CodeWarrior for ARMv7 Tracing and Analysis User Guide
## Contents

### Chapter 1 Introduction
- 1.1 Overview .......................................................... 5
- 1.2 Accompanying documentation ........................................ 6

### Chapter 2 Tracing
- 2.1 Configuring and collecting trace .................................................. 7
  - 2.1.1 Creating a new project ....................................................... 7
  - 2.1.2 Configuring trace ............................................................ 8
  - 2.1.3 Collecting trace data .......................................................... 13
  - 2.1.4 Collecting trace data using an Attach configuration .............. 14
- 2.2 Viewing trace data using Analysis Results view ......................... 16
  - 2.2.1 Trace viewer ................................................................. 19
  - 2.2.2 Timeline viewer ............................................................ 20
    - 2.2.2.1 Selection Mode ....................................................... 21
    - 2.2.2.2 Zoom Mode ........................................................... 21
    - 2.2.2.3 Full View ............................................................... 21
    - 2.2.2.4 Edit Groups ............................................................ 21
  - 2.2.3 Code Coverage viewer .................................................... 22
    - 2.2.3.1 Summary table ....................................................... 23
    - 2.2.3.2 Details table ........................................................... 25
  - 2.2.4 Performance viewer ....................................................... 27
  - 2.2.5 Call Tree viewer .......................................................... 29

### Chapter 3 Collecting and Viewing Linux Trace
- 3.1 Collecting Linux trace using CodeWarrior ............................... 31
- 3.2 Viewing Linux trace collected without using CodeWarrior ............ 42

### Chapter 4 Linux Kernel Debug Print Tool
- 4.1 Filtering debug print messages ................................................ 51
Chapter 1
Introduction

The CodeWarrior for ARMv7® V10.x Tracing and Performance Analysis tool allows you to analyze and collect data of an application.

You can use this collected data to identify the bottlenecks, such as slow execution of routines or heavily-used routines within the application. This visibility can help you understand how your application runs, as well as identify operational problems.

The UI Platform Configurator tool reads the user settings from the input XML file and transforms them into target access memory writes to the configuration registers.

The figure below shows the core trace path for LS1021A within the ARMv7 architecture.

Figure 1: Core trace path for LS1021A

The UI Platform Configurator configures the ARMv7 cores Embedded Trace Macrocell (ETM), Funnel, Embedded Trace FIFO (ETF), and Embedded Trace Router (ETR) modules. The ETF stores and forwards trace data using a dedicated RAM buffer. This reduces trace loss by absorbing spikes in trace data. The ETR redirects trace to the system bus for collection from alternative channels. For ARMv7, it is DDR memory.

The destination of the raw trace is either the internal trace buffer of the ETF module or a user defined buffer in DDR.

This manual describes how the CodeWarrior for ARMv7 V10.x Tracing and Performance Analysis tool can be used for the LS1020A, LS1021A, LS1022A, and LS1024A devices. This chapter presents an overview of this manual and introduces you to the Tracing and Performance Analysis tools.

This chapter contains the following sections:

• Overview on page 5

• Accompanying documentation on page 6

1.1 Overview

Each chapter of this manual describes a different area of Tracing and Performance Analysis tools.

The table below describes each chapter in the manual.
Table 1: Manual Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>Introduces Tracing and Performance Analysis tool (this chapter)</td>
</tr>
<tr>
<td><strong>Tracing</strong> on page 7</td>
<td>Describes the trace collection process and how to use the trace data for identifying the bottlenecks</td>
</tr>
<tr>
<td><strong>Collecting and Viewing Linux Trace</strong> on page 31</td>
<td>Describes how you to collect and view satrace with and without using CodeWarrior</td>
</tr>
<tr>
<td><strong>Linux Kernel Debug Print Tool</strong> on page 45</td>
<td>Describes how Debug Print Tool works. The tool is independent of CodeWarrior and does not require a debug session</td>
</tr>
</tbody>
</table>

1.2 Accompanying documentation

The Documentation page describes the documentation included in this version of CodeWarrior Development Studio for QorIQ LS series - ARM V7 ISA.

You can access the Documentation page by:

- Opening START_HERE.html from the `<CWInstallDir>\CW_ARMv7\ARMv7\Help` folder
- Choosing Help > Documentation from the CodeWarrior IDE menu bar
Chapter 2
Tracing

Tracing is a technique that obtains diagnostic and performance information about a program's execution. This information is typically used for debugging purposes, and additionally, depending on the type and detail information contained in a trace log, to diagnose common problems with the software. The trace log includes information about the trace source, type of event, description of the event and time stamp value.

This chapter contains the following sections:

- Configuring and collecting trace on page 7
- Viewing trace data using Analysis Results view on page 16

2.1 Configuring and collecting trace

To collect trace, you need a project to define your application, a launch configuration to set up a debug connection to the target, and a trace configuration to define the type of trace data you would like to collect.

This section contains the following subsections:

- Creating a new project on page 7
- Configuring trace on page 8
- Collecting trace data on page 13
- Collecting trace data using an Attach configuration on page 14

2.1.1 Creating a new project

You need to create a new CodeWarrior bareboard project or open an existing project before you start collecting the trace data.

