Contents

Section number | Title | Page
--- | --- | ---

Chapter 1
Introduction

1.1 Release Notes | | 9
1.2 Accompanying Documentation | | 9

Chapter 2
Getting Started

2.1 Profiling and Analysis Tools | | 11
2.2 CodeWarrior Interface | | 12
2.3 Data Collection | | 14
  2.3.1 Trace Data | | 14
  2.3.2 Critical Code Data | | 14
  2.3.3 Timeline Data | | 15
  2.3.4 Performance Data | | 15
  2.3.5 Call Tree | | 15
  2.3.6 Profiling Data | | 15

Chapter 3
Collecting Data

3.1 Creating New Project | | 17
  3.1.1 Using HCS08 Target | | 18
  3.1.2 Using ColdFire V1 Target | | 21
  3.1.3 Using Kinetis Target | | 23
  3.1.4 Using ColdFire V4e | | 26
  3.1.5 Using MPC56xx Target | | 29
3.2 Configuring Launcher | | 31
  3.2.1 Configure HCS08 Target | | 31
  3.2.2 Configure ColdFire V1 Target | | 38
  3.2.3 Configure Kinetis Target | | 42
    3.2.3.1 Configuring Kinetis Cortex M0+ Core | | 47
<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.3.2</td>
<td>Difference Between Kinetis Cores Cortex M4 and Cortex M0+</td>
<td>49</td>
</tr>
<tr>
<td>3.2.3.3</td>
<td>Trace Collection on Kinetis</td>
<td>50</td>
</tr>
<tr>
<td>3.2.3.3.1</td>
<td>Embedded Trace Macrocell (ETM)</td>
<td>50</td>
</tr>
<tr>
<td>3.2.3.3.1.1</td>
<td>Triggering Trace</td>
<td>51</td>
</tr>
<tr>
<td>3.2.3.3.2</td>
<td>Instrumentation Trace Macrocell (ITM)</td>
<td>52</td>
</tr>
<tr>
<td>3.2.3.3.2.1</td>
<td>Timestamps</td>
<td>53</td>
</tr>
<tr>
<td>3.2.3.3.2.2</td>
<td>Synchronization</td>
<td>53</td>
</tr>
<tr>
<td>3.2.3.3.3</td>
<td>Embedded Trace Buffer (ETB)</td>
<td>54</td>
</tr>
<tr>
<td>3.2.3.3.4</td>
<td>J-Trace</td>
<td>54</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Configuring Advanced Settings on Kinetis</td>
<td>55</td>
</tr>
<tr>
<td>3.2.5</td>
<td>Configure MPC56xx Target</td>
<td>68</td>
</tr>
<tr>
<td>3.2.6</td>
<td>Configure S12Z Target</td>
<td>68</td>
</tr>
<tr>
<td>3.2.7</td>
<td>Configure ColdFire V2-V4 Targets</td>
<td>71</td>
</tr>
<tr>
<td>3.2.8</td>
<td>Configure DSC Target</td>
<td>73</td>
</tr>
<tr>
<td>3.3</td>
<td>Collecting Data</td>
<td>76</td>
</tr>
<tr>
<td>3.4</td>
<td>Viewing Data</td>
<td>77</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Data Trace Import Dialog Box</td>
<td>79</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Controlling Trace Generation</td>
<td>82</td>
</tr>
<tr>
<td>3.4.2.1</td>
<td>Resume/Suspend Toggle Button</td>
<td>82</td>
</tr>
<tr>
<td>3.4.2.2</td>
<td>Reset Button</td>
<td>83</td>
</tr>
<tr>
<td>4.1</td>
<td>Trace Data</td>
<td>85</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Finding and Filtering Trace Data</td>
<td>87</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Exporting Trace Data</td>
<td>90</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Configuring Time Unit and Time Format</td>
<td>91</td>
</tr>
<tr>
<td>4.1.4</td>
<td>Customizing Trace Data Viewer</td>
<td>92</td>
</tr>
<tr>
<td>4.2</td>
<td>Timeline</td>
<td>94</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Selection Mode</td>
<td>95</td>
</tr>
</tbody>
</table>
4.2.2 Zoom Mode..........................................................................................................................................................96
4.2.3 Full View............................................................................................................................................................97
4.2.4 Edit Groups........................................................................................................................................................97
  4.2.4.1 Add/Remove Function...................................................................................................................................98
  4.2.4.2 Edit Address Range of Function..................................................................................................................98
  4.2.4.3 Change Color..............................................................................................................................................98
  4.2.4.4 Add/Remove Group.......................................................................................................................................98
  4.2.4.5 Merge Groups/Functions..............................................................................................................................100
4.2.5 Configure Table..................................................................................................................................................101
4.3 Critical Code Data...................................................................................................................................................101
  4.3.1 Critical Code Tab.............................................................................................................................................103
  4.3.2 File Coverage Tab..........................................................................................................................................105
4.4 Performance Data....................................................................................................................................................106
4.5 Call Tree...............................................................................................................................................................109
4.6 Importing Trace Data Offline..................................................................................................................................111

Chapter 5
Setting Tracepoints (HCS08)

5.1 Conditions for Starting/Stopping Triggers........................................................................................................117
5.2 Trace Modes............................................................................................................................................................120
  5.2.1 Setting Triggers in Continuously Mode...........................................................................................................121
    5.2.1.1 Instruction at Address A, Then Instruction at Address B are Executed.......................................................121
    5.2.1.2 Instruction at Address A or Address B is Executed.......................................................................................126
    5.2.1.3 Instruction Inside Range from Address A to Address B is Executed..............................................................127
    5.2.1.4 Instruction at Address A is Executed and Value on Data Bus Match............................................................131
    5.2.1.5 Instruction at Address A is Executed and Value on Data Bus Mismatch.......................................................133
    5.2.1.6 Memory Access Triggers............................................................................................................................133
  5.2.2 Setting Triggers in Automatically Mode........................................................................................................136
    5.2.2.1 From the Disassembly View......................................................................................................................136
    5.2.2.2 On Data and Memory...............................................................................................................................139
<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2.2.3</td>
<td>LOOP1 Mode</td>
<td>142</td>
</tr>
<tr>
<td>5.2.2.4</td>
<td>Instruction Outside Range from Address A to Address B is Executed</td>
<td>145</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Setting Triggers in Collect Data Trace Mode</td>
<td>148</td>
</tr>
<tr>
<td>5.2.3.1</td>
<td>Capture Read/Write Values at Address B</td>
<td>148</td>
</tr>
<tr>
<td>5.2.3.2</td>
<td>Capture Read/Write Values at Address B, After Access at Address A</td>
<td>152</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Collecting Trace in Profile-Only Mode</td>
<td>155</td>
</tr>
<tr>
<td>5.2.5</td>
<td>Collecting Trace in Expert Mode</td>
<td>156</td>
</tr>
<tr>
<td>5.3</td>
<td>Enabling and Disabling the Tracepoints</td>
<td>158</td>
</tr>
<tr>
<td></td>
<td><strong>Chapter 6</strong></td>
<td></td>
</tr>
<tr>
<td>6.1</td>
<td>Conditions for Starting/Stopping Triggers</td>
<td>161</td>
</tr>
<tr>
<td>6.2</td>
<td>Trace Modes</td>
<td>163</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Setting Triggers in Continuous Mode</td>
<td>163</td>
</tr>
<tr>
<td>6.2.1.1</td>
<td>Trace From Trigger A Onward</td>
<td>163</td>
</tr>
<tr>
<td>6.2.1.2</td>
<td>Trace From Trigger A to Trigger B</td>
<td>166</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Setting Triggers in Automatic (One-buffer) Mode</td>
<td>170</td>
</tr>
<tr>
<td>6.2.2.1</td>
<td>Trace From Trigger A Onward</td>
<td>170</td>
</tr>
<tr>
<td>6.2.2.2</td>
<td>Trace From Trigger A to Trigger B</td>
<td>173</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Setting Triggers in Profile-Only Mode</td>
<td>174</td>
</tr>
<tr>
<td>6.2.4</td>
<td>Setting Triggers in Expert Mode</td>
<td>175</td>
</tr>
<tr>
<td>6.3</td>
<td>Tracepoints on Data and Memory</td>
<td>177</td>
</tr>
<tr>
<td>6.3.1</td>
<td>From Variables View</td>
<td>177</td>
</tr>
<tr>
<td>6.3.2</td>
<td>From Memory View</td>
<td>179</td>
</tr>
<tr>
<td>6.4</td>
<td>Enable and Disable Tracepoints</td>
<td>181</td>
</tr>
<tr>
<td></td>
<td><strong>Chapter 7</strong></td>
<td></td>
</tr>
<tr>
<td>7.1</td>
<td>Setting Hardware Tracepoints</td>
<td>183</td>
</tr>
<tr>
<td>7.1.1</td>
<td>From Source Code - Kinetis Cortex M4 Core</td>
<td>184</td>
</tr>
<tr>
<td>7.1.2</td>
<td>From Source Code - Kinetis Cortex M0+ Core</td>
<td>188</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>7.1.3</td>
<td>From Trace and Profile Tab - Kinetis Cortex M4 Core</td>
<td>192</td>
</tr>
<tr>
<td>7.1.4</td>
<td>From Trace and Profile Tab - Kinetis Cortex M0+ Core</td>
<td>196</td>
</tr>
<tr>
<td>7.2</td>
<td>Setting Software Tracepoints</td>
<td>198</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Setting Software Tracepoints Manually</td>
<td>198</td>
</tr>
<tr>
<td>7.2.1.1</td>
<td>Using Freescale Toolchain</td>
<td>199</td>
</tr>
<tr>
<td>7.2.1.2</td>
<td>Using GCC Toolchain</td>
<td>203</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Setting Software Tracepoints Automatically</td>
<td>206</td>
</tr>
<tr>
<td>7.3</td>
<td>Viewing Tracepoints</td>
<td>209</td>
</tr>
<tr>
<td>7.3.1</td>
<td>View Full Path of Tracepoint Attribute</td>
<td>210</td>
</tr>
<tr>
<td>7.3.2</td>
<td>Group Tracepoints</td>
<td>210</td>
</tr>
<tr>
<td>7.3.3</td>
<td>Define Working Sets</td>
<td>211</td>
</tr>
<tr>
<td>7.3.4</td>
<td>Add New Analysispoint</td>
<td>212</td>
</tr>
<tr>
<td>7.3.5</td>
<td>Enable/Disable Tracepoints</td>
<td>213</td>
</tr>
<tr>
<td>7.3.6</td>
<td>Navigate to Tracepoint Line</td>
<td>214</td>
</tr>
<tr>
<td>7.3.7</td>
<td>Remove Tracepoints</td>
<td>214</td>
</tr>
<tr>
<td>7.3.8</td>
<td>Context Menu</td>
<td>214</td>
</tr>
<tr>
<td></td>
<td><strong>Chapter 8</strong> Data Visualization</td>
<td></td>
</tr>
<tr>
<td>8.1</td>
<td>Creating DSC Project</td>
<td>217</td>
</tr>
<tr>
<td>8.2</td>
<td>Configuring for Data Visualization</td>
<td>218</td>
</tr>
<tr>
<td>8.3</td>
<td>Setting Analysispoints for Data Visualization</td>
<td>219</td>
</tr>
<tr>
<td>8.3.1</td>
<td>Setting Data Analysispoints on Memory View</td>
<td>220</td>
</tr>
<tr>
<td>8.3.2</td>
<td>Setting Data Analysispoints on Variables View</td>
<td>221</td>
</tr>
<tr>
<td>8.3.3</td>
<td>Setting Register Analysispoints on Registers View</td>
<td>221</td>
</tr>
<tr>
<td>8.4</td>
<td>Collecting and Viewing Data</td>
<td>222</td>
</tr>
<tr>
<td></td>
<td><strong>Chapter 9</strong> Launching Scripts</td>
<td></td>
</tr>
<tr>
<td>9.1</td>
<td>Run Sample Python Script</td>
<td>229</td>
</tr>
<tr>
<td>9.2</td>
<td>Collect Trace Using Jython</td>
<td>231</td>
</tr>
</tbody>
</table>
Chapter 10
Simple Instrumentation Profiling on ColdFire V2 - V4e Targets

