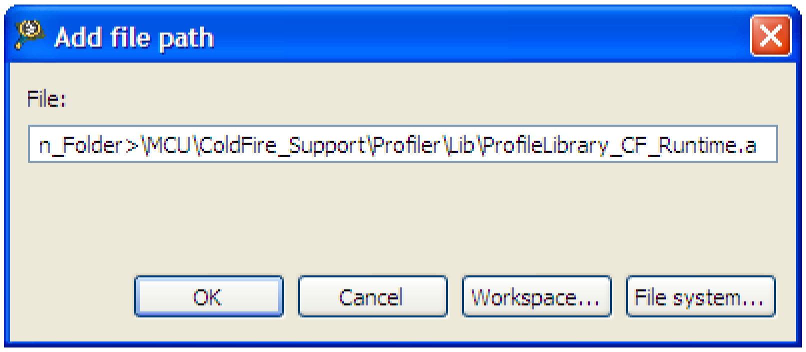
   The **Properties for <project name>** dialog box appears.

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 the **Add..** 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.


Add file path

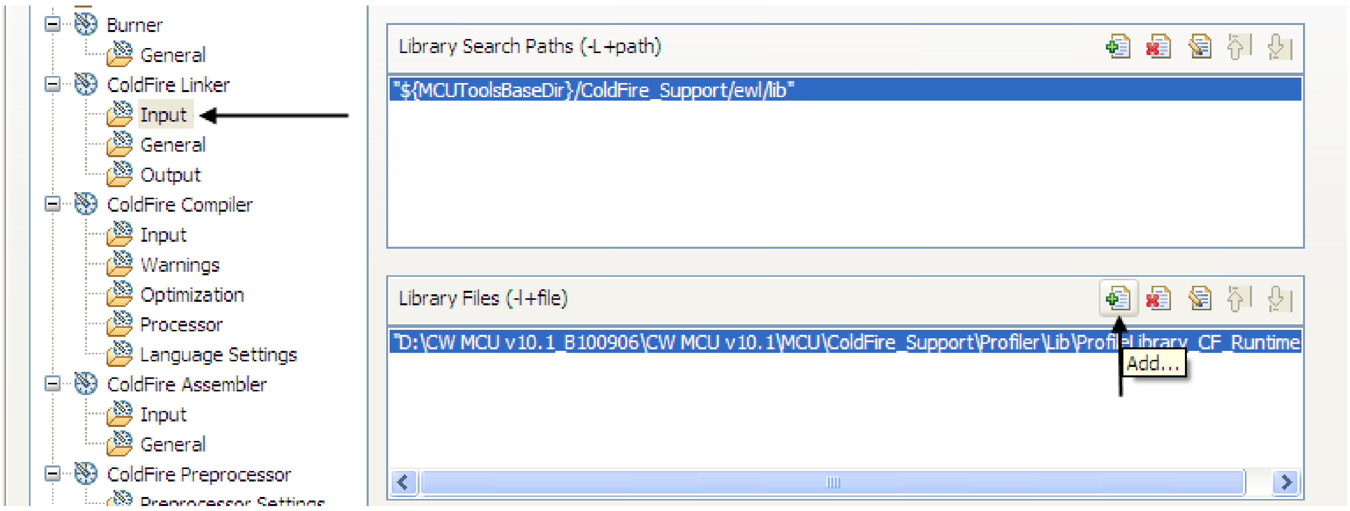
File:

n_Folder>\MCU\ColdFire_Support\Profiler\Lib\ProfileLibrary_CF_Runtime.a

Figure 10-2. 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.



"${MCUToolsBaseDir}/ColdFire_Support/ewl/lib"

"D:\CW MCU v10.1_B100906\CW MCU v10.1\MCU\ColdFire_Support\Profiler\Lib\ProfileLibrary_CF_Runtime

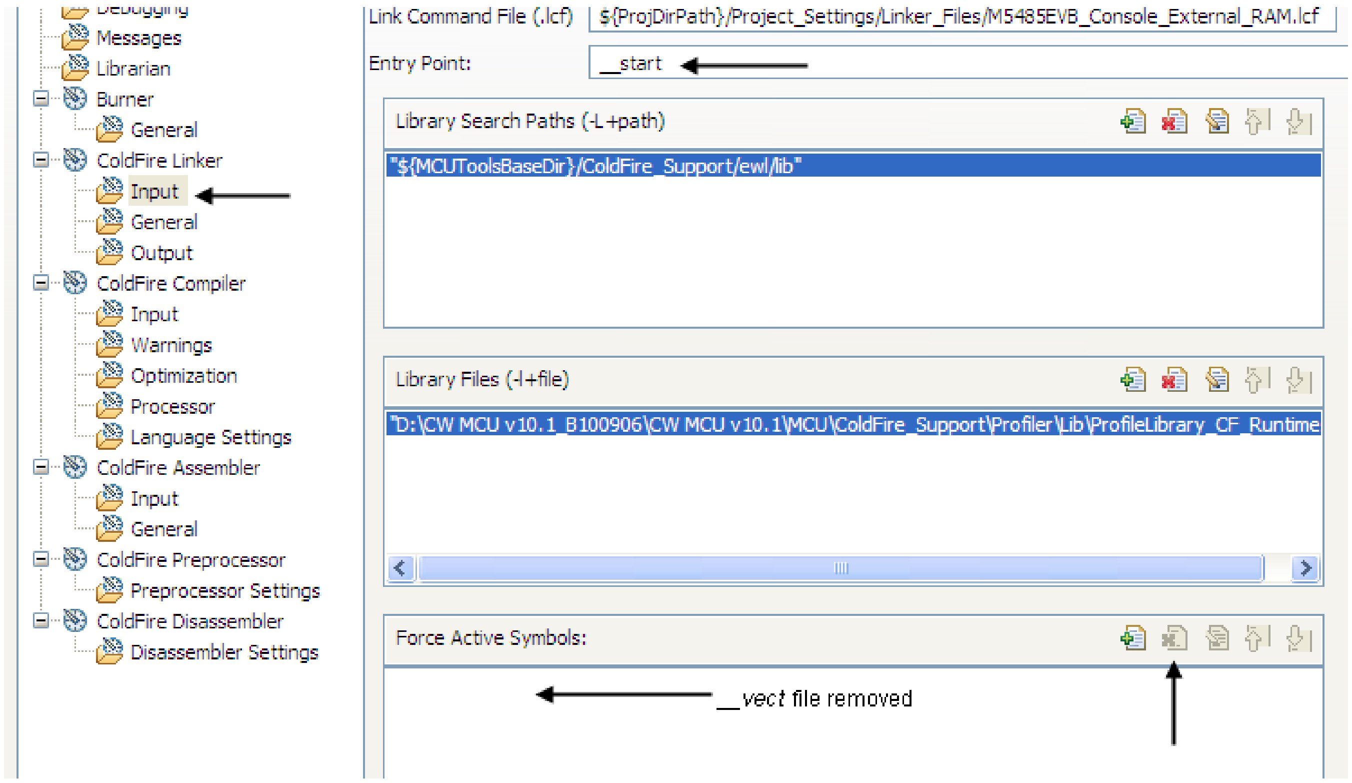
Figure 10-3. Tool Settings Page of Properties Dialog Box

