TASKING®

TASKING Embedded Profiler User Guide

MA160-857 (v1.0r3) November 20, 2018

Table of Contents

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, V2.4 [2017-10, Infineon]
- AURIX™ TC38x Appendix, V2.5.1 [2018-04, Infineon]
- AURIX™ TC39x-B Appendix, V2.5.1 [2018-04, 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 Altium.

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.

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 Altium 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, Altium 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.

1. Download the **getcid** program from http://www.tasking.com/support/tlm/download.shtml.

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
   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 Altium 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.

http://www.tasking.com/support/tlm/faq.shtml

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

If your question is not there, please contact your nearest Altium 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 licence 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` .. `lic4.tasking.com` at the TCP ports 8080, 8936 or 80. Make sure that the firewall in your network has transparent access enabled for one of these ports.

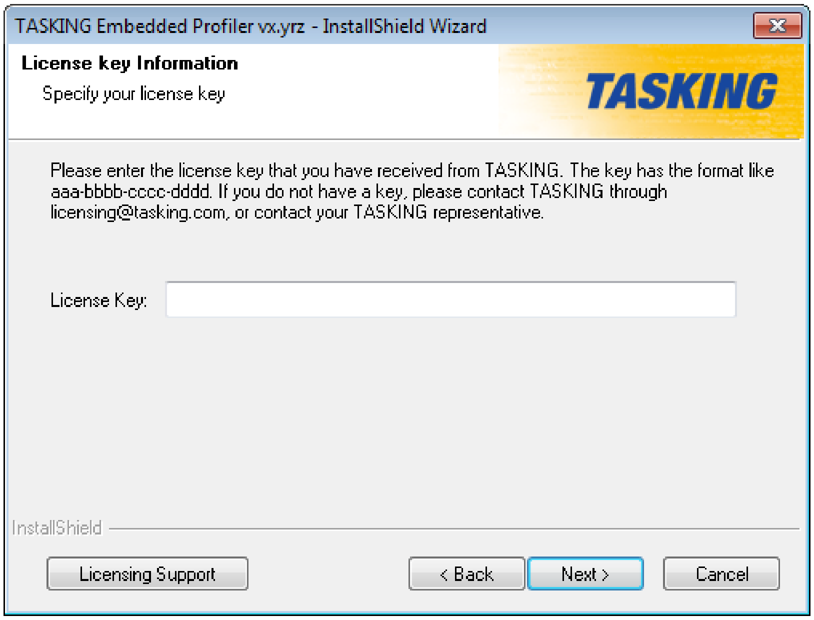
1.2.3.2. Installing Server Based Licenses (Floating or Node-Locked)

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 (product code SW000089).

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

   The installation program asks you to enter the license information.


TASKING Embedded Profiler vx.yrz - InstallShield Wizard

License key Information

Specify your license key

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

License Key:

3. In the **License key** field enter the license key you have received from Altium 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.

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

5. 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.

6. (For local license server only) specify the **Server name** and **Port number** of the local license server.

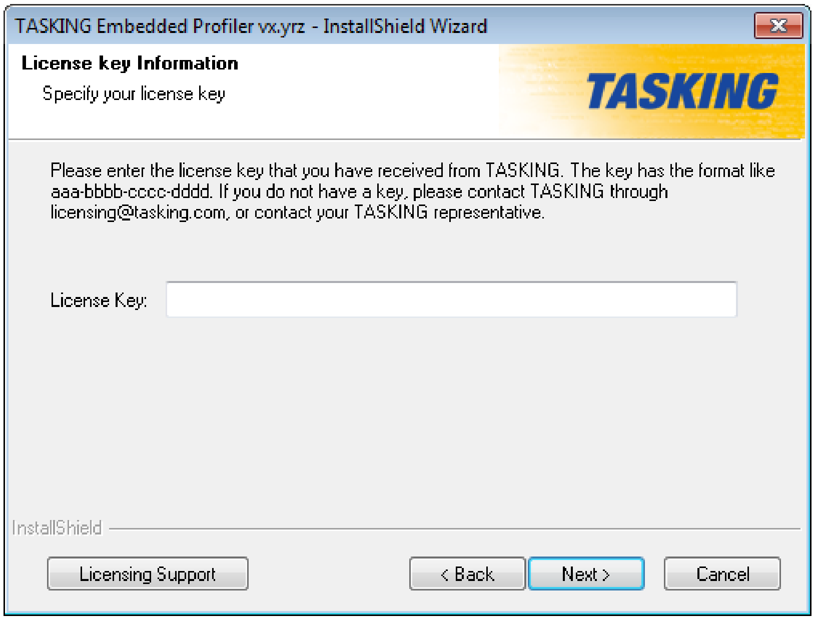
7. Click **Finish** to complete the installation.

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 instruction that appear on your screen.

   The installation program asks you to enter the license information.



TASKING Embedded Profiler vx.yrz - InstallShield Wizard

License key Information

Specify your license key

Please enter the license key that you have received from TASKING. The key has the format like aaa-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 Altium 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 content** field enter the contents of the license file you have received from Altium.

   The license data is stored in the file licfile.txt in the etc directory of the product.
5. Click **Finish** to complete the installation.

Chapter 2. Introduction to the TASKING Embedded Profiler

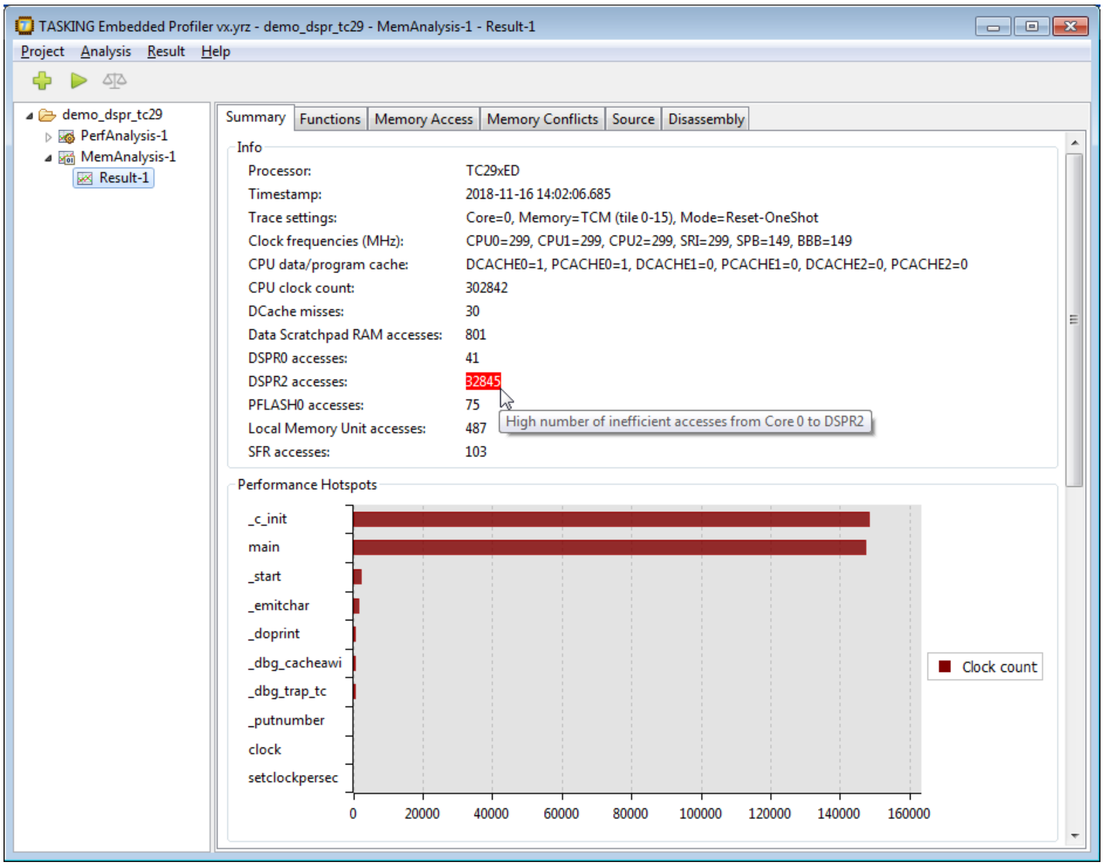
After your application has been verified, thoroughly tested and debugged, and by itself behaves correctly, you may still run into performance and timing issues. Many timing issues can be addressed simply by improving the performance of the applications that caused a missed deadline. Furthermore, by reducing 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.

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.



Processor:	TC29xED
Timestamp:	2018-11-16 14:02:06.685
Trace settings:	Core=0, Memory=TCM (tile 0-15), Mode=Reset-OneShot
Clock frequencies (MHz):	CPU0=299, CPU1=299, CPU2=299, SRI=299, SPB=149, BBB=149
CPU data/program cache:	DCACHE0=1, PCACHE0=1, DCACHE1=0, PCACHE1=0, DCACHE2=0, PCACHE2=0
CPU clock count:	302842
DCache misses:	30
Data Scratchpad RAM accesses:	801
DSPR0 accesses:	41
DSPR2 accesses:	32845
PFLASH0 accesses:	75
Local Memory Unit accesses:	487
SFR accesses:	103

High number of inefficient accesses from Core 0 to DSPR2

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
- Memory access analysis
- Function-level 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 Device Access Server (DAS) v7.0 and Device Access Port (DAP) miniWiggler

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. You can run this type of analysis on the whole application or select specific functions.

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. This is the fastest analysis.

2.1. Emulation Device (ED)

The standard TriCore/AURIX™ processors (production devices) lack debug trace functionality. However, this functionality is very useful when you develop and test your application. Therefore pin compatible Emulation Devices (ED) are available. An Emulation Device has an Emulation Extension Chip (EEC) added to the same silicon, which is accessible through the JTAG or DAP interface. The TASKING Embedded Profiler supports the on-chip trace feature of the Emulation Device. See the processor documentation for detailed information about the device.

Some Production Devices, such as the TC29x, are equipped with a mini-MCDS, which 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. See the processor documentation for detailed information about the device.

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 a detailed naming convention see the AURIX™ Product Naming PDF on the Infineon website.

2.2. Trace Support

The TASKING Embedded Profiler uses the Multi-Core Debug Solution (MCDS) for on-chip trace support. For detailed information about MCDS we recommend that you read the processor documentation belonging to the Emulation Device.

Trace memory

Trace information is stored in a dedicated trace buffer. With an Emulation Device 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. Where 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 mini-MCDS 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 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 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 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. 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.

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.

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 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 TC29xB 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

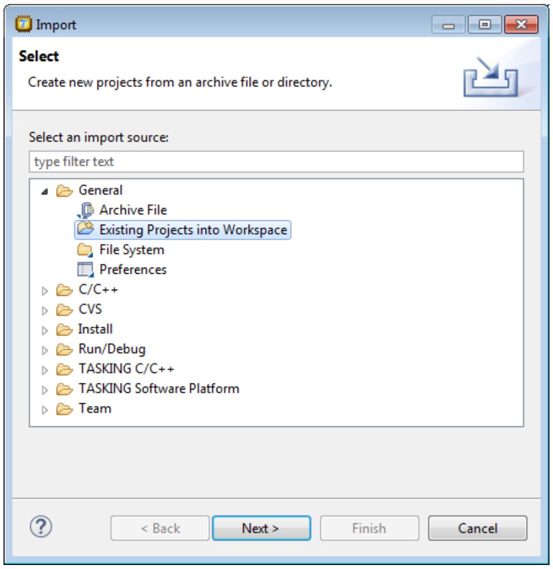
Before you can use the TASKING Embedded Profiler, you must have an application ELF file with debug information and the application must be downloaded onto a target board.

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

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.


Import

Select

Create new projects from an archive file or directory.

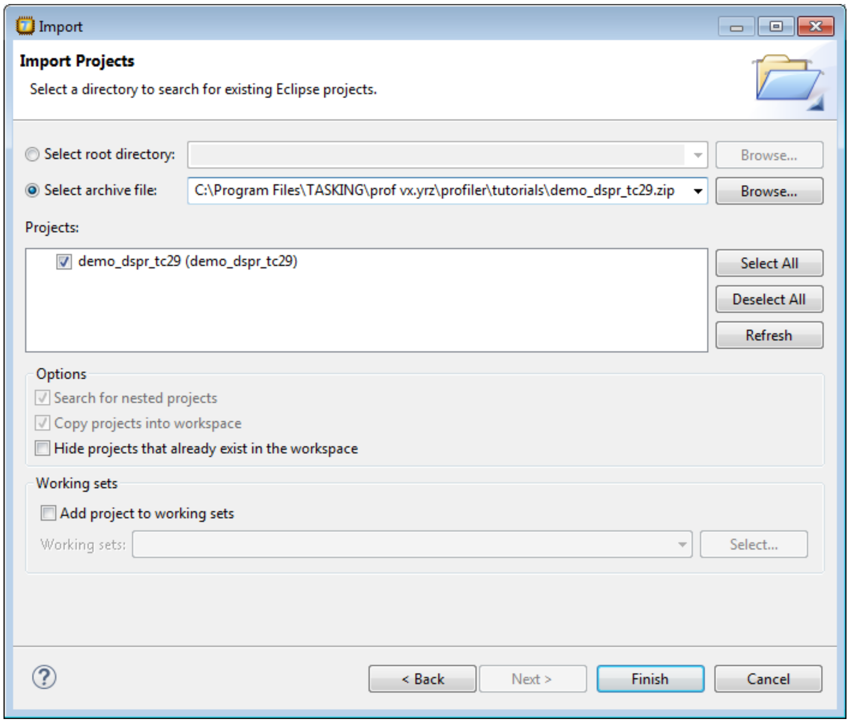
Select an import source:

type filter text

- General
  - Archive File
  - Existing Projects into Workspace
  - File System
  - Preferences
- C/C++
- CVS
- Install
- Run/Debug
- TASKING C/C++
- TASKING Software Platform
- Team

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

   The Import Projects dialog appears.


Import

Import Projects

Select a directory to search for existing Eclipse projects.

○ Select root directory:

● Select archive file: C:\Program Files\TASKING\prof vx.yrz\profiler\tutorials\demo_dspr_tc29.zip

Projects:

- [✓] demo_dspr_tc29 (demo_dspr_tc29)

Options

- [✓] Search for nested projects
- [✓] Copy projects into workspace
- [ ] Hide projects that already exist in the workspace

Working sets

- [ ] Add project to working sets

Working sets:

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

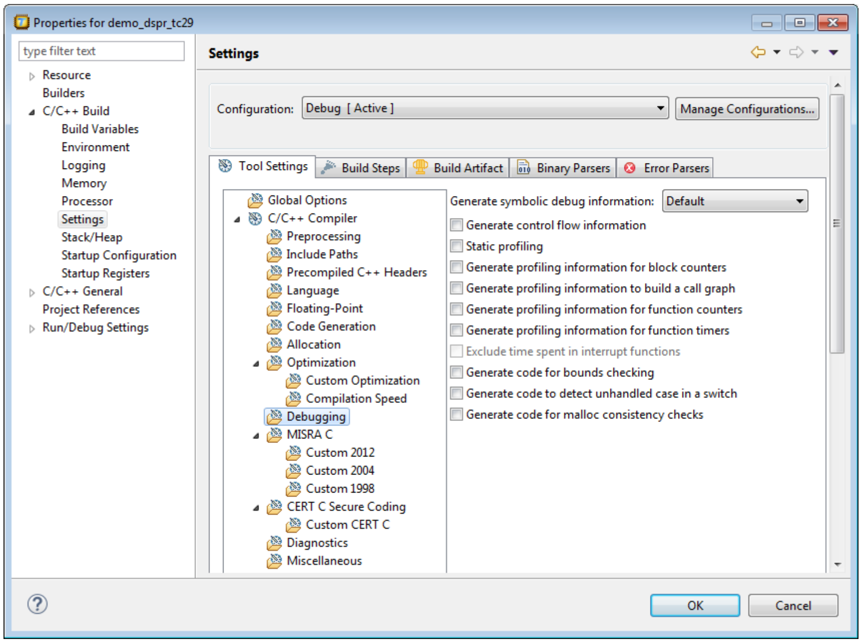
The resulting application ELF file must contain debug information. The demo projects already have debugging enabled by default. So, for the demo projects you can skip this step. For your own project, make sure that debugging is enabled.

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

   The Properties for demo_dspr_tc29 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**.


Properties for demo_dspr_tc29

Settings

Configuration: Debug [Active]

Tool Settings | Build Steps | Build Artifact | Binary Parsers | Error Parsers

Generate symbolic debug information: Default

- [ ] Generate control flow information
- [ ] Static profiling
- [ ] Generate profiling information for block counters
- [ ] Generate profiling information to build a call graph
- [ ] Generate profiling information for function counters
- [ ] Generate profiling information for function timers
- [ ] Exclude time spent in interrupt functions
- [ ] Generate code for bounds checking
- [ ] Generate code to detect unhandled case in a switch
- [ ] Generate code for malloc consistency checks

Build the project

- From the **Project** menu, select **Build demo_dspr_tc29**, or click from the toolbar.