10.1 Include Profiler Library and Files.................................................................238
10.2 Configure Project for Profiling.................................................................240
10.3 Modify Source Code.............................................................................243
10.4 Debug Application and Collect Profiling Information......................253
10.5 View Profiling Results.........................................................................254
  10.5.1 Flat View..........................................................................................256
  10.5.2 Tree View.........................................................................................256
  10.5.3 Class View.......................................................................................257

Chapter 11
11.1 Trace Collection with Breakpoints.........................................................259

Chapter 12
Configuring Trace Registers in Source Code

12.1 HCS08.................................................................................................263
12.2 ColdFire V1........................................................................................264

Chapter 13
Low Power WAIT Mode

13.1 Configure Low Power WAIT State.......................................................269
13.2 View Low Power WAIT Results.........................................................271
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 v10.x. 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 v10.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-PCNXUS, 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.
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 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 > MC9S08QE128**.
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.

   ![CodeWarrior Projects View - HCS08 Project](image)

   **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 to CodeWarrior Development Studio for Microcontrollers V10.x Targeting Manual located at C:\Freescale\CW MCU v10.x \MCU\Help.

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.
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](image)

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 to CodeWarrior Development Studio for Microcontrollers V10.x Targeting Manual located at C:\Freescale\CW MCU v10.x \MCU\Help.

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.
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](image)

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 V10.x Targeting Manual located at C:\Freescale\CW MCU v10.x \MCU\Help.

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**.
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](image)

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.

### 3.1.5 Using MPC56xx Target

To create a new Microcontrollers project using the MPC56xx target:


   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](image)

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

**NOTE**

Use the same steps to configure launcher for e200 and DSC targets.

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 - Main Page](image)

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

### Table 3-1. Trace and Profile Options for HCS08

<table>
<thead>
<tr>
<th>Group</th>
<th>Options</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>User Options</strong></td>
<td><strong>Enable Logging</strong></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td><strong>Configuration Set in User Code</strong></td>
<td>When checked, lets you configure trace registers from the application without using the <strong>Trace and Profile</strong> page. In this scenario, you can write the appropriate registers in the source code to configure the trace mode and triggers.</td>
</tr>
</tbody>
</table>

Table continues on the next page...
Table 3-1. Trace and Profile Options for HCS08 (continued)

<table>
<thead>
<tr>
<th>Group</th>
<th>Options</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace Mode Options</td>
<td>Collect Program Trace</td>
<td>Consists of these options:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <strong>Continuously</strong> - 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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <strong>Automatically</strong> - 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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <strong>LOOP1 Mode</strong> - 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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For more information, refer <em>MC9S08QE128 Reference Manual</em>. <strong>NOTE:</strong> The <strong>LOOP1 Mode</strong> 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.</td>
</tr>
<tr>
<td></td>
<td>Collect Data Trace</td>
<td>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).</td>
</tr>
<tr>
<td></td>
<td>Profile-Only</td>
<td>When selected, collects trace by sampling the program counter (PC) from time to time.</td>
</tr>
</tbody>
</table>

*Table continues on the next page...*
Table 3-1. Trace and Profile Options for HCS08 (continued)

<table>
<thead>
<tr>
<th>Group</th>
<th>Options</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expert</td>
<td></td>
<td>When selected, enables the <strong>Configure Expert Settings</strong> 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: <a href="http://www.freescale.com/">http://www.freescale.com/</a></td>
</tr>
<tr>
<td>Trace Start/Stop Conditions</td>
<td>No Trigger</td>
<td>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 <a href="http://www.freescale.com/">Setting Tracepoints (HCS08)</a></td>
</tr>
<tr>
<td></td>
<td>Collect Trace From Trigger</td>
<td>Starts collecting trace when the triggers generate, that is when the condition for A and B is met. For more information on tracepoints, refer <a href="http://www.freescale.com/">Setting Tracepoints (HCS08)</a></td>
</tr>
<tr>
<td></td>
<td>Keep Last Buffer Before Trigger</td>
<td>When checked, overwrites the trace buffer during trace collection before the trigger is hit. When trigger is hit, trace starts collecting, gets appended to the existing buffer, and only the last part of the buffer is displayed in the <strong>Trace Data</strong> viewer. It gets enabled in the <strong>Continuously</strong> mode when the <strong>Collect Trace From Trigger</strong> option is selected.</td>
</tr>
<tr>
<td></td>
<td>Break on FIFO Full</td>
<td>While debugging, suspends the application automatically when buffer gets full. The checkbox gets enabled in the <strong>Automatically</strong> mode when the <strong>Collect Trace From Trigger</strong> option is selected.</td>
</tr>
<tr>
<td></td>
<td>Collect Trace Until Trigger</td>
<td>Starts collecting trace and stops when the condition for triggers, A and B is met. This option is not enabled in the <strong>Continuously</strong> mode. For more information on tracepoints, refer <a href="http://www.freescale.com/">Setting Tracepoints (HCS08)</a></td>
</tr>
<tr>
<td></td>
<td>Break on Trigger Hit</td>
<td>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 <strong>Collect Trace Until Trigger</strong> option is selected.</td>
</tr>
<tr>
<td>Trigger Type</td>
<td></td>
<td>Contains various conditions of triggers, A and B for starting/stopping trace collection.</td>
</tr>
</tbody>
</table>

*Table continues on the next page...*
### Table 3-1. Trace and Profile Options for HCS08 (continued)

<table>
<thead>
<tr>
<th>Group</th>
<th>Options</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>For more information on tracepoints, refer Setting Tracepoints (HCS08).</td>
</tr>
<tr>
<td>Instruction at Address A is Executed</td>
<td></td>
<td>Starts trace from the address or source line corresponding to trigger A. For more information, refer Setting Triggers in Automatically Mode.</td>
</tr>
<tr>
<td>Instruction at Address A or Address B is Executed</td>
<td></td>
<td>Starts trace from the address or source line corresponding to trigger A or trigger B whichever occurs first. For more information, refer Instruction at Address A or Address B is Executed.</td>
</tr>
<tr>
<td>Instruction Inside Range from Address A to Address B is Executed</td>
<td></td>
<td>Starts trace when any instruction in the range between trigger address A and trigger address B is executed. That is, when ([\text{address at trigger A}] \leq [\text{current address}] \leq [\text{address at trigger B}]). For more information, refer Instruction Inside Range from Address A to Address B is Executed.</td>
</tr>
<tr>
<td>Instruction Outside Range from Address A to Address B is Executed</td>
<td></td>
<td>Starts trace when any instruction outside the range between trigger address A and trigger address B is executed. That is, when ([\text{current address}] &lt; [\text{address at trigger A or address at trigger B}] &lt; [\text{current address}]). For more information, refer Instruction Outside Range from Address A to Address B is Executed.</td>
</tr>
<tr>
<td>Instruction at Address A, Then Instruction at Address B are Executed</td>
<td></td>
<td>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.</td>
</tr>
</tbody>
</table>

**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.

**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.
<table>
<thead>
<tr>
<th>Group</th>
<th>Options</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction at Address A is Executed and</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>Value on Data Bus Match</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>Value to Compare on Data Bus</td>
<td>Contains the value that you specify to be matched or not matched with the opcode read from trigger A address.</td>
<td></td>
</tr>
<tr>
<td>Capture Read/Write Values at Address B</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>Capture Read/Write Values at Address B,</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>After Access at Address A</td>
<td>Trigger Selection</td>
<td>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</td>
</tr>
</tbody>
</table>

Table continues on the next page...
Table 3-1. Trace and Profile Options for HCS08 (continued)

<table>
<thead>
<tr>
<th>Group</th>
<th>Options</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>you press resume. In this mode, the output of the comparator must propagate through an opcode tracking circuit before triggering FIFO actions.</td>
</tr>
<tr>
<td>Memory Access</td>
<td></td>
<td>When selected, allows memory access to both variables and instructions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For more information, refer Memory Access Triggers.</td>
</tr>
</tbody>
</table>

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

**Table 3-2. Trace and Profile Options for ColdFire V1**

<table>
<thead>
<tr>
<th>Group</th>
<th>Options</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Options</td>
<td>Enable Logging</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>Configuration Set in User Code</td>
<td>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.</td>
</tr>
</tbody>
</table>
Table 3-2. Trace and Profile Options for ColdFire V1 (continued)

<table>
<thead>
<tr>
<th>Group</th>
<th>Options</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select Trace Mode</td>
<td>Continuous</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>Automatic (One-buffer)</td>
<td>When selected, captures only the last instructions executed before the target gets suspended. It is totally unintrusive.</td>
</tr>
<tr>
<td></td>
<td>Halt the Target when Trace Buffer Gets Full</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>Profile-Only. Sample PC every cycles</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>Expert</td>
<td>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 <a href="http://www.freescale.com/">http://www.freescale.com/</a>.</td>
</tr>
<tr>
<td>Trace Start/Stop Conditions</td>
<td></td>
<td>Includes various conditions of triggers, A, B, and C, for starting and stopping trace. For more information, refer Conditions for Starting/Stopping Triggers.</td>
</tr>
<tr>
<td>Target PC Address</td>
<td>2 Bytes</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>3 Bytes</td>
<td>Select this default and recommended option if the PC address length in your program exceeds 16-bits (64K).</td>
</tr>
</tbody>
</table>

Table continues on the next page...
### Table 3-2. Trace and Profile Options for ColdFire V1 (continued)

<table>
<thead>
<tr>
<th>Group</th>
<th>Options</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>This feature is not supported for the <strong>Expert</strong> trace mode.</td>
</tr>
<tr>
<td>Trace data values</td>
<td>Read Data</td>
<td>Traces the values of data operands being read from the memory.</td>
</tr>
<tr>
<td></td>
<td>Write Data</td>
<td>Traces the values of data operands being written to the memory.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This feature is not supported for the <strong>Profile-Only</strong> and <strong>Expert</strong> trace modes.</td>
</tr>
</tbody>
</table>

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

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

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

*Table continues on the next page...*
### Table 3-3. Trace and Profile Options for Kinetis Cortex M4 Core (continued)

<table>
<thead>
<tr>
<th>Group</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>buffer overflow. A trace entry of <em>Trace buffer read finished</em> in the <strong>Trace Data</strong> viewer marks the point where an individual buffer ends.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>NOTE:</strong> This option is not the equivalent of continuous mode; in the <strong>Keep all trace buffers</strong> 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.</td>
</tr>
<tr>
<td>TPIU</td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>SWO</td>
<td></td>
<td>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 <strong>Debug Port Interface</strong> should be set as SWD.</td>
</tr>
<tr>
<td>Core clock</td>
<td></td>
<td>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 100Mhz works at 25Mhz, which is the default value. You can change the core clock value according to the requirements.</td>
</tr>
<tr>
<td>Tracelink</td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>Low Power Profiling</td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>Enable Logging</td>
<td></td>
<td>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.</td>
</tr>
</tbody>
</table>

