Arm® Streamline

Version 7.4

Target Setup Guide for Bare-metal Applications

arm

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
101815_0704_00_en

Arm® Streamline

Target Setup Guide for Bare-metal Applications

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.

Release Information

Document History

Issue	Date	Confidentiality	Change
0701-00	25 September 2019	Non-Confidential	New document for v7.1.
0701-01	30 October 2019	Non-Confidential	Updated document for v7.1.
0702-00	14 February 2020	Non-Confidential	New document for v7.2.
0703-00	29 May 2020	Non-Confidential	New document for v7.3.
0704-00	21 August 2020	Non-Confidential	New document for v7.4.

Non-Confidential Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. **No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.**

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, third party patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Arm, then the click through or signed written agreement prevails over and supersedes the conflicting provisions of these terms. This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm's trademark usage guidelines at *http://www.arm.com/company/policies/trademarks*.

Copyright © 2019, 2020 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Web Address

developer.arm.com

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Contents

Arm® Streamline Target Setup Guide for Bare-metal Applications

Preface

About this book 7

Chapter 1 Bare-metal Support

1.1 Bare-metal support overview 1-10

Chapter 2 Profiling with the bare-metal agent

2.1 Profiling with Barman 2-12
2.2 Data storage 2-13
2.3 Profiling with on-target RAM buffer 2-14
2.4 Profiling with System Trace Macrocell 2-20
2.5 Profiling with Instrumentation Trace Macrocell 2-23
2.6 Profiling with Embedded Trace Macrocell 2-25
2.7 Interfacing with Barman 2-29
2.8 Custom counters 2-40
2.9 Using the bare-metal generation mechanism from the command line 2-42

Chapter 3 Profiling with Instruction Trace

3.1 Importing instruction trace 3-44
3.2 Instruction trace notes and restrictions 3-47

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter 4 **Examples**
4.1 Examples using Barman 4-49

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Preface

This preface introduces the *Arm® Streamline Target Setup Guide for Bare-metal Applications*.

It contains the following:

- *About this book* on page 7.

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

About this book

This book describes how to set up Arm® Streamline on a bare-metal target.

Using this book

This book is organized into the following chapters:

Chapter 1 Bare-metal Support

Describes the bare-metal support available within Streamline.

Chapter 2 Profiling with the bare-metal agent

Chapter 3 Profiling with Instruction Trace

This section describes how to import an instruction trace, and restrictions related to this task.

Chapter 4 Examples

Glossary

The Arm® Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the *Arm® Glossary* for more information.

Typographic conventions

italic

Introduces special terminology, denotes cross-references, and citations.

bold

Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`

Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

`mono`space

Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

`monospace italic`

Denotes arguments to monospace text where the argument is to be replaced by a specific value.

`monospace bold`

Denotes language keywords when used outside example code.