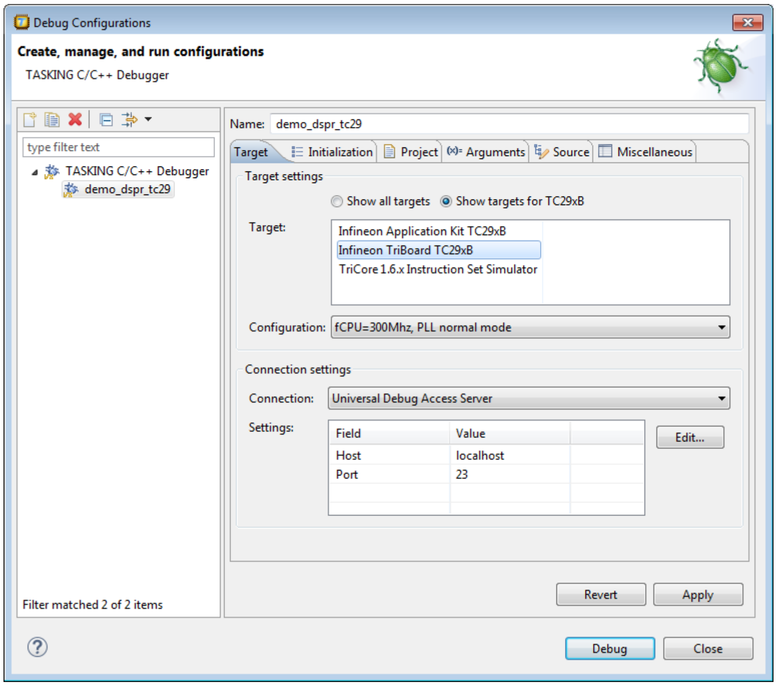
Run the debugger to flash the application onto the target board

1. Connect the Infineon TriBoard TC29xB to your computer. See the documentation that came with the board for more information.

2. From the **Debug** menu, select **Debug** *project*.

   Alternatively you can click the button in the main toolbar.

   Before you can debug a project, you need a Debug launch configuration. Such a configuration, identified by a name, contains all information about the debug project: which debugger is used, which project is used, which binary debug file is used, ... and so forth. So, initially the Debug Configurations dialog appears.


Debug Configurations

Create, manage, and run configurations

TASKING C/C++ Debugger

Name: demo_dspr_tc29

Target settings

○ Show all targets ● Show targets for TC29xB

Target:
- Infineon Application Kit TC29xB
- Infineon TriBoard TC29xB
- TriCore 1.6.x Instruction Set Simulator

Configuration: fCPU=300Mhz, PLL normal mode

Connection settings

Connection: Universal Debug Access Server

Settings:

Field	Value
Host	localhost
Port	23

3. On the **Target** tab, select the **Infineon Triboard TC29xB** and click **Debug**.

   The TASKING Debug perspective is associated with the TASKING C/C++ Debugger. Because the TASKING C/C++ perspective is still active, Eclipse asks to open the TASKING Debug perspective.

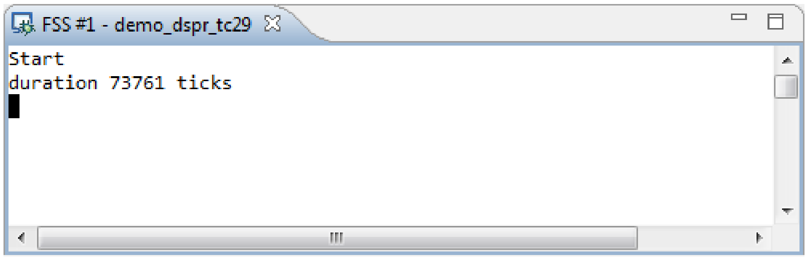
4. Optionally, enable the option **Remember my decision** and click **Yes**.

   The debug session is launched. This may take a few seconds.

5. From the **Debug** menu, select **Resume** (▶) to run the application on the target board.

   The output of the application appears in the FSS (File System Simulation) view.

6. Inspect the FSS view and notice the number of ticks.



```
Start
duration 73761 ticks
```

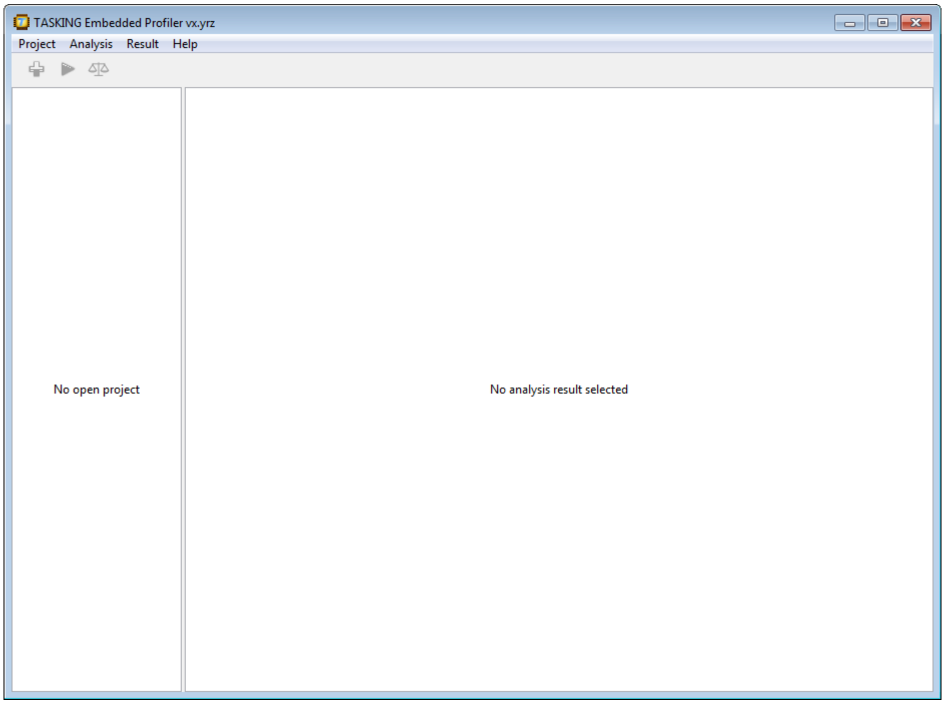
7. From the **Debug** menu, select **Terminate** (🟥) to stop the debugging session. This is necessary to free the connection with the target board.

3.2. Analyze Project in TASKING Embedded Profiler

Now it is time to start analyzing the demo project.

Create a project

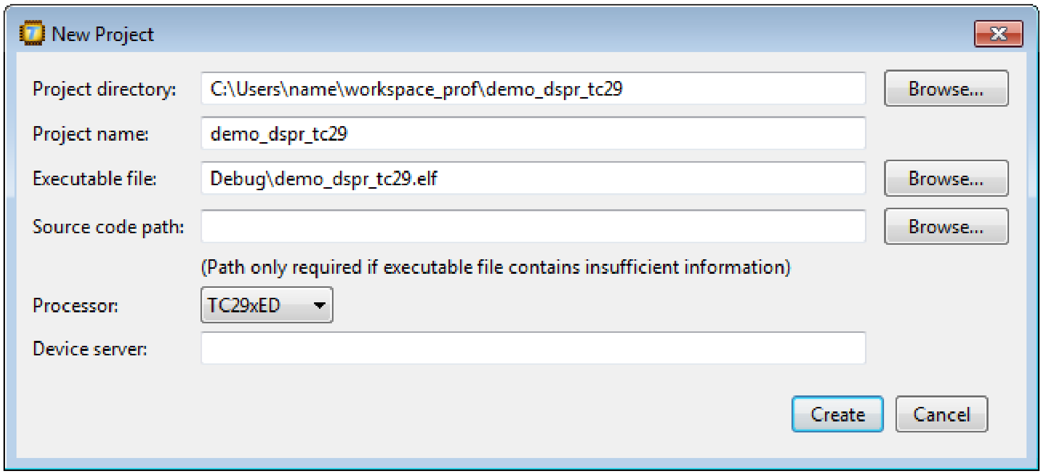
1. Start the TASKING Embedded Profiler.



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.



Project directory: C:\Users\name\workspace_prof\demo_dspr_tc29
Project name: demo_dspr_tc29
Executable file: Debug\demo_dspr_tc29.elf
Source code path:
(Path only required if executable file contains insufficient information)
Processor: TC29xED
Device server:

3. In the **Project directory** field, specify the directory where you want to store the Embedded Profiler project file (file with extension `.EmbProf`).

4. In the **Project name** field, enter the name of the project (for example, you can use the same name as the Eclipse project, `demo_dspr_tc29`).

5. In the **Executable file** field, specify the name of the ELF file. This file is usually relative to the project directory. If the executable file is stored in another directory, the full path name is shown.

6. 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.

7. Select the **Processor**. For example, `TC29xED`.

8. For the **Device server**, enter the server name (leave blank for `localhost`).

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

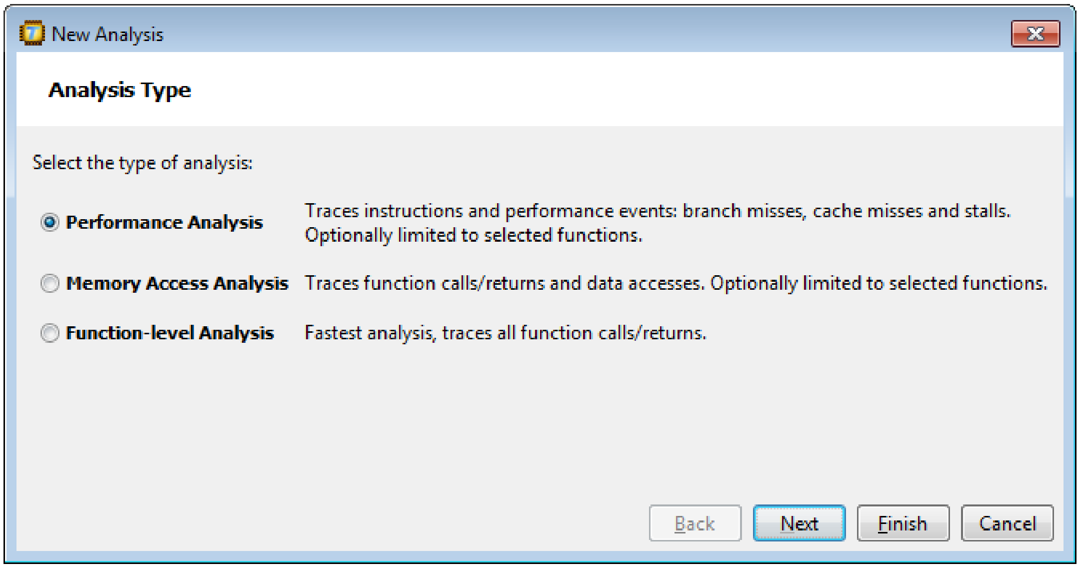
   The new project is created and opened.

   demo_dspr_tc29

Create a Performance analysis

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

   The New Analysis wizard appears.



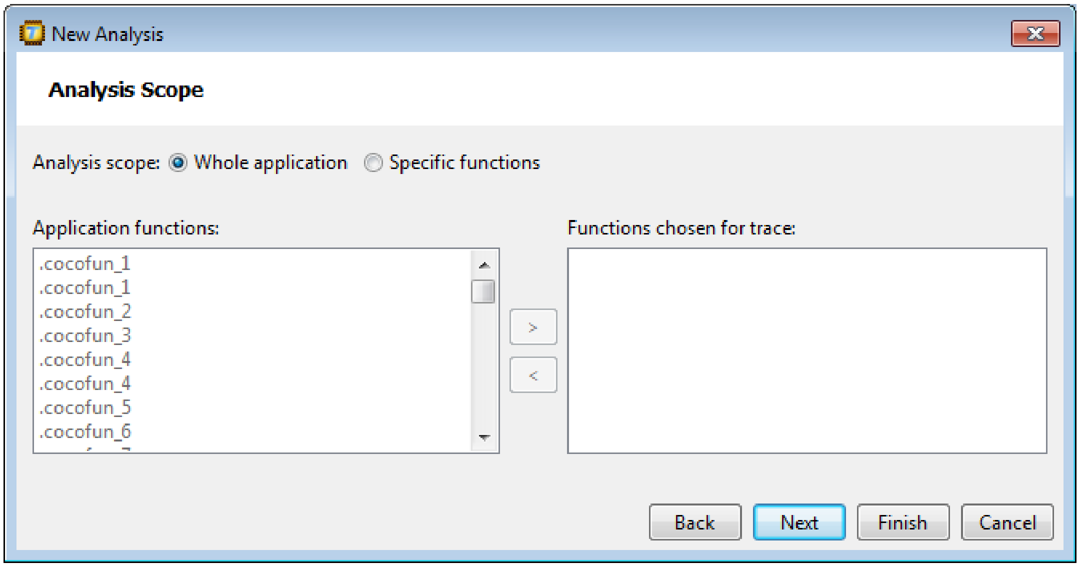
Analysis Type

Select the type of analysis:

[✓ **Performance Analysis**]	Traces instructions and performance events: branch misses, cache misses and stalls. Optionally limited to selected functions.
Memory Access Analysis	Traces function calls/returns and data accesses. Optionally limited to selected functions.
Function-level Analysis	Fastest analysis, traces all function calls/returns.

2. Three types of analyses are possible. Select **Performance Analysis** and click **Next**.

   The Analysis Scope page appears.



Analysis Scope

Analysis scope: [✓ Whole application] Specific functions

Application functions:

.cocofun_1
.cocofun_1
.cocofun_2
.cocofun_3
.cocofun_4
.cocofun_4
.cocofun_5
.cocofun_6

Functions chosen for trace:

3. For this tutorial select **Whole Application**. If you select **Specific Functions**, select one or more **Application functions** and click **>**.

Note that the number of Application functions you can select, is limited by the hardware. Usually, you can select a maximum of 4 functions.

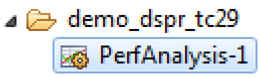
4. Click **Next**.

   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.

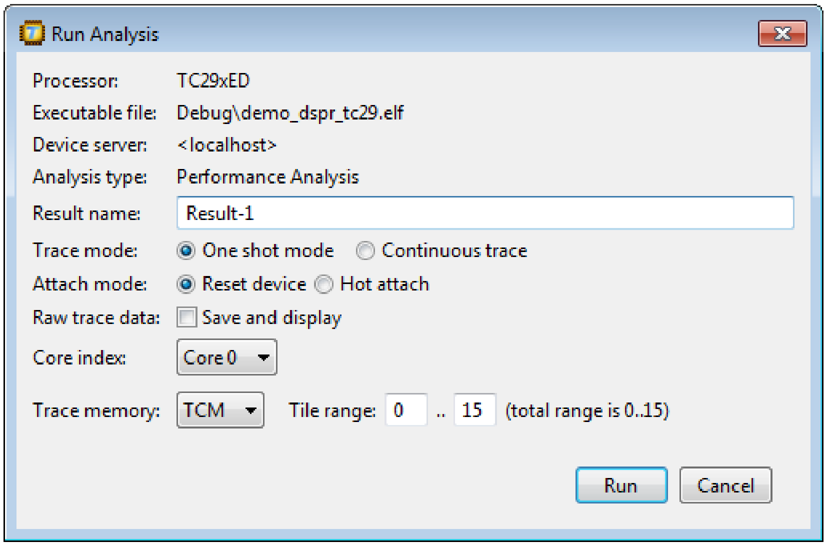


Run the analysis

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.



Run Analysis

Processor: TC29xED
Executable file: Debug\demo_dspr_tc29.elf
Device server: <localhost>
Analysis type: Performance Analysis
Result name: Result-1
Trace mode: [✓ One shot mode] Continuous trace
Attach mode: [✓ Reset device] Hot attach
Raw trace data: Save and display
Core index: Core 0
Trace memory: TCM Tile range: 0 .. 15 (total range is 0..15)

3. Enter an analysis **Result name** (default `Result-` and a sequence number).

4. Select a **Trace 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. Optionally enable **Save and display** 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.

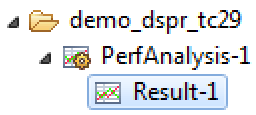
7. In the **Core index** field, select the TriCore core for which you want to run the analysis.

8. Select the type of **Trace 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.

9. 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.

10. Click **Run**.

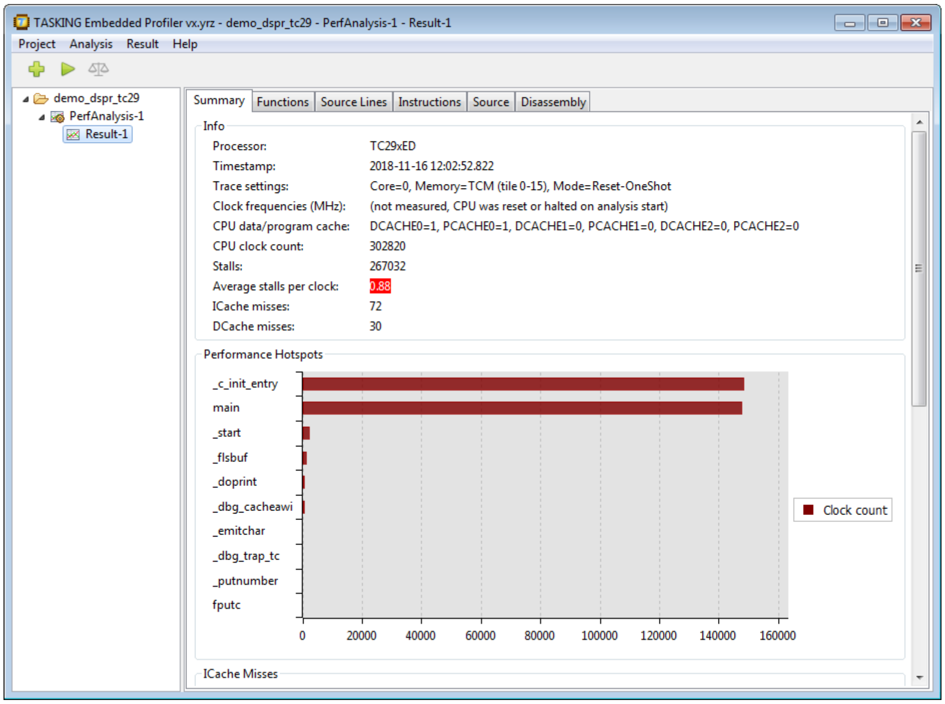
    The analysis starts. After the analysis is finished the result is present in the project tree.