10.2 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 the **Delete** button.



Link Command File (.lcf) ${ProjDirPath}/Project_Settings/Linker_Files/M5485EVB_Console_External_RAM.lcf

Entry Point: __start

Library Search Paths (-L+path)

"${MCUToolsBaseDir}/ColdFire_Support/ewl/lib"

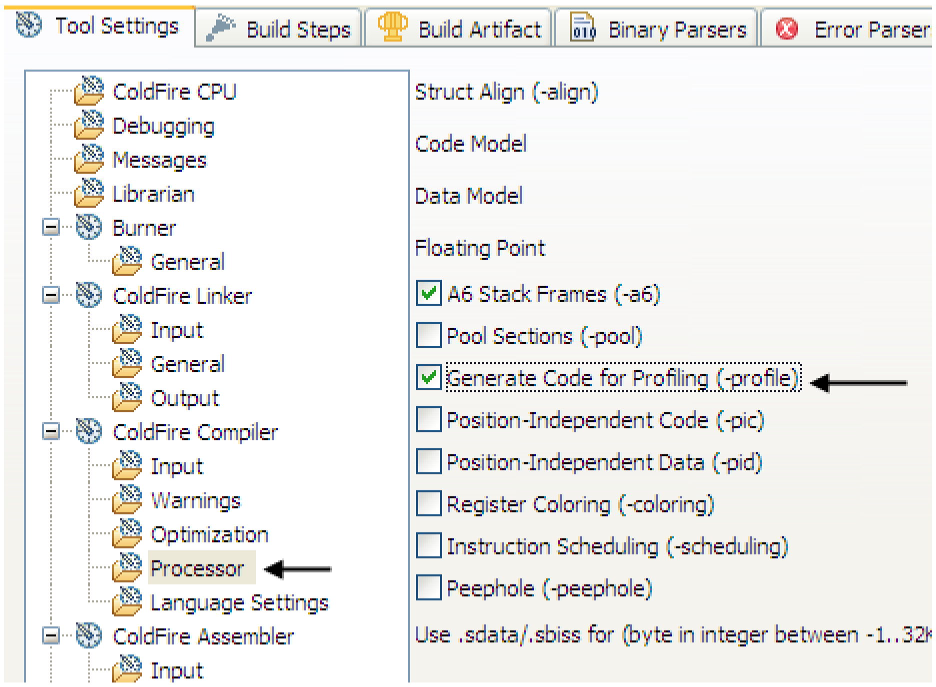
Library Files (-l+file)

"D:\CW MCU v10.1_B100906\CW MCU v10.1\MCU\ColdFire_Support\Profiler\Lib\ProfileLibrary_CF_Runtime

Force Active Symbols:

Figure 10-4. 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.



Struct Align (-align)

Code Model

Data Model

Floating Point

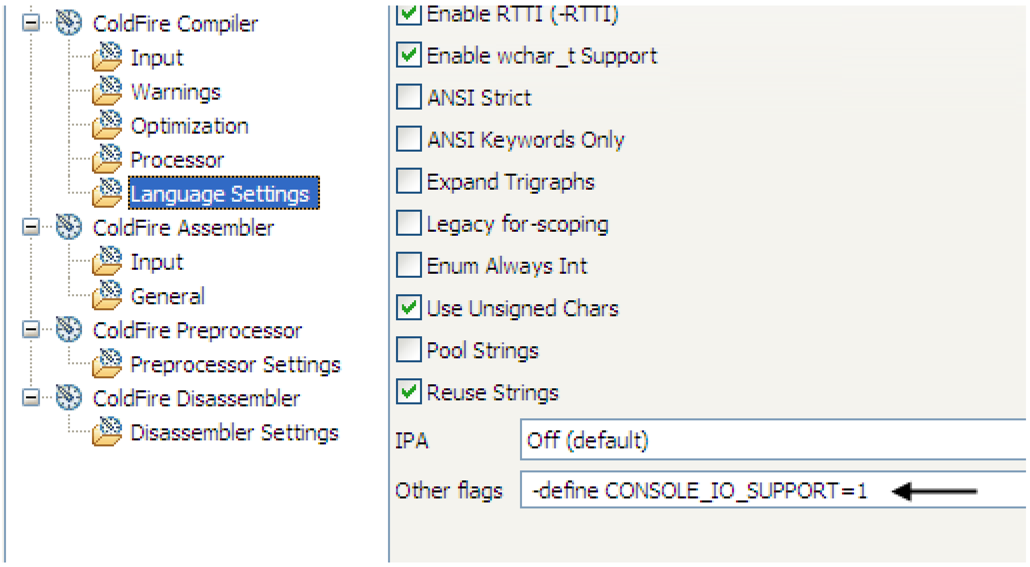
- [✓] A6 Stack Frames (-a6)
- [ ] Pool Sections (-pool)
- [✓] Generate Code for Profiling (-profile)
- [ ] Position-Independent Code (-pic)
- [ ] Position-Independent Data (-pid)
- [ ] Register Coloring (-coloring)
- [ ] Instruction Scheduling (-scheduling)
- [ ] Peephole (-peephole)