To create a new bareboard project:

1. Start the CodeWarrior IDE.
2. From the IDE menu bar, choose File > New > CodeWarrior Bareboard Project Wizard.
   
   The Create a CodeWarrior Bareboard Project page appears.
3. Enter a name for the new project in the Project Name text box.
4. The Location text box shows the default workspace location. To change this location, clear the Use default location checkbox and click Browse to choose a new location.
5. Use the subsequent dialog to specify a new location. Click OK.
   
   The dialog returns you to the Create a CodeWarrior Bareboard Project page, which now shows the new location.
6. Click Next.
   
   The Processor page appears.
7. Select the required processor and click **Next**.

8. The **Debug Target Settings** page appears. Choose a board from the **Board** menu and click **Next**.

9. Do not change the default settings of the rest of the pages and click **Next** and **Finish**.

10. From the IDE menu bar, choose **Project** > **Build Project**.

    The **Build Project** dialog appears; the build tools generate an executable program.

    **NOTE**
    For more information on creating a CodeWarrior project, see *CodeWarrior for ARMv7 Targeting Manual (CW_ARMv7_Targeting_Manual.pdf)* in the `<CWInstallDir>` `\CW_ARMv7\ARMv7\Help\PDF` folder, where `<CWInstallDir>` is the installation directory for the CW4NET installer.

### 2.1.2 Configuring trace

You need to define the trace configuration before debugging the application for trace collection.

To define a trace configuration:

1. Choose **Run** > **Debug Configurations** from the CodeWarrior IDE menu bar.
The Debug Configurations dialog appears.

2. In the left pane of this dialog, expand the CodeWarrior tree control.

3. Select the launch configuration corresponding to your project. For example, Project1-core0_RAM_LS1021AQDS_Download.

A set of tabbed configuration panels appears in the right pane of the dialog.

![Debug Configurations dialog](image)

4. Click the Trace and Profile tab.

The page appears with two tabs:

- The Overview tab displays the information about the ARMv7 architecture.

- The Basic tab displays the predefined settings. For Platform Configuration Settings, you can apply different values based on what you want to achieve. The values are called profiles, and the framework allows creation of different profiles for a configuration block. The profiles dialog is used to create new, rename, delete, or edit the settings of a profile for a configuration provider.
5. The **Trace and Profile** checkbox is selected for the trace session to start immediately on debug launch. If **User Code** checkbox is selected, it lets you upload trace into CodeWarrior when trace session is started from user code or tools other than CodeWarrior. That is, the application where you make your own configuration of trace registers in the trace hardware explicitly and CodeWarrior is not involved. If **Upload trace on terminate or disconnect** checkbox is selected, the trace data is automatically collected when debug session is terminated. In the combo box, you can see a default platform configuration file having same name as you have for launch configuration.

6. You can create, rename, edit, delete, or export the platform configuration file. The **Export** button allows you to export the platform configuration file to other platform for later use.

