**Table of Contents**

Manual Purpose and Structure .............................................................. v

1. Installing the Software ........................................................................ 1
   1.1. Installation for Windows ................................................................. 1
   1.2. Licensing ...................................................................................... 1
       1.2.1. Obtaining a License ................................................................. 3
       1.2.2. Frequently Asked Questions (FAQ) ........................................ 3
       1.2.3. Installing a License ................................................................. 3

2. Introduction to the TASKING Embedded Profiler .............................. 9
   2.1. Trace Support .............................................................................. 11
       2.1.1. MCDS .................................................................................... 13
       2.1.2. miniMCDS ........................................................................... 14
       2.1.3. MCDS Light .......................................................................... 14
   2.2. Trace Limitations .......................................................................... 14

3. Tutorial .............................................................................................. 17
   3.1. Prepare Demo Project in Eclipse ................................................. 18
   3.2. Analyze Project in TASKING Embedded Profiler ....................... 20
   3.3. Fix the Problem ........................................................................... 31
   3.4. Verify Fix in TASKING Embedded Profiler ................................... 32
   3.5. Compare Results ......................................................................... 34
   3.6. Export Results ............................................................................ 35

4. Effects on Profiling Analysis Results .............................................. 37
   4.1. Differences in Analysis Results Due to Compiler and Linker Optimizations ... 37
       4.1.1. Tail Call Optimization .............................................................. 37
       4.1.2. Code Compaction (Reverse Inlining) .................................... 39
       4.1.3. Automatic Function Inlining .................................................. 39
       4.1.4. Delete Duplicate Code and Delete Duplicate Constant Data .... 39
   4.2. Effects of Interrupt Handlers on Interrupted Functions ............ 40

5. Using the TASKING Embedded Profiler ........................................... 41
   5.1. Run the Embedded Profiler in Interactive Mode ....................... 41
   5.2. Run the Embedded Profiler from the Command Line .................. 42
       5.2.1. Command Line Tutorial ......................................................... 43
   5.3. What to Do if Your Application Does not Start on a Board? .... 45

6. Reference .......................................................................................... 47
   6.1. Settings Dialog ........................................................................... 47
   6.2. Analysis Scope Page ..................................................................... 48
   6.3. Run DMA Load Analysis Dialog ............................................... 52
   6.4. Progress Dialog .......................................................................... 53

6.5. Summary Tab .................................................................................. 54
       6.5.1. Configuration ........................................................................ 56
       6.5.2. Info ..................................................................................... 56
       6.5.3. Performance Hotspots Clocks .............................................. 58
       6.5.4. Source Coverage ................................................................. 59
       6.5.5. ICache Miss Count ............................................................... 59
       6.5.6. DCache Miss Count .............................................................. 59
       6.5.7. Memory Access ................................................................... 60
       6.5.8. Memory Conflicts ............................................................... 61
       6.5.9. DMA ................................................................................... 61
       6.5.10. DMA Channel ................................................................. 62
6.5.11. DMA Per Period ................................................................. 63
6.5.12. DMA Channel Per Period .................................................. 63
6.6. Functions Tab ........................................................................... 64
6.7. Source Lines Tab ................................................................. 65
6.8. Instructions Tab ..................................................................... 66
6.9. Memory Access Tab ............................................................ 66
6.10. Memory Conflicts Tab ......................................................... 68
6.11. Source Tab ............................................................................ 68
6.12. Disassembly Tab ................................................................. 69
6.13. Raw Trace Data Tab ............................................................ 71
6.14. DMA Load Tab ..................................................................... 74
6.15. Timeline Tab ........................................................................ 75
Manual Purpose and Structure

Manual Purpose

You should read this manual if you want to know:

• how to use the TASKING Embedded Profiler
• the features of the TASKING Embedded Profiler

Manual Structure

Chapter 1, Installing the Software
Explains how to install and license the TASKING Embedded Profiler.

Chapter 2, Introduction to the TASKING Embedded Profiler
Contains an introduction to the TASKING Embedded Profiler and contains an overview of the features.

Chapter 3, Tutorial
Contains a step-by-step tutorial how to use the demo projects with the TASKING Embedded Profiler.

Chapter 4, Effects on Profiling Analysis Results
Describes the differences in analysis results due to compiler optimizations and explains the effects of interrupt handlers on interrupted functions.

Chapter 5, Using the TASKING Embedded Profiler
Explains how to use the TASKING Embedded Profiler. You can run the TASKING Embedded Profiler in two ways, via an interactive graphical user interface (GUI) or via the command line.

Chapter 6, Reference
Contains an overview of all the fields and columns in an analysis result output.

Related Publications

• Getting Started with the TASKING VX-toolset for TriCore
• TASKING VX-toolset for TriCore User Guide
• AURIX™ TC21x/TC22x/TC23x Family User’s Manual, V1.1 [2014-12, Infineon]
• AURIX™ TC26x A-Step User’s Manual, V1.1 [2013-12, Infineon]
• AURIX™ TC26x B-Step User’s Manual, V1.2 [2014-02, Infineon]
• AURIX™ TC27x User’s Manual, V1.4 [2013-11, Infineon]
• AURIX™ TC27x B-Step User’s Manual, V1.4.1 [2014-02, Infineon]
• AURIX™ TC27x C-Step User’s Manual, V2.2 [2014-12, Infineon]
• AURIX™ TC27x D-Step User’s Manual, V2.2 [2014-12, Infineon]
• AURIX™ TC29x A-Step User’s Manual, V1.1.1 [2014-01, Infineon]
• AURIX™ TC29x B-Step User’s Manual, V1.3 [2014-12, Infineon]
• AURIX™ TC3xx Target Specification, V2.5.1 [2018-04, Infineon]
• AURIX™ TC3xx User’s Manual, V2.0.0 [2021-02, Infineon]
• AURIX™ TC33xEXT User’s Manual Appendix, V1.6.0 [2020-08, Infineon]
• AURIX™ TC35x User’s Manual Appendix, V1.6.0 [2020-08, Infineon]
• AURIX™ TC37x User’s Manual Appendix, V1.6.0 [2020-08, Infineon]
• AURIX™ TC37xEXT User’s Manual Appendix, V1.6.0 [2020-08, Infineon]
• AURIX™ TC38x User’s Manual Appendix, V1.6.0 [2020-08, Infineon]
• AURIX™ TC39x-B User’s Manual Appendix, V1.6.0 [2020-08, Infineon]
Chapter 1. Installing the Software

This chapter guides you through the installation process of the TASKING® Embedded Profiler. It also describes how to license the software.

In this manual, TASKING Embedded Profiler and Embedded Profiler are used as synonyms.

1.1. Installation for Windows

System Requirements

Before installing, make sure the following minimum system requirements are met:

- 64-bit version of Windows 7 or higher
- 2 GHz Pentium class processor
- 4 GB memory
- 500 MB free hard disk space
- Screen resolution: 1024 x 768 or higher

Installation

1. If you received a download link, download the software and extract its contents.
   - or -
   If you received an USB flash drive, insert it into a free USB port on your computer.

2. Run the installation program (setup.exe).

   The TASKING Setup dialog box appears.

3. Select a product and click on the Install button. If there is only one product, you can directly click on the Install button.

4. Follow the instructions that appear on your screen. During the installation you need to enter a license key, this is described in Section 1.2, Licensing.

1.2. Licensing

TASKING products are protected with TASKING license management software (TLM). To use a TASKING product, you must install that product and install a license.

The following license types can be ordered from TASKING.
Node-locked license

A node-locked license locks the software to one specific computer so you can use the product on that particular computer only.

For information about installing a node-locked license see Section 1.2.3.2, Installing Server Based Licenses (Floating or Node-Locked) and Section 1.2.3.3, Installing Client Based Licenses (Node-Locked).

Floating license

A floating license is a license located on a license server and can be used by multiple users on the network. Floating licenses allow you to share licenses among a group of users up to the number of users (seats) specified in the license.

For example, suppose 50 developers may use a client but only ten clients are running at any given time. In this scenario, you only require a ten seats floating license. When all ten licenses are in use, no other client instance can be used. Also a linger time is in place. This means that a license seat is locked for a period of time after a user has stopped using a client. The license seat is available again for other users when the linger time has finished.

For information about installing a floating license see Section 1.2.3.2, Installing Server Based Licenses (Floating or Node-Locked).

License service types

The license service type specifies the process used to validate the license. The following types are possible:

- **Client based** (also known as 'standalone'). The license is serviced by the client. All information necessary to service the license is available on the computer that executes the TASKING product. This license service type is available for node-locked licenses only.

- **Server based** (also known as 'network based'). The license is serviced by a separate license server program that runs either on your companies' network or runs in the cloud. This license service type is available for both node-locked licenses and floating licenses.

  Licenses can be serviced by a cloud based license server called "Remote TASKING License Server". This is a license server that is operated by TASKING. Alternatively, you can install a license server program on your local network. Such a server is called a "Local TASKING License Server". You have to configure such a license server yourself. The installation of a local TASKING license server is not part of this manual. You can order it as a separate product (SW000089).

  The benefit of using the Remote TASKING License Server is that product installation and configuration is simplified.

  Unless you have an IT department that is proficient with the setup and configuration of licensing systems we recommend to use the facilities offered by the Remote TASKING License Server.
1.2.1. Obtaining a License

You need a license key when you install a TASKING product on a computer. If you have not received such a license key follow the steps below to obtain one. Otherwise, you cannot install the software.

Obtaining a server based license (floating or node-locked)

- Order a TASKING product from TASKING or one of its distributors.

  A license key will be sent to you by email or on paper.

If your node-locked server based license is not yet bound to a specific computer ID, the license server binds the license to the computer that first uses the license.

Obtaining a client based license (node-locked)

To use a TASKING product on one particular computer with a license file, TASKING needs to know the computer ID that uniquely identifies your computer. You can do this with the `getcid` program that is available on the TASKING website. The detailed steps are explained below.


2. Execute the `getcid` program on the computer on which you want to use a TASKING product. The tool has no options. For example,

   C:\Tasking\getCid_version
   Computer ID: 5Dzm-L9+Z-WFbO-aMkU-5Dzm-L9+Z-WFbO-aMkU-MDAy-Y2Zm

   The computer ID is displayed on your screen.

3. Order a TASKING product from TASKING or one of its distributors and supply the computer ID.

   A license key and a license file will be sent to you by email or on paper.