Use .sdata/.sbiss for (byte in integer between -1..32k

Figure 10-5. 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 entry: `-define CONSOLE_IO_SUPPORT=1`



- [✓] Enable RTTI (-RTTI)
- [✓] Enable wchar_t Support
- [ ] ANSI Strict
- [ ] ANSI Keywords Only
- [ ] Expand Trigraphs
- [ ] Legacy for-scoping
- [ ] Enum Always Int
- [✓] Use Unsigned Chars
- [ ] Pool Strings
- [✓] Reuse Strings

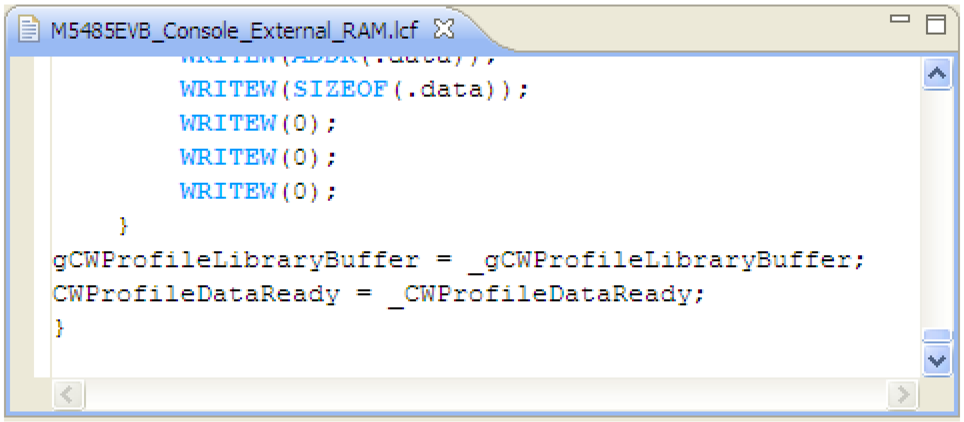
IPA: Off (default)

Other flags: -define CONSOLE_IO_SUPPORT=1

Figure 10-6. 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;
```



```
        WRITEW(SIZEOF(.data));
        WRITEW(0);
        WRITEW(0);
        WRITEW(0);
    }
gCWProfileLibraryBuffer = _gCWProfileLibraryBuffer;
CWProfileDataReady = _CWProfileDataReady;
}
```

Figure 10-7. Modification in M5485EVB_Console_External_RAM.lcf File

13. Save and close the file.

This configures the project for profiling.

10.3 Modify Source Code

Replace the source code of the `main.cpp` file of your project with the source code shown in the listing below. To open the `main.cpp` file:

1. Expand the **Sources** node of your project in the **CodeWarrior Projects** view.
2. Double-click `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: 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;

}
```

10.4 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.

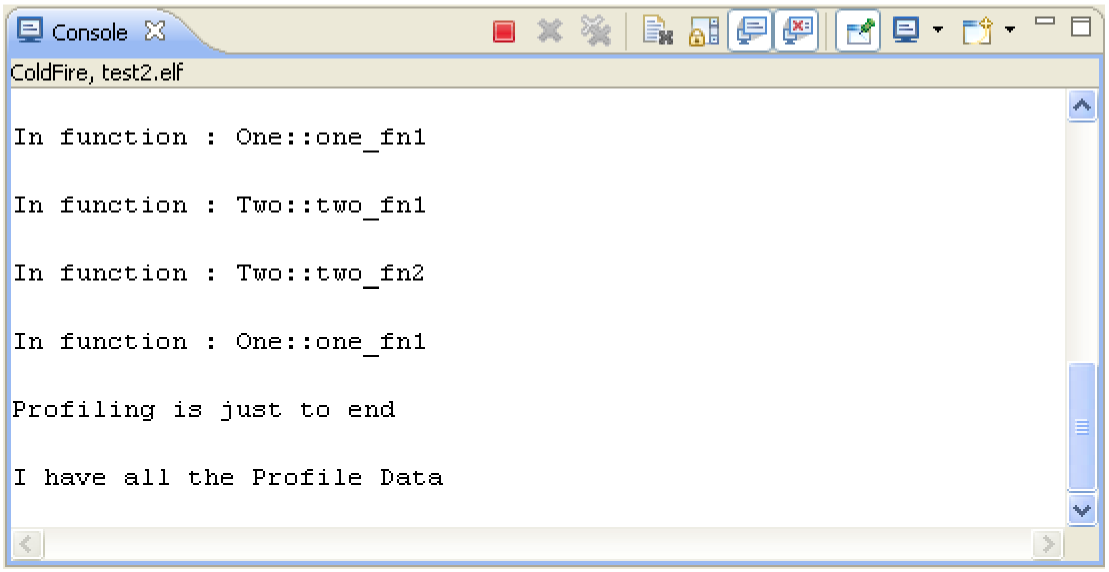
The project halts at the `main()` function.