`<and>`

Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:

```
MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

SMALL CAPITALS

Used in body text for a few terms that have specific technical meanings, that are defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Feedback

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to *errata@arm.com*. Give:

- The title *Arm Streamline Target Setup Guide for Bare-metal Applications*.
- The number 101815_0704_00_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Other information

- *Arm® Developer*.
- *Arm® Information Center*.
- *Arm® Technical Support Knowledge Articles*.
- *Technical Support*.
- *Arm® Glossary*.

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter 1
Bare-metal Support

Describes the bare-metal support available within Streamline.

It contains the following section:

- *1.1 Bare-metal support overview* on page 1-10.

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1.1 Bare-metal support overview

Bare-metal support allows Streamline to visualize elements of the system state of a target device that is running with no operating system or a light-weight real-time operating system.

For bare-metal support, you can profile your application using either of the following:

- The agent *Barman*.
- Instruction Trace.

Barman consists of two C source files that you build into the executable that runs on the target device. A configuration and generation utility generates these files.

To profile with Instruction Trace, import a trace of the instructions that your application executed. Streamline can then analyze this trace.

Related references

2.1 Profiling with Barman on page 2-12
Chapter 3 Profiling with Instruction Trace on page 3-43

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter 2
Profiling with the bare-metal agent

It contains the following sections:

- *2.1 Profiling with Barman* on page 2-12.
- *2.2 Data storage* on page 2-13.
- *2.3 Profiling with on-target RAM buffer* on page 2-14.
- *2.4 Profiling with System Trace Macrocell* on page 2-20.
- *2.5 Profiling with Instrumentation Trace Macrocell* on page 2-23.
- *2.6 Profiling with Embedded Trace Macrocell* on page 2-25.
- *2.7 Interfacing with Barman* on page 2-29.
- *2.8 Custom counters* on page 2-40.
- *2.9 Using the bare-metal generation mechanism from the command line* on page 2-42.

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.1 Profiling with Barman

Barman consists of two C source files, `barman.c` and `barman.h`, that you build into the executable that runs on the target device. A configuration and generation utility generates these files.

To use Barman, you must modify your existing executable to do the following:

- Initialize Barman at runtime.
- Periodically call the data collection routines that Barman provides.
- Optionally, stop the capture.
- Optionally, extract the raw data that Barman collects and provide it to Streamline for analysis.