*Table continues on the next page...*
### Table 3-3. Trace and Profile Options for Kinetis Cortex M4 Core (continued)

<table>
<thead>
<tr>
<th>Group</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Continuous Trace Collection</td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>ETM</td>
<td>Collect Program Trace</td>
<td>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).</td>
</tr>
<tr>
<td>ITM</td>
<td>Collect Instrumentation trace</td>
<td>Enables/disables trace output from the Instrumentation Trace Macrocell (ITM) block. For more information, refer Instrumentation Trace Macrocell (ITM).</td>
</tr>
</tbody>
</table>
| 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.
Figure 3-16. Setting Trace Buffer Size

7. Click **Apply**.

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

### 3.2.3.4 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.

![Figure 3-18. Selecting Debug Port Interface for JTrace](image)

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**.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug port interface</td>
<td>SWD</td>
</tr>
<tr>
<td>Delay after reset [ms]</td>
<td>JTAG</td>
</tr>
<tr>
<td>JTAG/SWD fixed speed [Khz]</td>
<td>SWD</td>
</tr>
<tr>
<td>JTAG/SWD speed</td>
<td>Auto</td>
</tr>
<tr>
<td>Reset</td>
<td>Kinetics</td>
</tr>
<tr>
<td>USB device</td>
<td>0</td>
</tr>
</tbody>
</table>

Refer **Collecting Data** to know how to collect data on Kinetis target. Refer **Viewing Data** to know how to view collected trace data.
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.
Figure 3-19. Preferences Dialog Box

The table below describes the various options available on the ETM Settings page.
<table>
<thead>
<tr>
<th>Group</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Settings</td>
<td>Stall Processor</td>
<td>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.</td>
</tr>
<tr>
<td>Trace All Branches</td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>Enable Timestamps</td>
<td></td>
<td>Enables timestamping when checked.</td>
</tr>
<tr>
<td>FIFOFULL Level (no. of bytes)</td>
<td></td>
<td>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.</td>
</tr>
</tbody>
</table>

**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)

<table>
<thead>
<tr>
<th>Group</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Stop Resource</td>
<td>Selects the corresponding DWT comparator to control trace stop.</td>
</tr>
<tr>
<td>Trigger Control</td>
<td></td>
<td>Defines an event that generates on meeting a condition and appears in the trace data.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>Function</td>
<td></td>
<td>Specifies the condition that must be met to generate a trigger signal on the trace port.</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td>Is the ETM event resource used for triggering. It can take four values:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DWT comparator - Any of the four comparators</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• ETM - Counter at zero - 16-bit counter reload value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Start/stop - Start or stop tracepoint set in the application</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Always true - Default option that triggers all the time</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B Same as A.</td>
</tr>
<tr>
<td></td>
<td>Index</td>
<td>Specifies the value of the comparator if you select DWT comparator in the A or B fields.</td>
</tr>
<tr>
<td></td>
<td>Collect Trace after Trigger</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>Collect Trace before Trigger</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>Collect Trace about Trigger</td>
<td>When selected, changes the value of Trigger Counter to 256 words in the Trigger Settings page.</td>
</tr>
</tbody>
</table>

Table continues on the next page...
### Table 3-4. Options Available in ETM Settings Page (continued)

<table>
<thead>
<tr>
<th>Group</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>Trace Enable Event</td>
<td></td>
<td>Enables the trace when an event occurs.</td>
</tr>
<tr>
<td><strong>NOTE:</strong> It is not guaranteed that trace will be generated exactly when the event occurs. It may have a few cycles delay.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Function</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>Same as A.</td>
</tr>
<tr>
<td></td>
<td>Index</td>
<td>Specifies the value of the comparator if you select DWT comparator in the A or B fields.</td>
</tr>
<tr>
<td>Timestamp Event</td>
<td></td>
<td>Generates a timestamp in trace when the event gets activated.</td>
</tr>
<tr>
<td></td>
<td>Function</td>
<td>Specifies the condition that must be met to generate the timestamp event.</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>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</td>
</tr>
</tbody>
</table>

*Table continues on the next page...*
### Table 3-4. Options Available in ETM Settings Page (continued)

<table>
<thead>
<tr>
<th>Group</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
</table>
|       |        | • 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**

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

**NOTE**

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

3. Click **ITM Settings** on left to display its corresponding options on right.
Figure 3-20. 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

<table>
<thead>
<tr>
<th>Group</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable ITM Tracing</td>
<td></td>
<td>Enables ITM tracing</td>
</tr>
<tr>
<td>Synchronize Packages</td>
<td></td>
<td>Enables packages synchronization when checked.</td>
</tr>
<tr>
<td>Timestamp</td>
<td></td>
<td>Enables ITM timestamping (delta).</td>
</tr>
<tr>
<td>No Timestamps</td>
<td></td>
<td>Disables timestamping.</td>
</tr>
<tr>
<td>Local Timestamps</td>
<td></td>
<td>Defines the number of ITM clock ticks on which the timestamp counts. ITM clock is a clock on 20 bits.</td>
</tr>
<tr>
<td></td>
<td>Timestamp Prescaler</td>
<td>Modifies the scaling of the timestamps clock. For example, if set to 16, the timestamp counts once every 16 clock ticks.</td>
</tr>
<tr>
<td>Global Timestamps</td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>Global Timestamp Frequency</td>
<td>Decides when the global cycle count is written in the trace buffer, that is every 128/8192 cycles or at every packet.</td>
</tr>
<tr>
<td>Enable Stimulus Registers</td>
<td></td>
<td>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 <strong>Select All</strong> or <strong>Clear All</strong> to check or clear all the checkboxes.</td>
</tr>
<tr>
<td>Cycle Counter</td>
<td>Enable</td>
<td>Enables cycle counter which counts the number of core cycles. The counting is suspended when the core halts in debug state.</td>
</tr>
<tr>
<td></td>
<td>Extension Counter Reload Value (hex)</td>
<td>Defines the cycle count event combining with Cycle Counter Tap and Synchronization Packet Rate.</td>
</tr>
</tbody>
</table>

*Table continues on the next page...*
<table>
<thead>
<tr>
<th>Group</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cycle Counter Tap</td>
<td>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. <strong>NOTE:</strong> Cycle count is a 32-bit, incrementing (up) cycle counter.</td>
</tr>
<tr>
<td></td>
<td>Synchronization Packet Rate</td>
<td>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 to 1 or 1 to 0).</td>
</tr>
<tr>
<td>Event Generation</td>
<td>Enable PC Sample</td>
<td>Controls PC sample event generation.</td>
</tr>
<tr>
<td></td>
<td>Enable Cycle Count Event Generation</td>
<td>Enables cycle count, allowing it to increment and generate synchronization and count events.</td>
</tr>
<tr>
<td></td>
<td>Enable Exception Trace</td>
<td>Enables exception trace that traces exception entry, exit and return to a pre-empted handler or thread.</td>
</tr>
<tr>
<td></td>
<td>Enable Exception Overhead Count</td>
<td>Enables exception overhead event. <strong>NOTE:</strong> 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.</td>
</tr>
<tr>
<td></td>
<td>Enable CPI Count</td>
<td>Enables the CPI count event.</td>
</tr>
<tr>
<td></td>
<td>Enable Sleep Overhead Count</td>
<td>Enables sleep overhead count event.</td>
</tr>
<tr>
<td></td>
<td>Enable LSU Count</td>
<td>Enables the Load Store Unit (LSU) count event. <strong>NOTE:</strong> The LSU counter increments on the additional cycles required to execute all load and store instructions.</td>
</tr>
<tr>
<td></td>
<td>Enable Folded Instruction Count</td>
<td>Enables the folded instruction count event.</td>
</tr>
</tbody>
</table>
Table 3-5. Options Available in ITM Settings Page

<table>
<thead>
<tr>
<th>Group</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOTE:</td>
<td>The folded instruction counter increments on any instruction that executes in zero cycles.</td>
<td></td>
</tr>
</tbody>
</table>

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

![Figure 3-21. ETB Settings Page](image)
The table below describes the various options available on the **ETM Settings** page.

**Table 3-6. Options Available in ETB Settings Page**

<table>
<thead>
<tr>
<th>Group</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable ETB Trace Capture</td>
<td>Option</td>
<td>Enables ETB tracing.</td>
</tr>
<tr>
<td>Continuous Trace Collection</td>
<td>Enable</td>
<td>Enables the continuous trace collection when checked. It also enables the <strong>Almost Full Level</strong> checkbox.</td>
</tr>
<tr>
<td></td>
<td>Almost Full Level (no. of words)</td>
<td>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.</td>
</tr>
<tr>
<td>Trace Formatting Settings</td>
<td>Enable Formatting</td>
<td>When checked, prevents triggers from being embedded into the formatted stream.</td>
</tr>
<tr>
<td></td>
<td>Continuous Formatting</td>
<td>Enables the continuous mode. In the ETB, this mode corresponds to the normal mode with the embedding of triggers.</td>
</tr>
<tr>
<td></td>
<td>Stop on Flush Completion</td>
<td>Stops trace formatting when a flush is completed. This forces the FIFO to drain off any partially completed packets.</td>
</tr>
<tr>
<td></td>
<td>Stop on Trigger Event</td>
<td>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.</td>
</tr>
</tbody>
</table>

*Table continues on the next page...*
<table>
<thead>
<tr>
<th>Group</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trigger Settings</td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>Trigger Counter (no. of words)</td>
<td>Defines the number of 32-bit words remaining to be stored in the ETB Trace RAM.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>NOTE:</strong> The hardware buffer can hold up to 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.</td>
</tr>
<tr>
<td></td>
<td>Indicate Trigger on TRIGN</td>
<td>Indicates a trigger on TRIGIN being asserted.</td>
</tr>
<tr>
<td></td>
<td>Indicate Trigger on Trigger Event</td>
<td>Indicates a trigger on a trigger event.</td>
</tr>
<tr>
<td></td>
<td>Indicate Trigger on Flush Completion</td>
<td>Indicates a trigger on flush completion.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
</tbody>
</table>

*Table continues on the next page...*
Table 3-6. Options Available in ETB Settings Page (continued)

<table>
<thead>
<tr>
<th>Group</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generate Flush Using FLUSHIN</td>
<td></td>
<td>Generates flush using the FLUSHIN interface.</td>
</tr>
<tr>
<td>Generate Flush Using Trigger Event</td>
<td></td>
<td>Generates flush using the trigger event.</td>
</tr>
</tbody>
</table>

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.

![Figure 3-22. Trace and Profile Tab Options of S12Z](image-url)
The table below describes the various **Trace and Profile** options for S12Z architecture.