Inspect the result of the Performance analysis

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

   The result appears in several tabs.


TASKING Embedded Profiler vx.yrz - demo_dspr_tc29 - PerfAnalysis-1 - Result-1
Project Analysis Result Help
demo_dspr_tc29
PerfAnalysis-1
Result-1
Summary Functions Source Lines Instructions Source Disassembly

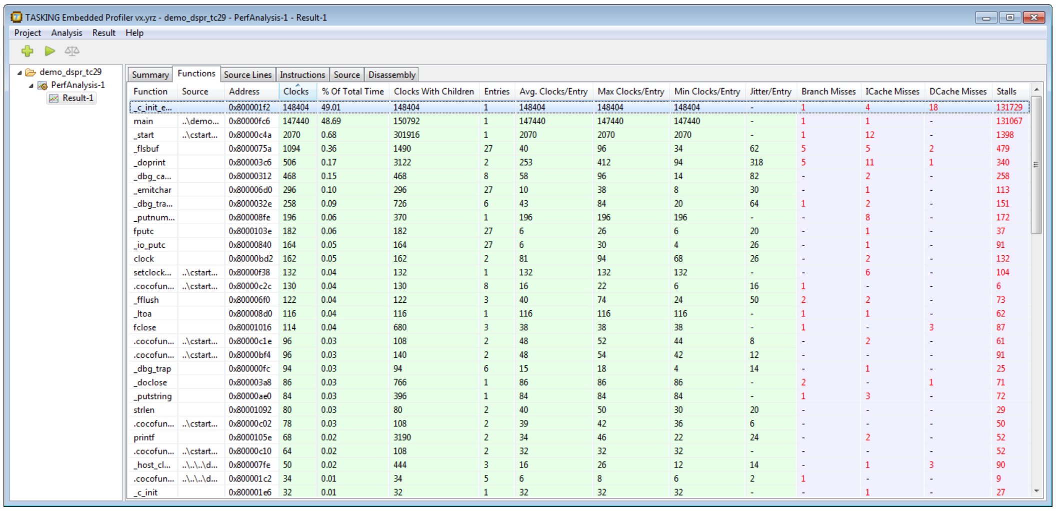
Info

Processor:	TC29xED
Timestamp:	2018-11-16 12:02:52.822
Trace settings:	Core=0, Memory=TCM (tile 0-15), Mode=Reset-OneShot
Clock frequencies (MHz):	(not measured, CPU was reset or halted on analysis start)
CPU data/program cache:	DCACHE0=1, PCACHE0=1, DCACHE1=0, PCACHE1=0, DCACHE2=0, PCACHE2=0
CPU clock count:	302820
Stalls:	267032
Average stalls per clock:	0.88
ICache misses:	72
DCache misses:	30

2. On the **Summary** tab, notice the high number of **Average stalls per clock** (0.88).

   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 `_c_init_entry` and `main`.

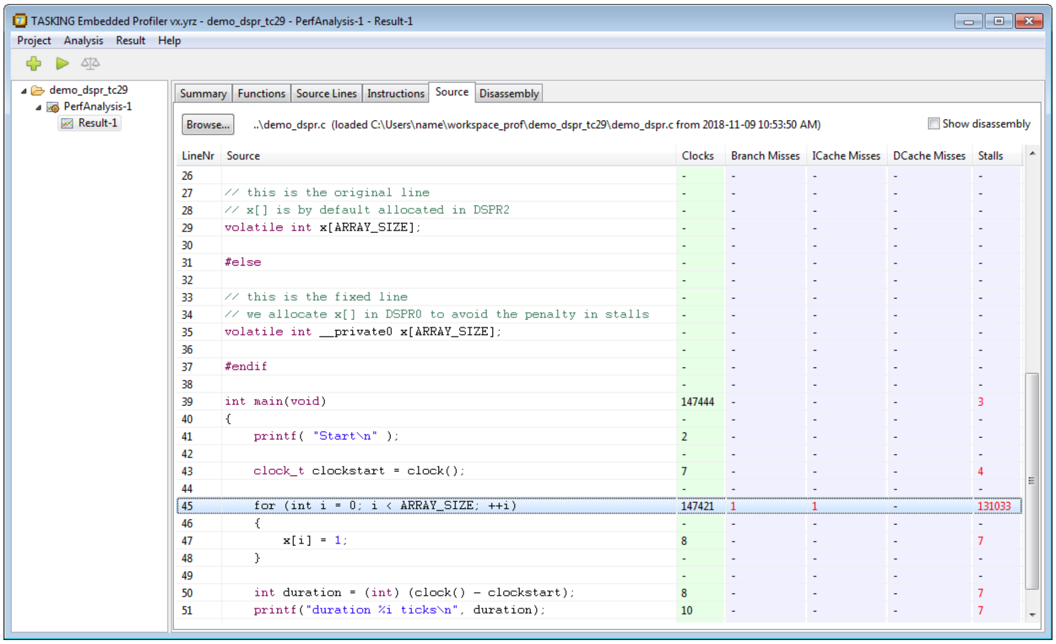


TASKING Embedded Profiler vx.yrz - demo_dspr_tc29 - PerfAnalysis-1 - Result-1

Function	Source	Address	Clocks	% Of Total Time	Clocks With Children	Entries	Avg. Clocks/Entry	Max Clocks/Entry	Min Clocks/Entry	Jitter/Entry	Branch Misses	ICache Misses	DCache Misses	Stalls
_c_init_e...		0x800001f2	148404	49.01	148404	1	148404	148404	148404	-	1	4	18	131729
main	..\demo...	0x80000fc6	147440	48.69	150792	1	147440	147440	147440	-	1	1	-	131067
_start	..\cstart...	0x80000c4a	2070	0.68	301916	1	2070	2070	2070	-	1	12	-	1398
_flsbuf		0x8000075a	1094	0.36	1490	27	40	96	34	62	5	5	2	479
_doprint		0x800003c6	506	0.17	3122	2	253	412	94	318	5	11	1	340
_dbg_ca...		0x80000312	468	0.15	468	8	58	96	14	82	-	2	-	258
_emitchar		0x800006d0	296	0.10	296	27	10	38	8	30	-	1	-	113
_dbg_tra...		0x8000032e	258	0.09	726	6	43	84	20	64	1	2	-	151
_putnum...		0x800008fe	196	0.06	370	1	196	196	196	-	-	8	-	172
fputc		0x8000103e	182	0.06	182	27	6	26	6	20	-	1	-	37
_io_putc		0x80000840	164	0.05	164	27	6	30	4	26	-	1	-	91
clock		0x80000bd2	162	0.05	162	2	81	94	68	26	-	2	-	132
setclock...	..\cstart...	0x80000f38	132	0.04	132	1	132	132	132	-	-	6	-	104
.cocofun...	..\cstart...	0x80000c2c	130	0.04	130	8	16	22	6	16	1	-	-	6
_fflush		0x800006f0	122	0.04	122	3	40	74	24	50	2	2	-	73
_ltoa		0x800008d0	116	0.04	116	1	116	116	116	-	1	1	-	62
fclose		0x80001016	114	0.04	680	3	38	38	38	-	1	-	3	87
.cocofun...	..\cstart...	0x80000c1e	96	0.03	108	2	48	52	44	8	-	2	-	61
.cocofun...	..\cstart...	0x80000bf4	96	0.03	140	2	48	54	42	12	-	-	-	91
_dbg_trap		0x800000fc	94	0.03	94	6	15	18	4	14	-	1	-	25
_doclose		0x800003a8	86	0.03	766	1	86	86	86	-	2	-	1	71
_putstring		0x80000ae0	84	0.03	396	1	84	84	84	-	1	3	-	72
strlen		0x80001092	80	0.03	80	2	40	50	30	20	-	-	-	29
.cocofun...	..\cstart...	0x80000c02	78	0.03	108	2	39	42	36	6	-	-	-	50
printf		0x8000105e	68	0.02	3190	2	34	46	22	24	-	2	-	52
.cocofun...	..\cstart...	0x80000c10	64	0.02	108	2	32	32	32	-	-	-	-	52
_host_cl...	..\..\..\d...	0x800007fe	50	0.02	444	3	16	26	12	14	-	1	3	90
.cocofun...	..\..\..\d...	0x800001c2	34	0.01	34	5	6	8	6	2	1	-	-	9
_c_init		0x800001e6	32	0.01	32	1	32	32	32	-	-	1	-	27

4. Double-click on `main`.

   The Source tab opens.



TASKING Embedded Profiler vx.yrz - demo_dspr_tc29 - PerfAnalysis-1 - Result-1

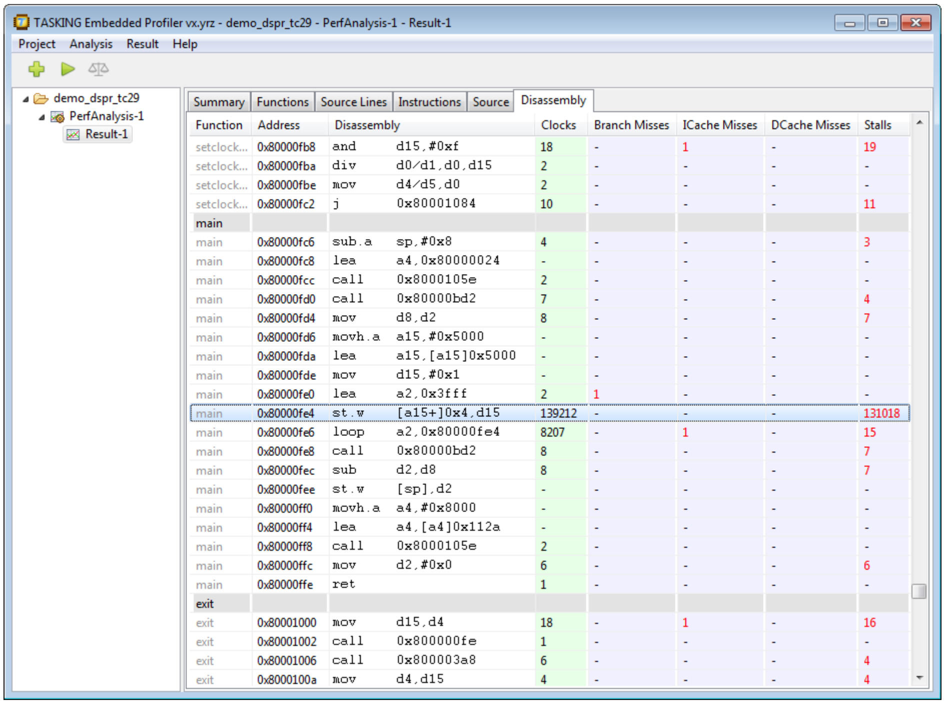
..\demo_dspr.c (loaded C:\Users\name\workspace_prof\demo_dspr_tc29\demo_dspr.c from 2018-11-09 10:53:50 AM)

LineNr	Source	Clocks	Branch Misses	ICache Misses	DCache Misses	Stalls
26		-	-	-	-	-
27	`// this is the original line`	-	-	-	-	-
28	`// x[] is by default allocated in DSPR2`	-	-	-	-	-
29	`volatile int x[ARRAY_SIZE];`	-	-	-	-	-
30		-	-	-	-	-
31	`#else`	-	-	-	-	-
32		-	-	-	-	-
33	`// this is the fixed line`	-	-	-	-	-
34	`// we allocate x[] in DSPR0 to avoid the penalty in stalls`	-	-	-	-	-
35	`volatile int __private0 x[ARRAY_SIZE];`	-	-	-	-	-
36		-	-	-	-	-
37	`#endif`	-	-	-	-	-
38		-	-	-	-	-
39	`int main(void)`	147444	-	-	-	3
40	`{`	-	-	-	-	-
41	`    printf( "Start\n" );`	2	-	-	-	-
42		-	-	-	-	-
43	`    clock_t clockstart = clock();`	7	-	-	-	4
44		-	-	-	-	-
45	`    for (int i = 0; i < ARRAY_SIZE; ++i)`	147421	1	1	-	131033
46	`    {`	-	-	-	-	-
47	`        x[i] = 1;`	8	-	-	-	7
48	`    }`	-	-	-	-	-
49		-	-	-	-	-
50	`    int duration = (int) (clock() - clockstart);`	8	-	-	-	7
51	`    printf("duration %i ticks\n", duration);`	10	-	-	-	7

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.



TASKING Embedded Profiler vx.yrz - demo_dspr_tc29 - PerfAnalysis-1 - Result-1

Project Analysis Result Help

- demo_dspr_tc29
  - PerfAnalysis-1
    - Result-1

Summary | Functions | Source Lines | Instructions | Source | Disassembly

Function	Address	Disassembly		Clocks	Branch Misses	ICache Misses	DCache Misses	Stalls
setclock...	0x80000fb8	and	d15,#0xf	18	-	1	-	19
setclock...	0x80000fba	div	d0/d1,d0,d15	2	-	-	-	-
setclock...	0x80000fbe	mov	d4/d5,d0	2	-	-	-	-
setclock...	0x80000fc2	j	0x80001084	10	-	-	-	11
main								
main	0x80000fc6	sub.a	sp,#0x8	4	-	-	-	3
main	0x80000fc8	lea	a4,0x80000024	-	-	-	-	-
main	0x80000fcc	call	0x8000105e	2	-	-	-	-
main	0x80000fd0	call	0x80000bd2	7	-	-	-	4
main	0x80000fd4	mov	d8,d2	8	-	-	-	7
main	0x80000fd6	movh.a	a15,#0x5000	-	-	-	-	-
main	0x80000fda	lea	a15,[a15]0x5000	-	-	-	-	-
main	0x80000fde	mov	d15,#0x1	-	-	-	-	-
main	0x80000fe0	lea	a2,0x3fff	2	1	-	-	-
main	0x80000fe4	st.w	[a15+]0x4,d15	139212	-	-	-	131018
main	0x80000fe6	loop	a2,0x80000fe4	8207	-	1	-	15
main	0x80000fe8	call	0x80000bd2	8	-	-	-	7
main	0x80000fec	sub	d2,d8	8	-	-	-	7
main	0x80000fee	st.w	[sp],d2	-	-	-	-	-
main	0x80000ff0	movh.a	a4,#0x8000	-	-	-	-	-
main	0x80000ff4	lea	a4,[a4]0x112a	-	-	-	-	-
main	0x80000ff8	call	0x8000105e	2	-	-	-	-
main	0x80000ffc	mov	d2,#0x0	6	-	-	-	6
main	0x80000ffe	ret		1	-	-	-	-
exit								
exit	0x80001000	mov	d15,d4	18	-	1	-	16
exit	0x80001002	call	0x800000fe	1	-	-	-	-
exit	0x80001006	call	0x800003a8	6	-	-	-	4
exit	0x8000100a	mov	d4,d15	4	-	-	-	4

Create and run a Memory access analysis

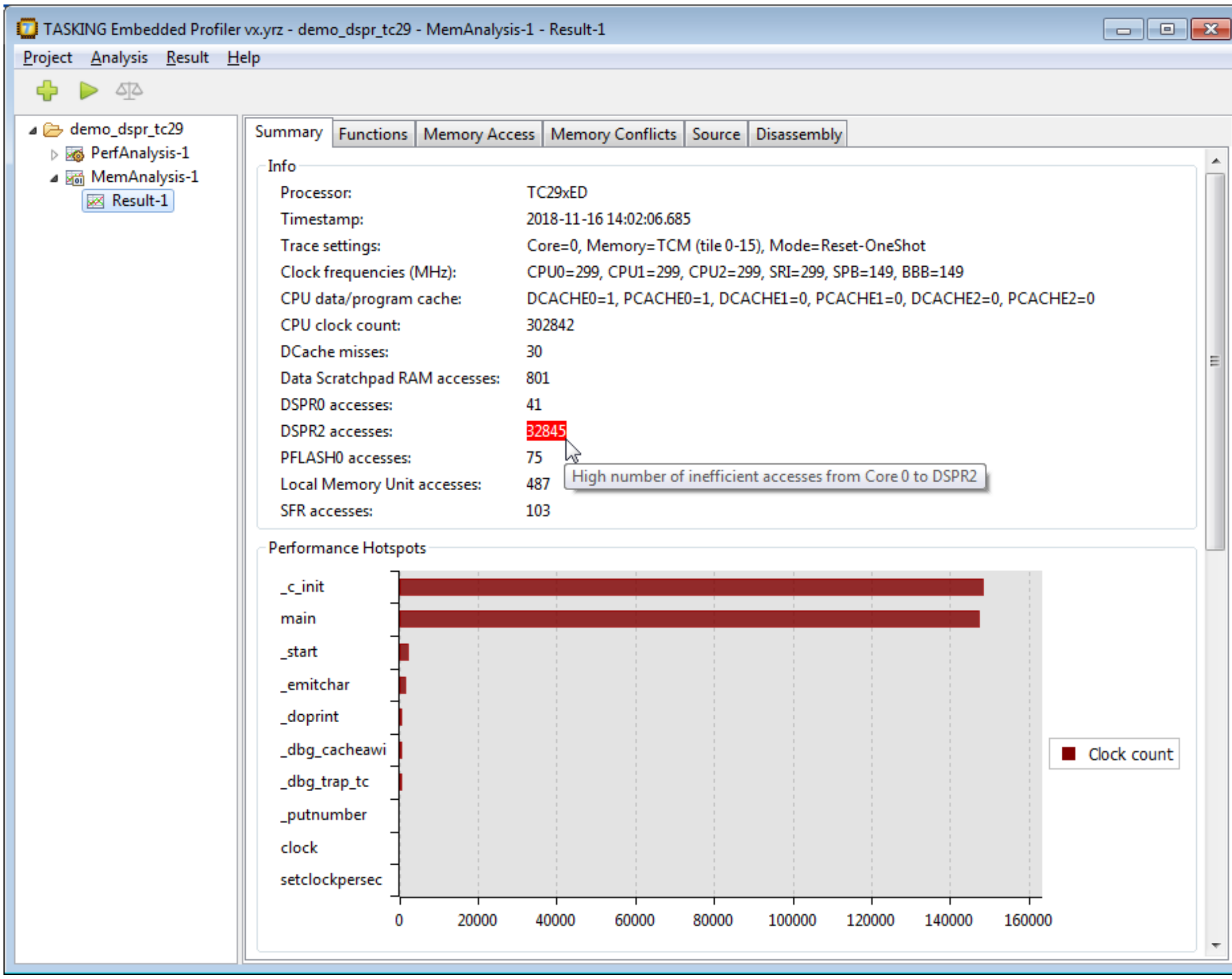
1. Repeat the steps described above with *Create a Performance analysis*, but in Step 2 select **Memory Access Analysis**.