Barman has the following features:

- It captures PMU counter values from Cortex®-A and Cortex-R class processors.
- It captures sampled PC values.
- It captures custom counters.
- It allows you to control the sample rate.
- It writes the data that it collects to memory.
- It has low data collection overhead.

Barman supports the following Arm architectures:

- Armv7-A
- Armv7-R
- Armv7-M
- Armv8-A, both AArch32 and AArch64.
- Armv8-R
- Armv8-M

Warning

Barman is only intended for use in a development environment. Arm does not recommend including Barman in a released product without performing a security audit of the source code.

Related concepts

2.2 Data storage on page 2-13

2.7 Interfacing with Barman on page 2-29

Related references

2.3 Profiling with on-target RAM buffer on page 2-14

2.4 Profiling with System Trace Macrocell on page 2-20

2.5 Profiling with Instrumentation Trace Macrocell on page 2-23

2.6 Profiling with Embedded Trace Macrocell on page 2-25

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.2 Data storage

Barman uses a simple abstraction layer for handling the storage of collected data. Typically, the data that Barman collects is stored in a RAM buffer on the target.

You can choose from the following data storage modes provided:

Linear RAM buffer mode
Data collection stops when the buffer is full. This mode ensures that no collected data is lost, but no further data can be recorded.

Circular RAM buffer mode
Data collection continues after the buffer is full and the oldest data is lost as the newest data overwrites it. This mode gives you control over when the data collection ends.

STM Interface
System Trace Macrocell (STM) data is collected on a DSTREAM device that is connected to the target, or by another similar method. You then dump the STM trace into a host directory, which you can import into Streamline for analysis.

ITM Interface
Instrumentation Trace Macrocell (ITM) data is collected on a DSTREAM device that is connected to the target, or by another similar method. You then dump the ITM trace into a host directory, which you can import into Streamline for analysis.

ETM Interface
Embedded Trace Macrocell (ETM) data is collected on a DSTREAM device that is connected to the target, or by another similar method. You then dump the ETM trace into a host directory, which you can import into Streamline for analysis.

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.3 Profiling with on-target RAM buffer

For Barman to be able to use either of the RAM buffer modes, you must provide the RAM buffer on the target device. The RAM buffer is a dedicated, contiguous area of RAM that Barman can write data to.

On multiprocessor systems, the RAM buffer must be at the same address for all processors. It is your responsibility to allocate memory for the RAM buffer, either statically or dynamically.