**NOTE**
You can create your own configuration file by using a template placed at:

```
<CWInstallDir>CW_ARmv7\ARMv7\sa_is\data\fsl.configs.sa.ls.configurators/```

7. If you want to create or manage new profiles, you can invoke the **Trace Configurations** dialog by clicking on the **Edit** button. You can see a tree viewer on the left side that lists all the profiles. The content in right pane is driven by the selection in the left pane.

- If you select the required core under the **Trace Generators** on the left pane, the entries corresponding to that configuration are displayed on the right pane.
The following settings can be configured for each core or trace generator:

- **Trace Scenarios**: Choose a trace scenario type from the **Trace Scenarios** panel.

- **General Settings**: Select **Timestamp** checkbox in the **General Settings** panel to enable timestamp. Timestamping is useful for correlating multiple trace sources. Timestamping is performed by the insertion of timestamp packets into the trace streams. It displays the value of platform global timestamp generator (64-bit wide).

- If you select **Data Streams** to collect trace data, the entries corresponding to the selected buffer are displayed on the right pane.
The following settings can be configured for each data stream:

- **Trace collection location**: Choose ETF or DDR as the trace collection location.

- **Trace collection mode**: Choose One buffer or Overwrite as the trace collection mode. If the One buffer option is chosen, then trace can only be collected until the buffer is filled. In this case, only the first buffer size part of the trace is kept. If the Overwrite option is chosen and the buffer is filled, then the pointer returns to the beginning of the buffer and overwrites the older trace. In this case, only the last buffer size part of the trace is kept.

In case the trace is collected in the DDR buffer, then you also need to specify the following options to set the address range for the DDR trace buffer:

- **Start address**: Indicates the start address of the trace buffer in the DDR memory.

- **Size**: Indicates the size of the DDR trace buffer in bytes.

**NOTE**
Each time you collect new trace data for a project, the existing trace data will be overwritten.
2.1.3 Collecting trace data

After creating a project and defining launch configuration and trace configuration, you can start a debug session and start collecting trace data.

You can perform trace collection tasks using the buttons available in the Debug view.

To collect trace data:

1. In the Debug Configurations dialog, click Debug.

   The IDE switches to the Debug perspective; the debugger downloads your program to the target development board and halts execution at the first statement of `main()`.

   In the editor view (center-left of perspective), the program counter icon on the marker bar points to the next statement to be executed.

   ![Debug perspective](image)

   **Figure 7:** Debug perspective

2. In the Debug view, click Resume.

   The data collection starts and the Resume button gets disabled. The debugger executes all statements, the program writes the strings in the Console view and then enters an infinite loop.

3. Let the application execute for several seconds and click Suspend if you want to stop the collection temporarily. Again, click Resume to start the trace collection and when it finishes, click Terminate.
2.1.4 Collecting trace data using an Attach configuration

This section explains how to collect trace using an Attach launch configuration.

The Attach launch configuration is useful when you want to connect to a running target without resetting the target or downloading a different application on it. For more information on Attach launch configuration, see *CodeWarrior for ARMv7 Targeting Manual (CW_ARMv7_Targeting_Manual.pdf)* in the `<CWInstallDir>\CW_ARMv7\ARMv7\Help\PDF` folder.

To configure your application for trace collection using the Attach launch configuration:

1. Create a project with the Attach launch configuration using the CodeWarrior Bareboard Project Wizard.
   
   a. From the IDE menu bar, choose File > New > CodeWarrior Bareboard Project Wizard.
   
   b. Enter a name for the new project in the Project Name text box.
   
   c. Click Next and select the required processor in the Processors page.
   
   d. Click Next and select the Attach checkbox in the Debug Target Settings page. Choose the Default option to create an Attach configuration based on default parameters.
e. Select the required **Connection Type** and click **Next**.
f. Follow the remaining steps of the wizard and click **Finish**.

2. Build the project.

3. Choose **Run > Debug Configurations** to open the **Debug Configurations** dialog.

4. Expand **CodeWarrior** in the left pane of the dialog and select the Attach launch configuration corresponding to your project.
The Attach configuration assumes that code is already running on the board and therefore does not run a target initialization file. Therefore, when attach launch configurations are used, ensure that the target is already running.

5. In the **Main** tab of the dialog, make sure that **Attach** is selected in the **Debug Session Type** group.
6. Open the **Trace and Profile** tab.
7. Select the **Trace and Profile** checkbox on the **Basic** tab to enable trace and profile options. The **Upload trace on terminate or disconnect** checkbox is selected, by default.
8. Click **Debug**.
9. Click **Resume** to start collecting the trace data.
10. Click **Suspend** after some time to stop trace collection.
11. Click the **Upload Trace** button to get the collected trace.
12. Open the **Trace** viewer to view the collected data.

### 2.2 Viewing trace data using Analysis Results view

The **Analysis Results** view provides access to various trace data results collected on an application.

The **Analysis Results** view shows and manages collected trace and profiling data. It provides link to open any supported viewer: **Trace**, **Timeline**, **Code Coverage**, **Performance**, and **Call Tree**. This view provides functions like save results, rename, and delete to help you to organize collected results.
To open the Analysis Results view:

1. Choose **Window > Show View > Other** from the CodeWarrior IDE menu bar. The **Show View** dialog appears.

2. Select **Software Analysis > Analysis Results** and click **OK**. The **Analysis Results** view appears.

**NOTE**
The Analysis Results view pops up and gets focus after successful trace collection. It does not refresh automatically.

*Figure 10: Analysis Results view*

The Analysis Results view toolbar provides the following options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Refresh]</td>
<td>Refreshes the displayed data.</td>
</tr>
<tr>
<td>![Expand All]</td>
<td>Expands all the nodes.</td>
</tr>
<tr>
<td>![Collapse All]</td>
<td>Collapses all the nodes.</td>
</tr>
<tr>
<td>![Select custom results folder]</td>
<td>Select custom results folder.</td>
</tr>
</tbody>
</table>

The Analysis Results view shortcut menu, which appears on right-click, provides the following options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Refresh</td>
<td>Refreshes the displayed data.</td>
</tr>
<tr>
<td>Expand All</td>
<td>Expands all the nodes.</td>
</tr>
<tr>
<td>Collapse All</td>
<td>Collapses all the nodes.</td>
</tr>
<tr>
<td>Results</td>
<td>Lets you select a custom trace results folder.</td>
</tr>
<tr>
<td>Copy Cell</td>
<td>Copies the currently selected cell. The name of the data source is copied to the clipboard.</td>
</tr>
</tbody>
</table>

*Table continues on the next page...*
### Table 3: Analysis Results view - shortcut menu commands (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy Line</td>
<td>Copies the complete line of the data source.</td>
</tr>
<tr>
<td>Save Results</td>
<td>Saves the trace results.</td>
</tr>
<tr>
<td>Delete Results</td>
<td>Deletes all the saved results.</td>
</tr>
</tbody>
</table>

The trace file can automatically be saved after trace collection completes or is stopped. To enable automatic saving of trace file, follow these steps:

1. Choose **Window > Preferences**. The **Preferences** dialog appears.
2. Click **Software Analysis** in the left pane of the **Preferences** dialog, and select the **Automatically save trace results** checkbox in the right pane, as shown in the figure below.

![Figure 11: Preferences dialog](image)

3. Click **OK** to apply the setting and close the **Preferences** dialog.

Now, when you will collect trace, the trace file will be saved in the `.AnalysisData` folder of the current workspace.

The **Analysis Results** view provides hyperlinks to open:

- **Trace viewer** on page 19
- **Timeline viewer** on page 20