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.



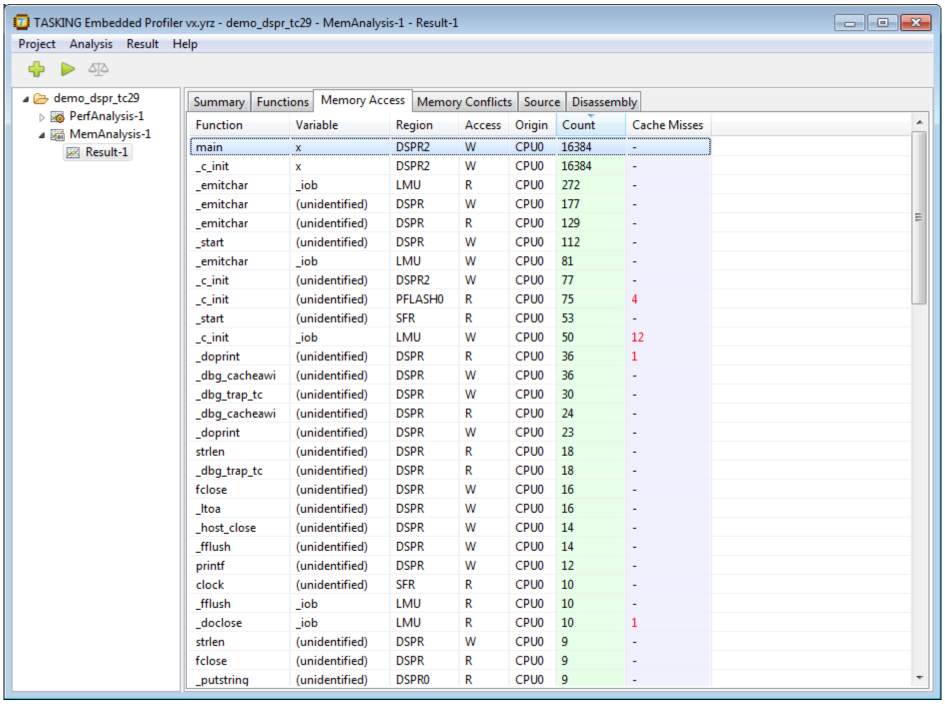
Processor:	TC29xED
Timestamp:	2018-11-16 14:02:06.685
Trace settings:	Core=0, Memory=TCM (tile 0-15), Mode=Reset-OneShot
Clock frequencies (MHz):	CPU0=299, CPU1=299, CPU2=299, SRI=299, SPB=149, BBB=149
CPU data/program cache:	DCACHE0=1, PCACHE0=1, DCACHE1=0, PCACHE1=0, DCACHE2=0, PCACHE2=0
CPU clock count:	302842
DCache misses:	30
Data Scratchpad RAM accesses:	801
DSPR0 accesses:	41
DSPR2 accesses:	32845
PFLASH0 accesses:	75
Local Memory Unit accesses:	487
SFR accesses:	103

High number of inefficient accesses from Core 0 to DSPR2

2. On the **Summary** tab, notice the high number of **DSPR2 accesses** (32845). 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 DSPR memory access is calculated as: *factor* * *total DSPR access*. In this case 0.05*(41+32845)=1644.3. You can change this threshold factor in the Settings dialog (**Project » Settings**). See Section 6.1, *Settings Dialog*.

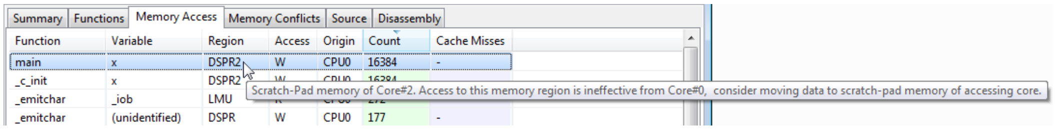
3. On the **Memory Access** tab and notice that `_c_init` and `main` both access variable `x` in DSPR2.



Function	Variable	Region	Access	Origin	Count	Cache Misses
main	x	DSPR2	W	CPU0	16384	-
_c_init	x	DSPR2	W	CPU0	16384	-
_emitchar	_iob	LMU	R	CPU0	272	-
_emitchar	(unidentified)	DSPR	W	CPU0	177	-
_emitchar	(unidentified)	DSPR	R	CPU0	129	-
_start	(unidentified)	DSPR	W	CPU0	112	-
_emitchar	_iob	LMU	W	CPU0	81	-
_c_init	(unidentified)	DSPR2	W	CPU0	77	-
_c_init	(unidentified)	PFLASH0	R	CPU0	75	4
_start	(unidentified)	SFR	R	CPU0	53	-
_c_init	_iob	LMU	W	CPU0	50	12
_doprint	(unidentified)	DSPR	R	CPU0	36	1
_dbg_cacheawi	(unidentified)	DSPR	W	CPU0	36	-
_dbg_trap_tc	(unidentified)	DSPR	W	CPU0	30	-
_dbg_cacheawi	(unidentified)	DSPR	R	CPU0	24	-
_doprint	(unidentified)	DSPR	W	CPU0	23	-
strlen	(unidentified)	DSPR	R	CPU0	18	-
_dbg_trap_tc	(unidentified)	DSPR	R	CPU0	18	-
fclose	(unidentified)	DSPR	W	CPU0	16	-
_ltoa	(unidentified)	DSPR	W	CPU0	16	-
_host_close	(unidentified)	DSPR	W	CPU0	14	-
_fflush	(unidentified)	DSPR	W	CPU0	14	-
printf	(unidentified)	DSPR	W	CPU0	12	-
clock	(unidentified)	SFR	R	CPU0	10	-
_fflush	_iob	LMU	R	CPU0	10	-
_doclose	_iob	LMU	R	CPU0	10	1
strlen	(unidentified)	DSPR	W	CPU0	9	-
fclose	(unidentified)	DSPR	R	CPU0	9	-
_putstring	(unidentified)	DSPR0	R	CPU0	9	-

4. Hover the mouse over DSPR2 in `main`.

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



Function	Variable	Region	Access	Origin	Count	Cache Misses
main	x	DSPR2	W	CPU0	16384	-
_c_init	x	DSPR2	W	CPU0	16384	
_emitchar	_iob	LMU	R	CPU0	272	
_emitchar	(unidentified)	DSPR	W	CPU0	177	-

Scratch-Pad memory of Core#2. Access to this memory region is ineffective from Core#0, consider moving data to scratch-pad memory of accessing core.

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:

```
#define FIXED   0
```

   into:

```
#define FIXED   1
```

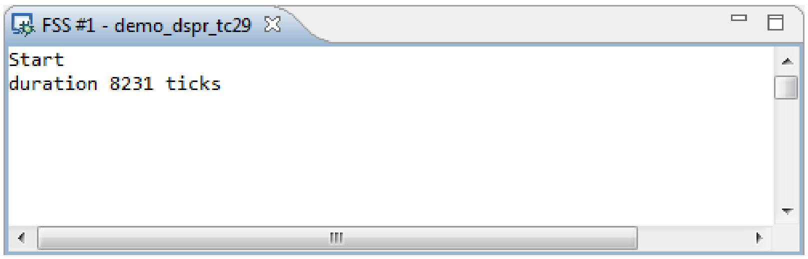
3. From the **Project** menu, select **Rebuild demo_dspr_tc29** ().

4. From the **Debug** menu, select **Debug** *project* ().

5. From the **Debug** menu, select **Resume** () to run the application on the target board.

   The output of the application appears in the FSS (File System Simulation) view.

6. Inspect the FSS view and notice the number of ticks has reduced significantly.



```
Start
duration 8231 ticks
```

7. From the **Debug** menu, select **Terminate** () to stop the debugging session. This is necessary to free the connection with the target board.

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

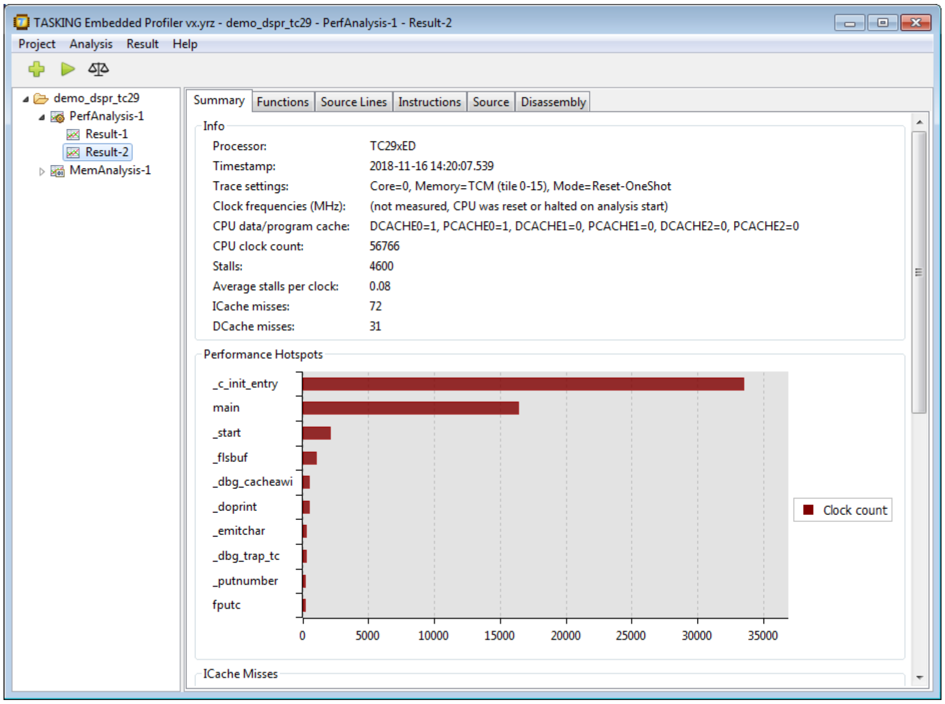
1. In the TASKING Embedded Profiler, select `PerfAnalysis-1`.

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 **Average stalls per clock** has reduced significantly from 0.88 to 0.08.


TASKING Embedded Profiler vx.yrz - demo_dspr_tc29 - PerfAnalysis-1 - Result-2

Info

Processor:	TC29xED
Timestamp:	2018-11-16 14:20:07.539
Trace settings:	Core=0, Memory=TCM (tile 0-15), Mode=Reset-OneShot
Clock frequencies (MHz):	(not measured, CPU was reset or halted on analysis start)
CPU data/program cache:	DCACHE0=1, PCACHE0=1, DCACHE1=0, PCACHE1=0, DCACHE2=0, PCACHE2=0
CPU clock count:	56766
Stalls:	4600
Average stalls per clock:	0.08
ICache misses:	72
DCache misses:	31

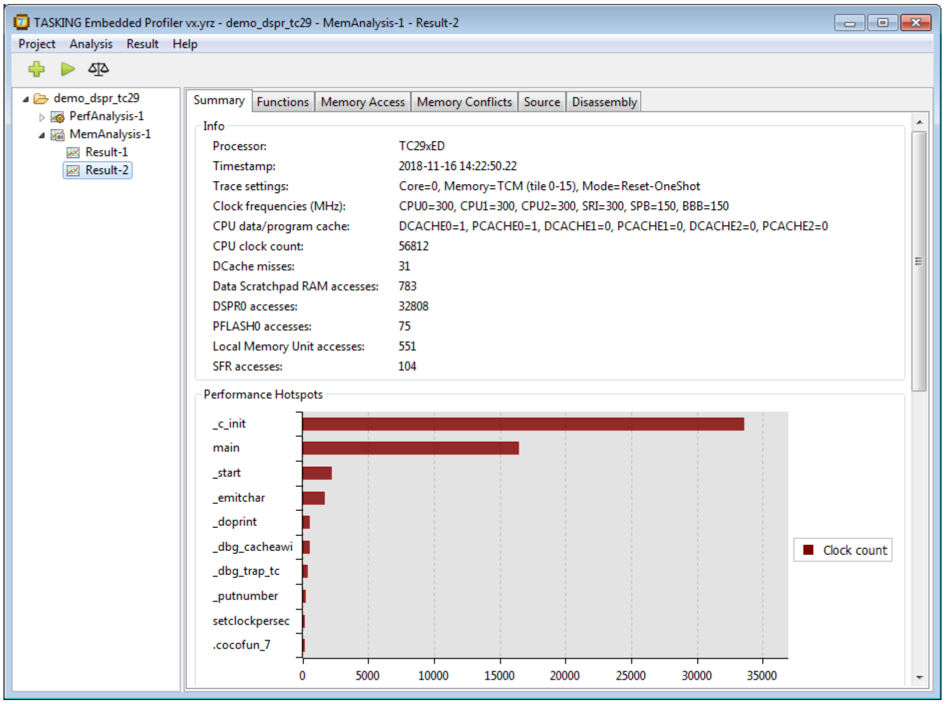
5. Also inspect the other tabs yourself to see the results.

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.

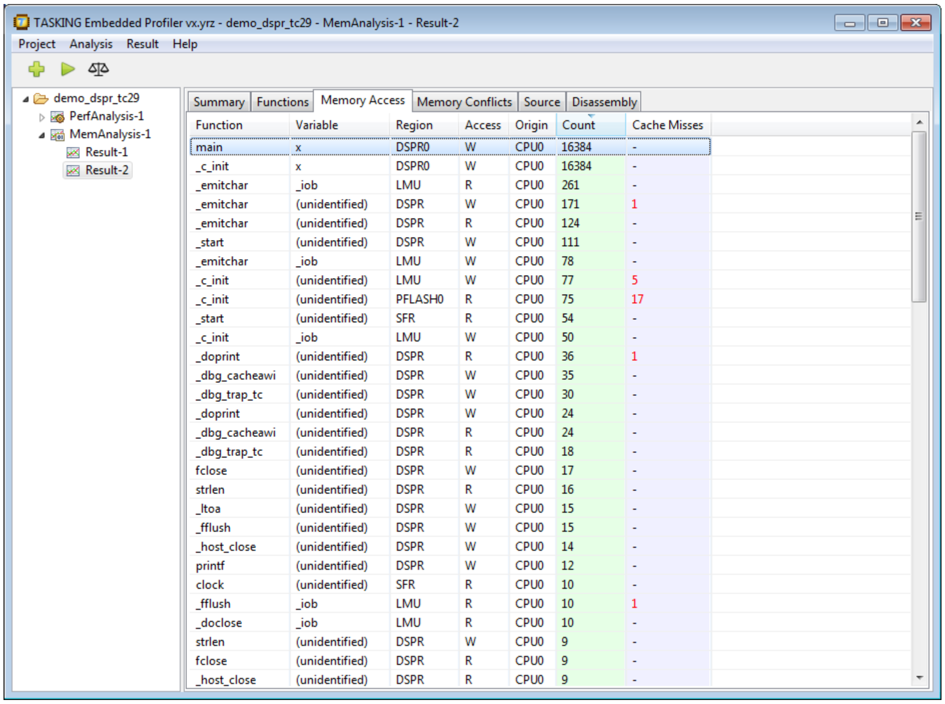
4. Select `Result-2` and notice that on the **Summary** tab, the accesses are now in DSPR0. And notice that on the **Memory Access** tab `_c_init` and `main` now both access variable `x` in DSPR0.