This section describes how to collect profiling data using the RAM buffer on the target device.

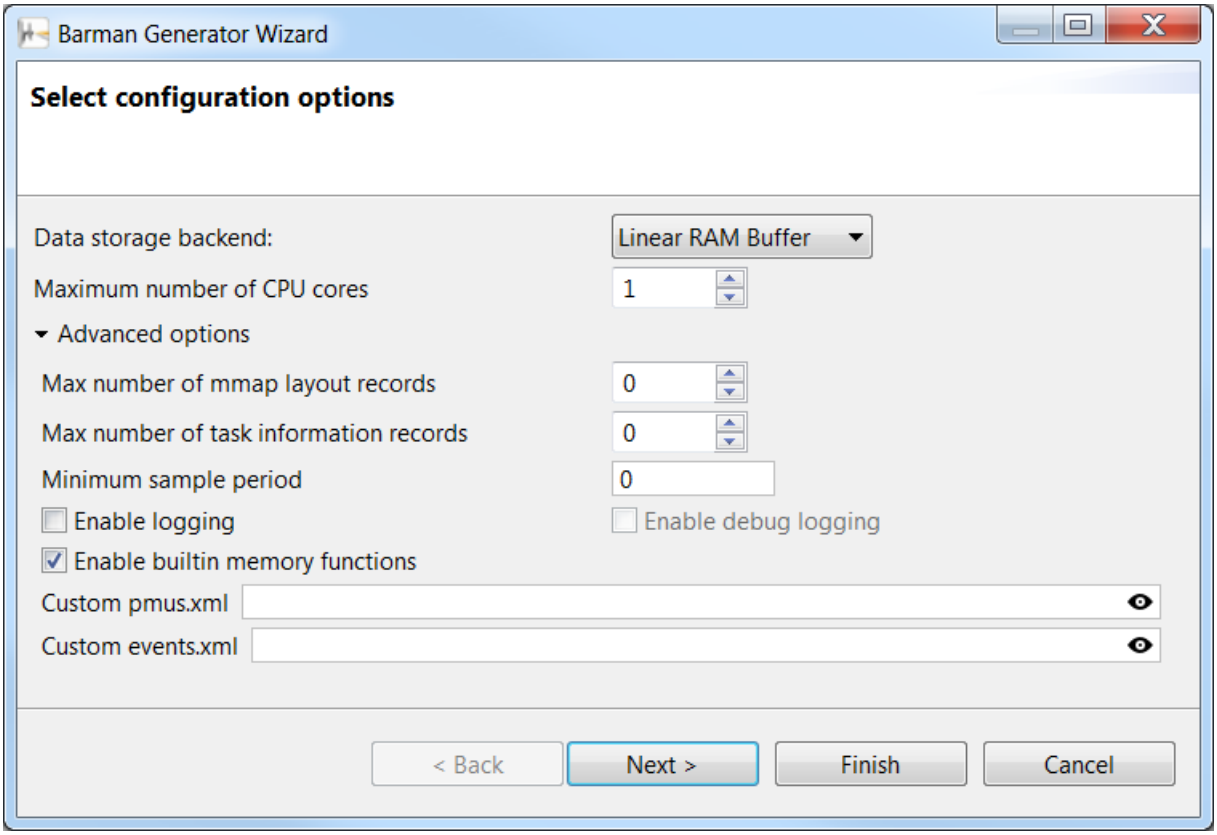
2.3.1 Configuring Barman

You must configure Barman with the configuration and generation utility before you compile the binary executable to be analyzed. Barman must then be built into the executable.

The configuration and generation utility is a wizard dialog available from the **Streamline** menu. The generated header and source files, and the configuration XML file, are then saved into a folder of your choice. The generation mechanism is also accessible from the command line.

Procedure

1. Access this utility from **Streamline** > **Generate Barman Sources**.
2. Configure the default configuration options, such as:
   - The number of processor elements.
   - Whether you intend to supply executable image memory map information.
   - Whether you intend to provide process or task level information (for example if you are running an RTOS).
   - The data storage mode (linear or circular RAM buffer).



Barman uses statically allocated, fixed sized headers for information such as details of the active processors on the system, and task, thread, and process information.

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Max number of mmap layout records and **Max number of task information records** are the maximum amount of space in the header for storing the task, thread, and process information. For example, if you have an RTOS with a fixed number of threads, specify the number of threads here. **Max number of mmap layout records** specifies the number of address mapping entries for mapping sections of the ELF image to addresses in memory. If you have a single ELF image that is physically mapped to memory, leave this value as zero.

The **Minimum sample period** is the minimum time in nanoseconds between samples. Set this value to be an integer multiple of the timer sampling rate. For example, if you have a fixed timer interrupt operating at 1000Hz, but due to memory constraints you want to sample at 100Hz, set **Minimum sample period** to 10000000. This value ensures that there is at least 10ms between each sample.

To provide your own implementation of the memory functions for Barman, for example `memcpy` and `memset`, deselect **Enable builtin memory functions**.

Note

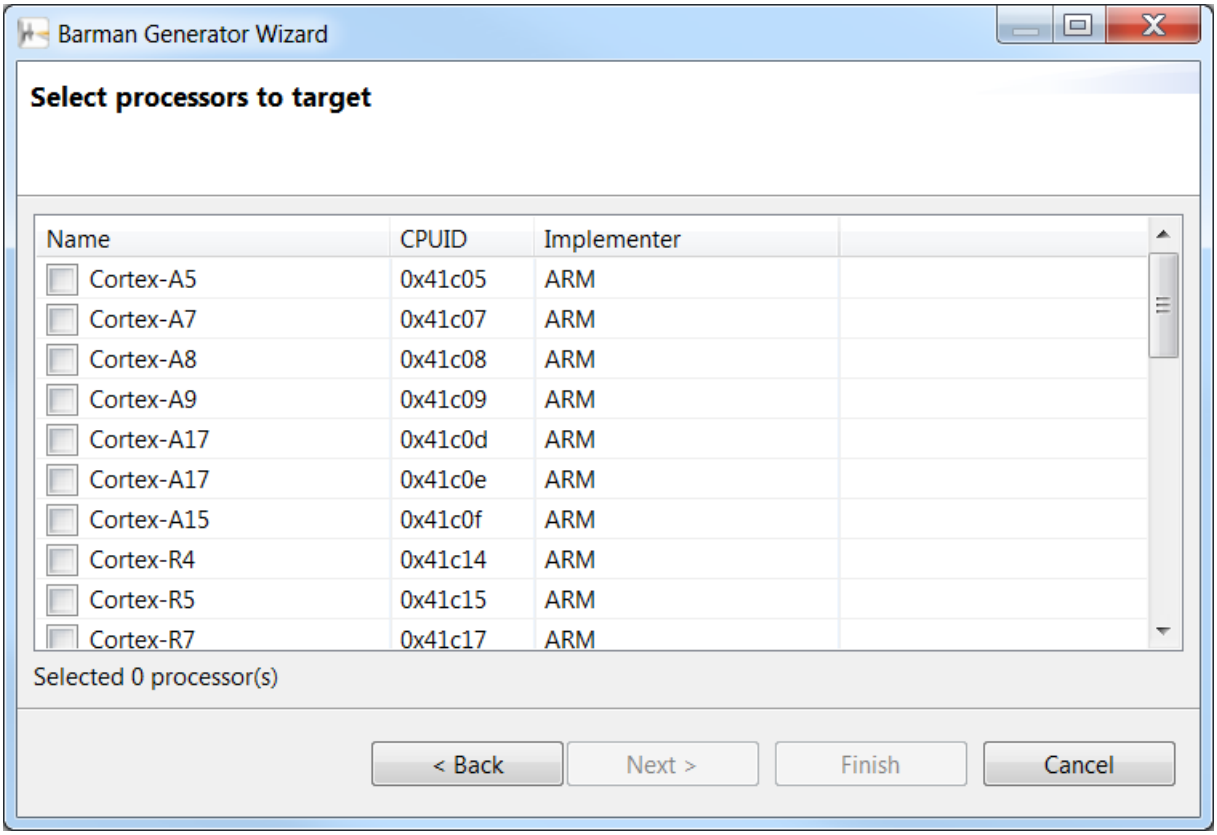
See *2.4 Profiling with System Trace Macrocell* on page 2-20 for information about using the **STM Interface** data storage backend.

See *2.5 Profiling with Instrumentation Trace Macrocell* on page 2-23 for information about using the **ITM Interface** data storage backend.

See *2.6 Profiling with Embedded Trace Macrocell* on page 2-25 for information about using the **ETM Interface** data storage backend.

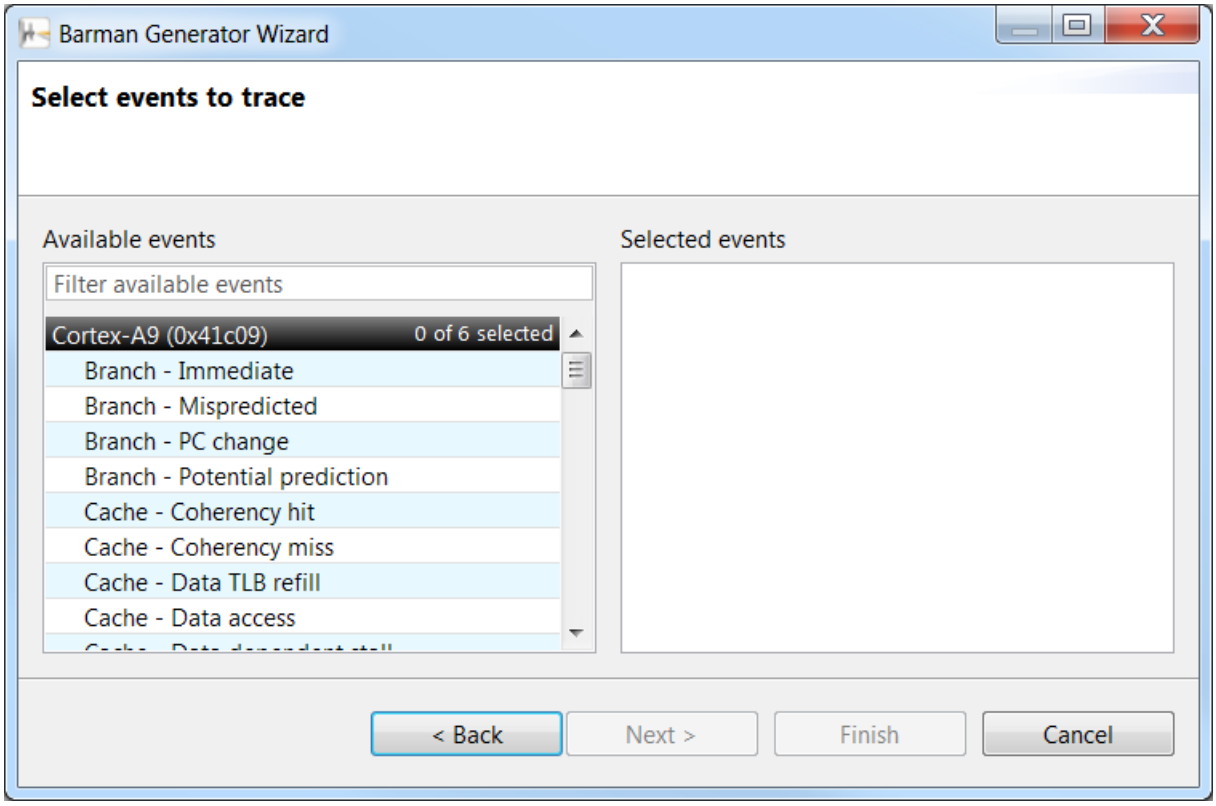