### Table 3-7. Trace and Profile Options for S12Z

<table>
<thead>
<tr>
<th>Group</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>User Options</strong></td>
<td>Enable Logging</td>
<td>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.</td>
</tr>
<tr>
<td><strong>Target Configuration</strong></td>
<td>Continuous trace collection</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>Enable Timestamps</td>
<td>Enables time stamping when checked.</td>
</tr>
<tr>
<td></td>
<td>Stop on buffer full</td>
<td>Suspends the target automatically and stops trace collection when buffer is full.</td>
</tr>
<tr>
<td><strong>Trace Mode</strong></td>
<td></td>
<td>Allows you to select any one of the following trace modes:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Normal - collects trace in normal mode and stores change of flow (COF) program counter (PC) addresses.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Pure PC - collects trace in normal mode without timestamps.</td>
</tr>
<tr>
<td><strong>Trigger Mode</strong></td>
<td></td>
<td>Allows you to select any one of the following triggers for the trigger set in the source code:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• End Trigger - collects trace data after the trigger is hit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Start Trigger - collects trace data until the trigger is hit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Middle Trigger - collects some data until the trigger is hit and some data after the trigger hits.</td>
</tr>
</tbody>
</table>

*Table continues on the next page...*
Table 3-7. Trace and Profile Options for S12Z (continued)

<table>
<thead>
<tr>
<th>Group</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace from: 0x</td>
<td></td>
<td>Specifies the address from which you want to collect trace data.</td>
</tr>
<tr>
<td>to: 0x</td>
<td></td>
<td>Specifies the address till which you want to collect trace data.</td>
</tr>
<tr>
<td>State 1, 2, 3</td>
<td>Next State for Comp A/B/C/D</td>
<td>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. <strong>NOTE:</strong> 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 &gt; Toggle Trace Trigger A/B/C/D.</td>
</tr>
</tbody>
</table>

**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.

![Image of Trace and Profile Tab Options]

**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**

<table>
<thead>
<tr>
<th>Group</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable Trace and Profile</td>
<td></td>
<td>Enables tracing and profiling for the application.</td>
</tr>
<tr>
<td>User Options</td>
<td>Keep all trace buffers</td>
<td>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.</td>
</tr>
</tbody>
</table>

*Table continues on the next page...*
Table 3-8. Trace and Profile Options for ColdFire V2-V4 (continued)

<table>
<thead>
<tr>
<th>Group</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable Logging</td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>Target Configuration</td>
<td>Target PC Address</td>
<td>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.</td>
</tr>
<tr>
<td>Trace data values</td>
<td>Read Data</td>
<td>Traces the values of data operands being read from the memory.</td>
</tr>
<tr>
<td></td>
<td>Write Data</td>
<td>Traces the values of data operands being written to the memory.</td>
</tr>
</tbody>
</table>

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

**Table 3-9. Trace and Profile Options for DSC**

<table>
<thead>
<tr>
<th>Group</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable Trace and Profile</td>
<td></td>
<td>Enables trace and profiling for the DSC application.</td>
</tr>
<tr>
<td>Enable Data Visualization</td>
<td></td>
<td>Enables data visualization. Data Visualization and tracing cannot be used simultaneously. For details, refer Data Visualization.</td>
</tr>
<tr>
<td>User Options</td>
<td>Enable Logging</td>
<td>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.</td>
</tr>
<tr>
<td>Select Trace Mode</td>
<td>Continuous</td>
<td>When selected, collects the trace data continuously.</td>
</tr>
<tr>
<td>Select Trace Mode</td>
<td>Automatic (One-buffer)</td>
<td>When selected, captures only the last instructions executed before the target gets suspended.</td>
</tr>
<tr>
<td>Collect Trace</td>
<td>Always</td>
<td>Collects trace with any of the following trigger conditions selected:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Always — captures all executed instructions in the application</td>
</tr>
</tbody>
</table>
Table 3-9. Trace and Profile Options for DSC

<table>
<thead>
<tr>
<th>Group</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>From Trigger A Onward</td>
<td>starts trace collection from where trigger A is set in the application</td>
</tr>
<tr>
<td></td>
<td>From Trigger B Onward</td>
<td>starts trace collection from where trigger B is set in the application</td>
</tr>
<tr>
<td></td>
<td>Until Trigger A</td>
<td>collects trace till Trigger A is set in the application</td>
</tr>
<tr>
<td></td>
<td>Until Trigger B</td>
<td>collects trace till Trigger B is set in the application</td>
</tr>
</tbody>
</table>

**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().

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.
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.

![Figure 3-27. Log View of Trace Collection](image)

**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.
5. Click **OK**.

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

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.
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.

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.

![Figure 3-32. Stop Toggle Button](image)

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.

![Figure 3-33. Reset Button in Software Analysis View](image)

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 Cf2-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 Data Viewer](image)

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>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>Displays the order number of the instructions.</td>
</tr>
<tr>
<td>Event Source</td>
<td>Displays program trace messages from ETM (Merlin) or special messages from ITM.</td>
</tr>
<tr>
<td>Description</td>
<td>Displays detailed information about the trace line.</td>
</tr>
<tr>
<td>Source</td>
<td>Displays the source function of the trace line if it is a call or a branch.</td>
</tr>
</tbody>
</table>

Table continues on the next page...
Table 4-1. Trace Data - Description of Fields (continued)

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

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:

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

### 4.1.1 Finding and Filtering Trace Data

Click the Search button in the Trace Data viewer to find and filter the trace data displayed in the Trace Data viewer. The Find dialog box appears. You can search for a particular set of events based on a specific criteria and filter the events within the Trace Data viewer.
To find particular events in the trace data:

1. In the **Find** text box, type the text you want to find in the trace data.
2. Click **Find**. The row containing the specified text gets selected in the **Trace Data** viewer.
3. Click **Advanced** if you want to specify a search criteria for your entry.
   - A list of columns of the **Trace Data** viewer appears.
4. Double-click the column in the list box on which you want to define the search criteria.
   - A list of operators corresponding to the added column are supported.
   - For numerical fields, standard equivalency operators are provided, that is ==, >, !=, >=, <, <=, depending upon the field.
For text fields, *contains*, *doesn-t-contain*, *doesn-t-equal*, and *equals* operators are provided.

- *contains* - tests that the character string entered is contained in the data to be searched
- *doesn-t-contain* - tests that the character string entered is not contained in the data to be searched.
- *doesn-t-equal* - tests that the character string entered does not exactly match the string in the data to be searched.
- *equals* - tests that the character string entered exactly matches the string in the data to be searched.

5. Select the required operator from the list box by double-clicking it.
6. Double-click <!--value--> and enter the value for comparison.
7. If you want to add more to the search expression, select the AND or OR operator and define your entry using the same steps.

8. Click **Find**.

   The focus will move to the first event that matches the search expression, and the corresponding row containing the event in the Trace Data viewer will be selected.

9. Press F3 on your keyboard to locate the next event that matches the search expression.
The search starting point is determined by the currently selected cell in the Trace Data viewer and the search direction. If no cell is currently selected then the search starts at the beginning of the trace data if the search direction is forward or at the end of the trace data if the search direction is backward.

If a cell is currently selected then the search begins in the row immediately after the selected row if the search direction is forward, or in the row immediately preceding the selected row if the search direction is backward.

To filter events:, repeat steps 3 - 8 and click the Filter button in the Find dialog box and repeat the steps 3 - 8. The data in the Trace Data viewer will be filtered to show only the trace events that match the filter expression.

4.1.2 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.

   ![Export Trace Data to CSV Dialog Box](image)

   **Figure 4-5. 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.

4.1.3 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 4-6. Configuring Time Unit](image)

- **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

**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.
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 4-7. Configuring Time Format](image)

**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.

### 4.1.4 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.
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.

<table>
<thead>
<tr>
<th>Event Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCU</td>
<td>Function main, address = 0xa4e.</td>
</tr>
<tr>
<td>MCU</td>
<td>Branch from main to entry. Source address = 0xa50. Ta...</td>
</tr>
<tr>
<td>MCU</td>
<td>Function entry, address = 0x982.</td>
</tr>
<tr>
<td>MCU</td>
<td>Branch from entry to Performance1. Source address = ...</td>
</tr>
<tr>
<td>MCU</td>
<td>Function Performance1, address = 0xa1c.</td>
</tr>
<tr>
<td>MCU</td>
<td>Branch from Performance1 to entry. Source address = ...</td>
</tr>
<tr>
<td>MCU</td>
<td>Function entry, address = 0x988.</td>
</tr>
<tr>
<td>MCU</td>
<td>Branch from entry to Recursive. Source address = 0x98...</td>
</tr>
<tr>
<td>MCU</td>
<td>Function Recursive, address = 0xa2a.</td>
</tr>
<tr>
<td>MCU</td>
<td>Branch from Recursive to Recursive. Source address = ...</td>
</tr>
<tr>
<td>MCU</td>
<td>Function Recursive, address = 0xa36.</td>
</tr>
<tr>
<td>MCU</td>
<td>Branch from Recursive to Recursive. Source address = ...</td>
</tr>
<tr>
<td>MCU</td>
<td>Function Recursive, address = 0xa2e.</td>
</tr>
</tbody>
</table>

**Figure 4-9. 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.
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.

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.

![Timeline Viewer Displaying Timeline Data](image)

**Figure 4-12. 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 4-13. Selection Mode to Measure Difference of Cycles Between Functions](image)

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.

![Edit Groups Dialog Box](image)

**Figure 4-14. Edit Groups Dialog Box**

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

- Add/Remove Function
- Edit Address Range of Function
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.

   ![Figure 4-15. Adding Group](image)

   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.

   ![Figure 4-16. Edit Groups Dialog Box After Editing Address Range and Color of Group](image)

   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.

![Timeline After Adding Group](image)

**Figure 4-17. 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.

![Merging Functions/Groups](image)

**Figure 4-18. 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.

![Critical Code Viewer](image)

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**

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

![Critical Code Viewer Screenshot](image)

**Figure 4-20. Statistics of Critical Code Data**

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Function</th>
<th>Coverage %</th>
<th>ASM Decision Coverage %</th>
<th>ASM Count</th>
<th>Time (CPU cycles)</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x060</td>
<td>InterruptTest</td>
<td>100 %</td>
<td>0 %</td>
<td>20</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>0x068</td>
<td>Launch</td>
<td>100 %</td>
<td>0 %</td>
<td>50</td>
<td>0</td>
<td>24</td>
</tr>
<tr>
<td>0x080</td>
<td>entry</td>
<td>85.37 %</td>
<td>75 %</td>
<td>921</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>0x0a4</td>
<td>PerformanceWork</td>
<td>100 %</td>
<td>100 %</td>
<td>95</td>
<td>0</td>
<td>50</td>
</tr>
<tr>
<td>0x18</td>
<td>Performance</td>
<td>100 %</td>
<td>0 %</td>
<td>36</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>0x20</td>
<td>Recursive</td>
<td>100 %</td>
<td>100 %</td>
<td>279</td>
<td>0</td>
<td>42</td>
</tr>
<tr>
<td>0x4c</td>
<td>main()</td>
<td>50 %</td>
<td>0 %</td>
<td>3</td>
<td>0</td>
<td>14</td>
</tr>
</tbody>
</table>
Table 4-3. Description of Statistics of Critical Code Data