TASKING Embedded Profiler vx.yrz - demo_dspr_tc29 - MemAnalysis-1 - Result-2
Project Analysis Result Help
demo_dspr_tc29
PerfAnalysis-1
MemAnalysis-1
Result-1
Result-2
Summary Functions Memory Access Memory Conflicts Source Disassembly

Info

Processor:	TC29xED
Timestamp:	2018-11-16 14:22:50.22
Trace settings:	Core=0, Memory=TCM (tile 0-15), Mode=Reset-OneShot
Clock frequencies (MHz):	CPU0=300, CPU1=300, CPU2=300, SRI=300, SPB=150, BBB=150
CPU data/program cache:	DCACHE0=1, PCACHE0=1, DCACHE1=0, PCACHE1=0, DCACHE2=0, PCACHE2=0
CPU clock count:	56812
DCache misses:	31
Data Scratchpad RAM accesses:	783
DSPR0 accesses:	32808
PFLASH0 accesses:	75
Local Memory Unit accesses:	551
SFR accesses:	104

Performance Hotspots


TASKING Embedded Profiler vx.yrz - demo_dspr_tc29 - MemAnalysis-1 - Result-2
Project Analysis Result Help
demo_dspr_tc29
PerfAnalysis-1
MemAnalysis-1
Result-1
Result-2
Summary Functions Memory Access Memory Conflicts Source Disassembly

Function	Variable	Region	Access	Origin	Count	Cache Misses
main	x	DSPR0	W	CPU0	16384	-
_c_init	x	DSPR0	W	CPU0	16384	-
_emitchar	_iob	LMU	R	CPU0	261	-
_emitchar	(unidentified)	DSPR	W	CPU0	171	1
_emitchar	(unidentified)	DSPR	R	CPU0	124	-
_start	(unidentified)	DSPR	W	CPU0	111	-
_emitchar	_iob	LMU	W	CPU0	78	-
_c_init	(unidentified)	LMU	W	CPU0	77	5
_c_init	(unidentified)	PFLASH0	R	CPU0	75	17
_start	(unidentified)	SFR	R	CPU0	54	-
_c_init	_iob	LMU	W	CPU0	50	-
_doprint	(unidentified)	DSPR	R	CPU0	36	1
_dbg_cacheawi	(unidentified)	DSPR	W	CPU0	35	-
_dbg_trap_tc	(unidentified)	DSPR	W	CPU0	30	-
_doprint	(unidentified)	DSPR	W	CPU0	24	-
_dbg_cacheawi	(unidentified)	DSPR	R	CPU0	24	-
_dbg_trap_tc	(unidentified)	DSPR	R	CPU0	18	-
fclose	(unidentified)	DSPR	W	CPU0	17	-
strlen	(unidentified)	DSPR	R	CPU0	16	-
_ltoa	(unidentified)	DSPR	W	CPU0	15	-
_fflush	(unidentified)	DSPR	W	CPU0	15	-
_host_close	(unidentified)	DSPR	W	CPU0	14	-
printf	(unidentified)	DSPR	W	CPU0	12	-
clock	(unidentified)	SFR	R	CPU0	10	-
_fflush	_iob	LMU	R	CPU0	10	1
_doclose	_iob	LMU	R	CPU0	10	-
strlen	(unidentified)	DSPR	W	CPU0	9	-
fclose	(unidentified)	DSPR	R	CPU0	9	-
_host_close	(unidentified)	DSPR	R	CPU0	9	-

3.5. Compare Results

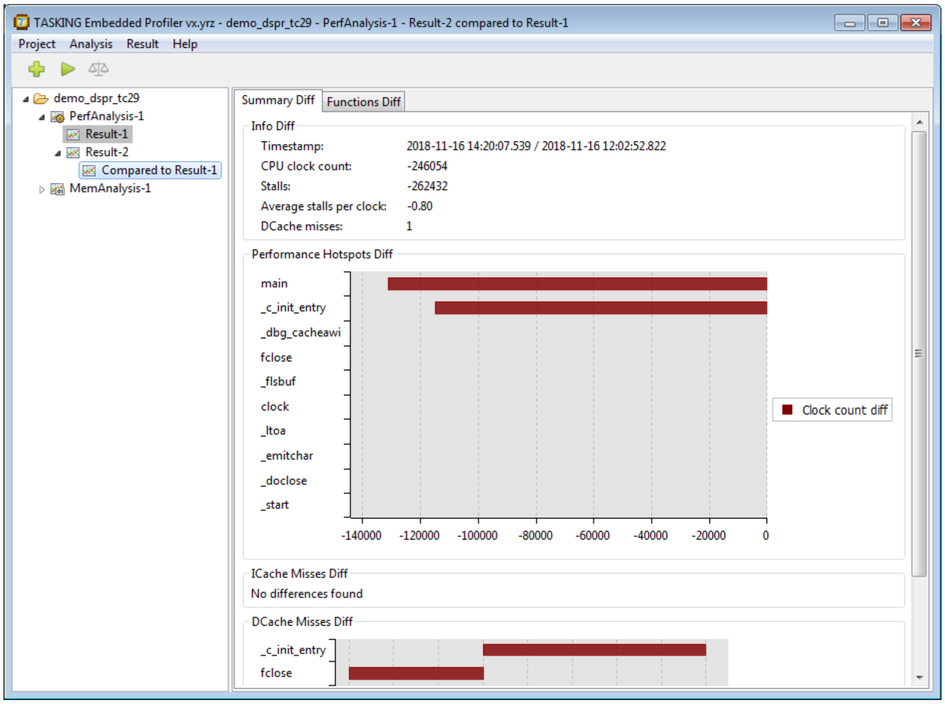
The Embedded Profiler has a feature to compare results. This is very useful to see the differences before and after a fix. Note that you can only compare results from the same analysis.

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".



TASKING Embedded Profiler vx.yrz - demo_dspr_tc29 - PerfAnalysis-1 - Result-2 compared to Result-1

Info Diff

Timestamp:	2018-11-16 14:20:07.539 / 2018-11-16 12:02:52.822
CPU clock count:	-246054
Stalls:	-262432
Average stalls per clock:	-0.80
DCache misses:	1

ICache Misses Diff

No differences found

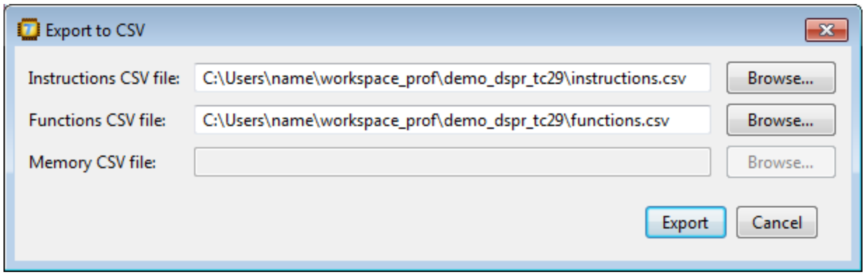
3.6. Export Results

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.



Export to CSV

Instructions CSV file: C:\Users\name\workspace_prof\demo_dspr_tc29\instructions.csv

Functions CSV file: C:\Users\name\workspace_prof\demo_dspr_tc29\functions.csv

Memory CSV file:

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 Optimizations

Analysis results may be different for functions in a performance analysis compared to a memory analysis or an 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. This optimization is enabled by default for the TASKING VX-toolset for TriCore. This optimization causes a leaf function that is called at the last line of a function to not show up in the analysis result while the function's code is executed. The reason for this is that the leaf function is entered with a jump instruction and the leaf function's return instruction performs the return that the calling function would have done.

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()`.

Because of the jump instead of a call, the TASKING Embedded Profiler will not detect `func_c()` in a memory analysis and function analysis, though the cycle count for `func_c()` is added to `func_b()`.

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

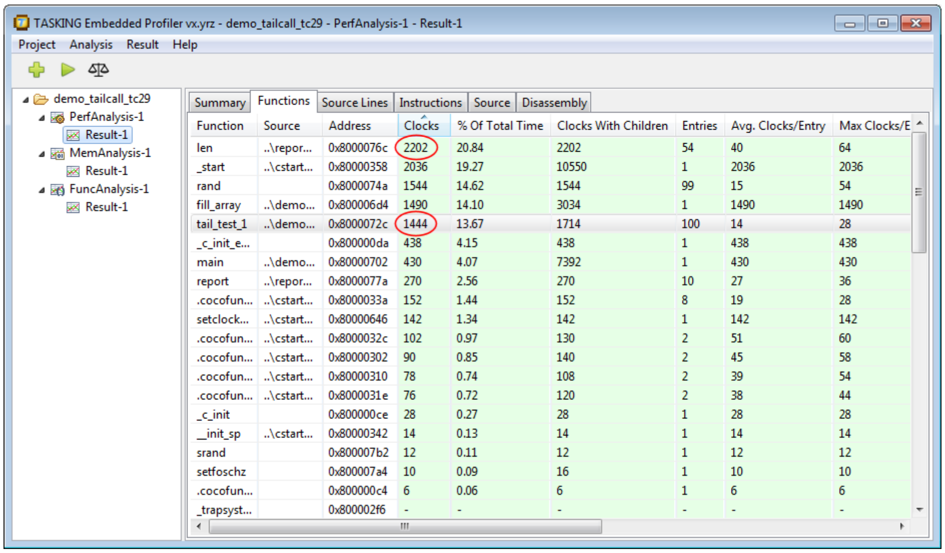
1. Import the `demo_tailcall` tutorial for the TC29x or TC39x the same way as explained for `demo_dspr` in Section 3.1, *Prepare Demo Project in Eclipse*. For this tutorial we use `demo_tailcall_tc29`. The tutorial already contains an Embedded Profiler project file.

2. Start the TASKING Embedded Profiler.

3. From the **Project** menu, select **Open Project**, and select `demo_tailcall_tc29.EmbProf`.

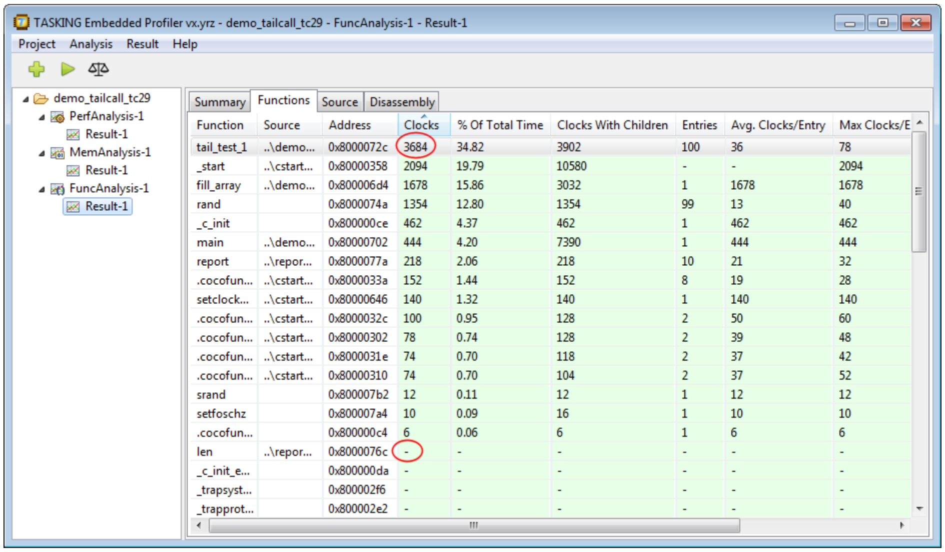
4. Inspect the **Functions** tab in `Result-1` of `PerfAnalysis-1`, `MemAnalysis-1` and `FuncAnalysis-1`.

   For the Performance Analysis `PerfAnalysis-1`, you can see there are 2202 clocks for function `len()` and 1444 clocks for function `tail_test_1()`.



Function	Source	Address	Clocks	% Of Total Time	Clocks With Children	Entries	Avg. Clocks/Entry	Max Clocks/E
len	..\repor...	0x8000076c	2202	20.84	2202	54	40	64
_start	..\cstart...	0x80000358	2036	19.27	10550	1	2036	2036
rand		0x8000074a	1544	14.62	1544	99	15	54
fill_array	..\demo...	0x800006d4	1490	14.10	3034	1	1490	1490
tail_test_1	..\demo...	0x8000072c	1444	13.67	1714	100	14	28
_c_init_e...		0x800000da	438	4.15	438	1	438	438
main	..\demo...	0x80000702	430	4.07	7392	1	430	430
report	..\repor...	0x8000077a	270	2.56	270	10	27	36
.cocofun...	..\cstart...	0x8000033a	152	1.44	152	8	19	28
setclock...	..\cstart...	0x80000646	142	1.34	142	1	142	142
.cocofun...	..\cstart...	0x8000032c	102	0.97	130	2	51	60
.cocofun...	..\cstart...	0x80000302	90	0.85	140	2	45	58
.cocofun...	..\cstart...	0x80000310	78	0.74	108	2	39	54
.cocofun...	..\cstart...	0x8000031e	76	0.72	120	2	38	44
_c_init		0x800000ce	28	0.27	28	1	28	28
__init_sp	..\cstart...	0x80000342	14	0.13	14	1	14	14
srand		0x800007b2	12	0.11	12	1	12	12
setfoschz		0x800007a4	10	0.09	16	1	10	10
.cocofun...		0x800000c4	6	0.06	6	1	6	6
_trapsyst...		0x800002f6	-	-	-	-	-	-

For the Memory Analysis `MemAnalysis-1` and the Function-level Analysis `FuncAnalysis-1`, you can see there are 3684 clocks for function `tail_test_1()` and no clocks for function `len()`.



Function	Source	Address	Clocks	% Of Total Time	Clocks With Children	Entries	Avg. Clocks/Entry	Max Clocks/E
tail_test_1	..\demo...	0x8000072c	3684	34.82	3902	100	36	78
_start	..\cstart...	0x80000358	2094	19.79	10580	-	-	2094
fill_array	..\demo...	0x800006d4	1678	15.86	3032	1	1678	1678
rand		0x8000074a	1354	12.80	1354	99	13	40
_c_init		0x800000ce	462	4.37	462	1	462	462
main	..\demo...	0x80000702	444	4.20	7390	1	444	444
report	..\repor...	0x8000077a	218	2.06	218	10	21	32
.cocofun...	..\cstart...	0x8000033a	152	1.44	152	8	19	28
setclock...	..\cstart...	0x80000646	140	1.32	140	1	140	140
.cocofun...	..\cstart...	0x8000032c	100	0.95	128	2	50	60
.cocofun...	..\cstart...	0x80000302	78	0.74	128	2	39	48
.cocofun...	..\cstart...	0x8000031e	74	0.70	118	2	37	42
.cocofun...	..\cstart...	0x80000310	74	0.70	104	2	37	52
srand		0x800007b2	12	0.11	12	1	12	12
setfoschz		0x800007a4	10	0.09	16	1	10	10
.cocofun...		0x800000c4	6	0.06	6	1	6	6
len	..\repor...	0x8000076c	-	-	-	-	-	-
_c_init_e...		0x800000da	-	-	-	-	-	-
_trapsyst...		0x800002f6	-	-	-	-	-	-
_trapprot...		0x800002e2	-	-	-	-	-	-

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** and in the **Custom Optimization** tab disable **Peephole optimizations**), 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.

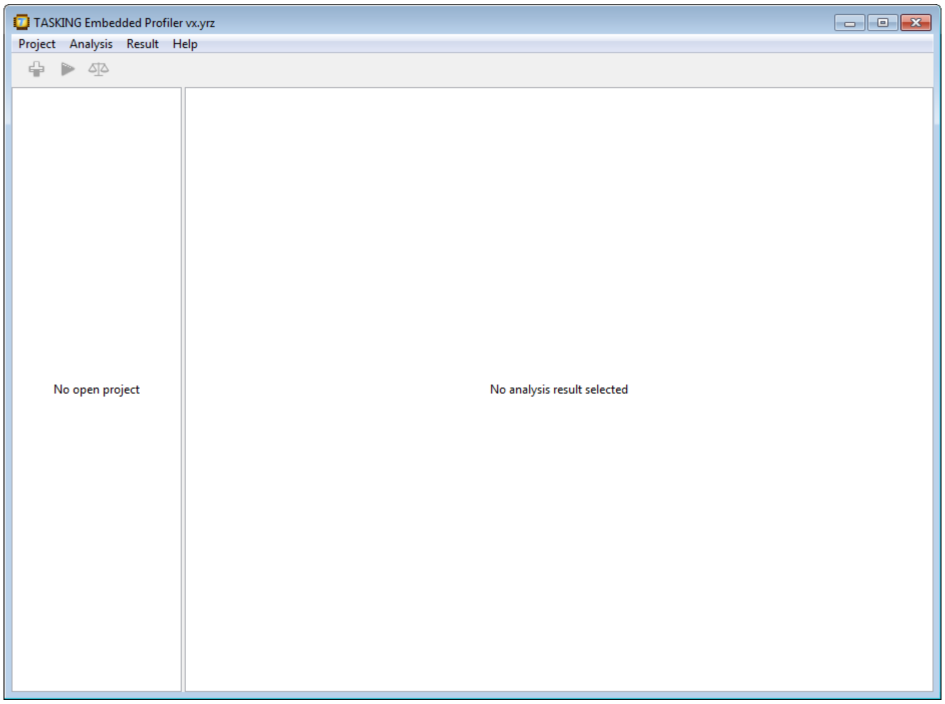
Interrupt handlers that do call function(s) are visible for all three analysis types. For function and memory analyses, the interrupt handler and its children are visible and have cycles accounted to them. These cycles are not added to the clocks with children of the interrupted function. For performance 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:

Option	Description
-? / **--help**	This option causes the program to display an overview of all command line options.
--compare=*result* **-m***result*	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 **--run** should be used together with this option.
--continuous **-c**	This option allows you to run the analysis in continuous trace mode. Without this option, the default is one shot mode.
--core=*core-nr*	This option allows you to specify the core index number. Without this option, the default is core 0.
--memorytype=*type* **-t***type*	This option allows you to specify the trace memory type. *type* can be TCM, XTM or TRAM.
--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.