See the gator protocol documentation in `<install_directory>/sw/streamline/protocol/gator/` for more information about `pmus.xml` and `events.xml`.

3. Select the target processor from the pre-defined list.

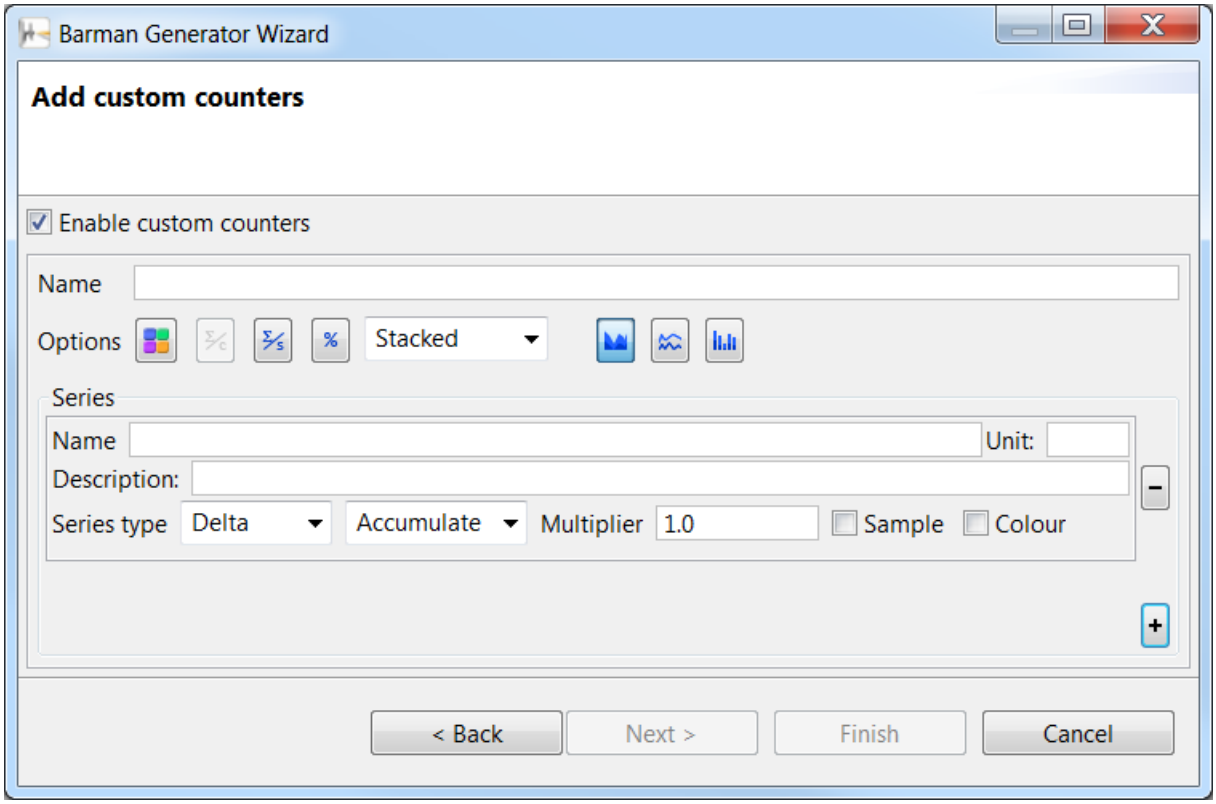


Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4. Select the PMU counters to collect during the capture session by double-clicking on them in the **Available events** list. Alternatively you can drag and drop the events into the **Selected events** list. To deselect events, drag and drop them back into the **Available events** list.

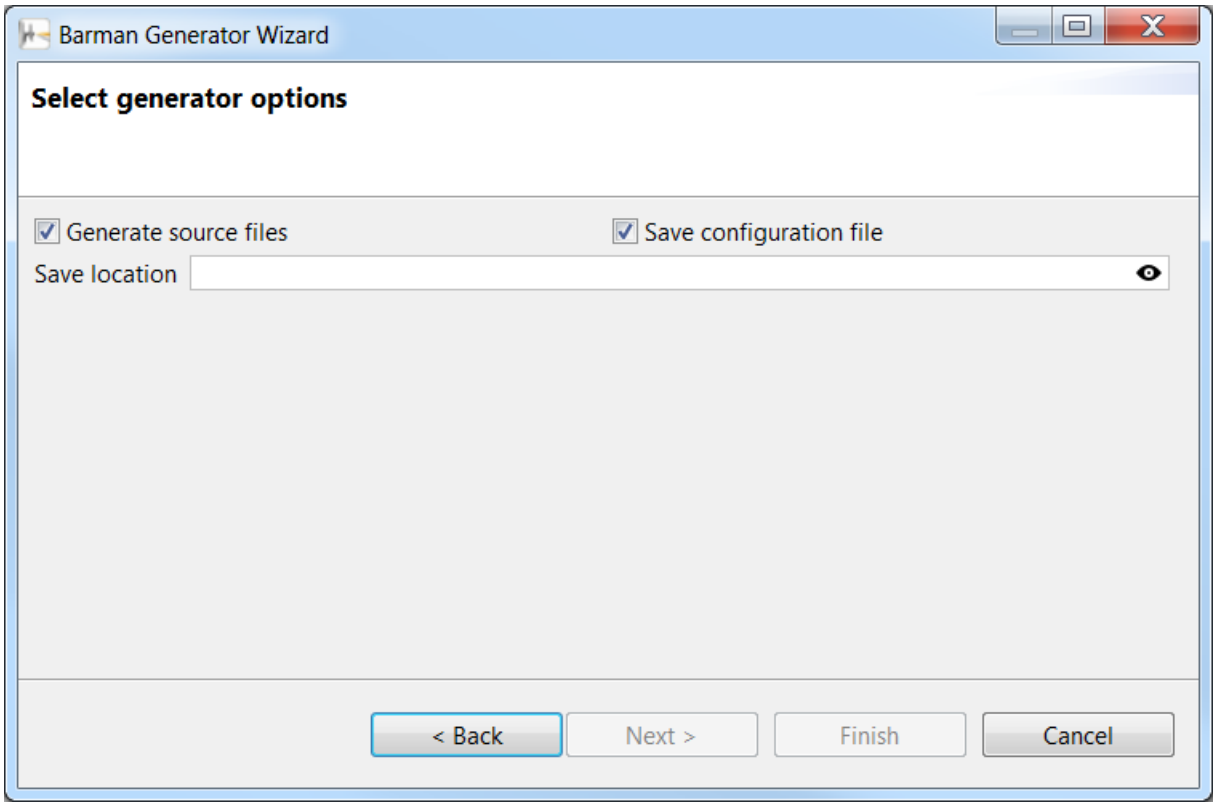


5. Add custom counters.

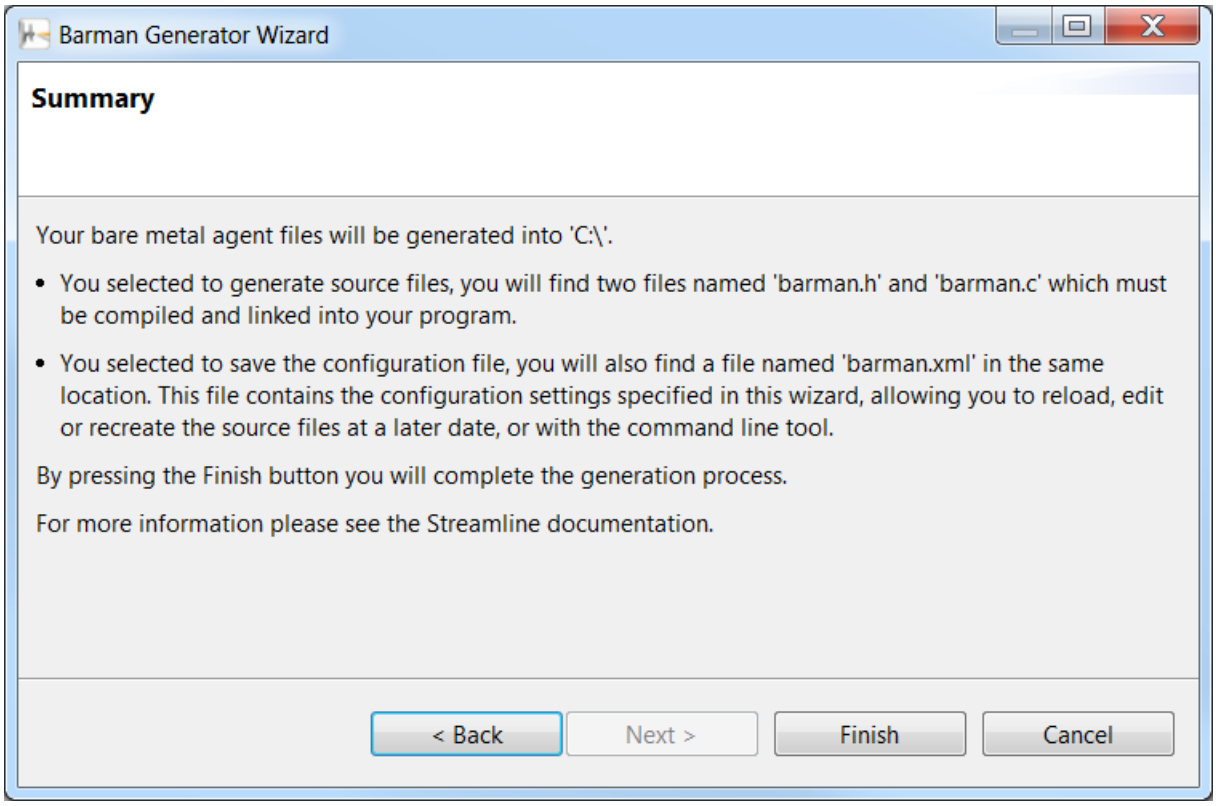


Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6. Select generator options.



7. Finish.



Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The setup process produces the following output:

- A configuration file, `barman.xml`, which contains the settings that were entered into the configuration wizard, and which can be used to reproduce the same configuration later.
- `barman.c`. You must compile and link this file into the bare-metal executable.
- `barman.h`. You must include this header when calling any of the functions within the agent. It also declares function prototypes for the functions you must implement.
- `barman_in_memory_helpers.py`. You can use this file as a use case script in Arm Development Studio. It helps you dump the contents of the in-memory capture buffer.

You need the compiler flag `--gnu` for `armcc` (Arm Compiler 5) to compile `barman.c`.

Related references

2.3.3 Barman use case script on page 2-18

2.3.2 Extracting and importing data

You must extract the data from the RAM buffer when the capture is complete.

For example, you could choose to do one of the following:

- Save the data to the file system of the target device, if one exists.
- Retrieve the data from RAM using JTAG during a debug session.
- Transfer the data over one of the available communication interfaces, for example ethernet or USB.

After extracting the raw data, give the data file a `.raw` extension. You can import this file into Streamline by clicking **Import Capture File(s)…**. The imported data is then available for Streamline to analyze.

If you added a custom `pmus.xml` or `events.xml` file during the configuration and generation stage, you must provide a copy of the same file into the `.apc` directory that is created for the imported capture. The files must be named `pmus.xml` and `events.xml` and must be placed in the directory alongside the `barman.raw` file for them to be detected and used.

2.3.3 Barman use case script

Streamline generates the file `barman_in_memory_helpers.py` with the Barman agent sources when you select an in-memory data storage backend. You can use it as a use case script in Arm Development Studio to help you dump the contents of the in-memory capture buffer.

Run the script with the following command:

```
usecase run "barman_in_memory_helpers.py" <usecase_command>
```

Two use case commands are available:

`get_parameters`
: Prints the current details of the buffer and information about how to dump it.

`dump`
: Dumps the contents of the memory buffer in a file that you specify with the option `--file <PATH>`.

Examples

The following examples show how to use these use case commands.

- To use the `get_parameters` use case command, enter:

```
usecase run "barman_in_memory_helpers.py" get_parameters
```

Output:

```
Barman memory buffer details:
    Base address:  0x0000000000001580
    Dump length:   1787404
    Bytes written: 1785996 of 67099264 (2.7%)

To dump this buffer use the command:
    dump memory <PATH> 0x1580 +1787404
```


Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
Or use the usecase command 'dump':
    usecase run "barman_in_memory_helpers.py" dump --file <PATH>
```

- To use the dump use case command, enter:

```
usecase run "barman_in_memory_helpers.py" dump --file barman.raw
```

Output:

```
Executing command:
    dump binary memory "barman.raw" 0x1580 +1787404

Memory successfully dumped to file barman.raw
```

Related tasks

2.3.1 Configuring Barman on page 2-14

Related information

Use case scripts

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.4 Profiling with System Trace Macrocell

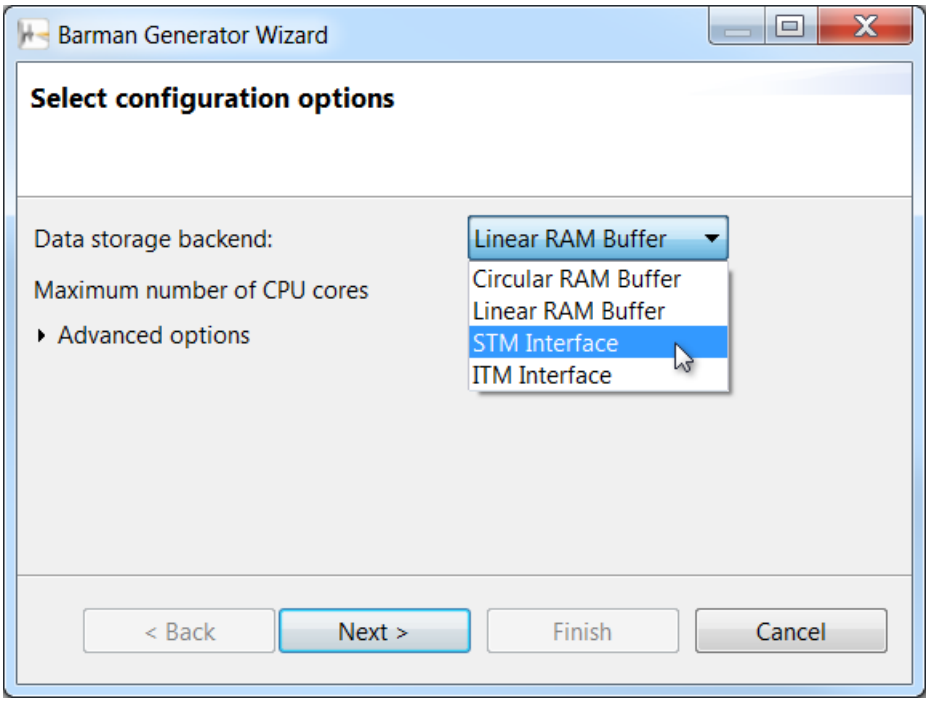
This section describes the collection of profiling data using System Trace Macrocell (STM).