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line/Address</td>
<td>Displays either the line number for each instruction in the source code or the address for the assembly code.</td>
</tr>
<tr>
<td>Instruction</td>
<td>Displays all the instructions executed in the selected function.</td>
</tr>
<tr>
<td>Coverage</td>
<td>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.</td>
</tr>
<tr>
<td>ASM Decision Coverage</td>
<td>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.</td>
</tr>
<tr>
<td>ASM Count</td>
<td>Displays the number of times each instruction is executed.</td>
</tr>
<tr>
<td>Time (CPU Cycles)</td>
<td>Displays the total number of clock cycles that each instruction in the function takes.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search</td>
<td>![Search icon]</td>
<td>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.</td>
</tr>
<tr>
<td>Previous function</td>
<td>![Previous function icon]</td>
<td>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.</td>
</tr>
<tr>
<td>Next function</td>
<td>![Next function icon]</td>
<td>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.</td>
</tr>
<tr>
<td>Export</td>
<td>![Export icon]</td>
<td>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.</td>
</tr>
</tbody>
</table>
Table 4-4. Buttons Available in Statistics View of Critical Code Tab (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configure table</td>
<td><img src="image" alt="image" /></td>
<td>Lets you show and hide column(s) of the critical code data. Click the button and select the <strong>Configure the table above</strong> option to show/hide columns of the top view or the <strong>Configure the table below</strong> option to show/hide columns of the bottom view. The <strong>Drag and drop to order columns</strong> dialog box appears in which you can check/uncheck the checkboxes corresponding to the available columns to show/hide them in the <strong>Critical Code</strong> viewer. The option also allows you to set CPU frequency and set time in cycles, milliseconds, microseconds, and nanoseconds.</td>
</tr>
<tr>
<td>Graphics</td>
<td><img src="image" alt="image" /></td>
<td>Lets you display the histograms in two colors for the <strong>CPU Cycles</strong> and <strong>ASM Count</strong> columns in the bottom view of the critical code data. Click the button and select the <strong>Assembly/Source &gt; ASM Count</strong> or <strong>Assembly/Source &gt; CPU Cycles</strong> option to display histograms in the <strong>ASM Count</strong> or <strong>CPU Cycles</strong> column. The colors in these columns differentiate source code with the assembly code.</td>
</tr>
<tr>
<td>Show code</td>
<td><img src="image" alt="image" /></td>
<td>Lets you display the assembly or mixed code in the statistics of the critical code data. Click the button and select the <strong>Assembly</strong> option to display the assembly code or the <strong>Mixed</strong> option to display the mixed code in the statistical details.</td>
</tr>
</tbody>
</table>

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

**Table 4-5. Description of File Coverage Data**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Displays the name of the file.</td>
</tr>
<tr>
<td>Number of Lines</td>
<td>Displays the size of the file in number of lines present in the file.</td>
</tr>
<tr>
<td>Lines Executed</td>
<td>Displays the number of lines executed in the file.</td>
</tr>
<tr>
<td>Coverage %</td>
<td>Displays the percentage of number of lines executed compared to the total number of lines, that is ((lines \text{ executed} / number \text{ of lines}) \times 100).</td>
</tr>
</tbody>
</table>

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.
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**

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

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caller</td>
<td>Name of the calling function.</td>
</tr>
<tr>
<td>Callee</td>
<td>Name of the function that is called by the calling function.</td>
</tr>
<tr>
<td>Num Calls</td>
<td>Number of times the caller called the callee.</td>
</tr>
<tr>
<td>Inclusive</td>
<td>Cumulative metric count during execution time spent from function entry to exit.</td>
</tr>
<tr>
<td>Min Inclusive</td>
<td>Minimum metric count during execution time spent from function entry to exit.</td>
</tr>
<tr>
<td>Max Inclusive</td>
<td>Maximum metric count during execution time spent from function entry to exit.</td>
</tr>
</tbody>
</table>

*Table continues on the next page...*
Table 4-7. Field Description of Call Site Performance Table (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg Inclusive</td>
<td>Average metric count during execution time spent from function entry to exit.</td>
</tr>
<tr>
<td>Percent Callee</td>
<td>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.</td>
</tr>
<tr>
<td>Percent Caller</td>
<td>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.</td>
</tr>
<tr>
<td>Call Site</td>
<td>Address from where the function was called.</td>
</tr>
</tbody>
</table>

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

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.

Figure 4-24. 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.
3. Click **Next** to display the **Import Trace** page of the **Import** wizard.
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 4-26 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.
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 4-26, 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 Dialog Box](image)

**Figure 4-28. 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
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

```c
#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()
```

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

**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.
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.

![Figure 5-2. Trace Data After Setting From Trigger in Continuously Mode](image)

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.

<table>
<thead>
<tr>
<th>Index</th>
<th>Event Source</th>
<th>Description</th>
<th>Source</th>
<th>Target</th>
<th>Type</th>
<th>TimeStamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MCU</td>
<td>Function main, address = 0x80b1.</td>
<td>main</td>
<td></td>
<td>Linear</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>MCU</td>
<td>Branch from main to f, Source address = 0x80b2. Target address = 0x8092.</td>
<td>main</td>
<td>f</td>
<td>Branch</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>MCU</td>
<td>Branch from f to main. Source address = 0x8092. Target address = 0x80b1.</td>
<td>f</td>
<td>main</td>
<td>Branch</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>MCU</td>
<td>Function main, address = 0x80b1.</td>
<td>main</td>
<td></td>
<td>Linear</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>MCU</td>
<td>Branch from main to f, Source address = 0x80b2. Target address = 0x8092.</td>
<td>main</td>
<td>f</td>
<td>Branch</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>MCU</td>
<td>Branch from f to main. Source address = 0x8092. Target address = 0x80b1.</td>
<td>f</td>
<td>main</td>
<td>Branch</td>
<td>24</td>
</tr>
<tr>
<td>6</td>
<td>MCU</td>
<td>Branch from main to f. Source address = 0x80b4. Target address = 0x8092.</td>
<td>main</td>
<td>f</td>
<td>Branch</td>
<td>29</td>
</tr>
<tr>
<td>7</td>
<td>MCU</td>
<td>Branch from f to main. Source address = 0x8092. Target address = 0x80b6.</td>
<td>f</td>
<td>main</td>
<td>Branch</td>
<td>35</td>
</tr>
<tr>
<td>8</td>
<td>MCU</td>
<td>Branch from main to f. Source address = 0x80b6. Target address = 0x8092.</td>
<td>main</td>
<td>f</td>
<td>Branch</td>
<td>40</td>
</tr>
<tr>
<td>9</td>
<td>MCU</td>
<td>Branch from f to main. Source address = 0x8092. Target address = 0x80b1.</td>
<td>f</td>
<td>main</td>
<td>Branch</td>
<td>46</td>
</tr>
<tr>
<td>10</td>
<td>MCU</td>
<td>Function main, address = 0x80b1.</td>
<td>main</td>
<td></td>
<td>Linear</td>
<td>56</td>
</tr>
</tbody>
</table>

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.
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 $f_1()$ function, which occurs first in execution.
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**

```c
#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)
{
```

---

**CodeWarrior Development Studio for Microcontrollers Version 10.x Profiling and Analysis Tools Users Guide, Rev. 10.6, 02/2014**

Freescale Semiconductor, Inc.
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 \texttt{ret = 1;} and trigger B at \texttt{ret = 2;} in the \texttt{PerformanceWork()} function.

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.

**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.

![Disassembly View](image1)

**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.

![Value to Compare on Data Bus](image2)

**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.

<table>
<thead>
<tr>
<th>Index</th>
<th>Event ...</th>
<th>Description</th>
<th>Call/Branch</th>
<th>Type</th>
<th>Timeout...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MCU</td>
<td>Function main, address = 0x8104.</td>
<td>main</td>
<td>Linear</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>MCU</td>
<td>Branch from main to Performance1. Source address = 0x8...</td>
<td>main</td>
<td>Performance... Branch</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>MCU</td>
<td>Branch from Performance1 to main. Source address = 0x8...</td>
<td>Performance... main</td>
<td>Branch</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>MCU</td>
<td>Function main, address = 0x8104.</td>
<td>main</td>
<td>Linear</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>MCU</td>
<td>Branch from main to Performance1. Source address = 0x8...</td>
<td>main</td>
<td>Performance... Branch</td>
<td>18</td>
</tr>
<tr>
<td>6</td>
<td>MCU</td>
<td>Branch from Performance1 to &lt;unknown&gt;, Source address = 0x8...</td>
<td>Performance... &lt;unknown&gt;</td>
<td>Branch</td>
<td>24</td>
</tr>
<tr>
<td>7</td>
<td>MCU</td>
<td>Function &lt;unknown&gt;, address = 0xfffffff.</td>
<td>&lt;unknown&gt;</td>
<td>Linear</td>
<td>27</td>
</tr>
<tr>
<td>8</td>
<td>MCU</td>
<td>Branch from main to Recursive. Source address = 0x810a...</td>
<td>main</td>
<td>Recursive</td>
<td>32</td>
</tr>
<tr>
<td>9</td>
<td>MCU</td>
<td>Function Recursive, address = 0x80e7.</td>
<td>Recursive</td>
<td>Linear</td>
<td>39</td>
</tr>
<tr>
<td>10</td>
<td>MCU</td>
<td>Branch from Recursive to Recursive. Source address = 0x8...</td>
<td>Recursive</td>
<td>Recursive</td>
<td>42</td>
</tr>
<tr>
<td>11</td>
<td>MCU</td>
<td>Function Recursive, address = 0x80f0.</td>
<td>Recursive</td>
<td>Linear</td>
<td>44</td>
</tr>
<tr>
<td>12</td>
<td>MCU</td>
<td>Branch from Recursive to Recursive. Source address = 0x8...</td>
<td>Recursive</td>
<td>Recursive</td>
<td>49</td>
</tr>
<tr>
<td>13</td>
<td>MCU</td>
<td>Function Recursive, address = 0x80e7.</td>
<td>Recursive</td>
<td>Linear</td>
<td>56</td>
</tr>
<tr>
<td>14</td>
<td>MCU</td>
<td>Branch from Recursive to Recursive. Source address = 0x8...</td>
<td>Recursive</td>
<td>Recursive</td>
<td>59</td>
</tr>
<tr>
<td>15</td>
<td>MCU</td>
<td>Function Recursive, address = 0x80f0.</td>
<td>Recursive</td>
<td>Linear</td>
<td>61</td>
</tr>
</tbody>
</table>

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 instructions 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 dropdown 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.

---

**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**.
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**

```c
#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 */
}
```

---

**Freescale Semiconductor, Inc.**
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()`.

![Disassembly View](image)

**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 $f_2()$ as highlighted in the figure, and continues till buffer trace gets full.

![Figure 5-13. Trace Data After Setting From Trigger in Automatically Mode](image-url)
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 dropdown 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().

![Disassembly View - Selecting Function Address](image)

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.

![Monitor Memory Dialog Box](image)

**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.

![Memory View](image)

**Figure 5-16. Memory View**

The Toggle HCS08 Trace Trigger dialog box appears.
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.

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.
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.
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)`.