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.
4. Click **Terminate** to stop the program execution.



```
In function : One::one_fn1

In function : Two::two_fn1

In function : Two::two_fn2

In function : One::one_fn1

Profiling is just to end

I have all the Profile Data
```

Figure 10-8. Console View

This is how you collect profiling data.

10.5 View Profiling Results

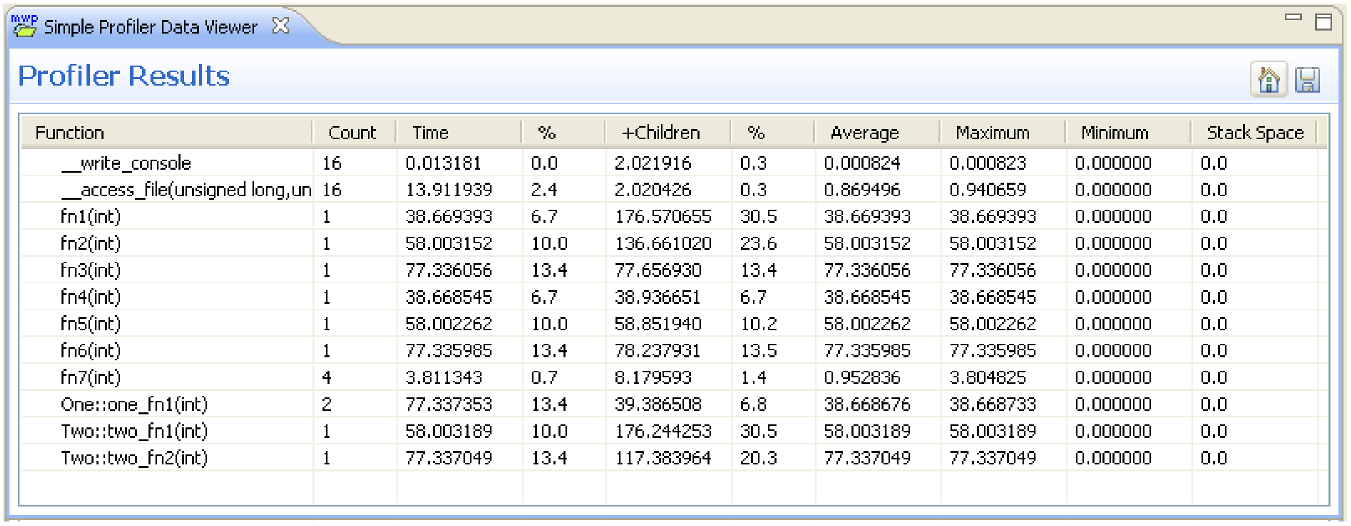
You use the **Simple Profiler Data Viewer** to view the profiler results. This viewer helps analyze the data of the executed program and determine what changes are appropriate to improve the performance of the application. Using the data display, you can:

- 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 **M5485EVB_Console_External_RAM** and double-click the `profiledump.mwp` file.

   The **Simple Profiler Data Viewer** appears.


Simple Profiler Data Viewer

Profiler Results

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

Figure 10-9. Simple Profiler Data Viewer

The **Simple Profiler Data Viewer** displays profiling information in the form of a table. The table below describes the fields of the profiling information.

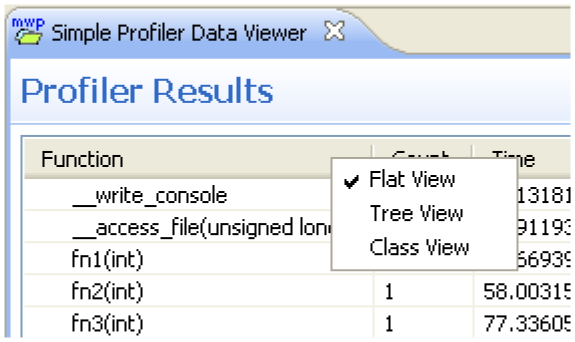
Table 10-1. Description of Profiler Results

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 any column of the **Simple Profiler Data Viewer** table and select the necessary option from the **View** context menu.


Simple Profiler Data Viewer

Profiler Results

Function	Count	Time
__write_console		13181
__access_file(unsigned lon		91193
fn1(int)		56939
fn2(int)	1	58.00315
fn3(int)	1	77.33605

Figure 10-10. Different Views of Profile Results

The three views for displaying profiling information are:

- Flat View
- Tree View
- Class View

10.5.1 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 10-9 shows the **Flat** view of the profiler results.