Further information about STM, including the Technical Reference Manual, can be found on *Arm Developer*.

2.4.1 STM workflow

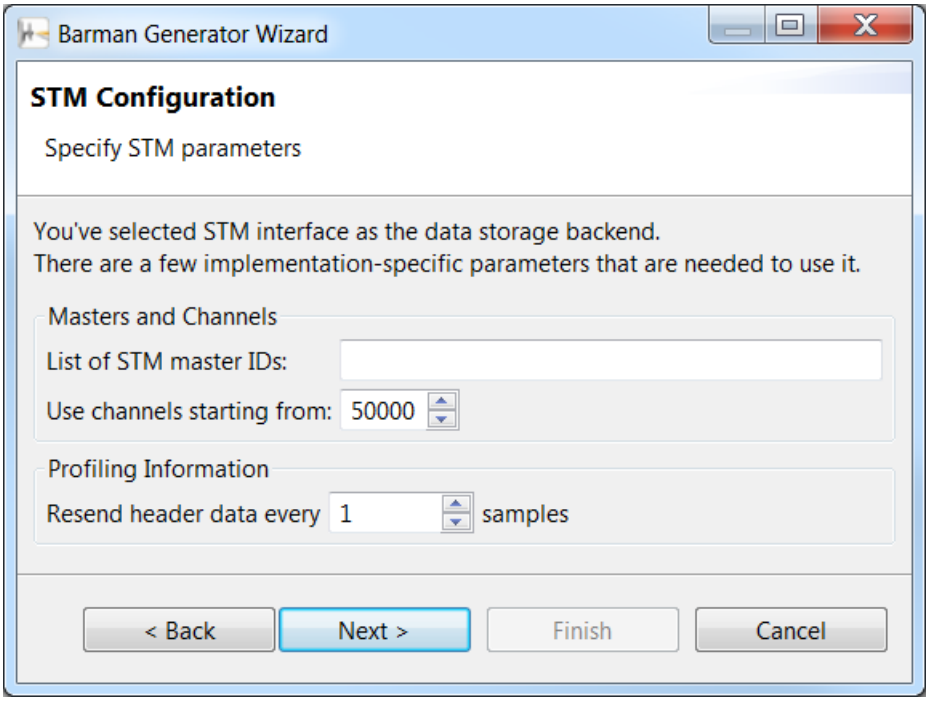
The workflow for STM involves a complex series of interactions between the applications involved.

1. Generate Barman agent code for STM using the **Barman Generator Wizard** dialog in Streamline.
   a. Select **STM Interface** as the data storage backend.



   b. Specify the STM parameters for your project.

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential



Note

Barman reserves the channels following the channel number that you specify. The number of channels reserved is the **Maximum number of CPU cores** specified on the previous page of the wizard.

c. Complete the remainder of the wizard as for a standard bare-metal project.
2. Add the Barman agent files that the wizard generates to your project.
3. Instrument your bare-metal application code with Barman agent calls (initialization, periodic sampling).
4. Compile and link your project.
5. Connect your target to a DSTREAM device.
6. Configure your target for collecting STM data into its RAM buffer.
7. Run the application on a target.
8. When you want to end the profiling, stop the application.
9. Dump the STM trace from the DSTREAM device into a directory.
10. Let Streamline import the trace file dump. Streamline reformats it and prepares it for analysis.

Note

- If you are using Arm Development Studio, you can dump the STM trace into a directory using the following command:

```
trace dump <directory> STM
```

- If you do not launch your bare-metal application from within Arm Development Studio, you must handle connecting to DSTREAM, obtaining the trace file, and importing it into Streamline.

Related tasks

2.3.1 Configuring Barman on page 2-14

2.4.2 Importing an STM trace

Import STM trace files into Streamline for analysis.

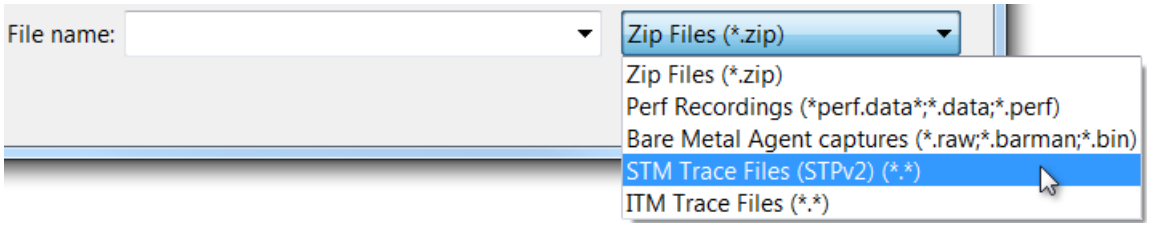
Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

You can also import the STM trace from the command line.