![Table showing trace data](image)

<table>
<thead>
<tr>
<th>Index</th>
<th>Event Source</th>
<th>Description</th>
<th>Call/Branch</th>
<th>Type</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MCU</td>
<td>Branch from Recursive to Recursive. Source address = 0x11...</td>
<td>Recursive</td>
<td>Branch</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>MCU</td>
<td>Function Recursive, address = 0x1910.</td>
<td>Recursive</td>
<td>Linear</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>MCU</td>
<td>Branch from Recursive to Recursive. Source address = 0x1...</td>
<td>Recursive</td>
<td>Branch</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>MCU</td>
<td>Function Recursive, address = 0x1917.</td>
<td>Recursive</td>
<td>Linear</td>
<td>17</td>
</tr>
<tr>
<td>4</td>
<td>MCU</td>
<td>Branch from Recursive to Recursive. Source address = 0x11...</td>
<td>Recursive</td>
<td>Branch</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>MCU</td>
<td>Function Recursive, address = 0x1910.</td>
<td>Recursive</td>
<td>Linear</td>
<td>22</td>
</tr>
<tr>
<td>6</td>
<td>MCU</td>
<td>Branch from Recursive to Recursive. Source address = 0x1...</td>
<td>Recursive</td>
<td>Branch</td>
<td>27</td>
</tr>
<tr>
<td>7</td>
<td>MCU</td>
<td>Function Recursive, address = 0x1917.</td>
<td>Recursive</td>
<td>Linear</td>
<td>34</td>
</tr>
<tr>
<td>8</td>
<td>MCU</td>
<td>Branch from Recursive to Recursive. Source address = 0x11...</td>
<td>Recursive</td>
<td>Branch</td>
<td>37</td>
</tr>
<tr>
<td>9</td>
<td>MCU</td>
<td>Function Recursive, address = 0x1910.</td>
<td>Recursive</td>
<td>Linear</td>
<td>39</td>
</tr>
<tr>
<td>10</td>
<td>MCU</td>
<td>Branch from Recursive to Recursive. Source address = 0x1...</td>
<td>Recursive</td>
<td>Branch</td>
<td>44</td>
</tr>
</tbody>
</table>

**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**

```c
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.

![Figure 5-23. Trace Data - Instruction Outside Range](image-url)
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** dropdown 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.
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.

   **Figure 5-26. Trace Collected at Address B**

<table>
<thead>
<tr>
<th>Index</th>
<th>Event Source</th>
<th>Description</th>
<th>Call/Branch Source</th>
<th>Type</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MCU</td>
<td>Data addr = 0x100 val = 0x0 (0)</td>
<td></td>
<td>Data Trace</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>MCU</td>
<td>Data addr = 0x100 val = 0x1 (1)</td>
<td></td>
<td>Data Trace</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>MCU</td>
<td>Data addr = 0x100 val = 0x1 (1)</td>
<td></td>
<td>Data Trace</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>MCU</td>
<td>Data addr = 0x100 val = 0x2 (2)</td>
<td></td>
<td>Data Trace</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>MCU</td>
<td>Data addr = 0x100 val = 0x2 (2)</td>
<td></td>
<td>Data Trace</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>MCU</td>
<td>Data addr = 0x100 val = 0x2 (2)</td>
<td></td>
<td>Data Trace</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>MCU</td>
<td>Data addr = 0x100 val = 0x3 (3)</td>
<td></td>
<td>Data Trace</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>MCU</td>
<td>Data addr = 0x100 val = 0x3 (3)</td>
<td></td>
<td>Data Trace</td>
<td>0</td>
</tr>
</tbody>
</table>

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.
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.

![Figure 5-28. Setting Trigger A in Collect Data Trace Mode](image)

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.

![Figure 5-29. Analysispoints View with Trigger A and Trigger B](image)

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.

<table>
<thead>
<tr>
<th>Index</th>
<th>Event Source</th>
<th>Description</th>
<th>Call/Branch Source</th>
<th>Type</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MCU</td>
<td>Data addr = 0x100 val = 0x2 (2)</td>
<td></td>
<td>Data Trace</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>MCU</td>
<td>Data addr = 0x100 val = 0x3 (3)</td>
<td></td>
<td>Data Trace</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>MCU</td>
<td>Data addr = 0x100 val = 0x3 (3)</td>
<td></td>
<td>Data Trace</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>MCU</td>
<td>Data addr = 0x100 val = 0x4 (4)</td>
<td></td>
<td>Data Trace</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>MCU</td>
<td>Data addr = 0x100 val = 0x4 (4)</td>
<td></td>
<td>Data Trace</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>MCU</td>
<td>Data addr = 0x100 val = 0x4 (4)</td>
<td></td>
<td>Data Trace</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>MCU</td>
<td>Data addr = 0x100 val = 0x5 (5)</td>
<td></td>
<td>Data Trace</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>MCU</td>
<td>Data addr = 0x100 val = 0x5 (5)</td>
<td></td>
<td>Data Trace</td>
<td>0</td>
</tr>
</tbody>
</table>

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.
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.

![Critical Code Viewer](image)

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 Dialog Box](image)

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.

![Figure 5-34. Disabling/Enabling the Trigger from Analysispoints View](image)

For detailed information on the **Analysispoints** view, refer [Viewing Tracepoints](#).
Enabling and Disabling the 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:

- Conditions for Starting/Stopping Triggers
- Trace Modes
- Tracepoints on Data and Memory
- Enable and Disable Tracepoints

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**

```c
#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.
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.
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.

![Figure 6-4. Setting Triggers A and B](image)

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

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.

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

<table>
<thead>
<tr>
<th>Index</th>
<th>Event Source</th>
<th>Description</th>
<th>Call/Branch</th>
<th>Type</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MCU</td>
<td>Function f, address = 0x76e</td>
<td>f</td>
<td>Linear</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>MCU</td>
<td>Branch from f to main. Source address = 0x772. Target ad...</td>
<td>f</td>
<td>main</td>
<td>Branch</td>
</tr>
<tr>
<td>2</td>
<td>MCU</td>
<td>Function main, address = 0x7e0</td>
<td>main</td>
<td>Linear</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>MCU</td>
<td>Branch from main to main. Source address = 0x7c4. Target...</td>
<td>main</td>
<td>main</td>
<td>Branch</td>
</tr>
<tr>
<td>4</td>
<td>MCU</td>
<td>Branch from main to f. Source address = 0x750. Target ad...</td>
<td>main</td>
<td>f</td>
<td>Branch</td>
</tr>
<tr>
<td>5</td>
<td>MCU</td>
<td>Function f, address = 0x7ee</td>
<td>f</td>
<td>Linear</td>
<td>37</td>
</tr>
<tr>
<td>6</td>
<td>MCU</td>
<td>Branch from f to main. Source address = 0x772. Target ad...</td>
<td>f</td>
<td>main</td>
<td>Branch</td>
</tr>
<tr>
<td>7</td>
<td>MCU</td>
<td>Function main, address = 0x7e0</td>
<td>main</td>
<td>Linear</td>
<td>51</td>
</tr>
<tr>
<td>8</td>
<td>MCU</td>
<td>Branch from main to main. Source address = 0x7c4. Target...</td>
<td>main</td>
<td>main</td>
<td>Branch</td>
</tr>
<tr>
<td>9</td>
<td>MCU</td>
<td>Branch from main to f. Source address = 0x750. Target ad...</td>
<td>main</td>
<td>f</td>
<td>Branch</td>
</tr>
<tr>
<td>10</td>
<td>MCU</td>
<td>Function f, address = 0x76e</td>
<td>f</td>
<td>Linear</td>
<td>66</td>
</tr>
</tbody>
</table>
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, in case 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.
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(;;)`.

![Disassembly View](image)

**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 $f_2()$ 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.
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.

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

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.

![Critical Code Data](image.png)

**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 Dialog Box](image-url)

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.

   ![](image)

   **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.

   ![](image)

   **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.
The variable \( a \) is accessed first time in the \( f_2() \) function. Therefore, after setting trigger C on the address of \( a \), the trace data starts collecting from \( f_2() \). 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.

---

<table>
<thead>
<tr>
<th>Index</th>
<th>Event Source</th>
<th>Description</th>
<th>Call/Branch Source</th>
<th>Type</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MCU</td>
<td>Function ( f_2 ), address = 0x796.</td>
<td>( f_2 )</td>
<td>Linear</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>MCU</td>
<td>Branch from ( f_2 ) to main. Source address = 0x79a....</td>
<td>( f_2 ) main</td>
<td>Branch</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>MCU</td>
<td>Function main, address = 0x7ca.</td>
<td>main</td>
<td>Linear</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>MCU</td>
<td>Branch from main to ( f_1 ). Source address = 0x7ce....</td>
<td>( f_1 ) main</td>
<td>Branch</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>MCU</td>
<td>Function ( f_1 ), address = 0x786.</td>
<td>( f_1 )</td>
<td>Linear</td>
<td>21</td>
</tr>
<tr>
<td>6</td>
<td>MCU</td>
<td>Branch from main to main. Source address = 0x7...</td>
<td>main</td>
<td>Branch</td>
<td>23</td>
</tr>
<tr>
<td>7</td>
<td>MCU</td>
<td>Function main, address = 0x7ca.</td>
<td>main</td>
<td>Linear</td>
<td>25</td>
</tr>
<tr>
<td>8</td>
<td>MCU</td>
<td>Branch from main to ( f_1 ). Source address = 0x7ce....</td>
<td>( f_1 ) main</td>
<td>Branch</td>
<td>38</td>
</tr>
<tr>
<td>9</td>
<td>MCU</td>
<td>Function ( f_1 ), address = 0x782.</td>
<td>( f_1 )</td>
<td>Linear</td>
<td>37</td>
</tr>
<tr>
<td>10</td>
<td>MCU</td>
<td>Branch from ( f_2 ) to main. Source address = 0x79a....</td>
<td>( f_2 ) main</td>
<td>Branch</td>
<td>42</td>
</tr>
<tr>
<td>11</td>
<td>MCU</td>
<td>Branch from main to main. Source address = 0x7...</td>
<td>main</td>
<td>Branch</td>
<td>44</td>
</tr>
</tbody>
</table>

---

**Figure 6-16. Trace Data After Setting Trace from Trigger C in Continuous Mode**

---

*Freescale Semiconductor, Inc.*
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.

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.
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
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**

```c
#include <stdio.h>

volatile int a;

void func() {
    a=0;
}

int main()
{
    int i;
    func();
    a=0;
    a=1;
}
```

**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.
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.
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.
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.

   ![Figure 7-3. Trace Results After Setting Hardware Tracepoints in Source Code - Start Address](image)

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.
### Setting Hardware Tracepoints

<table>
<thead>
<tr>
<th>Index</th>
<th>Event Source</th>
<th>Description</th>
<th>Call/Branch</th>
<th>Type</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>ETM</td>
<td>Function main, address = 0x1ff8266.</td>
<td>main</td>
<td>Linear</td>
<td>2852996694983</td>
</tr>
<tr>
<td>17</td>
<td>ETM</td>
<td>Function main, address = 0x1ff8272.</td>
<td>main</td>
<td>Linear</td>
<td>2852996694983</td>
</tr>
<tr>
<td>18</td>
<td>ETM</td>
<td>Function main, address = 0x1ff8276.</td>
<td>main</td>
<td>Linear</td>
<td>2852996694987</td>
</tr>
<tr>
<td>19</td>
<td>ETM</td>
<td>Function main, address = 0x1ff827c.</td>
<td>main</td>
<td>Linear</td>
<td>2852996694991</td>
</tr>
<tr>
<td>20</td>
<td>ETM</td>
<td>Function main, address = 0x1ff8280.</td>
<td>main</td>
<td>Linear</td>
<td>2852996694994</td>
</tr>
<tr>
<td>22</td>
<td>JTM</td>
<td>Asserted DWT packet - Cyc counter overflowed</td>
<td>Signal</td>
<td>73</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>ETM</td>
<td>Function main, address = 0x1ff8284.</td>
<td>main</td>
<td>Linear</td>
<td>2852996694994</td>
</tr>
<tr>
<td>24</td>
<td>ETM</td>
<td>Function main, address = 0x1ff8286.</td>
<td>main</td>
<td>Linear</td>
<td>2852996694999</td>
</tr>
<tr>
<td>25</td>
<td>JTM</td>
<td>Asserted DWT packet - Cyc counter overflowed - timest...</td>
<td>Signal</td>
<td>136</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>ETM</td>
<td>Function main, address = 0x1ff828a.</td>
<td>main</td>
<td>Linear</td>
<td>2852996694998</td>
</tr>
</tbody>
</table>