Option	Description
--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.

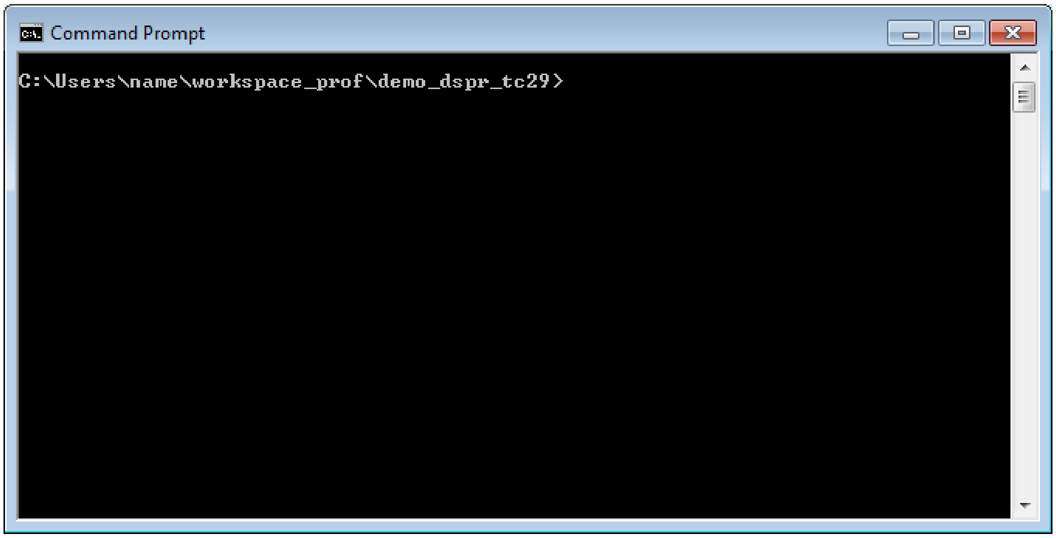
5.2.1. Command Line Tutorial

In this section we use tutorial `demo_dspr_tc29` with the delivered `demo_dspr_tc29.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_tc29`.



```
C:\Users\name\workspace_prof\demo_dspr_tc29>
```

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_tc29` using one shot trace mode, enter:

   ```
   EmbProfCmd --run=PerfAnalysis-1 demo_dspr_tc29.EmbProf
   ```

   The results are exported to the CSV files `demo_dspr_tc29_functions.csv` and `demo_dspr_tc29_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_tc29.EmbProf` file.

2. To run a performance analysis on `demo_dspr_tc29` using one shot trace mode and compare the results with `original`, enter:

   ```
   EmbProfCmd --run=PerfAnalysis-1 --compare=original demo_dspr_tc29.EmbProf
   ```

   The results of the comparison are exported to the CSV file `demo_dspr_tc29_diff_functions.csv`. If all value fields are zero, this indicates that the results are identical. This should be the case with this example.

3. To run a performance analysis on `demo_dspr_tc29` using one shot trace mode and compare the results with `fixed`, enter:

   ```
   EmbProfCmd --run=PerfAnalysis-1 --compare=fixed demo_dspr_tc29.EmbProf
   ```