When you have received your TASKING product, you are now ready to install it.

1.2.2. Frequently Asked Questions (FAQ)

If you have questions or encounter problems you can check the support page on the TASKING website.


This page contains answers to questions for the TASKING license management system TLM.

If your question is not there, please contact your nearest TASKING Sales & Support Center or Value Added Reseller.

1.2.3. Installing a License

The license setup procedure is done by the installation program.
If the installation program can access the internet then you only need the license key. Given the license key the installation program retrieves all required information from the remote TASKING license server. The install program sends the license key and the computer ID of the computer on which the installation program is running to the remote TASKING license server, no other data is transmitted.

If the installation program cannot access the internet the installation program asks you to enter the required information by hand. If you install a node-locked client based license you should have the license file at hand (see Section 1.2.1, Obtaining a License).

Floating licenses are always server based and node-locked licenses can be server based. All server based licenses are installed using the same procedure.

1.2.3.1. Configure the Firewall in your Network

For using the TASKING license servers the TASKING license manager tries to connect to the Remote TASKING servers lic1.tasking.com, lic2.tasking.com, lic3.tasking.com, lic4.tasking.com at the TCP ports 8080, 8936 or 80. Make sure that the firewall in your network is transparently enabled for one of these ports.

1.2.3.2. Installing Server Based Licenses (Floating or Node-Locked)

If you do not have received your license key, read Section 1.2.1, Obtaining a License before you continue.

1. If you want to use a local license server, first install and run the local license server before you continue with step 2. You can order a local license server as a separate product (SW000089).

2. Install the TASKING product and follow the instructions that appear on your screen.

The installation program asks you to enter the license information.

3. In the License Key field enter the license key you have received from TASKING and click Next to continue.
The installation program tries to retrieve the license information from a remote TASKING license server. Wait until the license information is retrieved. If the license information is retrieved successfully subsequent dialogs are already filled-in and you only have to confirm the contents of the dialogs by clicking the Next button. If the license information is not retrieved successfully you have to enter the information by hand.

4. Select your **License Type** and click Next to continue. If the license type is already filled in and grayed out, you can just click Next to continue.

   You can find the license type in the email or paper that contains the license key.

5. (For floating licenses only) Select **Remote TASKING license server** to use one of the remote TASKING license servers, or select **Local TASKING license server** for a local license server. The latter requires optional software.

   (For local license server only) specify the **Server name** and **Server port** of the local license server.

   Note that a Node-locked server based license always uses the Remote TASKING license server.

6. Click Next and follow the rest of the instructions to complete the installation.

### 1.2.3.3. Installing Client Based Licenses (Node-Locked)

If you do not have received your license key and license file, read Section 1.2.1, *Obtaining a License* before continuing.

1. Install the TASKING product and follow the instructions that appear on your screen.

   The installation program asks you to enter the license information.

   **License Key Information**
   Specify your license key

   Please enter the license key that you have received from TASKING. The key has the format like aaaa-bbbb-cccc-dddd. If you do not have a key, please contact TASKING through licensing@tasking.com, or contact your TASKING representative.

   License Key: 

2. In the **License Key** field enter the license key you have received from TASKING and click Next to continue.
The installation program tries to retrieve the license information from a remote TASKING license server. Wait until the license information is retrieved. If the license information is retrieved successfully subsequent dialogs are already filled-in and you only have to confirm the contents of the dialogs by clicking the Next button. If the license information is not retrieved successfully you have to enter the information by hand.

3. Select **Node-locked client based license** and click **Next** to continue.
4. In the **License File Contents** field enter the contents of the license file you have received from TASKING.

   *The license data is stored in the file licfile.txt in the etc directory of the product (<install_dir>\etc).*

5. Click **Next** and follow the rest of the instructions to complete the installation.
Chapter 2. Introduction to the TASKING Embedded Profiler

After your application has been verified, thoroughly tested and debugged, and by itself behaves correctly, you may still run into performance and timing issues. Many timing issues can be addressed simply by improving the performance of the applications that caused a missed deadline. Furthermore, by reducing the core load of your applications you may be able to go for a device that is cheaper because it has fewer cores. A way to address these issues is performance tuning.

With performance tuning we refer to optimizing your application for a specific target device. Common situations where performance tuning of your application makes sense are:

- You are using self-made libraries that are called a lot and thus have a big impact on overall application performance.
- You develop/adapt low level drivers and basic software (BSW) components.
- You are close to or above your core load budget limit.
- You have a timing problem in your schedule that could be fixed by speeding up specific tasks but want to avoid changing the schedule.
- You want to try and target a smaller electronic control unit (ECU) in order to save costs.
- You care about easily and cost effectively tracking and improving the performance of your code on target devices.
- You want your tests to cover most of your source code to lower the probability of undetected software bugs.

Embedded hardware platforms are too complex for the average software developer to predict or understand the performance of his code. In order to optimize code for a specific platform (cores plus peripherals), developers need feedback from the hardware on which specific part of their code is suboptimal (in terms of memory consumption, jitter, execution time, ...) and what is the root cause of the performance impact. The TASKING Embedded Profiler is a smart profiling tool that provides this feedback.

The TASKING Embedded Profiler communicates with an embedded processor (CPU) to gather real-time tracing and performance data. The tool gives an overview over the current clock settings — no need to get an oscilloscope to verify that the clocks are configured properly for a benchmark run. After verification of correct clock setup, you are guided through a few easy steps that pinpoint the source lines that have the greatest performance impact. The tool indicates the root cause of the performance impact and gives simple instructions on how to address the problem. The data is presented in graphics and tables and into computer readable formats.
After applying the suggested mitigation, you can use the TASKING Embedded Profiler to confirm that the problem has indeed been fixed. With the default settings of the tool this all happens non-intrusively with real data collected from the application running on the real device. Using such a performance tuning tool, non-expert users can often highly speed up untuned applications.

Features of the TASKING Embedded Profiler

- Performance analysis
- Flow analysis
- Memory access analysis
- Function-level analysis
- DMA load analysis
- Compare analysis runs of the same kind
- Organize analyses and results in projects
- Load/store analysis results
- Graphical user interface (GUI) and command line support
• Support for the latest Device Access Server driver (DAS, see www.infineon.com/das) and support for all Device Access Port (DAP) miniWigglers that are supported by the DAS drivers. All debug interfaces supported by the Infineon miniWigger can be used to connect to the target hardware. This can be 10-pin DAP / 20-pin Automotive JTAG connector or 10-pin DAP / 16-pin OCDSL1, depending on which miniWiggler version is used.

Performance analysis

This type of analysis traces instructions and performance events. It measures the CPU clock count and it finds branch misses, cache misses and stalls due to memory access delays or pipeline hazards. It also provides detailed coverage information. You can run this type of analysis on the whole application or select specific functions.

Flow Analysis

This type of analysis traces all flow changes, including all branches, function calls and function returns. This is the fastest analysis to have detailed coverage information.

Memory access analysis

This type of analysis traces function calls, function returns and data accesses. You can run this type of analysis on the whole application or select specific functions.

Function-level analysis

This type of analysis traces all function calls and function returns. It also provides brief coverage information about functions by means of functions invoked or not. This is the fastest analysis.

DMA load analysis

This type of analysis traces both DMA (Direct Memory Access) load and flow changes. For the DMA trace hardware is needed that supports OTGB (OCDS Trigger/Trace Bus on product chip) or OTGM (OCDS Trigger/Trace Multiplexer on product chip). For the flow trace a free CPU on a Trace Source is needed.

2.1. Trace Support

The standard TriCore/AURIX™ processors (production devices) lack debug trace functionality. However, this functionality is very useful when you develop and test your application.

The TASKING Embedded Profiler uses the Multi-Core Debug Solution (MCDS) for on-chip trace support. The following table shows the devices that are supported by the TASKING Embedded Profiler.
<table>
<thead>
<tr>
<th>Device</th>
<th>MCDS type</th>
<th>Trace memory</th>
<th>DMA Load supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC23xED</td>
<td>MCDS</td>
<td>TCM / XTM</td>
<td>Yes</td>
</tr>
<tr>
<td>TC26xED</td>
<td>MCDS</td>
<td>TCM / XTM</td>
<td>Yes</td>
</tr>
<tr>
<td>TC27xED</td>
<td>MCDS</td>
<td>TCM</td>
<td>Yes</td>
</tr>
<tr>
<td>TC29x</td>
<td>miniMCDS</td>
<td>TRAM</td>
<td>No</td>
</tr>
<tr>
<td>TC29xED</td>
<td>MCDS</td>
<td>TCM / XTM</td>
<td>Yes</td>
</tr>
<tr>
<td>TC33xEXT</td>
<td>MCDSLight</td>
<td>TCM / XTM</td>
<td>Yes</td>
</tr>
<tr>
<td>TC35x</td>
<td>MCDSLight</td>
<td>TCM / XTM</td>
<td>Yes</td>
</tr>
<tr>
<td>TC37x</td>
<td>miniMCDS</td>
<td>TRAM</td>
<td>No</td>
</tr>
<tr>
<td>TC37xEXT</td>
<td>MCDS</td>
<td>TCM / XTM</td>
<td>Yes</td>
</tr>
<tr>
<td>TC38x</td>
<td>miniMCDS</td>
<td>TRAM</td>
<td>No</td>
</tr>
<tr>
<td>TC39xAED</td>
<td>MCDS</td>
<td>TCM / XTM</td>
<td>Yes, but CPU0 cannot be used</td>
</tr>
<tr>
<td>TC39xED</td>
<td>MCDS</td>
<td>TCM / XTM</td>
<td>Yes</td>
</tr>
</tbody>
</table>

TC33xED is sometimes used in Infineon documentation for TC33xEXT.

TC37xED is sometimes used in Infineon documentation for TC37xEXT.

TC39xED is sometimes used in Infineon documentation for TC39x.

**Naming convention**

You can see by the name on the processor what type of device it is. For example, with SAK-TC299TE the last letter indicates the "Feature Package". If this letter is an 'E' or 'F' you have an Emulation Device. For the TC3xx devices, if the "Feature Package" letter is an 'E' you have an Emulation Device.