- **Code Coverage viewer** on page 22
- **Performance viewer** on page 27
- **Call Tree viewer** on page 29
2.2.1 Trace viewer

The Trace viewer displays the decoded trace event data generated during the data collection phase.

To view trace data:

1. In the Analysis Results view, expand the project name.
   
   All trace collections performed for the project are displayed.

2. Click the Trace hyperlink, and view the trace data in the Trace viewer.

   Figure 12: Trace viewer

The data in Trace viewer is organized in a way to ease the evaluation of tracing information and navigation through the events in the sequence they were logged. This data can be very complex, and the size of the decoded data can be very large, up to approximately 40 GB. The Trace viewer is constrained by the size of the decoded data. Currently, the SAE can iterate over a maximum of $2^{32}$ items of raw Nexus trace. Each item of undecoded trace can be associated with zero, one, or multiple decoded trace events.

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

The following table describes the Trace viewer fields.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>Each decoded event has a unique number starting with 1.</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: Trace viewer fields (continued)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Specifies the type of event that has occurred, such as Info, Branch, Error,</td>
</tr>
<tr>
<td></td>
<td>Function Return and Function Call.</td>
</tr>
<tr>
<td>Description</td>
<td>Displays detailed information about the trace line.</td>
</tr>
<tr>
<td>Address</td>
<td>Displays the starting address of the target function.</td>
</tr>
<tr>
<td>Destination</td>
<td>Displays the end address of the target function.</td>
</tr>
<tr>
<td>Timestamp</td>
<td>Specifies the timestamp value that is expressed as clock ticks. Depending</td>
</tr>
<tr>
<td></td>
<td>on the configuration of the trace source, this can be relative to the event's</td>
</tr>
<tr>
<td></td>
<td>trace source or relative to all events logged from all sources.</td>
</tr>
</tbody>
</table>

2.2.2 Timeline viewer

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 Analysis Results view, expand the project name.
   The data source is listed under the project name.

2. Click the Timeline hyperlink.
   The Timeline viewer appears.

   Figure 13: Timeline viewer

   ![Timeline viewer](image-url)

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 Timeline viewer also displays the following buttons:

- **Selection Mode** on page 21
- **Zoom Mode** on page 21
- **Full View** on page 21
- **Edit Groups** on page 21
2.2.2.1 Selection Mode
The **Selection Mode** allows you to mark points in the function bars in the timeline graph 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 14: Selection mode to measure difference of cycles between functions](image)

You might view a difference in the time cycles displayed in the **Timeline** viewer and the **Code Coverage** 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.

2.2.2.2 Zoom Mode
The **Zoom Mode** allows you to zoom-in and zoom-out in the timeline graph.

Click **Zoom Mode** and then click 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.

2.2.2.3 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.

2.2.2.4 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, click **Edit Groups**. The **Edit Groups** dialog appears.
2.2.3 Code Coverage viewer

The Code Coverage viewer allows you to analyze the flat profile of your application either at a functional level or at a file level.

To view code coverage data:

1. In the Analysis Results view, expand the project name.
   - All trace collections performed for the project appear.
2. Click the Code Coverage hyperlink.
   - The Code Coverage viewer appears.
The Code Coverage viewer divides the critical code data into two tabular views: Summary Table and Details Table.

This section contains the following subsections:

- Summary table on page 23
- Details table on page 25

### 2.2.3.1 Summary table

The Summary table of the Code Coverage viewer displays the summary of the functions executed in the application.

The Summary table provides tree and flat view structures to display code coverage data. The tree view is the default structure in which the functions are grouped by source file. You can expand or collapse in the column to view the functions contained in the corresponding source file. In a flat view structure, all functions are displayed individually. You can switch between tree view or flat view by right-clicking on a column name of the Summary table and selecting the Switch to tree/flat view option.

The Summary table contains the fields as described in the table below. You can switch to ASM instructions statistics or source lines statistics alternatively using the button available on the toolbar on the right side of the Summary table. The columns are movable; you can drag and drop the columns to move them according to your requirements. The table below shows the metrics for ASM level coverage with assembly instructions coverage percentage and total number of assembly instructions per function/module. Clicking the button to switch to source lines statistics shows metrics for source line coverage with number of source lines covered, not covered, partially covered, and total number of source lines per function/module.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>File/Function</td>
<td>Displays the name of the function that has executed.</td>
</tr>
<tr>
<td>Address</td>
<td>Displays the start address of the function.</td>
</tr>
<tr>
<td>Covered ASM %</td>
<td>Displays the percentage of number of assembly instructions executed from the total number of assembly instructions per function or per source file.</td>
</tr>
<tr>
<td>Not Covered ASM %</td>
<td>Displays the percentage of number of assembly instructions not executed from the total number of assembly instructions per function or per source file.</td>
</tr>
<tr>
<td>Total ASM instructions</td>
<td>Displays the total number of assembly instructions per function and per source file.</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. Therefore, if a function has two conditional instructions, one with 100% and another with 50% decision coverage, the decision coverage would be $\frac{100 + 50}{2} = 75%$. It is calculated only for assembly instructions and not for C source code.</td>
</tr>
<tr>
<td>Time</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>
In the Code Coverage viewer, all functions in all files associated with the project are displayed irrespective of coverage percentage. However, the 0% coverage functions do not appear in the Performance and Call Tree viewers because these functions are not considered to be computed and are not a part of caller-called pair.

### Table 6: Buttons available in Summary table of Code Coverage viewer

<table>
<thead>
<tr>
<th>Name</th>
<th>Button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Previous function</td>
<td><img src="image" alt="" /></td>
<td>Lets you view the details of the previous function that was selected in the Summary table before the currently selected function. Click it to view the details of the previous function.</td>
</tr>
<tr>
<td>Next function</td>
<td><img src="image" alt="" /></td>
<td>Lets you view the details of the next function that was selected in the Summary table. Note: The Previous and Next buttons are contextual and go to previous/next function according to the history of selections. So if you select a single line in the view, these buttons will be disabled because there is no history.</td>
</tr>
<tr>
<td>Export</td>
<td><img src="image" alt="" /></td>
<td>Lets you export the Code Coverage data in a CSV or html format. Click the button to choose between Export to CSV or Export to HTML options. The Export to CSV option lets you export data of both Summary and Details tables. The exported html file contains the statistics for all the source files/functions from the Summary table along with the statistics of source, assembly or mixed instructions.</td>
</tr>
</tbody>
</table>

Table continues on the next page...
Table 6: Buttons available in Summary table of Code Coverage viewer (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configure table</td>
<td></td>
<td>Lets you show and hide column(s) of the Code Coverage data. Click the button and select the appropriate option to show/hide columns of the Summary/Details table. The Drag and drop to order columns dialog appears in which you can select/deselect the checkboxes corresponding to the available columns to show/hide them in the Code Coverage viewer. The option also allows you to set CPU frequency and set time in cycles, milliseconds, microseconds, and nanoseconds.</td>