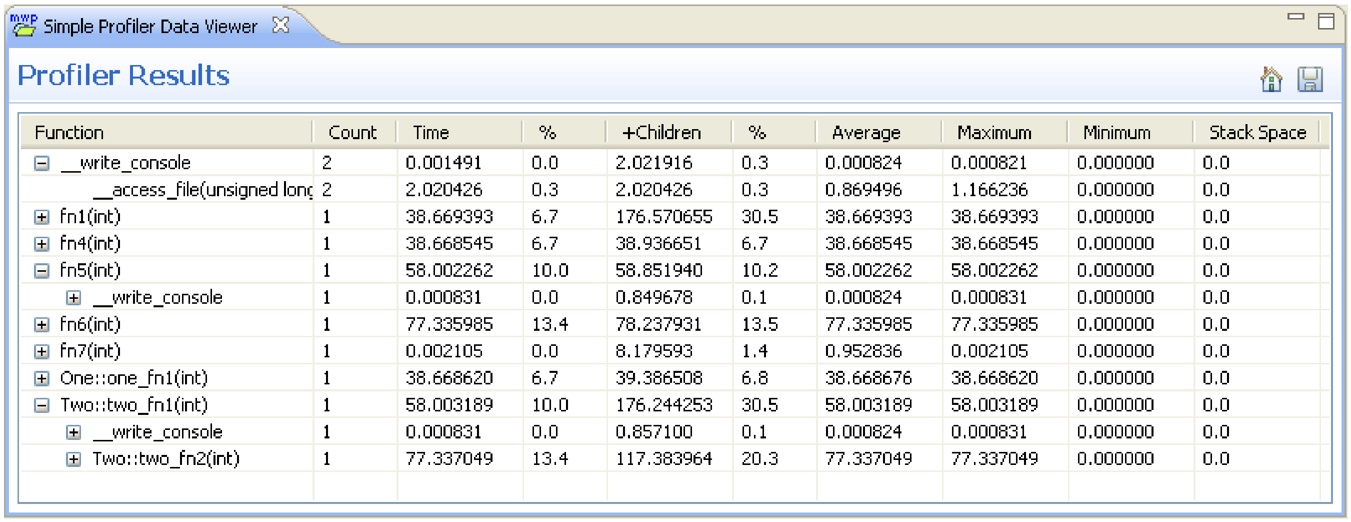
The **Flat** view is particularly useful for comparing functions to check which function takes the longest time to execute. The **Flat** view is also useful for finding a performance problem with a small function that is called from many different places in the program. This view helps you look for the functions that make heavy demands in time or raw number of calls.

10.5.2 Tree View

The **Tree** view displays the detailed profile data as shown in Figure 10-10. 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.



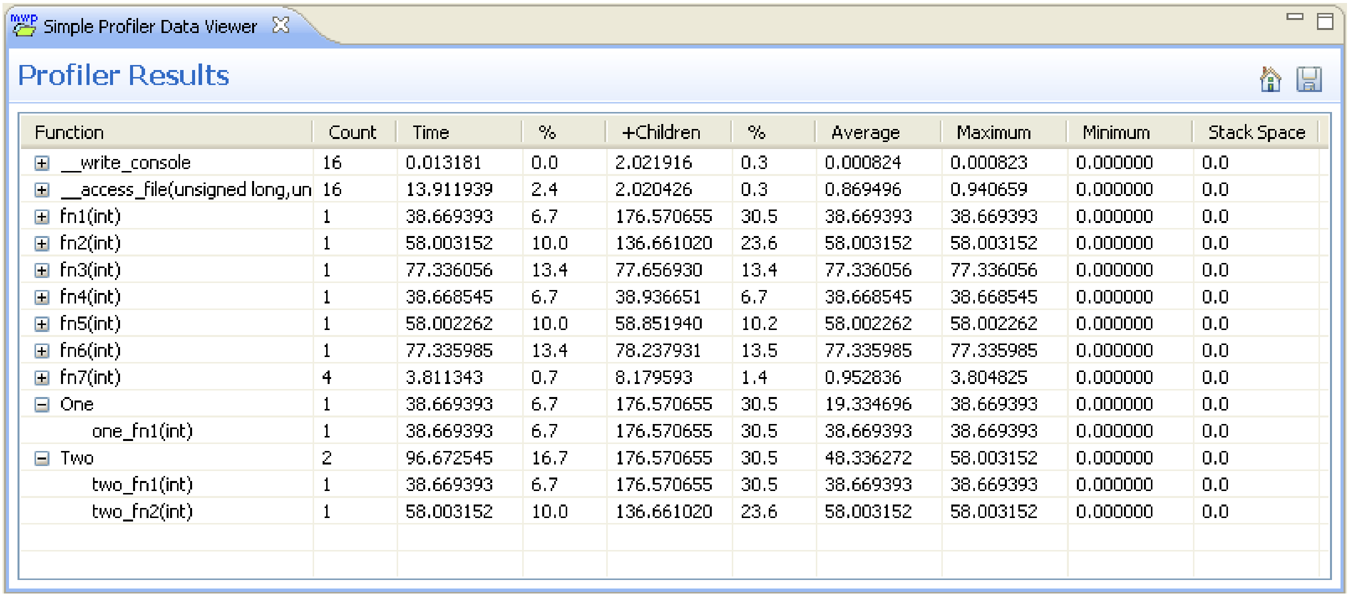
Function	Count	Time	%	+Children	%	Average	Maximum	Minimum	Stack Space
⊟ __write_console	2	0.001491	0.0	2.021916	0.3	0.000824	0.000821	0.000000	0.0
__access_file(unsigned lonç	2	2.020426	0.3	2.020426	0.3	0.869496	1.166236	0.000000	0.0
⊞ fn1(int)	1	38.669393	6.7	176.570655	30.5	38.669393	38.669393	0.000000	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

Figure 10-11. Simple Profiler Data Viewer - Tree View

10.5.3 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. The figure below shows the **Class** view of the profiler results.