For a detailed naming convention see the Infineon website:

- AURIX™ Product Naming

**Trace memory**

Trace information is stored in a dedicated trace buffer. With an Emulation Device (ED) you can allocate part of the Emulation Memory (EMEM) as trace buffer memory. The Emulation Memory is divided in RAM blocks, the so-called 'tiles', which can be used as Calibration or Trace memory. These memory tiles consists of TCM, XCM and XTM. TCM (Trace Calibration Memory) can be used for Trace memory or Calibration, XCM (Extended Calibration Memory) can only be used for Calibration memory and XTM (Extended Trace Memory) can only be used for Trace memory.

Production Devices that are equipped with miniMCDS use TRAM for trace memory.

Which trace memory you can select depends on the selected processor.
Tile memory range

For TCM, you can choose which part of the Emulation Memory should be used for tracing. For XTM always both tiles are used for tracing.

Be careful that the same tile memory range used for tracing is not used by the target application, as this can lead to unexpected trace results. The number of tiles vary per Emulation Device.

Trace mode

When you run a trace analysis in the TASKING Embedded Profiler, you can set the trace mode:

- **One shot mode.** In this mode the analysis will run until the trace buffer is full, or when the application finishes or when you stop the analysis manually. This is non-intrusive, meaning that the trace does not interfere the running processor. After the trace has stopped the Embedded Profiler reads the collected data.

- **Continuous trace.** In this mode the analysis will run until the application finishes or when you stop the analysis manually. This mode is intrusive, meaning that the processor is stopped temporarily every time the trace buffer has been filled, so that the Embedded Profiler can read the collected data. After that the processor continues execution and continues writing to the trace buffer.

Raw trace data

Raw trace data is included as a service to advanced users who are familiar with the Infineon Multi-Core Debug Solution and who want to examine program flow. Raw trace data is useful, for example, to see why stall cycles are assigned to instructions that do not access memory. This can be the case when an instruction is target of a branch. Raw trace data is displayed in a separate tab. The Raw Trace Data tab has a search field that you can use to search through the address column. It has buttons to search the Next, Previous, First and Last occurrence of the specified address. It does not support wildcards or regular expressions.

Attach mode

When you run a trace analysis in the TASKING Embedded Profiler, you can set the attach mode:

- **Reset device.** In this mode the device is reset first and then the analysis starts.

- **Hot attach.** In this mode the analysis will start at the current execution position of the running application.

2.1.1. MCDS

MCDS (Multi-Core Debug Solution) uses ED (Emulation Devices). An Emulation Device has an Emulation Extension Chip (EEC) added to the same silicon, which is accessible through the JTAG or DAP interface (EEC is only for devices that have tile range supports like MCDS and MCDSLight.). The TASKING Embedded Profiler supports the on-chip trace feature of the Emulation Device.

The MCDS has support for a maximum of three processor observation blocks (POBs). This means that, depending on the device, up to three cores can be traced at once. For the hardware that can trace three
cores at the same time one of the three cores must be core 0. For detailed information about MCDS we recommend that you read the processor documentation.

2.1.2. miniMCDS

Some devices that do not have MCDS come with miniMCDS. miniMCDS is a subset of the on-chip trace feature that is available on Emulation Devices. The mini-MCDS memory is not suitable for safety related data and must not be used for data storage by safety applications.

The miniMCDS devices have no EEC and therefore only TRAM memory. These kind of devices have only one POB, this means that only one core can be traced at the same time. See the processor documentation for detailed information about the device.

2.1.3. MCDS Light

In AURIX 2G devices a new type of Multi-Core Debug Solution is introduced, called MCDS Light. MCDS Light is a subset of MCDS. The difference with MCDS is that MCDS Light has a maximum of two POBs, which means that a maximum of two cores can be traced at once.

One of the advantages of MCDS Light over miniMCDS is that it comes with EMEM and thus is not limited to the very small TRAM memory.

2.2. Trace Limitations

The MCDS reports events with a timestamp resolution of half of the CPU clock. This means that even when every instruction is traced it is not always 100% possible to know to which instruction the reported clock cycles belong. Beyond that there are cases where the MCDS summarizes its reports even though a report per instruction was requested.

See the following worst case example for the start of a function with multiple 1 cycle instructions, where it looks as if clock cycles are missing:

<table>
<thead>
<tr>
<th>LineNo</th>
<th>Source</th>
<th>Clocks</th>
<th>Coverage %</th>
<th>Branch Misses</th>
<th>ICache Misses</th>
<th>DCache Misses</th>
<th>Stalls</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>/* file_1.c */</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>#include &lt;stdio.h&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>#include &lt;time.h&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>void func_1(void)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>__nop();</td>
<td>15</td>
<td>100</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>12</td>
</tr>
<tr>
<td>9</td>
<td>0x80000072 __nop</td>
<td>15</td>
<td>✓</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>12</td>
</tr>
<tr>
<td>10</td>
<td>__nop();</td>
<td>-</td>
<td>100</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>0x80000074 __nop</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>__nop();</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

The first \texttt{nop()} has a high number of clocks reported, caused by instruction cache miss and the pipeline penalty of the call. The MCDS cannot report on the second and third \texttt{nop()}. It is important to realize that
the clocks spent in those were reported as part of the count for the first `nop()` and not somehow dropped. Instructions without clocks that can still be covered have a help popup explanation when you hover the mouse over the empty clock.

**Tracing regions**

When you are tracing regions, the entry/exit from a region misses clocks which are detected before the MCDS detects the region has been entered, or after the region has been left. This is a hardware limitation and in practice means counts for instructions will be zero. For a short function this can mean the whole function is being reported as covered without any clocks being assigned at all.

**Differences between different analysis types on the same embedded program**

When you run different analysis types on the same embedded program, the output can be different:

- Performance analysis vs. flow analysis will not differ that much. Mostly clocks assigned to performance events (stalls, cache misses) will sometimes move between caller/callee.

- Function analysis can differ more, especially as it does not see tail call optimizations and interrupts. See Chapter 4, *Effects on Profiling Analysis Results.*
Chapter 3. Tutorial

The profiler\tutorials directory of the TASKING Embedded Profiler installation contains several examples. They serve as a good starting point for your own profiling analysis project. All examples are present for the TC29xB and the TC39xB.

• demo_dspr - A project demonstrating how defaulting to the wrong scratch pad memory results in a penalty in stalls.

• demo_dcache - A project demonstrating how multiple passes over a large buffer can cause many data cache misses.

• demo_concurrent - A project demonstrating how accessing the same memory from multiple cores causes stalls.

• demo_tailcall - A project demonstrating a possible difference in analysis results between a performance analysis and a memory analysis or function analysis. This is due to the tail call elimination optimization of the C compiler. Tail call elimination is part of the peephole optimization of the C compiler. See Section 4.1.1, Tail Call Optimization.

• demo_dma_simple - A project demonstrating DMA transfers for four channels in a row that is repeated four times.

When the Timeline check box is set when the DMA analysis is started the channel activity will be shown in a 'Timeline' view. See Section 6.15, Timeline Tab.

The DMA simple example uses DMA channel 1, 2, 3 and 4 with a relative payload in size 1, 2, 3 and 4. When looking at the 'DMA Load' view this payload size can be seen as the amount of time spent transferring data for channel 1 is half that of channel 2, a third of channel 3 and a quarter of channel 4. See Section 6.14, DMA Load Tab.

The DMA transfers of this example do not overlap nor interrupt each other so the number of activations will be 4 for each channel. Note: on TriCore the highest DMA channel number has the highest priority and interrupts DMA transfers of channels with a lower number.

All examples come with TASKING Embedded Profiler projects (files with the .EmbProf extension), with pre-run analyses. You can open a project in the TASKING Embedded Profiler to inspect the various analysis results, without having to run the examples on a target board.

All examples also contain an ELF file (.elf) and an Intel Hex file (.hex), so that you can also use the TASKING Embedded Profiler, TASKING Embedded Debugger or a flash tool to flash an example application on a target board. Note that these files are for the original example, without any fixes.

In this tutorial we will use the demo_dspr example for the TC39xB to go through the process of preparing your project from scratch, running a profiling analysis, fixing the problem and rerunning a profiling analysis to see the improvement. After this tutorial you can use the other tutorials yourself in a similar way.
3.1. Prepare Demo Project in Eclipse

Before you can use the TASKING Embedded Profiler, you must have an application ELF file with debug information.

The example projects delivered with the TASKING Embedded Profiler are Eclipse projects suitable for the TASKING VX-toolset for TriCore v6.2r1 or newer. For this part of the tutorial it is assumed that you have this toolset version or newer installed.

Connect the target board

- Connect the Infineon TriBoard TC39xB to your computer. See the documentation that came with the board for more information.

Import an example project

1. Start the TASKING VX-toolset for TriCore Eclipse IDE.
2. From the File menu, select Import.

   The Import dialog appears.

3. Select General » Existing Projects into Workspace and click Next.

   The Import Projects dialog appears.
4. Click **Select archive file** and browse to the example ZIP file delivered with the TASKING Embedded Profiler.

5. Leave the other settings in this dialog as is and click **Finish**.

   *The project will be added to your workspace.*

You can now examine the source files, build the project (for your target) and flash the application.

**Examine source file**

1. In the C/C++ Projects view double-click on the source file `demo_dspr.c`.

   *The file will be opened in the source editor.*

2. Examine the source file and make sure that the following define has the value 0:

   ```
   #define FIXED   0
   ```

   This define is used to demonstrate the different profiler results before and after fixing the source file.
Set project options

The resulting application ELF file must contain debug information. The demo projects already have debugging enabled by default. So, for the demo projects you can skip this step. For your own project, make sure that debugging is enabled.

1. From the Project menu, select Properties for. Alternatively, you can click the button.

   "The Properties for demo_dspr_tc39 dialog appears."

2. If not selected, expand C/C++ Build and select Settings to access the TriCore tool settings.

3. On the Tool Settings tab, expand C/C++ Compiler » Debugging, set option Generate symbolic debug information to Default or Full and click OK.

Build the project

• From the Project menu, select Build demo_dspr_tc39, or click from the toolbar.

3.2. Analyze Project in TASKING Embedded Profiler

Now it is time to start analyzing the demo project.

Create a project

1. Start the TASKING Embedded Profiler.
The TASKING Embedded Profiler window is divided into two panes. The left pane is reserved for the project tree and the right pane is reserved for analysis results.