   The results of the comparison are exported to the CSV file `demo_dspr_tc29_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_tc29.EmbProf
   ```

5. To run an analysis on a specific core, use option **--core=***core-nr*. For the TC29x derivative your can use the values 0, 1 and 2. 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_tc29.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_tc29.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:

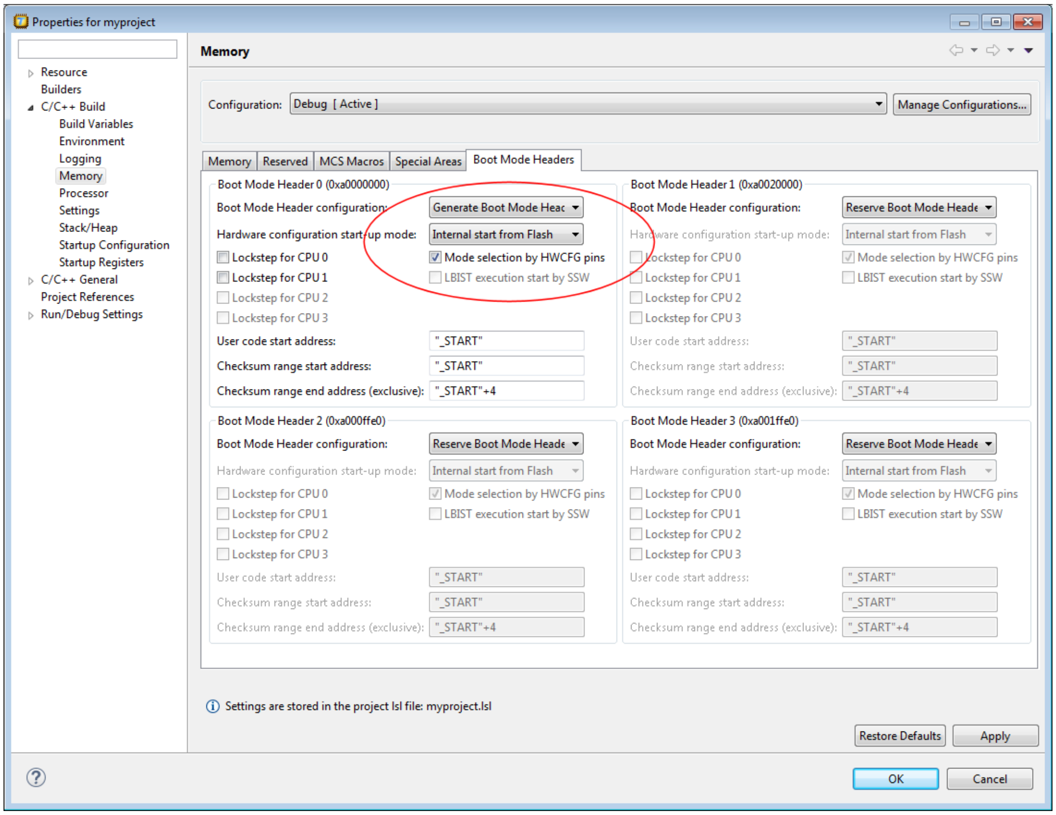
```
Trace error: cannot find code at address address
Do you want to continue the run?
```

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**.



Configuration: Debug [Active]

Boot Mode Header 0 (0xa0000000)
Boot Mode Header configuration: Generate Boot Mode Heac
Hardware configuration start-up mode: Internal start from Flash
☐ Lockstep for CPU 0 ☑ Mode selection by HWCFG pins
☐ Lockstep for CPU 1 ☐ LBIST execution start by SSW
☐ Lockstep for CPU 2
☐ Lockstep for CPU 3
User code start address: "_START"
Checksum range start address: "_START"
Checksum range end address (exclusive): "_START"+4

Boot Mode Header 1 (0xa0020000)
Boot Mode Header configuration: Reserve Boot Mode Heade
Hardware configuration start-up mode: Internal start from Flash
☐ Lockstep for CPU 0 ☑ Mode selection by HWCFG pins
☐ Lockstep for CPU 1 ☐ LBIST execution start by SSW
☐ Lockstep for CPU 2
☐ Lockstep for CPU 3
User code start address: "_START"
Checksum range start address: "_START"
Checksum range end address (exclusive): "_START"+4

Boot Mode Header 2 (0xa000ffe0)
Boot Mode Header configuration: Reserve Boot Mode Heade
Hardware configuration start-up mode: Internal start from Flash
☐ Lockstep for CPU 0 ☑ Mode selection by HWCFG pins
☐ Lockstep for CPU 1 ☐ LBIST execution start by SSW
☐ Lockstep for CPU 2
☐ Lockstep for CPU 3
User code start address: "_START"
Checksum range start address: "_START"
Checksum range end address (exclusive): "_START"+4

Boot Mode Header 3 (0xa001ffe0)
Boot Mode Header configuration: Reserve Boot Mode Heade
Hardware configuration start-up mode: Internal start from Flash
☐ Lockstep for CPU 0 ☑ Mode selection by HWCFG pins
☐ Lockstep for CPU 1 ☐ LBIST execution start by SSW
☐ Lockstep for CPU 2
☐ Lockstep for CPU 3
User code start address: "_START"
Checksum range start address: "_START"
Checksum range end address (exclusive): "_START"+4

Settings are stored in the project lsl file: myproject.lsl

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.

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

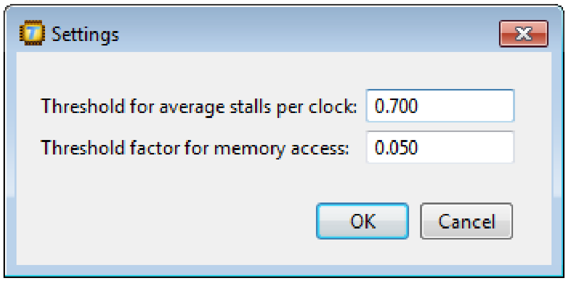
6.1. Settings Dialog

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.



Threshold for average stalls per clock: 0.700
Threshold factor for memory access: 0.050

2. Change the value(s) and click **OK**.

   All results will be updated to reflect the new thresholds.

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:

*Threshold factor for memory access * total DSPR access = Threshold for memory access*

This means, for example, when DSPR0 accesses is 50 and DSPR2 accesses is 31950 the total DSPR access is 32000, and the value of DSPR2 of 31950 will be marked red because it is higher than 0.05*32000=1600. Also other memory accesses that are higher than 1600 will be marked red.

6.2. Summary Tab

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

Performance analysis

- Info
- Performance hotspots
- ICache misses
- DCache misses

Memory access analysis

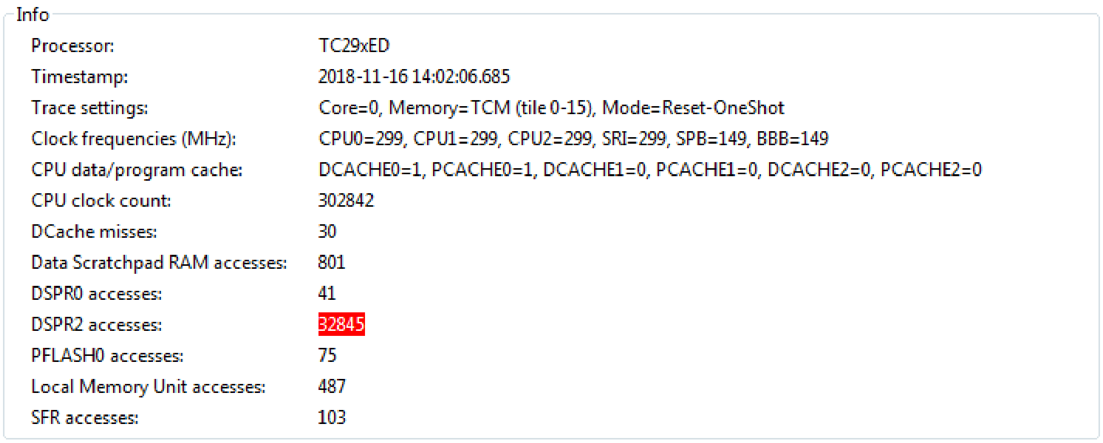
- Info
- Performance hotspots
- DCache misses
- Memory access
- Memory conflicts

Function-level analysis

- Info
- Performance hotspots

6.2.1. Info

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



Info

Processor:	TC29xED
Timestamp:	2018-11-16 14:02:06.685
Trace settings:	Core=0, Memory=TCM (tile 0-15), Mode=Reset-OneShot
Clock frequencies (MHz):	CPU0=299, CPU1=299, CPU2=299, SRI=299, SPB=149, BBB=149
CPU data/program cache:	DCACHE0=1, PCACHE0=1, DCACHE1=0, PCACHE1=0, DCACHE2=0, PCACHE2=0
CPU clock count:	302842
DCache misses:	30
Data Scratchpad RAM accesses:	801
DSPR0 accesses:	41
DSPR2 accesses:	32845
PFLASH0 accesses:	75
Local Memory Unit accesses:	487
SFR accesses:	103

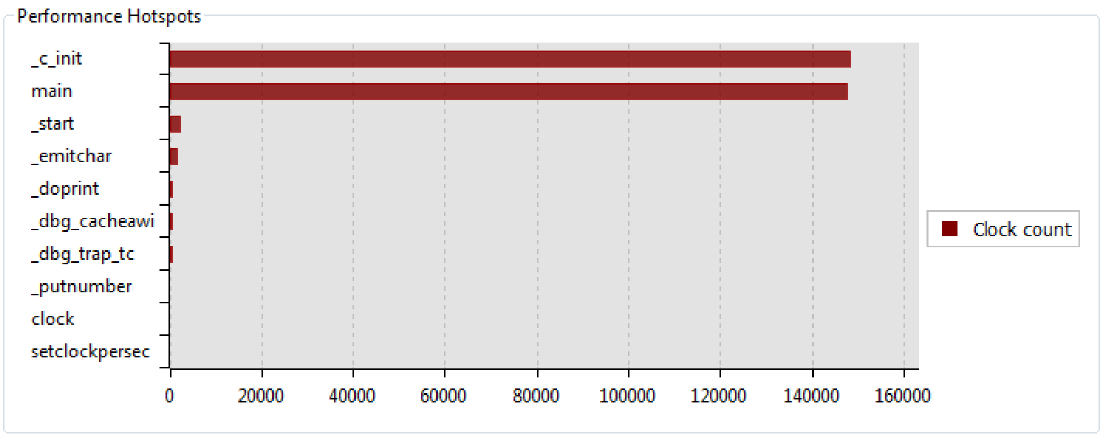
Information	**Description**	**Perf Analysis**	**Mem Analysis**	**Func Analysis**
Processor	The name of the selected processor device	✓	✓	✓
Timestamp	The date and time the analysis was run	✓	✓	✓
Trace settings	The TriCore core (0, 1, 2, ...) the analysis was run for, the trace memory and tile range used and the trace mode	✓	✓	✓
Clock frequencies	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.	✓	✓	✓
CPU data/program cache	The CPU 0, 1, 2, ... data cache (DCache) and program cache (PCache) settings. DCACHE0=1 means CPU0.DCACHE is enabled, PCACHE1=0means CPU1.PCACHE is disabled. The values are read at the start of the analysis before any reset.	✓	✓	✓
CPU clock count	The number of CPU clock cycles on the board it took to run the analysis	✓	✓	✓
Stalls	The number of clock cycles the CPU stalls on branch misses, ICache misses and/or DCache misses	✓		
Average stalls per clock	The average of stalls / CPU clock count	✓		

Information	Description	Perf Analysis	Mem Analysis	Func Analysis
ICache misses	The number of failed attempts to read or write instructions from the instruction cache (ICache)	✓		
DCache misses	The number of failed attempts to read or write data from the data cache (DCache)	✓	✓	
Data Scratchpad RAM accesses	The number of read or write accesses to Data Scratchpad RAM, where the core could not be determined		✓	
DSPR*x* accesses	The number of read or write accesses to Data Scratchpad RAM *x*, where *x* can be 0 .. 5		✓	
PFLASH*x* accesses	The number of read or write accesses to flash memory		✓	
External Bus Unit memory accesses	The number of read or write accesses to the EBU		✓	
Local Memory Unit accesses	The number of read or write accesses to the LMU		✓	
Program Memory Unit accesses	The number of read or write accesses to the PMU		✓	
SFR accesses	The number of read or write accesses to Special Function registers		✓	

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*.

6.2.2. Performance Hotspots

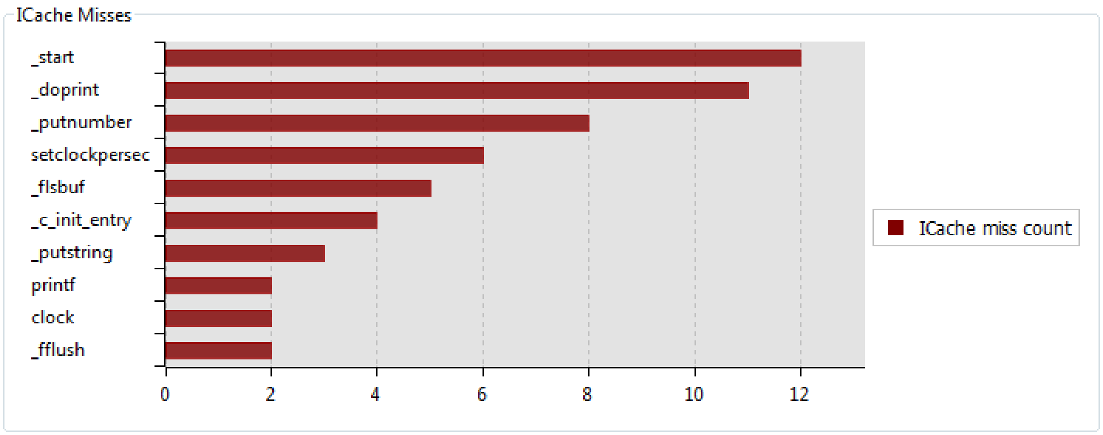
The **Performance hotspots** part of the Summary tab shows the functions with the highest clock count. This chart is available for all analysis types. As you can see in the following example, most of the time is spent in the functions `_c_init` and `main`.



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

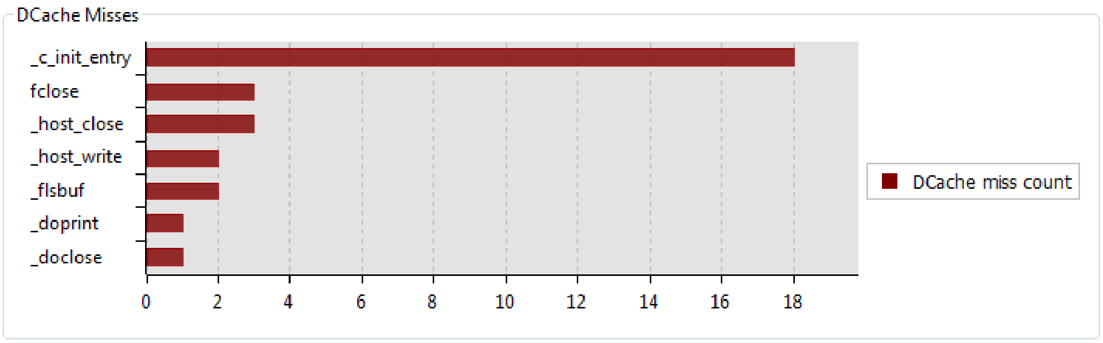
6.2.3. ICache Misses

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



6.2.4. DCache Misses

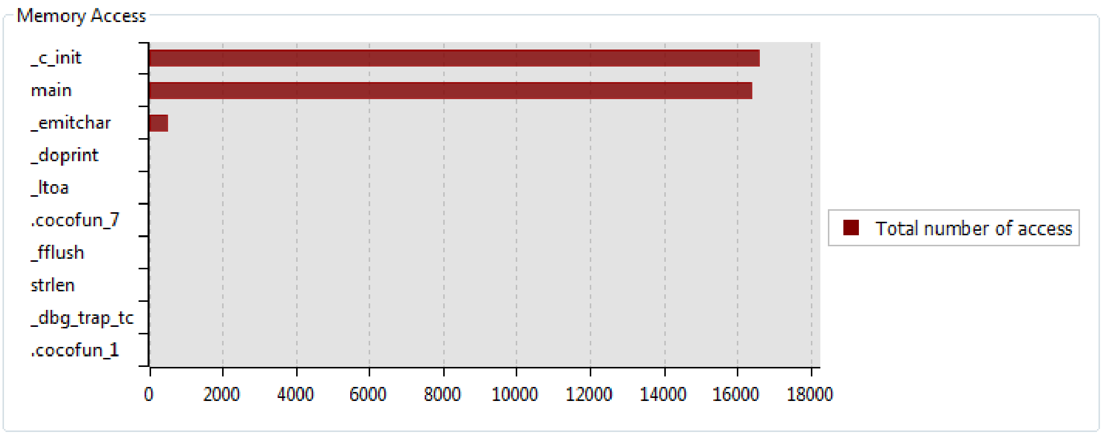
The **DCache misses** part of the Summary tab show a DCache Miss chart. It shows the functions with the highest number of data cache (DCache) misses. This chart is available for performance analyses and memory access analyses.



6.2.5. 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.

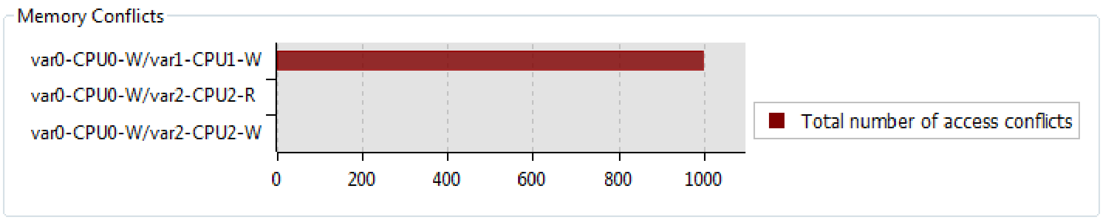


6.2.6. Memory Conflicts

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.



6.3. 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:

Column	Description	Perf Analysis	Mem Analysis	Func Analysis
Function	The name of the measured function	✓	✓	✓
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	✓	✓	✓
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	✓	✓	✓
Clocks With Children	The total number of CPU clocks spent in the function and call tree descendents	✓	✓	✓
Entries	The total number of times the function is called	✓	✓	✓
Avg. Clocks/Entry	The average number of CPU clocks spent in a function per function entry	✓	✓	✓
Max Clocks/Entry	The highest number of CPU clocks spent in a function per function entry	✓	✓	✓
Min Clocks/Entry	The lowest number of CPU clocks spent in a function per function entry	✓	✓	✓
Jitter/Entry	The difference between the highest and lowest number of CPU clocks spent in a function. This is the difference of the previous two columns.	✓	✓	✓

Column	Description	Perf Analysis	Mem Analysis	Func Analysis
Branch Misses	The total number of branch misses	✓		
ICache Misses	The total number of instruction cache misses	✓		
DCache Misses	The total number of data cache misses	✓	✓	
Stalls	The total number of stalls due to memory access delays or pipeline hazards	✓		

6.4. 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:

Column	Description
Line	The source line number, function name and relative path to the source file where the problem occurred
Clocks	The total number of CPU clocks spent on the source line
Branch Misses	The total number of branch misses
ICache Misses	The total number of instruction cache misses
DCache Misses	The total number of data cache misses
Stalls	The total number of stalls due to memory access delays or pipeline hazards

6.5. 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:

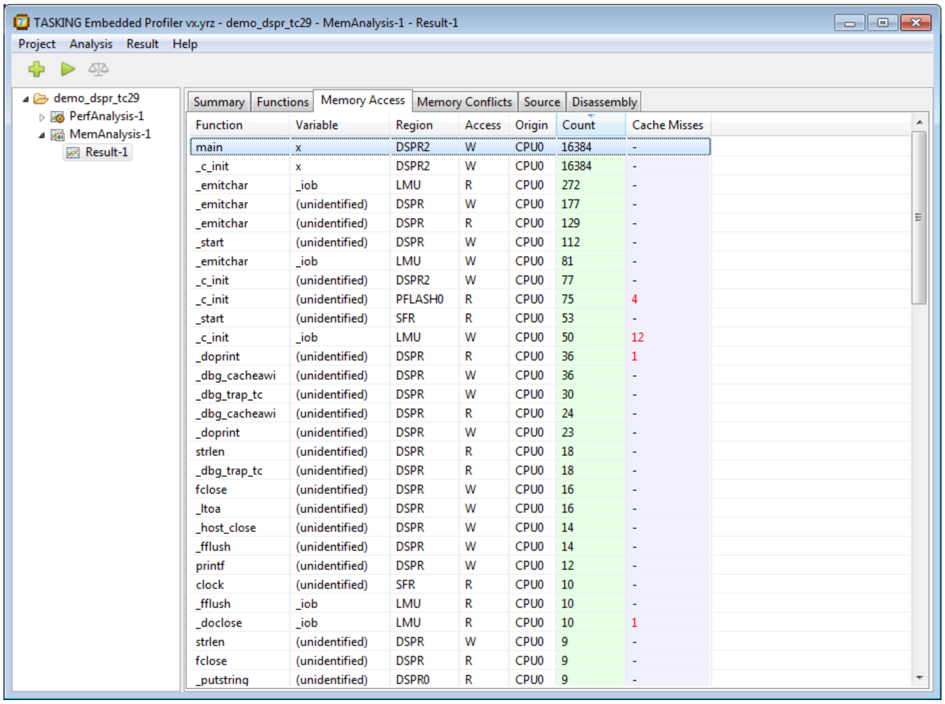
Column	Description
Address	The instruction address and function name where the problem occurred
Clocks	The total number of CPU clocks spent on the instruction

Column	Description
Branch Misses	The total number of branch misses
ICache Misses	The total number of instruction cache misses
DCache Misses	The total number of data cache misses
Stalls	The total number of stalls due to memory access delays or pipeline hazards

6.6. 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.



Function	Variable	Region	Access	Origin	Count	Cache Misses
main	x	DSPR2	W	CPU0	16384	-
_c_init	x	DSPR2	W	CPU0	16384	-
_emitchar	_iob	LMU	R	CPU0	272	-
_emitchar	(unidentified)	DSPR	W	CPU0	177	-
_emitchar	(unidentified)	DSPR	R	CPU0	129	-
_start	(unidentified)	DSPR	W	CPU0	112	-
_emitchar	_iob	LMU	W	CPU0	81	-
_c_init	(unidentified)	DSPR2	W	CPU0	77	-
_c_init	(unidentified)	PFLASH0	R	CPU0	75	4
_start	(unidentified)	SFR	R	CPU0	53	-
_c_init	_iob	LMU	W	CPU0	50	12