Simple Profiler Data Viewer

Profiler Results

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

Figure 10-12. Simple Profiler Data Viewer - Class View

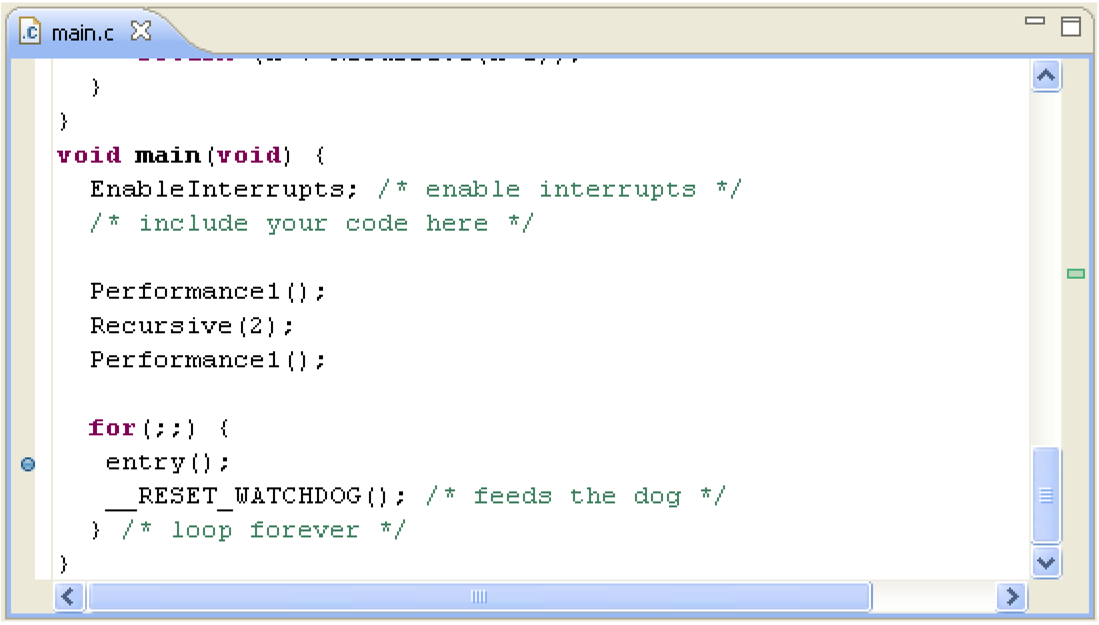
Chapter 11

11.1 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. The following steps demonstrate how to set a breakpoint in the editor area and collect trace data on the ColdFire V1 target:

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 editor area, double-click the marker bar corresponding to the instruction on which you want to set the breakpoint.

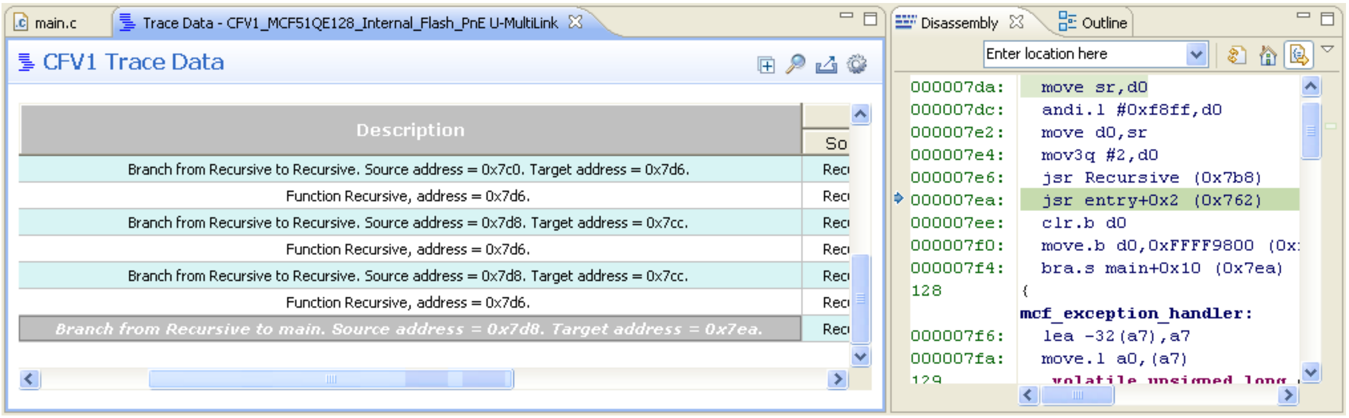


```
  }
}
void main(void) {
  EnableInterrupts; /* enable interrupts */
  /* include your code here */

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

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

Figure 11-1. 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 Viewing Data to view the trace data.
10. Press **Resume** to resume trace collection.

The figure below shows that trace collection stops at the address where the breakpoint was set.



Description	So
Branch from Recursive to Recursive. Source address = 0x7c0. Target address = 0x7d6.	Rec
Function Recursive, address = 0x7d6.	Rec
Branch from Recursive to Recursive. Source address = 0x7d8. Target address = 0x7cc.	Rec
Function Recursive, address = 0x7d6.	Rec
Branch from Recursive to Recursive. Source address = 0x7d8. Target address = 0x7cc.	Rec
Function Recursive, address = 0x7d6.	Rec
Branch from Recursive to main. Source address = 0x7d8. Target address = 0x7ea.	Rec