2. From the Project menu, select **New Project**.

*The New Project dialog appears.*

```plaintext
<table>
<thead>
<tr>
<th>Directory:</th>
<th>C:\Users\name\workspace_prof\demo_dspr_tc39</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name:</td>
<td>demo_dspr_tc39</td>
</tr>
<tr>
<td>Executable file</td>
<td>Debug\demo_dspr_tc39.elf</td>
</tr>
<tr>
<td>Verify embedded image</td>
<td>☑</td>
</tr>
<tr>
<td>Source code path:</td>
<td></td>
</tr>
<tr>
<td>Processor:</td>
<td>TC39xED</td>
</tr>
<tr>
<td>Device server:</td>
<td></td>
</tr>
<tr>
<td>JTAG frequency:</td>
<td>1 MHz</td>
</tr>
</tbody>
</table>
```
3. In the **Directory** field, specify the directory where you want to store the Embedded Profiler project file (file with extension `.EmbProf`).

4. In the **Name** field, enter the name of the project, for example `demo_dspr_tc39`. By default, the name is derived from the selected directory, but you can change it.

5. In the **Executable file** field, specify the name of the ELF file. For standard TASKING projects this file is usually in the Debug directory relative to the project directory. If the executable file is stored in another directory, the full path name is shown.

6. Enable **Verify embedded image** to compare the contents of the flash image to the ELF file before a run is started. If there is a difference you are asked if the flash should be updated before the run starts, unless you also enable **Always flash if different**.

7. Optionally specify a **Source code path** (a semi-colon separated directory list). Normally, the location of the source files is taken from the ELF file.

8. Select the **Processor**. For example, TC39xED.

9. For the **Device server**, enter the name or address of a remote PC that the TriCore hardware is connected to (leave blank for localhost). See also the DAS documentation.

10. Optionally specify the **JTAG frequency** in MHz.

11. Leave the rest of the dialog as is and click **Create**.

    *The new project is created and opened.*

    demo_dspr_tc39

**Create a Performance analysis**

1. From the **Analysis** menu, select **New Analysis**.

    *The New Analysis wizard appears.*
2. Several types of analyses are possible. Select **Performance Analysis** and click **Next**.

*The Analysis Scope page appears.*
3. For this tutorial leave this page as is, this means that the whole application will be analyzed (the full address range). If you select one or more **Application functions** or specify an **Address region** and click **Add**, tracing is limited to those functions and address ranges.

   For details about the Analysis Scope page see Section 6.2, **Analysis Scope Page**.

   Note that the number of trace ranges is limited by the hardware. Usually, you can select a maximum of 4 trace ranges. A trace range can contain several functions and address regions.

4. Click **Next**.

   *The Analysis Name page appears.*

5. Specify the analysis name. A default name has already been filled in based on the analysis type and a sequence number, but you can specify your own name.

6. Click **Finish**.

   *The new analysis is created and is visible in the project tree.*

### Run the analysis

1. In the project tree select the analysis you want to run.

2. From the **Analysis** menu, select **Run Analysis**.

   *The Run Analysis dialog appears.*
3. In the **Core** field, select the TriCore core(s) for which you want to run the analysis.

4. Select a trace **Buffer mode**. A **One shot mode** trace ends when the hardware trace buffer is full, or when the application finishes or when you stop the analysis manually. A **Continuous trace** ends when the application finishes or when you stop the analysis manually. This mode is intrusive, meaning that the processor is stopped temporarily every time the trace buffer has been filled, so that the profiler can read the collected data. After that the processor continues execution and continues writing to the trace buffer.

5. Select an **Attach mode**. With **Reset device**, tracing starts by running the program in the embedded device from the reset vector. With **Hot attach**, tracing starts by continuing tracing from the current program counter location.

6. Select the type of **Memory** that should be used for tracing, **TCM** (Trace Calibration Memory) or **XTM** (Extended Trace Memory). Production Devices that are equipped with mini-MCDS always use TRAM for trace memory.

7. For trace calibration memory (TCM) on emulation devices only, enter a trace memory **Tile range**. Trace calibration memory (TCM) of emulation devices consists of a consecutive number of tiles.
Select the first and last tile index you want to use for trace memory. The tile size and trace buffer size are listed as information.

8. Enter an analysis result **Name** (default Result- and a sequence number).

9. Optionally **Save** additional **Raw trace** data. Raw trace data is for advanced users who want to examine program flow. Raw trace data is useful, for example, to see why stall cycles are assigned to instructions that do not access memory. If you enable this option, an extra **Raw Trace Data** tab appears in the analysis result.

10. Click **Run**.

   *The default setting is that before a run is started the contents of the flash is compared to the ELF file and you are asked if the flash should be updated before the run starts. You can change this in the project settings. When the ELF file is flashed, the analysis starts. After the analysis is finished the result is present in the project tree.*

1. **Inspect the result of the Performance analysis**

   In the project tree select the result you want to inspect (PerfAnalysis-1, Result-1).

   *The result appears in several tabs.*
2. On the **Summary** tab, notice the number of **CPU clock counts** (311,408), **Stalls** (107,106), **Average stalls per clock** (0.34) and **DCache misses** (4,138).

If the value is marked red or not depends on a threshold. For the average stalls per clock, the default threshold is 0.7. You can change this threshold value in the Settings dialog (**Project » Settings**). See Section 6.1, **Settings Dialog**.

3. On the **Functions** tab, notice the high number of **Stalls** with functions **main**, **_c_init_entry** and **_start**.
4. Double-click on `main`.  
The Source tab opens.

5. Notice the high number of stalls is in the `for` loop.

6. Enable Show disassembly on the Source tab to show disassembly intermixed with the source lines, or open the Disassembly tab. When you double-click on an assembly instruction in the Source tab, the Disassembly tab is opened automatically at the right position. Notice that the stalls are related to memory access.
Create and run a Memory access analysis

1. Repeat the steps described above with Create a Performance analysis, but in Step 2 select Memory Access Analysis.

2. Run the new analysis similar as described above with Run the analysis.

Inspect the result of the Memory access analysis

1. In the project tree select the result you want to inspect (MemAnalysis-1, Result-1).

   The result appears in several tabs.
2. On the **Summary** tab, notice the high number of **Local Memory Unit accesses** (33,322). When you hover the mouse over a value that is marked, a context sensitive help box with additional information can appear.

   If the value is marked red or not depends on a threshold factor. The default threshold factor is 0.05. The threshold for LMU memory access is calculated as: `factor * Local Data Scratch Pad RAM accesses`. In this case `0.05 * 600 = 30`. You can change this threshold factor in the Settings dialog (**Project » Settings**). See **Section 6.1, Settings Dialog**.

3. On the **Memory Access** tab and notice that `main` and `_c_init` both access variable `x` in LMU.
4. Hover the mouse over LMU in main.

A context sensitive help box appears with a suggestion to solve the problem.

3.3. Fix the Problem

Now that we have analyzed the problem, we can fix it.

1. In the TASKING TriCore Eclipse IDE, double-click on the source file demo_dspr.c.

   The file will be opened in the source editor.

2. Change the following source line:

   ```c
   #define FIXED   0
   ```

   into:

   ```c
   #define FIXED   1
   ```
3. From the **Project** menu, select **Rebuild demo_dspr_tc39** to generate a new ELF file.

### 3.4. Verify Fix in TASKING Embedded Profiler

Now that we have fixed the problem, we can use the TASKING Embedded Profiler to rerun both the Performance analysis and the Memory access analysis mentioned in **Section 3.2, Analyze Project in TASKING Embedded Profiler** and see the new results of the analyses.

**Rerun the Performance analysis and inspect the result**

1. In the TASKING Embedded Profiler, select **PerfAnalysis-1**.
2. From the **Analysis** menu, select **Run Analysis**.
3. Click **Run**.
   
   *This creates a Result-2.*

4. Select **Result-2** and notice that on the **Summary** tab, the number of **CPU clock counts**, **Stalls**, **Average stalls per clock** and **DCache misses** have reduced significantly.
5. Also inspect the other tabs yourself to see the results.

**Rerun the Memory access analysis**

1. In the TASKING Embedded Profiler, select MemAnalysis-1.
2. From the **Analysis** menu, select **Run Analysis**.
3. Click **Run**.
   
   *This creates a Result-2.*

4. Select **Result-2** and notice that on the **Summary** tab, the accesses are now in DSPR0. And notice that on the **Memory Access** tab, _c_init and main now both access variable x in DSPR0.*

![Tasking Embedded Profiler screenshot showing memory access analysis results.](image-url)
3.5. Compare Results

The Embedded Profiler has a feature to compare results. This is very useful to see the differences before and after a fix. Note that you can only compare results from the same analysis.

1. In the TASKING Embedded Profiler, select a result. For example, Result-2 of PerfAnalysis-1.

2. From the Result menu, select Compare Results.

3. Select another result, for example Result-1. The results you can select are marked yellow.

   The comparison starts and a difference report is created. The numbers in the report are calculated as the "first selected result" minus the "second selected result".
3.6. Export Results

You can export analysis results and comparison results to comma separated values (CSV) files. You can choose to export instructions, functions or memory depending on the analysis type.

1. In the TASKING Embedded Profiler, select a result. For example, Result-1 of PerfAnalysis-1.

2. From the Result menu, select Export to CSV.

   The Export to CSV dialog appears.

3. Enter the filename(s) and click Export.
Chapter 4. Effects on Profiling Analysis Results

This chapter describes the differences in analysis results due to compiler optimizations and explains the effects of interrupt handlers on interrupted functions.

4.1. Differences in Analysis Results Due to Compiler and Linker Optimizations

4.1.1. Tail Call Optimization

Analysis results may be different for functions in a performance analysis or flow analysis compared to a memory analysis or function analysis. This can happen due to the tail call optimization of the C compiler, which is part of the peephole optimization of the C compiler (option -Oy). This optimization is enabled by default for the TASKING VX-toolset for TriCore.

This optimization replaces the call to the leaf function with a jump instruction. The leaf function's return instruction then performs the return that the calling function would have done.

The function analysis and memory analysis do see that the return belongs to the leaf function, but do not know about the jump instruction to a leaf function. The cycles up to the return instruction are added to the leaf function. Therefore the number of cycles for the calling function are less than expected.

Without the tail call optimization, the normal function flow is: func_a() calls func_b() which calls func_c(). func_c() returns to func_b() which returns to func_a().

```
func_a()
  |__ func_b()
    |__ func_c()
```

With tail call optimization, the function flow becomes: func_a() calls func_b() which jumps to func_c(). func_c() returns to func_a().

For an example of this behavior, see the demo_tailcall tutorial.