**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**

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

#define MAX_IT 5
#define SIMPLE

typedef int(*FUNC_TYPE)(int);

void Launch(FUNC_TYPE, int);
void entry();
```

Freescale Semiconductor, Inc.
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)
{
#ifdef 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:
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.
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.
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.
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**.

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**.
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()`.

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.

![Timeline Results](image)

**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 from 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,.`
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 = 0x1FF0000, LENGTH = 0x00001E0
   m_text (RX) : ORIGIN = 0x1FF01E0, LENGTH = 0x00020000-0x00001E0
   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);
   }
   ```
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;
__SF_INIT = . + 0x00000000;
_heap_addr = __SF_INIT;
_heap_size = 0x00000000;

__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 your 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.
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:

---

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

<table>
<thead>
<tr>
<th>Index</th>
<th>Event Source</th>
<th>Description</th>
<th>Call/Branch Source</th>
<th>Call/Branch Target</th>
<th>Type</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ETM</td>
<td>Trigger packet - ET8</td>
<td>Info</td>
<td>Linear</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>ETM</td>
<td>SYNC packet - ETM</td>
<td>Info</td>
<td>Linear</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>ETM</td>
<td>ISYNC PACKET - ETM - tracing enabled, addr=0xffff832e , ...</td>
<td>Custom</td>
<td>Linear</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>ETM</td>
<td>Trigger packet - ETM</td>
<td>Info</td>
<td>Linear</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>ETM</td>
<td>Function setupTracepoints, address = 0xffff833a.</td>
<td>setupTracepoints</td>
<td>Linear</td>
<td>131018795</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>ETM</td>
<td>Function setupTracepoints, address = 0xffff8330.</td>
<td>setupTracepoints</td>
<td>Linear</td>
<td>131018795</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>ETM</td>
<td>Function setupTracepoints, address = 0xffff8336.</td>
<td>setupTracepoints</td>
<td>Linear</td>
<td>131018795</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>ETM</td>
<td>Function setupTracepoints, address = 0xffff8338.</td>
<td>setupTracepoints</td>
<td>Linear</td>
<td>131018795</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>ETM</td>
<td>Function setupTracepoints, address = 0xffff833a.</td>
<td>setupTracepoints</td>
<td>Linear</td>
<td>131018795</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>ETM</td>
<td>Branch from setupTracepoints to setupTracepoints. Source ... setuptracepoints... setupTracepoints...</td>
<td>Branch</td>
<td>Linear</td>
<td>131018795</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>ETM</td>
<td>Function setupTracepoints, address = 0xffff837a.</td>
<td>setupTracepoints</td>
<td>Linear</td>
<td>131018797</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>ETM</td>
<td>Function setupTracepoints, address = 0xffff8376.</td>
<td>setupTracepoints</td>
<td>Linear</td>
<td>131018797</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>ETM</td>
<td>Branch from setupTracepoints to sa_interrupt_handler. Source ... setuptracepoints... sa_interrupt...</td>
<td>Branch</td>
<td>Linear</td>
<td>131018809</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>ETM</td>
<td>Function sa_interrupt_handler, address = 0xffff8390.</td>
<td>sa_interrupt...</td>
<td>Linear</td>
<td>131018812</td>
<td></td>
</tr>
</tbody>
</table>

**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.

---
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:

```c
sa_interrupt_handler,
```

5. Save the `kinetis_sysinit.c` file.

---

Figure 7-18. Interrupt Vector Table
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](image)

```c
/* Specify the memory areas */
MEMORY
{
  m_interrupts  (rx) : ORIGIN = 0x1FFFF000, LENGTH = 0x100
  m_text        (rx) : ORIGIN = 0x1FFFF0100, LENGTH = 64K-0x1ED0  /* 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:

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

**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.

*CodeWarrior Development Studio for Microcontrollers Version 10.x Profiling and Analysis Tools Users Guide, Rev. 10.6, 02/2014*
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.

![Figure 7-22. Analysispoints View](image)

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
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.

![Figure 7-23. Analysispoints View - Group by Projects](image)
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.

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.
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 **Toggle Default** 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 **Toggle Default** 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.

---

Figure 7-25. Analysispoints Grouped by Working Sets
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.

**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.

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.
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:
Configuring for Data Visualization

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.
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.

This topic contains the following sub-topics:

- Setting Data Analysispoints on Memory 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.

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

```c
#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");

```

CodeWarrior Development Studio for Microcontrollers Version 10.x Profiling and Analysis Tools Users Guide, Rev. 10.6, 02/2014

Freescale Semiconductor, Inc.
printf("======================================

");
print_array(arr, SIZE);

#ifndef NO_PRINT

for (i = 0; i < SIZE - 1; i++)
    for (j = i; j < SIZE; j++)
        if (arr[i] > arr[j])
            swap(&arr[i], &arr[j]);

#endif

void print_array(int arr[], int length) {

    int i;

    printf("Array = ");

    for (i = 0; i < length; i++) {
        printf("%d ", arr[i]);
    }

    printf("\n");
}

return (0);
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 analysis points.
   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 \( F_{\text{arr}}, F_i, F_j \) and click OK.
   c. Select \( F_i \) and select Add Visualization > Add Data Visualization.

       The Add Data Visualization dialog box appears.

   d. Accept the default settings and click OK.
   e. Select \( F_j \), 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.
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 \( F_j \) 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 analysis point to the **Analysis points** view.

5. Set the register analysis point.
   
a. Open the **Registers** view.

b. Expand **Core Registers > A1**, right-click, and select **Add Visualization > Add Data Visualization**.
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.
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 analysis point.
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, $F_j$ 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>C\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.

   ![Figure 9-1. Select New Interpreter Option in Jython Consoles View](image)

   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, for 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**

```c
#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) {
```

CodeWarrior Development Studio for Microcontrollers Version 10.x Profiling and Analysis Tools Users Guide, Rev. 10.6, 02/2014

Freescale Semiconductor, Inc.
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:

```python
>>> from scripting_sa_mcu import TestCollectContinuousHcs08
>>> TestCollectContinuousHcs08.runTraceDemo()
```

![Jython Consoles](image)

**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.
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:

```python
>>> 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:

```python
>>> 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

```plaintext
```

The following output is shown in the Jython console and four `.csv` files are created in the **Analysis Data** directory of your workspace.

![Figure 9-4. Running Commands in Jython Console](image)
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.

```python
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
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.

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 dialog box](image)

**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.
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.
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.

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`

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;
   ```

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:

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

**Listing: Sample source code used for profiling**

```c
#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
```

```
#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 Definition---------------*/

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++ )
    {
    }
  }
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");
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.
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.
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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>Name of the function or routine.</td>
</tr>
<tr>
<td>Count</td>
<td>Number of times the function was called.</td>
</tr>
<tr>
<td>Time</td>
<td>Time spent in the function itself without counting any time in functions called by this function.</td>
</tr>
<tr>
<td>%</td>
<td>Percentage of the total time spent in the function.</td>
</tr>
<tr>
<td>+Children</td>
<td>Time spent in the function and all the functions it calls.</td>
</tr>
<tr>
<td>%</td>
<td>Percentage of the total time spent in the function and all the functions it calls.</td>
</tr>
<tr>
<td>Average</td>
<td>Average time for each function invocation, that is <code>Time</code> divided by the number of times the function was called.</td>
</tr>
<tr>
<td>Maximum</td>
<td>Longest time for an invocation of the function.</td>
</tr>
<tr>
<td>Minimum</td>
<td>Shortest time for an invocation of the function.</td>
</tr>
<tr>
<td>Stack Space</td>
<td>Largest size (in bytes) of the stack when the function is called.</td>
</tr>
</tbody>
</table>

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.
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.

![Simple Profiler Data Viewer - Tree View](image)

### 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.
### Figure 10-12. Simple Profiler Data Viewer - Class View

#### Profiler Results

<table>
<thead>
<tr>
<th>Function</th>
<th>Count</th>
<th>Time</th>
<th>%</th>
<th>+Children</th>
<th>%</th>
<th>Average</th>
<th>Maximum</th>
<th>Minimum</th>
<th>Stack Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>__initw_console</td>
<td>16</td>
<td>0.013181</td>
<td>0.0</td>
<td>2.021916</td>
<td>0.3</td>
<td>0.000824</td>
<td>0.000823</td>
<td>0.000000</td>
<td>0.0</td>
</tr>
<tr>
<td>__access_file(unsigned long un</td>
<td>16</td>
<td>13.511939</td>
<td>2.4</td>
<td>2.020426</td>
<td>0.3</td>
<td>0.665986</td>
<td>0.940659</td>
<td>0.000000</td>
<td>0.0</td>
</tr>
<tr>
<td>fn1(int)</td>
<td>1</td>
<td>38.665993</td>
<td>6.7</td>
<td>176.570655</td>
<td>33.5</td>
<td>38.665993</td>
<td>38.665993</td>
<td>0.000000</td>
<td>0.0</td>
</tr>
<tr>
<td>fn2(int)</td>
<td>1</td>
<td>58.003152</td>
<td>10.0</td>
<td>136.661020</td>
<td>23.6</td>
<td>58.003152</td>
<td>58.003152</td>
<td>0.000000</td>
<td>0.0</td>
</tr>
<tr>
<td>fn3(int)</td>
<td>1</td>
<td>77.336056</td>
<td>13.4</td>
<td>77.656030</td>
<td>13.4</td>
<td>77.336056</td>
<td>77.336056</td>
<td>0.000000</td>
<td>0.0</td>
</tr>
<tr>
<td>fn4(int)</td>
<td>1</td>
<td>39.665945</td>
<td>6.7</td>
<td>36.936551</td>
<td>6.7</td>
<td>39.665945</td>
<td>39.665945</td>
<td>0.000000</td>
<td>0.0</td>
</tr>
<tr>
<td>fn5(int)</td>
<td>1</td>
<td>58.002262</td>
<td>10.0</td>
<td>58.851940</td>
<td>10.2</td>
<td>58.002262</td>
<td>58.002262</td>
<td>0.000000</td>
<td>0.0</td>
</tr>
<tr>
<td>fn6(int)</td>
<td>1</td>
<td>77.335985</td>
<td>13.4</td>
<td>76.237931</td>
<td>13.5</td>
<td>77.335985</td>
<td>77.335985</td>
<td>0.000000</td>
<td>0.0</td>
</tr>
<tr>
<td>fn7(int)</td>
<td>4</td>
<td>3.611343</td>
<td>0.7</td>
<td>0.179593</td>
<td>1.4</td>
<td>0.352036</td>
<td>3.004625</td>
<td>0.000000</td>
<td>0.0</td>
</tr>
<tr>
<td>One</td>
<td>1</td>
<td>38.665993</td>
<td>6.7</td>
<td>176.570655</td>
<td>33.5</td>
<td>38.665993</td>
<td>38.665993</td>
<td>0.000000</td>
<td>0.0</td>
</tr>
<tr>
<td>one_fn1(nt)</td>
<td>1</td>
<td>38.665993</td>
<td>6.7</td>
<td>176.570655</td>
<td>33.5</td>
<td>38.665993</td>
<td>38.665993</td>
<td>0.000000</td>
<td>0.0</td>
</tr>
<tr>
<td>Two</td>
<td>2</td>
<td>96.675245</td>
<td>16.7</td>
<td>176.570655</td>
<td>33.5</td>
<td>48.336272</td>
<td>58.003152</td>
<td>0.000000</td>
<td>0.0</td>
</tr>
<tr>
<td>two_fn1(nt)</td>
<td>1</td>
<td>38.665993</td>
<td>6.7</td>
<td>176.570655</td>
<td>33.5</td>
<td>38.665993</td>
<td>38.665993</td>
<td>0.000000</td>
<td>0.0</td>
</tr>
<tr>
<td>two_fn2(nt)</td>
<td>1</td>
<td>58.002152</td>
<td>10.0</td>
<td>136.661020</td>
<td>23.6</td>
<td>58.002152</td>
<td>58.002152</td>
<td>0.000000</td>
<td>0.0</td>
</tr>
</tbody>
</table>
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.
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.