</tr>
<tr>
<td>Collapse/Expand all files</td>
<td></td>
<td>Lets you expand or collapse all files in the Summary table.</td>
</tr>
<tr>
<td>Filter files</td>
<td></td>
<td>Allows you to choose the list of files to be displayed in the Summary table.</td>
</tr>
<tr>
<td>Switch to executable source lines statistics/Switch to ASM instructions statistics</td>
<td></td>
<td>Lets you switch between source lines or ASM instructions to be displayed in the Summary table.</td>
</tr>
</tbody>
</table>

2.2.3.2 Details table

The Details table of the Code Coverage viewer displays the statistics for all the instructions (source and disassembly) executed in a particular source file.

Click a hyperlinked file/function in the Summary table of the Code Coverage viewer to view the corresponding statistics for the instructions executed in that file/function. For example, the statistics of the `strlen()` function are shown in the figure below.

![Figure 18: Details table](image)

The table below describes the fields of the Details table.
Table 7: Details table - description of fields

<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 source files, displays if the instructions were covered, not covered, or partially covered.</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>

When you double-click the instruction in the Details Table, the corresponding source file is opened. To disable the path mapping option when you double-click the instruction, follow these steps:

1. Choose Window > Preferences. The Preferences dialog appears.
2. Click Software Analysis in the left pane of the Preferences dialog, and select the Do not locate file for path mapping checkbox in the right pane, as shown in the figure below.

3. Click OK to apply the setting and close the Preferences dialog.

You can perform the following actions on the Details table.
• **Search** - Lets you search for a particular text in the Details table. 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.

• **Graphics** - Lets you display the histograms in two colors for the ASM Count and Time columns in the bottom view of the Code Coverage data. Click the button and select the Assembly/Source > ASM Count or Assembly/Source > Time option to display histograms in the ASM Count or Time column. The colors in these columns differentiate source code with the assembly code.

• **Show code** - Lets you display the assembly, source or mixed code in the statistics of the Code Coverage data.

### 2.2.4 Performance viewer

The **Performance** viewer displays the metric and invocation information for each function that executes in the application.

To view performance data:

1. In the **Analysis Results** view, expand the project name.
   - All trace collections performed for the project are displayed.
2. Click the **Performance** hyperlink.
   - The **Performance** viewer appears.

![Performance viewer](image)

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

• The top view presents function performance data in the **Summary** 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 **Summary** 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 **Summary** table.

• The bottom view or the **Details** table presents call pair data for the function selected in the **Summary** table. The **Details** table 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 table below describes the fields of the **Details** table. You cannot sort the columns of this table.
Table 8: Field description of Summary 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 9: Field description of Details 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 Callee</td>
<td>Number of times the caller called the callee.</td>
</tr>
<tr>
<td>Inclusive Callee</td>
<td>Cumulative metric count during execution time spent from function entry to exit.</td>
</tr>
<tr>
<td>Min Inclusive Callee</td>
<td>Minimum metric count during execution time spent from function entry to exit.</td>
</tr>
<tr>
<td>Max Inclusive Callee</td>
<td>Maximum metric count during execution time spent from function entry to exit.</td>
</tr>
<tr>
<td>Avg Inclusive Callee</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 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. For details on these icons, see Table 6. Buttons available in Summary table of Code Coverage viewer on page 24.
2.2.5 Call Tree viewer

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

To view call tree data:

1. In the Analysis Results view, expand the project name.
   All trace collections performed for the project are displayed.
2. Click the Call Tree hyperlink.
   The Call Tree viewer appears.

   Figure 21: Call Tree viewer

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

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

The table below describe the fields of Call Tree data. The columns are movable; you can move the columns to the left or right of another column 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</td>
</tr>
<tr>
<td></td>
<td>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 (Microseconds; 50.0 MHz)</td>
<td>Cumulative count during execution time spent from function entry to exit.</td>
</tr>
</tbody>
</table>
You can perform the following actions using the buttons available on the toolbar of the Call Tree viewer:

- **Export to dot** - Exports call tree data in the .dot format using the button.
Chapter 3
Collecting and Viewing Linux Trace

The Linux satrace tool allows you to collect the trace of a program without using any hardware probe. The tool encapsulates the trace configurator and probe. It is delivered as a standalone component containing an executable and some shared libraries. The Linux satrace tool is independent of CodeWarrior; it does not provide any GUI. This document proposes an integration solution for this command-line utility into CodeWarrior and also how to collect trace without using CodeWarrior.

This chapter contains the following sections:

- Collecting Linux trace using CodeWarrior on page 31
- Viewing Linux trace collected without using CodeWarrior on page 42

3.1 Collecting Linux trace using CodeWarrior

This section explains how to collect Linux trace by establishing a Remote System Explorer (RSE) connection with the board.

To collect trace using the Linux satrace tool, perform these steps:

1. Start CW for ARMv7 and create a CodeWarrior Linux Project.
2. Write the application using the Editor.
3. Build the project to generate an ELF file, as shown in the figure below.

![Figure 22: ELF file](image)

4. Create a Linux connection using RSE:
   a. To open RSE, click **Window > Open Perspective > Other > Remote System Explorer**.
   b. On the toolbar of the **Remote Systems** view, click **Define a connection to remote system**. The RSE wizard starts.
   c. Expand **General** and select **Linux** option from the list.
d. Click **Next**.

![RSE wizard screenshot](image)

**Figure 23:** RSE wizard

e. The **Remote Linux System Connection** page appears. Specify the host name and connection name and click **Next**.
f. The **Files** page appears. Select the **ssh.files** checkbox and click **Next**.
Figure 25: Files page

The Processes page appears. Select the `processes.shell.linux` checkbox and click Next.
h. The **Shells** page appears. Select the **ssh.shells** checkbox and click **Finish**.
In the Remote Systems view, you can see that the connection with the board has been established. The connection name is **linux-connection**.

Figure 27: Shells page
i. Select **Sftp Files** in the **Remote Systems** view and specify the port number in the **Properties** window, as shown in the figure below.

![Remote Systems view](image)

**Figure 29: Setting port number**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connected</td>
<td>Yes</td>
</tr>
<tr>
<td>Name</td>
<td>Sftp Files</td>
</tr>
<tr>
<td>Number of children</td>
<td>2</td>
</tr>
<tr>
<td>Port</td>
<td>9900</td>
</tr>
<tr>
<td>Type</td>
<td>Subsystem</td>
</tr>
<tr>
<td>User ID</td>
<td>root (Inherited)</td>
</tr>
<tr>
<td>Version</td>
<td></td>
</tr>
</tbody>
</table>

j. Connect to the Linux system by right-clicking **Sftp Files** and choosing **Connect**.

k. Create a folder under **Sftp Files > My Home** where you can store the files related to your current project.

l. Now, copy your ELF file from the **CodeWarrior Projects** view and paste it into the folder you just created, as shown in the figure below.
5. Right-click the parent folder where you have pasted the ELF file, choose **Add Trace Support** from the context menu to copy the satrace into a directory.

6. Right-click the parent folder and choose **Launch Terminal** from the context menu.

7. In the **Terminals** view, run the satrace using the following command:

```
./linux.armv7.satrace/bin/ls.linux.satrace -v ./Test_Linux.elf
```
The tracing starts and collects the data in the *.cwzsa file under the parent folder.

8. Right-click the parent folder and choose **Refresh**. You can see the *.cwzsa file under the parent folder, as shown below.
9. Double-click the *.cwzsa file. The **Import** wizard starts, as shown in the figure below.
10. Click **Finish** to end the **Import** wizard. The file is imported and it is displayed in the **Analysis Results** view.

11. Click the **Trace** link under the **Trace** column in the **Analysis Results** view to view the trace data, as shown in the figure below.

The trace data file opens in the **Trace** viewer showing the trace results, as shown in the figure below.
3.2 Viewing Linux trace collected without using CodeWarrior

This section explains how to view Linux trace data collected from a board without using CodeWarrior.

To explain how to view Linux trace, this section uses trace data collected from the LS102xA TWR board without using CodeWarrior. See AN5001 for instructions on how to collect Linux trace without using CodeWarrior.

The standalone tracing tool generates a *.cwzsa file, an archive type that can be imported and fully decoded using ARMv7 decoder or ARMv7 CodeWarrior. To view the generated segfault.cwzsa file in CodeWarrior for ARMv7, perform these steps:

1. Drag the segfault.cwzsa file into the CodeWarrior IDE. The Import wizard starts.
2. Click Finish to end the Import wizard. The file is imported and it is displayed in the Analysis Results view.
3. Click the Trace link under the Trace column in the Analysis Results view to view the trace data. The decoding of trace and profiling data starts. The decoded trace data is displayed in Trace viewer, as shown in the figure below.
Figure 36: Trace data

You can also find the profiling data files in the same folder where `segfault.cwzsa` file is present. You can drag-and-drop the profiling data files in CodeWarrior to view the profiling data in the Timeline, Code Coverage, Performance, and Call Tree viewers. The Timeline, Code Coverage, Performance, and Call Tree viewers are explained in Viewing trace data using Analysis Results view on page 16.
Chapter 4
Linux Kernel Debug Print Tool

This chapter describes how to work with the Linux Kernel Debug Print tool.

The Linux Kernel Debug Print tool consists of the following two components that work together to perform the debug print operation:

- Target server, which is responsible for collecting Kernel Ring Buffer log in unformatted way
- Host, which periodically requests kernel log data from the server and displays it in a view

This tool's main objective is to provide a user-friendly way of monitoring kernel activity in a CodeWarrior console. It is composed of several modules:

- **Target side:**
  - Debug Print server – Reads on demand the Kernel Ring Buffer log. It also clears the log and sends it to the clients using TCP/IP connection. It collects the redirected printf output from the user space applications.
  - Debug Print dynamic library - Is responsible for redirection of the application's printf messages to the target server.

- **Host side:**
  - Debug Print probe – Is the actual client of the Debug Print server; it can be started from the Debug Print view. When started, it reads periodically the kernel log data from the server and sends it to the Debug Print view to display the kernel log data and other communication messages.
  - Debug Print view – Displays the log data and other communication messages in a user-friendly manner, also allows you to filter the displayed data on the basis of timestamp, module name/application path and pid, or a custom string contained in each log message.

**NOTE**
The tool is independent of CodeWarrior and does not require a debug session.

The ARM binaries have been compiled with tool chain gcc-linaro-arm-linux-gnueabihf-4.8-2013.12_linux and LS1 SDK version 1.1.

Before working on the Debug Print tool, check that TCP/IP communication is established between the host and the target. Below are the steps that are performed in order to see the functionality of the Debug Print tool.

1. Deploy the Software Analysis target binaries on the target. For example, if you have the target root filesystem on NFS, you can copy ls.target.server and libls.debugprint.lib.so* to the host location [NFS_PATH]/home/root.

2. The debug print target server cross-compiled for ARM is located in CodeWarrior in directory: [CWInstallDir]/CW_ARMv7/ARMv7/sa_ls/linux.armv7.debugprint/bin, which needs to be copied on the target (for example, to the home directory). The server requires sudo access (default user on target is root) and requires a single argument; the port number on which clients will listen. If not specified, it will start on the default port 5000.

   Start a ssh console on the target and then start the server:

   ```
   # ssh root@target_ip_address
   # ./ls.target.server
   ```
3. The dynamic library cross-compiled for ARM is located in CodeWarrior directory at: 
\[\text{CWInstallDir}/\text{CW_ARmv7/ARMv7/linux.armv7.debugprint/lib}\], which needs to be copied on the target. This library must be loaded by the Linux loader before the C runtime, when you are running the user space applications that need to be monitored by setting the environment variable, \textit{LD_PRELOAD}.

Start a new ssh console on the target and run a test application (in this case, test.arm) to see its original output:

```
# ssh root@target_ip_address
# ./test.arm
```

Then preload the debug print library and run the test application again:

```
# export LD_PRELOAD=/libls.linux.debugprint.lib.so
# ./test.arm
```

You will notice next time that the test application will not output anything on the console. The output is sent to the target server.
4. On the host machine, open the **Debug Print** view. The Debug Print Probe can be started from the **Debug Print** view and it communicates using TCP/IP connection with the server. When started, it reads periodically the kernel log data from the server and sends it to the **Debug Print** view to display. To start the **Debug Print** view, select **Window > Show View > Other > Software Analysis > Debug Print**.

The table below describes the icons displayed on the Debug Print viewer.

<table>
<thead>
<tr>
<th>Icons</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Clear All]</td>
<td>Removes all text from the view.</td>
</tr>
<tr>
<td>![Start/Stop]</td>
<td>Two-state button used for starting and stopping the Debug Print monitor task.</td>
</tr>
<tr>
<td>![Scroll Lock/Unlock]</td>
<td>Two-state button used for locking and unlocking the scrollbar. If the scrollbar is unlocked, it would always auto-scroll to the latest Debug Print message.</td>
</tr>
<tr>
<td>![Configure]</td>
<td>Opens a dialog for entering the server address and port.</td>
</tr>
<tr>
<td>![Create Debug Print Filters]</td>
<td>Opens a dialog for configuring what information is to be displayed in the <strong>Debug Print</strong> view (specific to timestamps, module name/application path and pid, other string patterns).</td>
</tr>
</tbody>
</table>

5. Click the **Configure** icon, enter the server address and port (for example, address 192.168.0.2, port 5000 – must be the same as for the server).

To configure Debug Print server, click **Configure** icon on the toolbar. The **Configure Debug Print** dialog appears. You can specify the server address, port number at which the server will listen to client, and the target description.

**Figure 38: Configure Debug Print dialog**

![Configure Debug Print dialog]

There is also a **Preference** page associated to this view, which can be accessed by clicking **Window > Preferences**, expand **Software Analysis** node and then select **Debug Print**.
The following **Debug Print** settings are available in the **Preference** dialog:

**Table 12: Debug Print settings**

<table>
<thead>
<tr>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum line count</td>
<td>Limits the number of lines the <strong>Debug Print</strong> view should display. If this</td>
</tr>
<tr>
<td></td>
<td>limit is exceeded, the old messages are deleted.</td>
</tr>
<tr>
<td>Log Debug Print contents to external</td>
<td>If selected, the messages will be appended to an external file besides</td>
</tr>
<tr>
<td>file</td>
<td>displaying them into the <strong>Debug Print</strong> view.</td>
</tr>
<tr>
<td>File name</td>
<td>Path for the external log file</td>
</tr>
</tbody>
</table>

**NOTE**

The ARM binaries are compiled with tool chain gcc-linaro-arm-linux-gnueabihf-4.8-2013.12_linux and LS1 SDK version 1.1.

6. Click the **Start** icon; you will see the kernel log messages are being populated in the view’s text area.
7. Open another console on the target in the same directory, preload the debug print library and run the test application:

```bash
# export LD_PRELOAD=~/.libls.linux.debugprint.lib.so
# ./test.arm
```

8. You will see the application messages getting appended in the Debug Print view:

```
```

9. To see the real time functionality of the Debug Print view, add some more messages to the view, both from kernel and the test application from the same console where the test application was running on the target:

```bash
# ./test.arm
# echo Hello World > /dev/kmsg
# ./test.arm
```
# echo Helloooooo > /dev/kmsg
# echo Hello World 2 > /dev/kmsg

**Figure 42:** Target console - sending messages

10. See the new messages displayed in the Debug Print text area as you enter them in the target shell:

**Figure 43:** Debug Print view - messages from server

This chapter contains the following section:

- Filtering debug print messages on page 51
4.1 Filtering debug print messages

The Linux Kernel Debug Print tool has a powerful filtering engine that allows you to see the desired information.

The filtering engine allows you to display data filtered by timestamp, module name/application path and PID, or a custom string contained by each log message. The Create Debug Print Filters configuration dialog allows creation of multiple filters, each of them able to match the module name, application path or PID of the messages displayed by the Debug Print view. These filters are OR-ed, which means that the view will display all messages which match at least one of the filters.

Figure 44: Create Debug Print Filters dialog

This dialog has three tabs:

- **ModuleName/Path** tab: Allows creation of new filters, by selecting from the Existing list a module name/application path, PID, or both (if available). Click Add Filter to add the filter in the Current Filters list. These filters can be qualified with a timestamp range or a string pattern.

The Existing list contains all the module names/application paths/PIDs from the messages already displayed in the Debug Print view. When you want to filter messages from a certain module or application that is not started or did not print any messages yet, you can manually enter the module name/path or PID in the Custom text box.

When no module filter is selected, and no global qualification is selected, (any) is displayed in the Current Filters, which means that no filter is applied (all messages are displayed).
Filtering debug print messages

- **Timestamp tab**: Allows adding timestamp qualification to the existing filters, or a global qualification if no other filter is created (that is a generic filter which applies to all messages, with all module names, paths and PIDs).

  After the user chooses the timestamp ranges in the Lower Limit/Upper Limit Spinners, you must click **Qualify** in order to add the timestamp qualification to all existing filters. If no filter exists, a global qualification is performed.

  ![Create Debug Print Filters dialog - Timestamp tab](image)

- **Other tab**: Allows adding other type of qualifications to existing filters, or a global qualification if no other filter is created. Currently, the only qualification in this tab is a string pattern which is searched in all the messages (except for timestamps and module names/paths/PIDs). After you input the string pattern, you must click **Qualify** in order to add this qualification to all the existing filters. If no filter exists, a global qualification is performed.
Figure 46: Create Debug Print Filters dialog - Other tab

Create filters for the Debug Print messages

You can select timestamp ranges, module names or paths, PID, or other string patterns to create complex filters.

Module | Timestamp | Other
Messages containing string

Current Filters
(any)

Add Filter  Qualify  Clear Filters

OK  Cancel