1. Import the demo_tailcall tutorial for the TC29x or TC39x the same way as explained for demo_dspr in Section 3.1, Prepare Demo Project in Eclipse. For this tutorial we use demo_tailcall_tc39. The tutorial already contains an Embedded Profiler project file.

2. Start the TASKING Embedded Profiler.

3. From the Project menu, select Open Project, and select demo_tailcall_tc39.EmbProf.

4. Inspect the Functions tab in Result-1 of PerfAnalysis-1, MemAnalysis-1, FuncAnalysis-1 and FlowAnalysis-1.
For the Performance Analysis PerfAnalysis-1 and the Flow Analysis FlowAnalysis-1, you can see there are 4902 clocks for function tail_test_1() and 3856 clocks for function len().

For the Memory Analysis MemAnalysis-1 and the Function-level Analysis FuncAnalysis-1, you can see there are 8890 clocks for function len() and 548 clocks for function tail_test_1().

5. Rebuild the example in the TriCore VX-toolset for TriCore with the peephole optimization disabled (C compiler option -OY, or in Eclipse select Project » Properties for » C/C++ Build » Settings » Tool Settings » C/C++ Compiler » Optimization » Optimization level » Custom Optimization
4.1.2. Code Compaction (Reverse Inlining)

The compiler optimization Code Compaction (C compiler option -Or) is the opposite of inlining functions: chunks of code that occur more than once, are transformed into a function. This reduces code size at the cost of execution speed.

For profiling results this will make functions that in source do not do function calls show up having a higher value for clocks with children.

These effects should be considered when looking at profiling results. Turn off code compaction with C compiler option -OR, or in Eclipse select Project » Properties for » C/C++ Build » Settings » Tool Settings » C/C++ Compiler » Optimization » Optimization level » Custom Optimization and in the Custom Optimization tab disable Code compaction), and run the analyses again in the Embedded Profiler to see the differences.

4.1.3. Automatic Function Inlining

The compiler optimization Automatic Function Inlining (C compiler option -Oi) will inline small functions that are not too often called. This reduces execution time at the cost of code size.

For profiling results this will make functions that in the source do function calls show up as not doing the functions calls. Instead the 'Clocks' count for that function becomes higher than it would when the actual function was not inlined. This also results in a lower than expected value for 'Clocks With Children'.

These effects should be considered when looking at profiling results. Turn off automatic function inlining with C compiler option -OI, or in Eclipse select Project » Properties for » C/C++ Build » Settings » Tool Settings » C/C++ Compiler » Optimization » Optimization level » Custom Optimization and in the Custom Optimization tab disable Automatic function inlining), and run the analyses again in the Embedded Profiler to see the differences.

4.1.4. Delete Duplicate Code and Delete Duplicate Constant Data

The linker optimizations Delete Duplicate Code (linker option -Ox) and Delete Duplicate Constant Data (linker option -Oy) remove code and constant data that is defined more than once from the resulting object file.

For profiling results this may result in a function call or a data reference that looks unexpected but is actually correct. The same code or the same data just another name.

These effects should be considered when looking at profiling results. Turn off these linker optimizations with linker option -OXY, or in Eclipse select Project » Properties for » C/C++ Build » Settings » Tool Settings » Linker » Optimization and disable Delete duplicate code and Delete duplicate data), and run the analyses again in the Embedded Profiler to see the differences.
4.2. Effects of Interrupt Handlers on Interrupted Functions

Interrupt handlers that do not call any functions (user functions, run-time functions and functions generated for code compaction) are not visible in function analyses and memory analyses and their clock cycles are added to the interrupted function. For performance analyses the interrupt function is visible and its cycles are added to the interrupted function, except for the first few instructions that are part of the interrupt vector table; the interrupt handler and children are visible and have cycles accounted to them.

Interrupt handlers that do call function(s) are visible for all three analysis types. For function and memory analyses, the interrupt handler and its children are visible and have cycles accounted to them. These cycles are not added to the clocks with children of the interrupted function. For performance analyses, the interrupt handler cycles (including children) are added to clocks with children of the interrupted function, except for the cycles accounted to the interrupt vector table; the interrupt handler and children are visible and have cycles accounted to them.
Chapter 5. Using the TASKING Embedded Profiler

You can run the TASKING Embedded Profiler in two ways, via an interactive graphical user interface (GUI) or via the command line. The GUI variant is useful in showing graphical analysis results with hints how to improve the code. The command line interface is useful in automated scripts and makefiles to generate analysis results in comma separated values (CSV) files.

5.1. Run the Embedded Profiler in Interactive Mode

To start the Embedded Profiler select Embedded Profiler from the Windows Start menu. The program starts with an empty window except for a menu bar and a toolbar at the top. The area below that consists of two panes. The left pane is used to display a project tree, with a project name, one or more analysis names and one or more result names. The right pane is used to display an analysis result. You can resize a pane by dragging one of its four corners and you can move a pane by dragging its title. You can drag the button toolbar to another place, for example vertically to the left side or even detach it from the main window.

Normal project management is available. You can create, open, edit, close or delete a project. A project filename will have the extension .EmbProf.
The steps to:

- create a project
- create an analysis
- run an analysis

are described in Section 3.2, Analyze Project in TASKING Embedded Profiler.

See also Section 3.5, Compare Results and Section 3.6, Export Results. For details about the Results see Chapter 6, Reference.

5.2. Run the Embedded Profiler from the Command Line

To run the Embedded Profiler from the command line use the EmbProfCmd batch file in a Windows Command Prompt. Enter the following command to see the usage:

```
EmbProfCmd --help
```

The general invocation syntax is:

```
EmbProfCmd options project.EmbProf
```

where, `project.EmbProf` refers to an existing Embedded Profiler project file.

The following options are available:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-? / --help</code></td>
<td>This option causes the program to display an overview of all command line options.</td>
</tr>
<tr>
<td><code>--compare=result</code> <code>--m=result</code></td>
<td>This option allows you to compare the results of a run with another result. You must specify the name of an existing reference result. Option <code>--run</code> should be used together with this option.</td>
</tr>
<tr>
<td><code>--continuous</code> <code>--c</code></td>
<td>This option allows you to run the analysis in continuous trace mode. Without this option, the default is one shot mode.</td>
</tr>
<tr>
<td><code>--core=core-nr</code></td>
<td>This option allows you to specify the core index number. Without this option, the default is core 0.</td>
</tr>
<tr>
<td><code>--flash</code> <code>--f</code></td>
<td>This option flashes the new ELF file if it differs from the loaded ELF file.</td>
</tr>
<tr>
<td><code>--frequency=frequency</code></td>
<td>This option allows you to specify the core frequency in Mhz. Without this option, the previously measured frequency is used.</td>
</tr>
<tr>
<td><code>--jtag=speed</code> <code>--s=speed</code></td>
<td>This option allows you to set the JTAG speed in MHz.</td>
</tr>
<tr>
<td><code>--memorytype=type</code> <code>--t=type</code></td>
<td>This option allows you to specify the trace memory type. <code>type</code> can be TCM, XTM or TRAM.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
</tbody>
</table>
|----------------------|---------------------------------------------------------------------------------------------------------------------------------------------
| --periodclocks= **clocks** | This option allows you to specify the duration of the trace in number of CPU **clocks**.                                                   |
| --periodcount= **count**  | This option allows you to set the number of periods that the analysis should run.                                                            |
| --periodseconds= **seconds** | This option allows you to specify the duration of the trace in **seconds**.                                                                  |
| --run= **analysis** -r **analysis** | This option allows you to run an existing analysis.                                                                                       |
| --server= **hostname** -s **hostname** | This option allows you to specify the device server name. If you omit this option, the default is **localhost**.                           |
| --tilerange= **from-to** -x **from-to** | This option allows you to specify the tile memory range for the TCM memory type.                                                            |
| --version -v          | This option shows the program version header.                                                                                              |

**To run an existing analysis**

Use the following syntax to run an existing analysis from the command line:

```
EmbProfCmd --run=**analysis** **project.EmbProf**
```

where, **project.EmbProf** refers to an existing Embedded Profiler project file.

**To run and compare an existing analysis**

Use the following syntax to run an existing analysis and compare the results with a previous result from the command line:

```
EmbProfCmd --run=**analysis** --compare=**result** **project.EmbProf**
```

where, **project.EmbProf** refers to an existing Embedded Profiler project file.

**5.2.1. Command Line Tutorial**

In this section we use tutorial **demo_dspr_tc39** with the delivered **demo_dspr_tc39.EmbProf** to illustrate the use of the command line options of the Embedded Profiler.

**Prepare command line**

Before you run the Embedded Profiler from the command line, follow these steps to configure the Windows command prompt.

1. Start the Windows Command Prompt and go to the workspace directory containing the tutorial **demo_dspr_tc39**.
2. Add the executable directory of the Embedded Profiler to the environment variable PATH. The executable directory is the profiler directory in the installation directory. Substitute version with the correct version number.

```
set PATH=%PATH%;"C:\Program Files\TASKING\prof version\profiler"
```

**Command line examples**

1. To run a performance analysis on demo_dspr_tc39 using one shot trace mode, enter:

```
EmbProfCmd --run=PerfAnalysis-1 demo_dspr_tc39.EmbProf
```

The results are exported to the CSV files demo_dspr_tc39_functions.csv and demo_dspr_tc39_instructions.csv. You can inspect these files with any text editor. The first line in a CSV file shows the columns that are used.

Note that the command line invocation does not add a new result entry to the demo_dspr_tc39.EmbProf file.

2. To run a performance analysis on demo_dspr_tc39 using one shot trace mode and compare the results with original, enter:

```
EmbProfCmd --run=PerfAnalysis-1 --compare=original demo_dspr_tc39.EmbProf
```

The results of the comparison are exported to the CSV file demo_dspr_tc39_diff_functions.csv. If all value fields are zero, this indicates that the results are identical.
3. To run a performance analysis on demo_dspr_tc39 using one shot trace mode and compare the results with fixed, enter:

   EmbProfCmd --run=PerfAnalysis-1 --compare=fixed demo_dspr_tc39.EmbProf

   The results of the comparison are exported to the CSV file demo_dspr_tc39_diff_functions.csv. Fields that contain zeros indicate no change. Fields with negative values indicate an improvement, fields with positive values indicate worse performance. In this example the comparison is worse, because we compare the original result (non-fixed sources) with a version where the sources have been fixed. Normally, you compare your results with a previous result.