```
000007da:   move sr,d0
000007dc:   andi.l #0xf8ff,d0
000007e2:   move d0,sr
000007e4:   mov3q #2,d0
000007e6:   jsr Recursive (0x7b8)
000007ea:   jsr entry+0x2 (0x762)
000007ee:   clr.b d0
000007f0:   move.b d0,0xFFFF9800 (0x
000007f4:   bra.s main+0x10 (0x7ea)
128         {
            mcf_exception_handler:
000007f6:   lea -32(a7),a7
000007fa:   move.l a0,(a7)
```

Figure 11-2. Trace Results After Setting Breakpoint

NOTE

Similarly, you can set breakpoints in the **Disassembly** view and collect the trace data.

Warning

Do not use breakpoints with triggers when collecting trace on the ColdFire V1 target. This is because there is only one hardware debug module on ColdFire V1 which is shared for setting hardware breakpoints or trace triggers. So you can either set breakpoint or trace trigger using this debug module.

On HCS08, if you are setting breakpoints with triggers for trace control, ensure that you do not to use more than one breakpoint. This is because on HCS08, two hardware debug modules, BDC (Background debug controller) and DBG (debug module) are used. Both debug modules can be used for setting hardware breakpoints, while only DBG can be used for setting triggers. The first breakpoint is set using BDC, any other breakpoints will use DBG and will conflict with the triggers.

On DSC and S12z platforms, breakpoints and triggers are mutually exclusive.

On the Kinetis platform, there are four DWT comparators which can be used for maximum four events. The Software Analysis tool checks which DWT comparators are already used for breakpoints and only uses the remaining ones for triggers.

Chapter 12
Configuring Trace Registers in Source Code

You can the configure trace registers in the source code without using the CodeWarrior Integrated Development Environment for both HCS08 and ColdFire V1 targets.

12.1 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 below:

   Listing: 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 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
```

12.2 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 below:
   Listing: 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 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 the listing below in the `main()` function.

Listing: 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: Configuration of trace registers for ColdFire V1. To configure trace in the Profile-Only mode, set the compiler switch to `TRACE_PCSYNC` in the source code shown in Listing: Configuration of trace registers for ColdFire V1.

Chapter 13
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

These steps are described in the following topics.

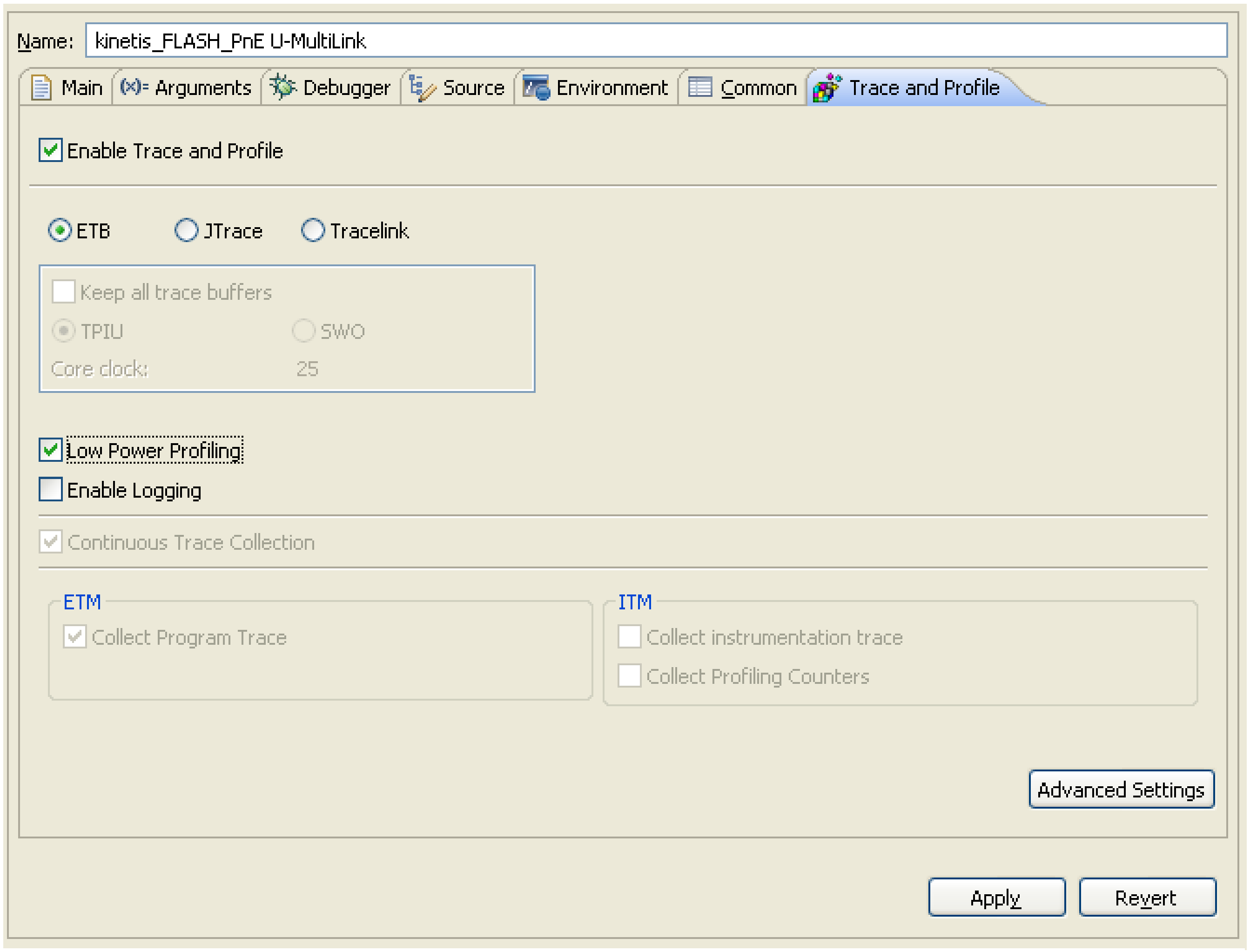
13.1 Configure Low Power WAIT State

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 **Low Power Profiling** checkbox.

   The **ETM** and **ITM** options will get disabled automatically. Also, the **Continuous Trace Collection** checkbox will get checked and disabled.