**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

---

**CodeWarrior Development Studio for Microcontrollers Version 10.x Profiling and Analysis Tools Users Guide, Rev. 10.6, 02/2014**

Freescale Semiconductor, Inc.
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 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 function 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**

```c
void main(void)
{
    EnableInterrupts;
    DBGT = 0x80;   // write debug trigger register
    DBGC = 0xC0;   // write debug control register

    for(;;) {
```


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**

   ```c
   #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);
#endif

#elseif TRACE_MODE == TRACE_CONTINUOUS  /* set continuous trace mode */
    wdebug(MCFDEBUG_CSR, 0x200);
    wdebug(MCFDEBUG_XCSR, 0x0);
    wdebug(MCFDEBUG_CSR2, 0x89);
#endif

#elseif 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_PBRM, 0x0);
wdebug(MCFDEBUG_AATR, 0xE401);
wdebug(MCFDEBUG_DBMR, 0x0);
wdebug(MCFDEBUG_TDR, 0x40006002);

#endif

// without trigger points
wdebug(MCFDEBUG_CSR, 0x200);
wdebug(MCFDEBUG_XCSR, 0x1);
wdebug(MCFDEBUG_CSR2, 0xC1);

#endif

#endif

#if TRACE_MODE == TRACE_CONTINUOUS /* set continuous trace mode */
#if ENABLE_TRIGGERS == True  // with trigger points

```

CodeWarrior Development Studio for Microcontrollers Version 10.x Profiling and Analysis Tools Users
Guide, Rev. 10.6, 02/2014
wdebug(MCFDEBUG_PBR0, 0x5F2); //set PBR0 register;trigger A at 0x5F2
wdebug(MCFDEBUG_PBR1, 0x6C8); //set PBR1 register;trigger B at 0x6C8
wdebug(MCFDEBUG_PBM, 0x0);
/without trigger points
wdebug(MCFDEBUG_AATR, 0xE401);
wdebug(MCFDEBUG_DB, 0x0);
wdebug(MCFDEBUG_DBM, 0x0);
wdebug(MCFDEBUG_TDR, 0x40006002);
#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.

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.

3. Click the Timeline hyperlink.

   The Timeline viewer appears displaying the low power WAIT results.
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

A
Add/Remove Function 98
Add/Remove Group 98
Add New Analysispoint 212
Advanced Settings on Kinetis 55
Almost Full Level (no. of words) 66
Analysispoints View 209
ASM Count 102
ASM Decision Coverage 104
ASM Decision Coverage% 102

B
Break on FIFO Full 35, 118
Break on Trigger Hit 35, 119
Breakpoint in Trace 259

C
Call Tree 15
Call Tree Viewer 110
Capture Read/Write Values at Address B 37, 148
Capture Read/Write Values at Address B, After Access at Address A 37, 152
Change Function Color 98
Choose columns 93
Class View 257
CodeWarrior Projects 13
ColdFire V1 Target 21
Collecting Data 76
Collecting Trace Data Using JTrace Connection 54
Collect Instrumentation trace 46
Collect Profiling Counters 46
Collect Profiling Information 253
Collect Trace From Trigger 35, 118
Collect Trace Until Trigger 35, 118
Comparator Settings
  Value 61
Configuration Set in User Code 33, 40, 263
Configure Advanced Settings on Kinetis 55
Configure Kinetis Cortex M0+ Core 47
Configure Project for Profiling 240
Configure table 105
Configure Time Format 91
Configure Time Unit 91
Configuring Launcher
  Configure ColdFire V1 Target 38
  Configure ColdFire V2-V4 Targets 71
  Configure DSC Target 73
  Configure HCS08 Target 31
  Configure Kinetis Target 42
  Configure MPC56xx Target 68

Configuring Launcher (index-continued-string)
  Configure S12Z Target 68
Console 13
Continuous Trace Collection 46
Copy Cell 79
Copy Line 79
Cortex M0+ 49
Cortex M4 49
Coverage% 102
Create column group 92
Creating new project
  Using ColdFire V1 Target 21
  Using ColdFire V4e Target 26
  Using HCS08 Target 18
  Using Kinetis Target 23
  Using MPC56xx Target 29
Critical Code Data 14
Critical Code Tab 103
Customizing Trace Data Viewer 92
Cycle Counter 63
Cycle Counter Tap 64

D
Data Analysispoints 219
Data Trace Import 79
Data Visualization 81, 217
Decoding Trace 115
Delete Results 79

E
Edit Address Range of Function 98
Embedded Trace Buffer (ETB) 54
Embedded Trace Macrocell (ETM) 46, 50
Enable Logging 33, 40
Enable Stimulus Registers 63
Enable Timestamps 58
Enabling and Disabling Tracepoints 158, 181, 213
ETM - Counter Reload Value 61
Event Generation 64
Export 104
Export Analysis Points 215
Export Trace Data 90
Extension Counter Reload Value (hex) 63

F
FIFOFULL Level (no. of bytes) 58
File Coverage Tab 105
Find and Filter 87
Flat View 256
Flush Control Settings 67
Freescale Toolchain 199
From trigger 118
From trigger in Automatically mode 136

GCC Toolchain 199, 203
Generate Comparator Match Event On 61
Global Timestamps 63
Go To File for Analysispoints 214
Graphics 105
Group By 210

H
Halt the Target when Trace Buffer Gets Full 41, 162
Hardware Tracepoints 183
HCS08 Target 18
Hide column 92

I
Ignore all Analysispoints 214
Import Analysis Points 216
Importing Trace Data Offline 111
Import wizard 111
Include Profiler Library and Files 238
Instruction at Address A, Then Instruction at Address B are Executed 36, 121
Instruction at Address A is Executed 36, 121
Instruction at Address A is Executed and Value on Data Bus Match 37, 131
Instruction at Address A is Executed and Value on Data Bus Mismatch 37, 133
Instruction at Address A or Address B is Executed 36, 126
Instruction Execute 37, 119
Instruction Inside Range from Address A to Address B is Executed 36, 127
Instruction Outside Range from Address A to Address B is Executed 36, 145
Instrumentation Trace Macrocell (ITM) 46, 52

J
JTrace 44
J-Trace 54
Jython 229

K
K10/K20 50 Mhz and 72 MHz derivatives 46
Keep all trace buffers 44, 72
Keep Last Buffer Before Trigger 35, 118
Kinetis Cortex M0+ Core 188, 196
Kinetis Cortex M4 Core 184, 193

L
Launching Scripts 229
Local Timestamps 63
Log 78
LOOP1 Mode 142
Low Power Profiling 45
Low Power WAIT Mode 269

M
Match Data Size 61
Memory Access 38, 119
Memory Access Triggers 134
Memory at Address A is Accessed 134
Merge Groups/Functions 100
Micro Trace Buffer (MTB) 50
MPC5668G/E 30

N
Next function 104
No. of bits to ignore 61
No Trigger 35

P
Performance Data 15
Performance Viewer 107
Previous function 104
Profile-Only 34
Profiling an application 237
Profiling and Analysis tools 11
Profiling Data 15
Profiling System 237

R
Register Analysispoints 219
Remote Launch 229
Remote Launching
Collect Trace Using Jython 231
Export Trace to CSV File 233
Modify Sample Python Script 235
Run Sample Python Script 229
Remove All Analysispoints 214
Remove Selected Analysispoint 214
Rename column 94
Reset Button 83
Resume/Suspend Toggle Button 82

S
S12Z Target 68
Save Results 79
Search 104
Select Default Working Set dialog box 212
Select Trace Mode - ColdFire V1 41
Set CPU Frequency 91
Setting Data Analysis points on Memory View 220
Setting Data Analysis points on Variables View 221
Setting Hardware Tracepoints
  From Source Code
    Kinetis Cortex M0+ Core 188
    Kinetis Cortex M4 Core 184
  From Trace and Profile Tab
    Kinetis Cortex M0+ Core 196
    Kinetis Cortex M4 Core 193
Setting Register Analysis points on Registers View 221
Setting Software Tracepoints 198
Setting Trigger from Disassembly View 136
Setting Trigger on Data and Memory 139
Setting Triggers from Memory View 179
Setting Triggers from Variables View 177
Setting Triggers on Data and Memory 177
Show code 105
Show Full Paths 210
Simple Profiler Data Viewer 13, 254, 255
Software Analysis 13
Software Analysis view 77
Software Tracepoints 183
Software Tracepoints Support dialog box 202, 207
Stall Processor 58
SWD 45
SWO 45
Synchronization 53
Synchronization Packet Rate 64

T

Target 87
Target PC Address
  2 Bytes 41
  3 Bytes 41
Time (CPU Cycles) 102
Timeline 15, 94
Timeline viewer
  Edit Groups 97
  Full View 97
  Selection Mode 96
  Zoom Mode 96
Timeline Viewer
  Configure Table 101
Timestamp 87
Timestamp Event
  A 60
  B 61
  Function 60
  Index 61
TPIU 45, 54
Trace Start/Stop Conditions - HCS08  35
Trace Start/Stop Control
  Start Resource  58
  Stop Resource  59
Tree View  256
Trigger Control
  A  59
  B  59
  Collect Trace about Trigger  59
  Collect Trace after Trigger  59
  Collect Trace before Trigger  59
  Function  59
  Index  59
Trigger Counter (no. of words)  67
Trigger Settings  67
Trigger Type  35

U

Ungroup columns  93
Until trigger  119
Until trigger in Automatically mode  136

V

Value to Compare on Data Bus  37
Viewing Data
  Call Tree  109
  Critical Code Data  101
  Performance Data  106
  Timeline  94
  Trace Data  85
Viewing Tracepoints  209
View Profiling Results  254

W

Working Sets  211
Information in this document is provided solely to enable system and software implementers to use Freescale products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. Freescale reserves the right to make changes without further notice to any products herein.

Freescale makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in Freescale data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer’s technical experts. Freescale does not convey any license under its patent rights nor the rights of others. Freescale sells products pursuant to standard terms and conditions of sale, which can be found at the following address: freescale.com/SalesTermsandConditions.

Freescale, the Freescale logo, CodeWarrior, ColdFire, ColdFire+, Kinetis, and Qorivva are trademarks of Freescale Semiconductor, Inc., Reg. U.S. Pat. & Tm. Off. All other product or service names are the property of their respective owners. ARM and ARM Cortex-M0+ are registered trademarks of ARM Limited.