4. To run an analysis using continuous trace mode use option --continuous. Be aware that this mode requires that the application ends and does not contain endless while loops. Otherwise an analysis run will not end.

   EmbProfCmd --run=PerfAnalysis-1 --compare=fixed --continuous demo_dspr_tc39.EmbProf

5. To run an analysis on a specific core, use option --core=core-nr. For the TC39xB derivative your can use the values 0 to 5. Be aware that a core needs to be enabled in the startup code of the application. Otherwise the analysis run will not terminate.

   EmbProfCmd --run=PerfAnalysis-1 --compare=fixed --continuous --core=0 demo_dspr_tc39.EmbProf

6. To specify a remote host to connect to the target, use option --server=hostname. The default, if you do not specify this option, is localhost.

   EmbProfCmd --run=PerfAnalysis-1 --compare=fixed --continuous --core=0 --server=myservername demo_dspr_tc39.EmbProf

5.3. What to Do if Your Application Does not Start on a Board?

When you profile an application and you encounter the error message:

Trace error: cannot find code at address address
Do you want to continue the run?

it might be the case that the application does not include a valid Boot Mode Header 0 (BMHD0) configuration, or that the start address in the Boot Mode Header on the target does not match the start address of the application. In order to fix this you need to initialize a Boot Mode Header for your target. But be careful, you need to know what you are doing, because wrong use of the Boot Mode Headers might brick the device. Therefore, we advice you to first read chapter 4 TC29x BootROM Content of the AURIX™ TC29x B-Step User's Manual, or similar chapter in the User's Manual for other devices. Also read sections 7.9.13 Boot Mode Headers, and section 9.7.1. Boot Mode Headers in the TriCore User Guide.
To initialize the Boot Mode Header using Eclipse in the TASKING VX-toolset for TriCore:

1. From the **Project** menu, select **Properties for » C/C++ Build » Memory**, and open the **Boot Mode Headers** tab.

2. In **Boot Mode Header 0**, from the **Boot Mode Header configuration**, select **Generate Boot Mode Header**.

3. Leave the other default settings untouched and select **OK**.

   This will initialize the Boot Mode Header to allow for stand-alone execution of the target.
Chapter 6. Reference

Every analysis result shows a number of tabs with information. What information is shown depends on the type of the analysis: performance analysis, memory access analysis or function-level analysis.

Furthermore there is a Settings dialog where you can specify values that influence the way information is shown in the analysis results. Also the Analysis Scope page of the New Analysis wizard is described in more detail.

This chapter contains a description of the Settings dialog and contains an overview of all the fields and columns in an analysis result.

6.1. Settings Dialog

In the Settings dialog you can specify values that influence the way information is shown in the analysis results.

To open the Settings dialog

1. From the Project menu, select Settings.

   *The Settings dialog appears.*

2. Change the values and/or colors and click OK.

   *All results will be updated to reflect the new thresholds.*
When you click the **Defaults** button all the values of the Settings dialog are reset to their initial values. By clicking on a color a color selector pops up where you can change the color.

When you run a Performance analysis, the value of **Threshold for average stalls per clock** determines when the **Average stalls per clock** value is marked red.

The **Threshold factor for memory access** is used to calculate the threshold for memory access in a Memory analysis:

\[
\text{Threshold factor for memory access} \times \text{Local Data Scratch Pad RAM accesses} = \text{Threshold for memory access}
\]

This means, for example, when Local Data Scratch Pad RAM accesses is 32000, and the value of DSPR2 of 31950 will be marked red because it is higher than \(0.05\times32000=1600\). Also other memory accesses that are higher than 1600 will be marked red.

In the **Graph colors** group you can change the colors of the graph lines.

When you run a DMA Load analysis, the settings in the **Load colors** group are visible on the DMA Load tab in the Load column. The load threshold values determine when the cells on the DMA Load tab in the Load column are changing their colors.

## 6.2. Analysis Scope Page

On the Analysis Scope page in the New Analysis wizard you can specify the application functions and address ranges you want to trace. If you do not specify anything, the whole application will be analyzed (the full address range).
The Analysis Scope page consists of two sides. On the left side you can select the Application functions or specify a (user defined) address region. The right side reflects the result of the selection and shows the resulting trace ranges. The bar in front of the right side indicates if the function or region is part of the trace (green color) or excluded from the trace (white color).

Hover the mouse over a field or button or area to get additional help information.

A function region is defined by a selected function from the ELF file, the actual address range is read from the ELF file upon start of the trace run. Their ranges cannot overlap.

A user region has a fixed address range defined by the user. Their ranges cannot overlap.

You can mark each function or user region on the right side as exclude (white box) or trace (blue box for user region, green box for function). A gray area means that no selection has been made.

Optionally, you can extend each function or user region to encompass the address range extending from its end to the next begin or end border of any other function or user range.

The resulting trace ranges are based on all (possibly extended) function regions and user regions, where for overlapping address ranges the function region exclude/trace status takes precedence over the user region. The ranges are shown as numbered dark green bars in front of the right side of the dialog, A dialog finish is only possible if the number of ranges does not exceed the hardware capabilities.
To filter the list of Application functions

Either

• enter part of the function name, or
• enter an address range, or
• click on a function or range on the right side and click **Filter** to get an initial filter which you can change afterwards.

*The Application functions list will show the filtered list.*

To add a function to the trace range

1. Select a function from the list of **Application functions**. You can use a filter first to make your selection easier
2. Click **Add**.

*The function will be part of the trace range.*

To add a user region to the trace range

1. In the **Address region** fields, enter the begin and end address of the region you want to be part of the trace. You can first click on a function or region on the right side and click **Set**, to have an initial range you can edit.
2. Click **Add**.

*The region will be part of the trace range.*

To extend a function or user region

1. Hover the mouse over an area on the right side to see which part can be extended.
2. Either double-click on the extension part, or select it first and then hit the spacebar to toggle the extension.

For example, see the figure above. If you double-click on the part between `main` and `printf`, function `main` will be extended and a (+) appears next to the function.
To include/exclude a function or user region from the trace range

1. Hover the mouse over an area on the right side to see which part can be excluded. Blue and green parts are part of the trace.

2. Either double-click on the function or region, or select it first and then hit the spacebar to toggle the exclude/trace.

   *The box/area turns white to indicate that it is no longer part of the trace.*

Note that you can use the **Invert all** button to toggle all exclude/trace parts.

**Final address range**

Note that the final address range of a function is only retrieved on trace start. Theoretically functions can swap order or move in or out of user ranges since the last time you have seen the wizard Analysis scope page. On the Run Analysis dialog the configured trace scope is shown without function addresses, only when the trace is started the ELF memory layout is retrieved and used to configure the trace ranges for that run.
6.3. Run DMA Load Analysis Dialog

In the tutorial in section Section 3.2, Analyze Project in TASKING Embedded Profiler is explained how to run an analysis. When you run a DMA Load analysis, the Run DMA Load Analysis dialog contains some extra fields which are described here.
In the Analysis Configuration you can specify a **Period size** in **CPU cycles** or **Seconds** and the **Number of periods** that is suitable for your application. After each period information is collected. If you select **Seconds** you can specify the **User specified** core **Frequency** in MHz. **Measured** means that the previously measured frequency is used.

Additionally you can choose to generate **Timeline** data. See Section 6.15, **Timeline Tab**.

### 6.4. Progress Dialog

The Progress dialog shows what is going on after you started a run. When tracing the Process dialog shows the number of clocks being traced.
The button can show the following texts:

- **Cancel**
  
  No trace data has been received yet. Clicking the **Cancel** button will end tracing without saving any result.

- **Stop**
  
  Trace data is being received. Clicking the **Stop** button will stop tracing but continue analyzing all data already traced by the hardware. This is the only way to halt a continuous trace for performance/flow/function/memory analysis if the embedded program does not end its main function or halt the processor. A DMA analysis could also end if a specific number of periods was requested.

- **Abort**
  
  Tracing has been halted but trace data received is still being analyzed. Clicking the **Abort** button will ask for confirmation to discard any traced data not analyzed yet, a result will be saved with only the analyzed data.

### 6.5. Summary Tab

On the Summary tab the following information is available for the different analysis types

**Performance analysis**

- Configuration
- Info
- Performance Hotspots Clocks
• Source Coverage
• ICache Miss Count
• DCache Miss Count

**Flow analysis**
• Configuration
• Info
• Performance Hotspots Clocks
• Source Coverage

**Memory access analysis**
• Configuration
• Info
• Performance Hotspots Clocks
• DCache Miss Count
• Memory Access
• Memory Conflicts

**Function-level analysis**
• Configuration
• Info
• Performance Hotspots Clocks

**DMA Load analysis**
• Configuration
• Info
• DMA Load
• DMA Channel Load
• DMA Load Per Period (only visible if there are multiple periods)
• DMA Channel Load Per Period (only visible if there are multiple periods)
6.5.1. Configuration

The **Configuration** part of the Summary tab contains the following information.

<table>
<thead>
<tr>
<th><strong>Configuration</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>TC39xED</td>
</tr>
<tr>
<td>Trace settings</td>
<td>Core=[0], Memory=TCM (tile 0-7), Mode=Reset-OneShot</td>
</tr>
<tr>
<td>Trace periods</td>
<td>60x1,000=60,000 cycles</td>
</tr>
</tbody>
</table>

Items that are marked red have additional information, hover the mouse over a value to see this additional information.

6.5.2. Info

The **Info** part of the Summary tab contains the following information.