Name: kinetis_FLASH_PnE U-MultiLink

[✓] Enable Trace and Profile

(●) ETB ( ) JTrace ( ) Tracelink

[ ] Keep all trace buffers
(●) TPIU ( ) SWO
Core clock: 25

[✓] Low Power Profiling
[ ] Enable Logging

[✓] Continuous Trace Collection

ETM
[✓] Collect Program Trace

ITM
[ ] Collect instrumentation trace
[ ] Collect Profiling Counters

Figure 13-1. Configuration of Low Power WAIT Mode

4. Click **Apply** to save the settings.

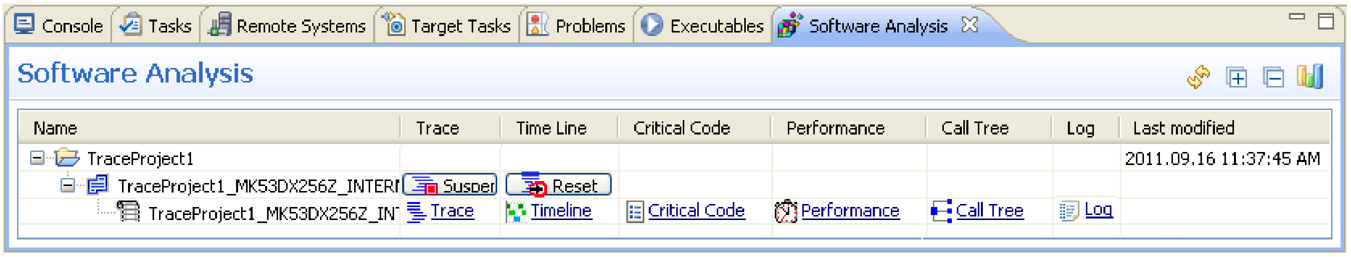
This configures the low power WAIT state of your project.

13.2 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 **Software Analysis** view. To view the low power WAIT results:

1. Open the **Software Analysis** view.
2. Expand the project name.

   The data source is listed under the project name.



Software Analysis

Name	Trace	Time Line	Critical Code	Performance	Call Tree	Log	Last modified
TraceProject1							2011.09.16 11:37:45 AM
TraceProject1_MK53DX256Z_INTERI	Suspen	Reset					
TraceProject1_MK53DX256Z_IN	Trace	Timeline	Critical Code	Performance	Call Tree	Log	

Figure 13-2. Software Analysis View

3. Click the **Timeline** hyperlink.

   The **Timeline** viewer appears displaying the low power WAIT results.

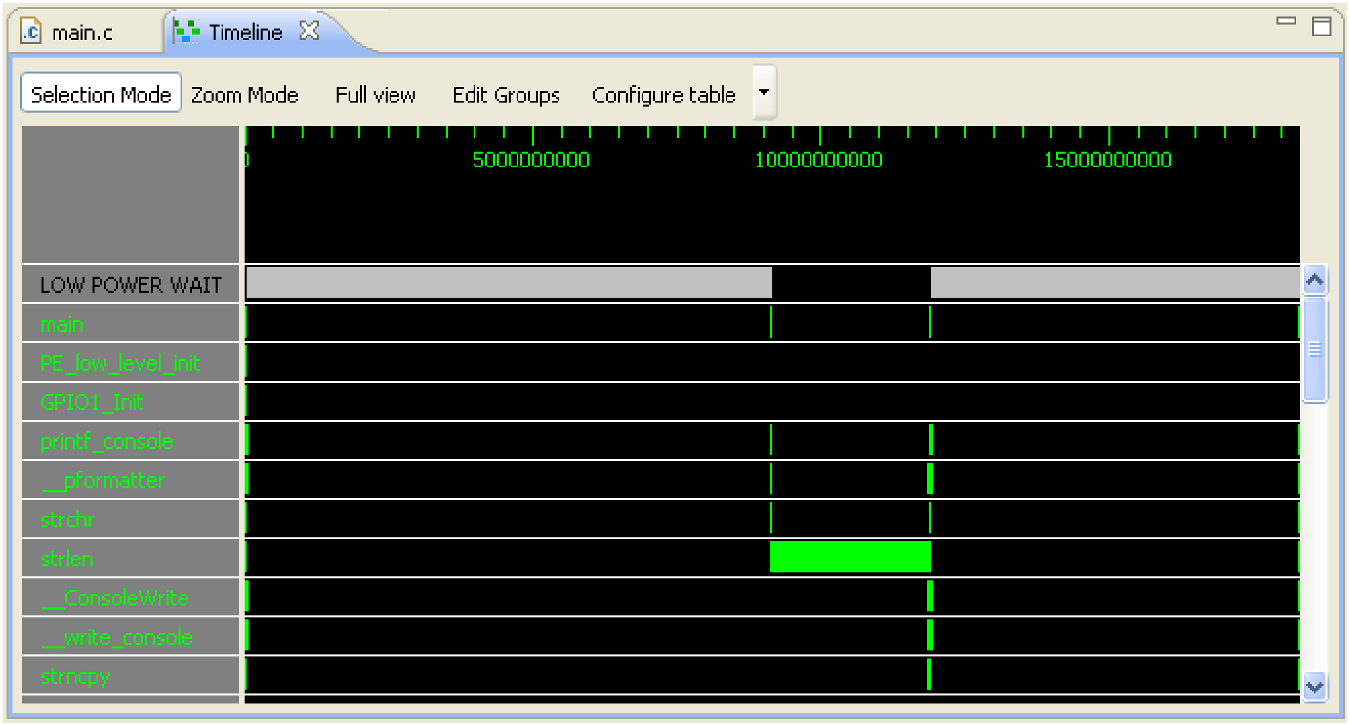


Figure 13-3. Timeline Viewer

The list of functions in the left panel of the **Timeline** 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

Document Number CWMCUSWAUG
Revision 11.x, 07/2017