_doprint	(unidentified)	DSPR	R	CPU0	36	1
_dbg_cacheawi	(unidentified)	DSPR	W	CPU0	36	-
_dbg_trap_tc	(unidentified)	DSPR	W	CPU0	30	-
_dbg_cacheawi	(unidentified)	DSPR	R	CPU0	24	-
_doprint	(unidentified)	DSPR	W	CPU0	23	-
strlen	(unidentified)	DSPR	R	CPU0	18	-
_dbg_trap_tc	(unidentified)	DSPR	R	CPU0	18	-
fclose	(unidentified)	DSPR	W	CPU0	16	-
_ltoa	(unidentified)	DSPR	W	CPU0	16	-
_host_close	(unidentified)	DSPR	W	CPU0	14	-
_fflush	(unidentified)	DSPR	W	CPU0	14	-
printf	(unidentified)	DSPR	W	CPU0	12	-
clock	(unidentified)	SFR	R	CPU0	10	-
_fflush	_iob	LMU	R	CPU0	10	-
_doclose	_iob	LMU	R	CPU0	10	1
strlen	(unidentified)	DSPR	W	CPU0	9	-
fclose	(unidentified)	DSPR	R	CPU0	9	-
_putstring	(unidentified)	DSPR0	R	CPU0	9	-

The Memory Access tab contains the following information:

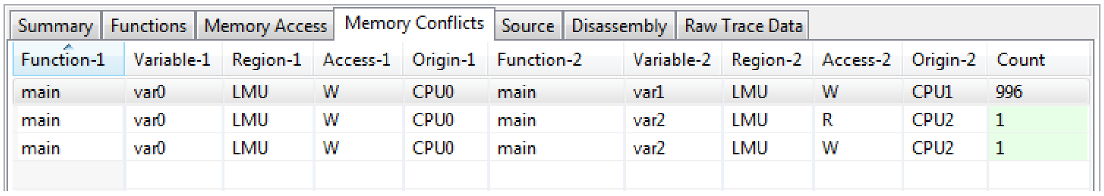
Column	Description
Function	The name of the function that contains the global variable.

Column	Description
Variable	The name of the global variable, if the address is associated with a variable, otherwise "(unidentified)" 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 **--emit-locals=+symbols**, or in Eclipse select **Project » Properties for » C/C++ Build » Settings » Tool Settings » Assembler » Symbols » Emit local non-EQU symbols**.
Region	The name of the memory
Access	The type of access read (R) or write (W)
Origin	The core from which the conflicting access originated
Count	The number of accesses
Cache Misses	The number of cache misses for this specific access

6.7. 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.



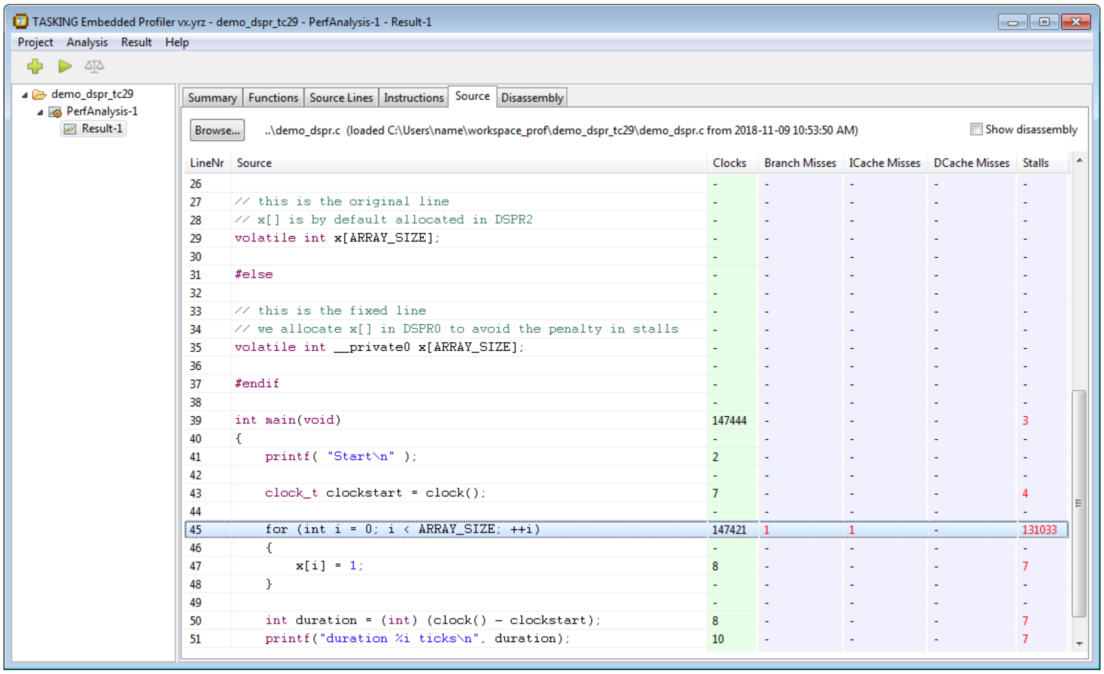
Function-1	Variable-1	Region-1	Access-1	Origin-1	Function-2	Variable-2	Region-2	Access-2	Origin-2	Count
main	var0	LMU	W	CPU0	main	var1	LMU	W	CPU1	996
main	var0	LMU	W	CPU0	main	var2	LMU	R	CPU2	1
main	var0	LMU	W	CPU0	main	var2	LMU	W	CPU2	1

The Memory Conflicts tab contains the following information:

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

6.8. 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.



..\demo_dspr.c (loaded C:\Users\name\workspace_prof\demo_dspr_tc29\demo_dspr.c from 2018-11-09 10:53:50 AM)

LineNr	Source	Clocks	Branch Misses	ICache Misses	DCache Misses	Stalls
26		-	-	-	-	-
27	// this is the original line	-	-	-	-	-
28	// x[] is by default allocated in DSPR2	-	-	-	-	-
29	volatile int x[ARRAY_SIZE];	-	-	-	-	-
30		-	-	-	-	-
31	#else	-	-	-	-	-
32		-	-	-	-	-
33	// this is the fixed line	-	-	-	-	-
34	// we allocate x[] in DSPR0 to avoid the penalty in stalls	-	-	-	-	-
35	volatile int __private0 x[ARRAY_SIZE];	-	-	-	-	-
36		-	-	-	-	-
37	#endif	-	-	-	-	-
38		-	-	-	-	-
39	int main(void)	147444	-	-	-	3
40	{	-	-	-	-	-
41	printf("Start\n");	2	-	-	-	-
42		-	-	-	-	-
43	clock_t clockstart = clock();	7	-	-	-	4
44		-	-	-	-	-
45	for (int i = 0; i < ARRAY_SIZE; ++i)	147421	1	1	-	131033
46	{	-	-	-	-	-
47	x[i] = 1;	8	-	-	-	7
48	}	-	-	-	-	-
49		-	-	-	-	-
50	int duration = (int) (clock() - clockstart);	8	-	-	-	7
51	printf("duration %i ticks\n", duration);	10	-	-	-	7

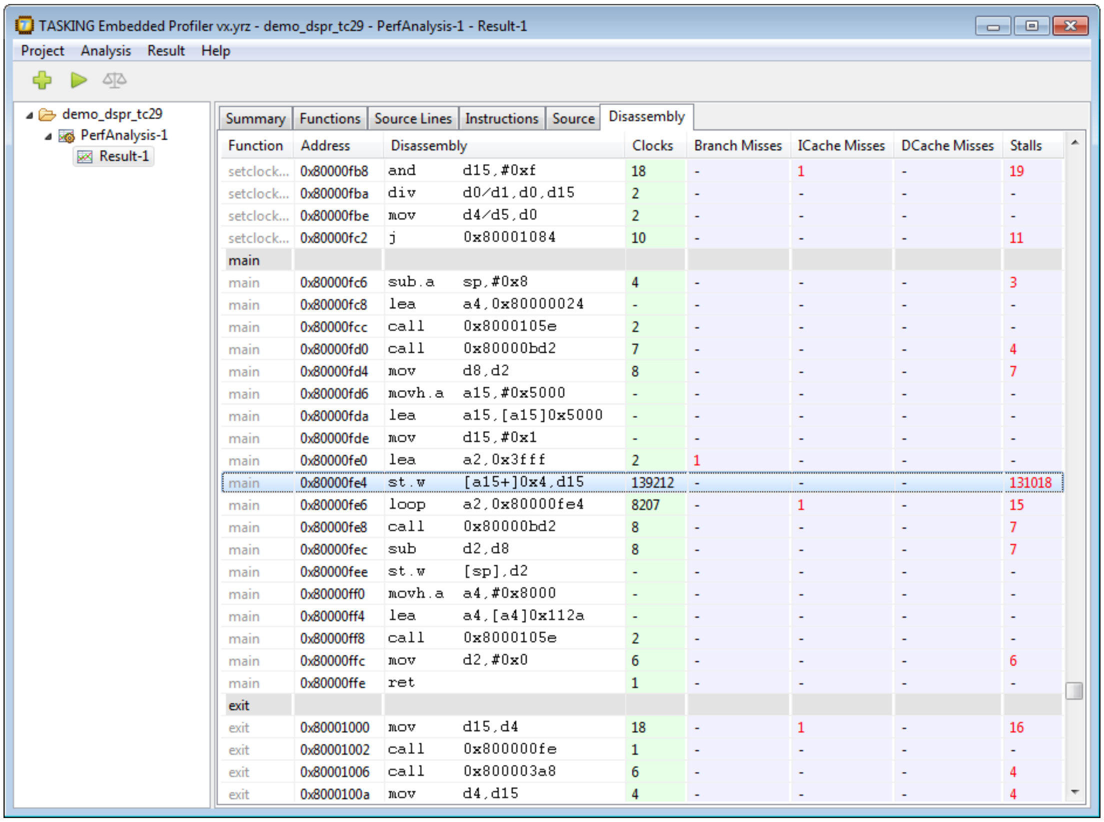
The columns are the same as explained in Section 6.4, *Source Lines Tab*. Red values indicate a miss or a stall. Hover the mouse over a value to see additional information.

With the **Browse** button you can open another source file.

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

6.9. 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.


TASKING Embedded Profiler vx.yrz - demo_dspr_tc29 - PerfAnalysis-1 - Result-1
Project Analysis Result Help
demo_dspr_tc29
PerfAnalysis-1
Result-1
Summary Functions Source Lines Instructions Source Disassembly

Function	Address	Disassembly	Clocks	Branch Misses	ICache Misses	DCache Misses	Stalls
setclock...	0x80000fb8	and d15,#0xf	18	-	1	-	19
setclock...	0x80000fba	div d0/d1,d0,d15	2	-	-	-	-
setclock...	0x80000fbe	mov d4/d5,d0	2	-	-	-	-
setclock...	0x80000fc2	j 0x80001084	10	-	-	-	11
main							
main	0x80000fc6	sub.a sp,#0x8	4	-	-	-	3
main	0x80000fc8	lea a4,0x80000024	-	-	-	-	-
main	0x80000fcc	call 0x8000105e	2	-	-	-	-
main	0x80000fd0	call 0x80000bd2	7	-	-	-	4
main	0x80000fd4	mov d8,d2	8	-	-	-	7
main	0x80000fd6	movh.a a15,#0x5000	-	-	-	-	-
main	0x80000fda	lea a15,[a15]0x5000	-	-	-	-	-
main	0x80000fde	mov d15,#0x1	-	-	-	-	-
main	0x80000fe0	lea a2,0x3fff	2	1	-	-	-
main	0x80000fe4	st.w [a15+]0x4,d15	139212	-	-	-	131018
main	0x80000fe6	loop a2,0x80000fe4	8207	-	1	-	15
main	0x80000fe8	call 0x80000bd2	8	-	-	-	7
main	0x80000fec	sub d2,d8	8	-	-	-	7
main	0x80000fee	st.w [sp],d2	-	-	-	-	-
main	0x80000ff0	movh.a a4,#0x8000	-	-	-	-	-
main	0x80000ff4	lea a4,[a4]0x112a	-	-	-	-	-
main	0x80000ff8	call 0x8000105e	2	-	-	-	-
main	0x80000ffc	mov d2,#0x0	6	-	-	-	6
main	0x80000ffe	ret	1	-	-	-	-
exit							
exit	0x80001000	mov d15,d4	18	-	1	-	16
exit	0x80001002	call 0x800000fe	1	-	-	-	-
exit	0x80001006	call 0x800003a8	6	-	-	-	4
exit	0x8000100a	mov d4,d15	4	-	-	-	4

The columns are the same as explained in Section 6.5, *Instructions Tab*. Red values indicate a miss or a stall. Hover the mouse over a value to see additional information.

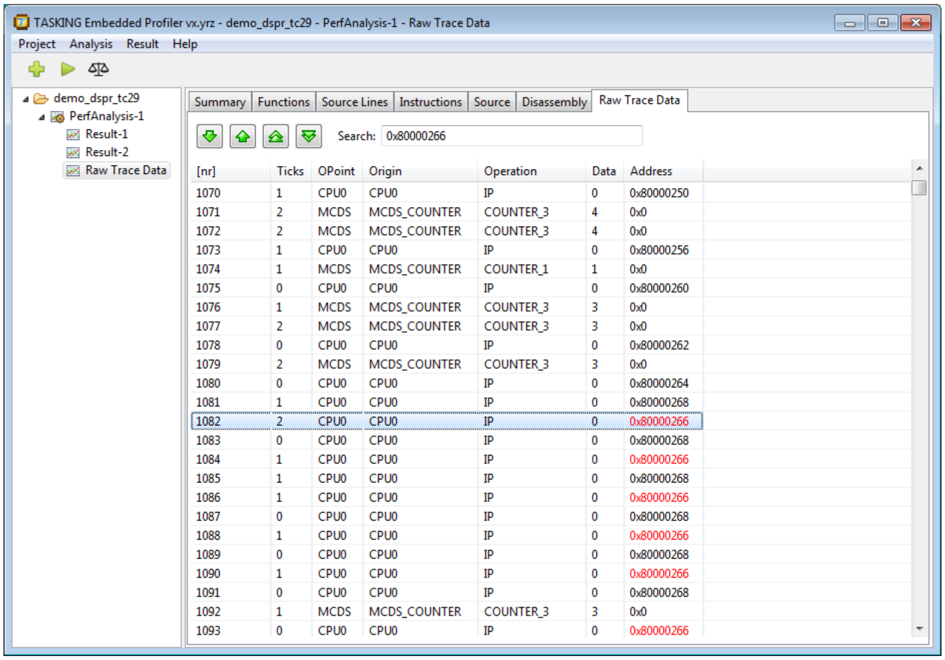
If you double-click on a row, the Raw Trace Data tab, if present, opens at the selected address.

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

6.10. Raw Trace Data Tab

The Raw Trace Data tab 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. This tab is available for all analysis types, but only when you enable **Save and display** raw trace data in the **Run Analysis** dialog.

Hover the mouse over a value to see additional information.



Search: 0x80000266

[nr]	Ticks	OPoint	Origin	Operation	Data	Address
1070	1	CPU0	CPU0	IP	0	0x80000250
1071	2	MCDS	MCDS_COUNTER	COUNTER_3	4	0x0
1072	2	MCDS	MCDS_COUNTER	COUNTER_3	4	0x0
1073	1	CPU0	CPU0	IP	0	0x80000256
1074	1	MCDS	MCDS_COUNTER	COUNTER_1	1	0x0
1075	0	CPU0	CPU0	IP	0	0x80000260
1076	1	MCDS	MCDS_COUNTER	COUNTER_3	3	0x0
1077	2	MCDS	MCDS_COUNTER	COUNTER_3	3	0x0
1078	0	CPU0	CPU0	IP	0	0x80000262
1079	2	MCDS	MCDS_COUNTER	COUNTER_3	3	0x0
1080	0	CPU0	CPU0	IP	0	0x80000264
1081	1	CPU0	CPU0	IP	0	0x80000268
1082	2	CPU0	CPU0	IP	0	0x80000266
1083	0	CPU0	CPU0	IP	0	0x80000268
1084	1	CPU0	CPU0	IP	0	0x80000266
1085	1	CPU0	CPU0	IP	0	0x80000268
1086	1	CPU0	CPU0	IP	0	0x80000266
1087	0	CPU0	CPU0	IP	0	0x80000268
1088	1	CPU0	CPU0	IP	0	0x80000266
1089	0	CPU0	CPU0	IP	0	0x80000268
1090	1	CPU0	CPU0	IP	0	0x80000266
1091	0	CPU0	CPU0	IP	0	0x80000268
1092	1	MCDS	MCDS_COUNTER	COUNTER_3	3	0x0
1093	0	CPU0	CPU0	IP	0	0x80000266

In the **Search** field you can enter an address to search for. All matches are marked red. With the buttons you can navigate to the Next, Previous, First or Last occurrence.

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

The Raw Trace Data tab contains the following information:

Column	Description
[nr]	The sequence number for every raw trace operation.
Ticks	The MCDS clock Ticks between trace messages. Please note that one Tick is equal to two CPU cycles.
OPoint	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.
Origin	The origin of the activity. In most cases this is the same as OPoint.
Operation	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*.
Data	The data written or read.
Address	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.

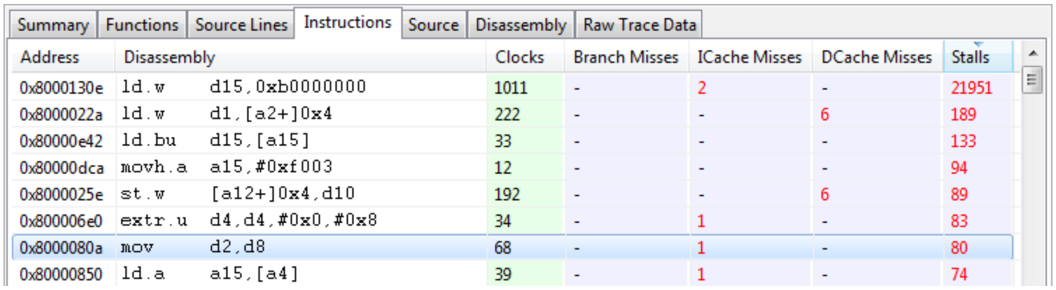
Example how to use raw trace data for analysis

1. Import the `demo_concurrent` example.

2. Run a **One shot mode** performance analysis with **Save and display** raw trace data enabled.

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

   Notice that near the top is a `mov d2, d8` instruction with a value of 80 stalls at address `0x8000080a`.

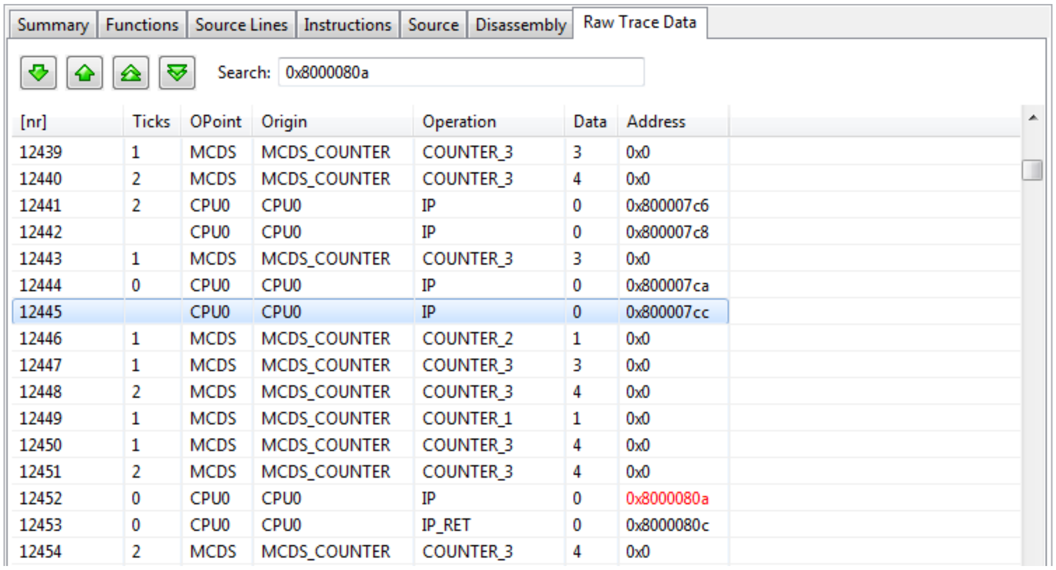


Summary | Functions | Source Lines | Instructions | Source | Disassembly | Raw Trace Data

Address	Disassembly		Clocks	Branch Misses	ICache Misses	DCache Misses	Stalls
0x8000130e	ld.w	d15,0xb0000000	1011	-	2	-	21951
0x8000022a	ld.w	d1,[a2+]0x4	222	-	-	6	189
0x80000e42	ld.bu	d15,[a15]	33	-	-	-	133
0x80000dca	movh.a	a15,#0xf003	12	-	-	-	94
0x8000025e	st.w	[a12+]0x4,d10	192	-	-	6	89
0x800006e0	extr.u	d4,d4,#0x0,#0x8	34	-	1	-	83
0x8000080a	mov	d2,d8	68	-	1	-	80
0x80000850	ld.a	a15,[a4]	39	-	1	-	74

4. In the **Raw Trace Data** tab, enter the address `0x8000080a` 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 `0x800007cc`.

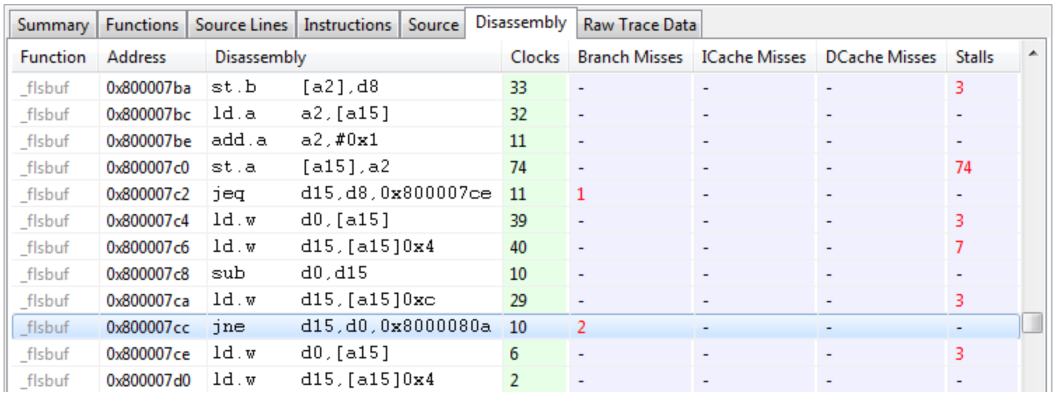


Summary | Functions | Source Lines | Instructions | Source | Disassembly | Raw Trace Data

Search: 0x8000080a

[nr]	Ticks	OPoint	Origin	Operation	Data	Address
12439	1	MCDS	MCDS_COUNTER	COUNTER_3	3	0x0
12440	2	MCDS	MCDS_COUNTER	COUNTER_3	4	0x0
12441	2	CPU0	CPU0	IP	0	0x800007c6
12442		CPU0	CPU0	IP	0	0x800007c8
12443	1	MCDS	MCDS_COUNTER	COUNTER_3	3	0x0
12444	0	CPU0	CPU0	IP	0	0x800007ca
12445		CPU0	CPU0	IP	0	0x800007cc
12446	1	MCDS	MCDS_COUNTER	COUNTER_2	1	0x0
12447	1	MCDS	MCDS_COUNTER	COUNTER_3	3	0x0
12448	2	MCDS	MCDS_COUNTER	COUNTER_3	4	0x0
12449	1	MCDS	MCDS_COUNTER	COUNTER_1	1	0x0
12450	1	MCDS	MCDS_COUNTER	COUNTER_3	4	0x0
12451	2	MCDS	MCDS_COUNTER	COUNTER_3	4	0x0
12452	0	CPU0	CPU0	IP	0	0x8000080a
12453	0	CPU0	CPU0	IP_RET	0	0x8000080c
12454	2	MCDS	MCDS_COUNTER	COUNTER_3	4	0x0

5. Double-click on the address and the view will switch to the **Disassembly** tab at the specified address, in this case `jne d15, d0, 0x8000080a`.



Function	Address	Disassembly		Clocks	Branch Misses	ICache Misses	DCache Misses	Stalls
_flsbuf	0x800007ba	st.b	[a2],d8	33	-	-	-	3
_flsbuf	0x800007bc	ld.a	a2,[a15]	32	-	-	-	-
_flsbuf	0x800007be	add.a	a2,#0x1	11	-	-	-	-
_flsbuf	0x800007c0	st.a	[a15],a2	74	-	-	-	74
_flsbuf	0x800007c2	jeq	d15,d8,0x800007ce	11	1	-	-	-
_flsbuf	0x800007c4	ld.w	d0,[a15]	39	-	-	-	3
_flsbuf	0x800007c6	ld.w	d15,[a15]0x4	40	-	-	-	7
_flsbuf	0x800007c8	sub	d0,d15	10	-	-	-	-
_flsbuf	0x800007ca	ld.w	d15,[a15]0xc	29	-	-	-	3
_flsbuf	0x800007cc	jne	d15,d0,0x8000080a	10	2	-	-	-
_flsbuf	0x800007ce	ld.w	d0,[a15]	6	-	-	-	3
_flsbuf	0x800007d0	ld.w	d15,[a15]0x4	2	-	-	-	-