<table>
<thead>
<tr>
<th><strong>Info</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Started at</td>
<td>May 26, 2021 6:48:34 PM</td>
</tr>
<tr>
<td>Consumed time</td>
<td>24 seconds</td>
</tr>
<tr>
<td>Clock frequencies (MHz):</td>
<td>CPU0= 100, CPU1=100, CPU2=100, CPU3=100, CPU4=100, CPU5=100, SRL=100, SPB=30, BBS=100</td>
</tr>
<tr>
<td>CPU data/program cache:</td>
<td>DCACHE0=0, PCACHE0=0, DCACHE1=0, PCACHE1=0, DCACHE2=0, PCACHE2=0, DCACHE1=0, PCACHE3=0, DCACHE3=0, DCACHE4=0, PCACHE4=0, DCACHE5=0, PCACHE5=0</td>
</tr>
<tr>
<td>CPU clock count</td>
<td>170,700</td>
</tr>
<tr>
<td>DCache misses</td>
<td>33</td>
</tr>
<tr>
<td>Local Data Scratch Pad RAM accesses</td>
<td>600</td>
</tr>
<tr>
<td>CPU0 Data Scratch Pad RAM accesses</td>
<td>32,961</td>
</tr>
<tr>
<td>PFLASH0 accesses</td>
<td>99</td>
</tr>
<tr>
<td>Local Memory Unit accesses</td>
<td>554</td>
</tr>
<tr>
<td>SFR accesses</td>
<td>2,546</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Information</strong></th>
<th><strong>Description</strong></th>
<th><strong>Perf</strong></th>
<th><strong>Flow</strong></th>
<th><strong>Mem</strong></th>
<th><strong>Func</strong></th>
<th><strong>DMA Load</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Started at</td>
<td>The date and time the analysis was run</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Consumed time</td>
<td>The time (in seconds) it took for the analysis to complete</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Information</td>
<td>Description</td>
<td>Perf</td>
<td>Flow</td>
<td>Mem</td>
<td>Func</td>
<td>DMA Load</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>------</td>
<td>------</td>
<td>-----</td>
<td>------</td>
<td>----------</td>
</tr>
<tr>
<td>Clock frequencies (MHz)</td>
<td>The values of several clock frequencies. The values are read at the start of the analysis before any reset. If the CPU was reset or halted at analysis start, the clock frequencies are not measured.</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CPU data/program cache</td>
<td>The CPU 0, 1, 2, ... data cache (DCache) and program cache (PCache) settings. DCACHE0=1 means CPU0.DCACHE is enabled, PCACHE1=0 means CPU1.PCACHE is disabled. The values are read at the start of the analysis before any reset.</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CPU clock count</td>
<td>The number of CPU clock cycles on the board it took to run the analysis</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Trace periods</td>
<td>The actual number of trace periods</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Period duration</td>
<td>The trace period size in cycles</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Stalls</td>
<td>The number of clock cycles the CPU stalls on branch misses, ICache misses and/or DCache misses</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average stalls per clock</td>
<td>The average of stalls / CPU clock count</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICache misses</td>
<td>The number of failed attempts to read or write instructions from the instruction cache (ICache)</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DCache misses</td>
<td>The number of failed attempts to read or write data from the data cache (DCache)</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total coverage %</td>
<td>Application coverage percentage, which indicates the number of executed instructions of the total number of instructions.</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Source coverage %</td>
<td>Source coverage percentage, which indicates the number of executed instructions of the total number of instructions belonging to source files.</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Local Data Scratch Pad RAM accesses</td>
<td>The number of read or write accesses to Data Scratch Pad RAM, where the core could not be determined</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
### Information

<table>
<thead>
<tr>
<th>Description</th>
<th>Perf</th>
<th>Flow</th>
<th>Mem</th>
<th>Func</th>
<th>DMA Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU(x) Data Scratch Pad RAM accesses</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The number of read or write accesses to Data Scratch Pad RAM (x), where (x) can be 0 .. 5</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PFLASH(x) accesses</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The number of read or write accesses to flash memory</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>External Bus Unit memory accesses</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The number of read or write accesses to the EBU</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Local Memory Unit accesses</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The number of read or write accesses to the LMU</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program Memory Unit accesses</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The number of read or write accesses to the PMU</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SFR accesses</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The number of read or write accesses to Special Function registers</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Items that are marked red are high values that may be improved. Hover the mouse over a value to see additional information. You can influence the thresholds in the Settings dialog. See Section 6.1, Settings Dialog.

### 6.5.3. Performance Hotspots Clocks

The Performance Hotspots Clocks part of the Summary tab shows the functions with the highest clock count. This chart is available for all analysis types. As you can see in the following example, most of the time is spent in the functions and main, _start and _c_init_entry.
If you double-click on a function, the Source tab opens at the selected function.

### 6.5.4. Source Coverage

The **Source Coverage** part of the Summary tab shows the functions with source and with the lowest coverage percentage. This chart is only available for performance analyses and flow analyses.

![Source Coverage Chart](image)

If you double-click on a function, the Source tab opens at the selected function.

### 6.5.5. ICache Miss Count

The **ICache Miss Count** part of the Summary tab shows the functions with the highest number of instruction cache (ICache) misses. This chart is available for performance analyses only.

![ICache Miss Count Chart](image)

### 6.5.6. DCache Miss Count

The **DCache Miss Count** part of the Summary tab shows the functions with the highest number of data cache (DCache) misses. This chart is available for performance analyses and memory access analyses.
6.5.7. Memory Access

The Memory Access part of the Summary tab shows the functions with the highest number of data accesses to memory. This chart is available for memory access analyses only.

Hover the mouse over a value to see additional information.
6.5.8. Memory Conflicts

The Memory Conflicts part of the summary tab shows the total number of access conflicts where two variables from different cores access the same memory at the same time. This is called concurrent access. The demo_concurrent tutorial delivered with the product demonstrates this problem. This chart is available for memory access analyses only.

The global variable name that accesses the memory, the core from which the conflicting access originated and the type of access read (R) or write (W) is listed for the two conflicting variables.

Hover the mouse over a value to see additional information.

6.5.9. DMA

The DMA part of the Summary tab shows the total DMA load percentage or activations as an average of both engines. This chart is available for DMA load analyses only.
Hover the mouse over a bar to see additional information.

If you double-click on a bar, the DMA Load tab opens.

**6.5.10. DMA Channel**

The **DMA Channel** part of the Summary tab shows the DMA load percentage or activations for the top most DMA channels. This chart is available for DMA load analyses only.

Hover the mouse over a bar to see additional information.

If you double-click on a bar, the DMA Load tab opens at the selected channel.
6.5.11. DMA Per Period

The **DMA Per Period** part of the Summary tab shows the total DMA load average or activations on both engines per period. This chart is available for DMA load analyses only.

Hover the mouse over a part of the line to see the load percentage.

If you double-click on a line, the DMA Load tab opens at the selected channel.

6.5.12. DMA Channel Per Period

The **DMA Channel Per Period** part of the Summary tab shows the DMA load percentage or activations for the top most DMA channel per period. This chart is available for DMA load analyses only.

Hover the mouse over a part of the line to see the load percentage.

You can click on a channel line to bring that channel forward. If you double-click on a line, the DMA Load tab opens at the selected channel.
6.6. Functions Tab

The Functions tab shows a list with all the measured functions. This tab is available in all analysis types. The performance analysis contains the most columns. Click on a column to sort the list according to the information in that column. If you double-click on a function, the Source tab opens at the selected function. If no source lines can be displayed, the Disassembly tab opens. Hover the mouse over a column to see additional information.

The Functions tab contains the following information:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Perf</th>
<th>Flow</th>
<th>Mem</th>
<th>Func</th>
</tr>
</thead>
</table>
| Function | The name of the measured function. Information about static functions is displayed where available, depending on the information in the ELF file:  
  • static functions with a known source file name. For example: _start(..\cstart.c)  
  • static functions without a known source file name having one implementation in the ELF file. For example: .cocofun_3  
  • static functions without a known source file name having multiple implementations in the ELF file. For example: .cocofun_1(0x80002c1e) | ✓    | ✓    | ✓   | ✓    |
<p>| Source   | The relative path to the source file as stored in the application ELF file | ✓    | ✓    | ✓   | ✓    |
| Address  | The address of the function in the application ELF file                     | ✓    | ✓    | ✓   | ✓    |
| Coverage %| The function coverage as a percentage of total function instructions. Note that the tool tip in this column shows the number of covered instructions versus the number of instructions. | ✓    | ✓    | ✓   | ✓    |
| Covered  | The function is called or not.                                              |      |      |     | ✓    |
| Clocks   | The total number of CPU clocks spent in the function                        | ✓    | ✓    | ✓   | ✓    |
| % Of Total Time | The application execution time spent in the function as a percentage of the total application execution time | ✓    | ✓    | ✓   | ✓    |</p>
<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Perf</th>
<th>Flow</th>
<th>Mem</th>
<th>Func</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clocks With Children</td>
<td>The total number of CPU clocks spent in the function and call tree descendent</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Entries</td>
<td>The total number of times the function is called</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Avg. Clocks/Entry</td>
<td>The average number of CPU clocks spent in a function per function entry</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Max Clocks/Entry</td>
<td>The highest number of CPU clocks spent in a function per function entry</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Min Clocks/Entry</td>
<td>The lowest number of CPU clocks spent in a function per function entry</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Jitter/Entry</td>
<td>The difference between the highest and lowest number of CPU clocks spent in a function. This is the difference of the previous two columns.</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Branch Misses</td>
<td>The total number of branch misses</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICache Misses</td>
<td>The total number of instruction cache misses</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DCache Misses</td>
<td>The total number of data cache misses</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stalls</td>
<td>The total number of stalls due to memory access delays or pipeline hazards</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 6.7. Source Lines Tab

The Source Lines tab shows a list with all the source lines of the measured functions where branch misses, instruction cache misses, data cache misses and/or stalls appear. This tab is available for performance analyses only. Click on a column to sort the list according to the information in that column. Hover the mouse over a column to see additional information.

If you double-click on a row, the Source tab opens at the selected source line.

The Source Lines tab contains the following information:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>The source line number, function name and relative path to the source file where the problem occurred</td>
</tr>
<tr>
<td>Clocks</td>
<td>The total number of CPU clocks spent on the source line</td>
</tr>
<tr>
<td>Coverage %</td>
<td>The source line coverage as a percentage of total source line instructions. Note that the tool tip in this column shows the number of covered instructions versus the number of instructions.</td>
</tr>
<tr>
<td>Branch Misses</td>
<td>The total number of branch misses</td>
</tr>
<tr>
<td>ICache Misses</td>
<td>The total number of instruction cache misses</td>
</tr>
</tbody>
</table>
### 6.8. Instructions Tab

The Instructions tab shows a list with all the instructions of the measured functions where branch misses, instruction cache misses, data cache misses and/or stalls appear. This tab is available for performance analyses only. Click on a column to sort the list according to the information in that column. Hover the mouse over a column to see additional information.

If you double-click on a row, the Disassembly tab opens at the selected instruction.

The Instructions tab contains the following information:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>The instruction address and function name where the problem occurred</td>
</tr>
<tr>
<td>Clocks</td>
<td>The total number of CPU clocks spent on the instruction</td>
</tr>
<tr>
<td>Branch Misses</td>
<td>The total number of branch misses</td>
</tr>
<tr>
<td>ICache Misses</td>
<td>The total number of instruction cache misses</td>
</tr>
<tr>
<td>DCache Misses</td>
<td>The total number of data cache misses</td>
</tr>
<tr>
<td>Stalls</td>
<td>The total number of stalls due to memory access delays or pipeline hazards</td>
</tr>
</tbody>
</table>

### 6.9. Memory Access Tab

The Memory Access tab shows the functions and variables and their data accesses to memory. This tab is available for memory access analyses only.

Hover the mouse over a value to see additional information.
The Memory Access tab contains the following information:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>The name of the function that contains the global variable. Information about static functions is displayed where available, depending on the information in the ELF file:</td>
</tr>
<tr>
<td></td>
<td>• static functions with a known source file name. For example: <code>_start(../cstart.c)</code></td>
</tr>
<tr>
<td></td>
<td>• static functions without a known source file name having one implementation in the ELF file. For example: <code>.cocofun_3</code></td>
</tr>
<tr>
<td></td>
<td>• static functions without a known source file name having multiple implementations in the ELF file. For example: <code>.cocofun_1(0x80002c1e)</code></td>
</tr>
<tr>
<td>Variable</td>
<td>The name of the global variable, if the address is associated with a variable, otherwise &quot;(unidentified)&quot; is shown. This may be because of function stack area, csa area, peripheral SFR area or another unknown area. Another possibility is that it is a local static variable which is not shown. In order to have static variables listed in the profiling analysis results, when building your application specify the assembler option <code>-emit-locals=symbols</code>, or in Eclipse select Project » Properties for » C/C++ Build » Settings » Tool Settings » Assembler » Symbols » Emit local non-EQU symbols.</td>
</tr>
<tr>
<td>Region</td>
<td>The name of the memory</td>
</tr>
</tbody>
</table>
6.10. Memory Conflicts Tab

The Memory Conflicts tab shows the conflicts where two variables from different cores access the same memory at the same time. This is called concurrent access. The demo_concurrent tutorial delivered with the product demonstrates this problem. This tab is available for memory access analyses only.

Hover the mouse over a value to see additional information.

The Memory Conflicts tab contains the following information:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function-1 / Function-2</td>
<td>The name of the first/second function that contains the global variable.</td>
</tr>
<tr>
<td>Variable-1 / Variable-2</td>
<td>The name of the first/second global variable</td>
</tr>
<tr>
<td>Region-1 / Region-2</td>
<td>The name of the first/second memory</td>
</tr>
<tr>
<td>Access-1 / Access-2</td>
<td>The type of access read (R) or write (W) for the first/second variable</td>
</tr>
<tr>
<td>Origin-1 / Origin-2</td>
<td>The core of the first/second variable from which the conflicting access originated</td>
</tr>
<tr>
<td>Count</td>
<td>The number of access conflicts</td>
</tr>
</tbody>
</table>

6.11. Source Tab

The Source tab shows the source code for the selected function. For performance analyses only, trace data is also present grouped by source line. For performance and flow analyses only, coverage data is also present.
The columns are the same as explained in Section 6.7, Source Lines Tab. Red values indicate a miss or a stall. Light red fields in the coverage columns indicate uncovered lines or instructions. Hover the mouse over a value to see additional information.

With the Browse button you can open another source file.

When you enable Show disassembly, the disassembly will be intermixed with the source lines.

### 6.12. Disassembly Tab

The Disassembly tab shows the instructions for the selected function. For performance analyses only, trace data is also present grouped by instruction address. For performance and flow analyses only, coverage data is also present.
The Disassembly tab contains the following information:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>The instruction address and function name where the problem occurred</td>
</tr>
<tr>
<td>Clocks</td>
<td>The total number of CPU clocks spent on the instruction</td>
</tr>
<tr>
<td>Covered</td>
<td>Instruction is executed or not</td>
</tr>
<tr>
<td>Branch Misses</td>
<td>The total number of branch misses</td>
</tr>
<tr>
<td>ICache Misses</td>
<td>The total number of instruction cache misses</td>
</tr>
<tr>
<td>DCache Misses</td>
<td>The total number of data cache misses</td>
</tr>
<tr>
<td>Stalls</td>
<td>The total number of stalls due to memory access delays or pipeline hazards</td>
</tr>
</tbody>
</table>

Red values indicate a miss or a stall. Light red fields in the Covered column indicate uncovered lines or instructions. Hover the mouse over a value to see additional information.

If you double-click on a row, the Raw Trace Data tab, if present, opens at the selected address.

Note that due to hardware constraints, a miss or a stall cannot always be linked to the exact assembly instruction.
6.13. Raw Trace Data Tab

The Raw Trace Data tab is included as a service to advanced users who are familiar with the Infineon Multi-Core Debug Solution and who want to examine program flow. Raw trace data is useful, for example, to see why stall cycles are assigned to instructions that do not access memory. This tab is available for all analysis types, but only when you enable Save Raw trace data in the Run Analysis dialog.

Hover the mouse over a value to see additional information.

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>The sequence number for every raw trace operation.</td>
</tr>
<tr>
<td>TimeStamp</td>
<td>The amount of clocks since the first traced IP record. 0 is start of analysis.</td>
</tr>
<tr>
<td>Ticks</td>
<td>The MCDS clock Ticks between trace messages. Please note that one Tick is equal to two CPU cycles.</td>
</tr>
<tr>
<td>OPoint</td>
<td>Displays the Observation Point of the trace data. The observation point is the physical data acquisition point inside the SoC (System-on-Chip). For example the CPU0, CPU1, SRI bus, and so on.</td>
</tr>
<tr>
<td>Origin</td>
<td>The origin of the activity. In most cases this is the same as OPoint.</td>
</tr>
</tbody>
</table>
**Example how to use raw trace data for analysis**

1. Import the `demo_concurrent` example.

2. Run a **One shot mode** performance analysis with Save additional data **Raw trace** enabled.

3. Open the **Instructions** tab and sort the **Stalls** column.

   Notice that near the top is a `loop a4, 0x800022be` instruction with a value of 123 stalls at address `0x800022c2`.

4. In the **Raw Trace Data** tab, enter the address `0x800022c2` in the **Search** box and search for the first occurrence.

   When searching through the raw trace data, it shows that the previous executed instruction is at address `0x800022c0`.

---

**TASKING Embedded Profiler User Guide**

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operation</strong></td>
<td>The operation being executed but not on the level of assembler mnemonics for program trace. It displays a more abstract type of the operation. For example, IP_CALL, IP_RET, MEMORY_READ, MEMORY_WRITE or one of the internal performance counters COUNTER_x.</td>
</tr>
<tr>
<td><strong>Data</strong></td>
<td>The data written or read.</td>
</tr>
<tr>
<td><strong>Address</strong></td>
<td>The pointer of the instruction (IP) which is being executed. If the Operation column displays an R/W Operation, the Address column displays the address where data is read or written to.</td>
</tr>
<tr>
<td><strong>Function/Variable</strong></td>
<td>The function or variable at the given address.</td>
</tr>
<tr>
<td><strong>Disassembly</strong></td>
<td>The disassembly at the given address.</td>
</tr>
</tbody>
</table>
5. Double-click on the address and the view will switch to the Disassembly tab at the specified address, in this case \texttt{st.w \; [a12+]0x4, \_d1}. 

Reference
6.14. DMA Load Tab

The DMA Load tab shows a table with all the measured clocks and load percentages for a DMA Load analysis.

Click on a column to sort the table according to the information in that column. For sub sorting keep the Ctrl key pressed while clicking on another column.

Uncheck the Details check box to hide the detail information. Uncheck the Totals check box to hide the totals (at least one of the 2 check boxes must be enabled, this will be forced by the program).

The totals over channels average the clocks and the load over both DMA engines. Which means a period with channel 1 50% busy on engine 1 and 25% on engine 2 shows as 75% total load for the channel, and as 37.5% load for DMA. Background info: a channel can only be busy on one engine at the time, while the DMA has a total capacity of two engines. Meaning that a single permanently busy channel uses 50% of the DMA.

Hover the mouse over a column to see additional information.

The DMA Load tab contains the following information:

<table>
<thead>
<tr>
<th>Information</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The measured channel</td>
</tr>
<tr>
<td>Period</td>
<td>The measured period</td>
</tr>
<tr>
<td>Activations</td>
<td>The number of times a channel becomes active in a period</td>
</tr>
<tr>
<td>Clocks</td>
<td>The measured total clocks</td>
</tr>
<tr>
<td>Min Load %</td>
<td>The lowest measured load in percentage</td>
</tr>
<tr>
<td>Max Load %</td>
<td>The highest measured load in percentage</td>
</tr>
<tr>
<td>Load %</td>
<td>The average measured load in percentage</td>
</tr>
<tr>
<td>Load</td>
<td>The average measured load displayed as color in percentage</td>
</tr>
</tbody>
</table>
6.15. Timeline Tab

The Timeline tab shows timing information for a DMA Load analysis. It is only available if you enabled the **Timeline** checkbox in the Run DMA Load Analysis dialog.

Hover all buttons for information. Hover or click the top right question mark for all non-button options. Use the Select and Show buttons to filter on what you want to see.

There are two different sort of lines:

- **group lines** - Show active values on the CPU or DMA. A group line shows as a diamond-shaped line.
- **value lines** - Each line shows the value being active or not on the parent CPU or DMA. A value line shows as an n-shaped light grey active zone.

If you click on a line it becomes blue, meaning that it is the current line. With the green buttons you can jump to events on the current line.

A vertical blue line indicates the position on the timeline. Zoom in to make it more precise.

Depending on the zoom level there is detailed information about an event. Hovering will display time information for the event (or multiple events if events are really happening simultaneously for that CPU/DMA). Ctrl-left-click displays the information in a separate window. If zooming out leads to multiple
consecutive events in a column, only the values are known, any order/timing/frequency information is lost (group line and value line both show as closed medium-gray box). Hovering will only display the values becoming active at least once in the column.

To measure the distance between events

1. Left-click on the first event you want to be part of the measurement.
   
   *The event becomes blue.*

2. Right-click on the second event you want to be part of the measurement.
   
   *The event becomes green and between red lines the result of the measurement is shown.*