Procedure

1. Click **Import Capture File(s)…** in the **Streamline Data** view.
2. Select the import file type **STM Trace Files (STPv2)**.



3. Select the trace file to import.
4. Click **Open** and a new dialog box opens.
5. Enter the location of the `barman.xml` file that the **Barman Generator Wizard** produced.

   This file contains information about how to find relevant data in the trace file. For example, the channel numbers used.
6. Click **OK**.

Streamline then reformats the data, and converts the STM trace file into a Barman agent raw file.

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

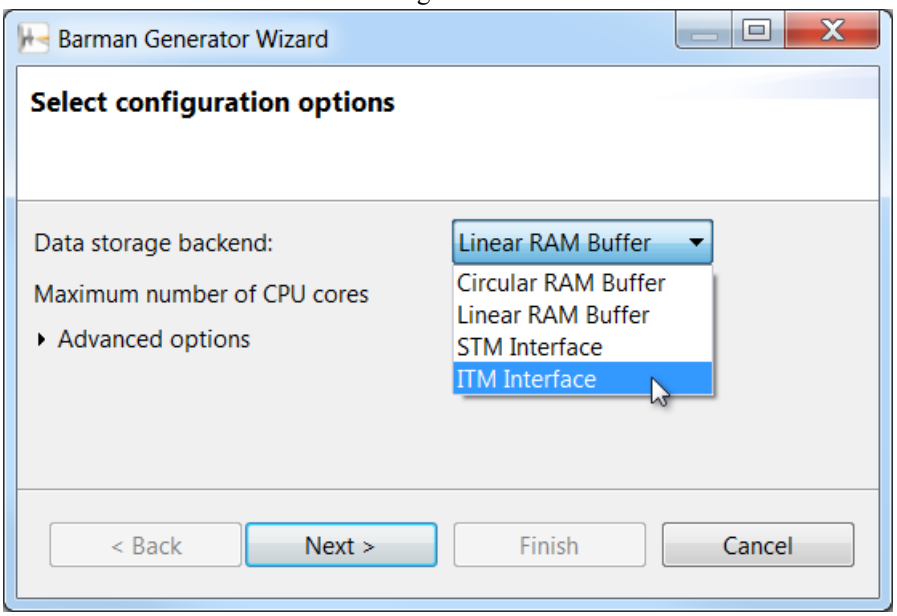
2.5 Profiling with Instrumentation Trace Macrocell

This section describes the collection of profiling data using Instrumentation Trace Macrocell (ITM).

2.5.1 ITM workflow

The workflow for ITM involves a complex series of interactions between the applications involved.

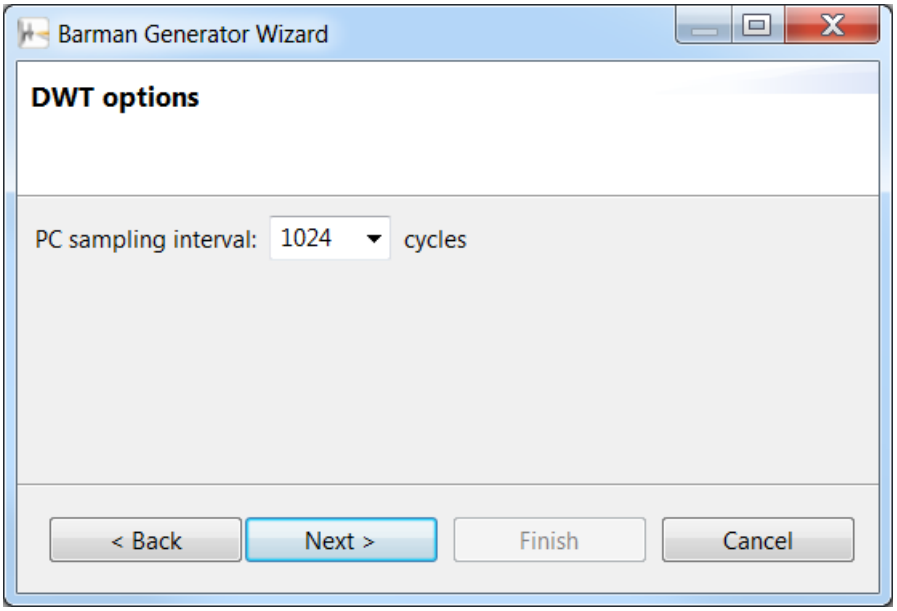
1. Generate Barman agent code for ITM using the **Barman Generator Wizard** dialog in Streamline.
   a. Select **ITM Interface** as the data storage backend.



Note

Barman uses ports 16-19 for ITM.

   b. Complete the remainder of the wizard as for a standard bare-metal project.
   c. If you selected a Cortex-M processor, select the number of cycles for the **PC sampling interval**.



2. Add the Barman agent files that the wizard generates to your project.

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3. Instrument your bare-metal application code with Barman agent calls (initialization, periodic sampling).
4. Compile and link your project.
5. Connect your target to a DSTREAM device.
6. Configure your target for collecting ITM data into its RAM buffer.
7. Run the application on a target.
8. When you want to end the profiling, stop the application.
9. Dump the ITM trace from the DSTREAM device into a directory.
10. Let Streamline import the trace file dump. Streamline reformats it and prepares it for analysis.

Note

- If you are using Arm Development Studio, you can dump the ITM trace into a directory using the following command:

```
trace dump <directory> ITM
```

- If you do not launch your bare-metal application from within Arm Development Studio, you must handle connecting to DSTREAM, obtaining the trace file, and importing it into Streamline.

Related tasks

2.3.1 Configuring Barman on page 2-14

2.5.2 Importing an ITM trace

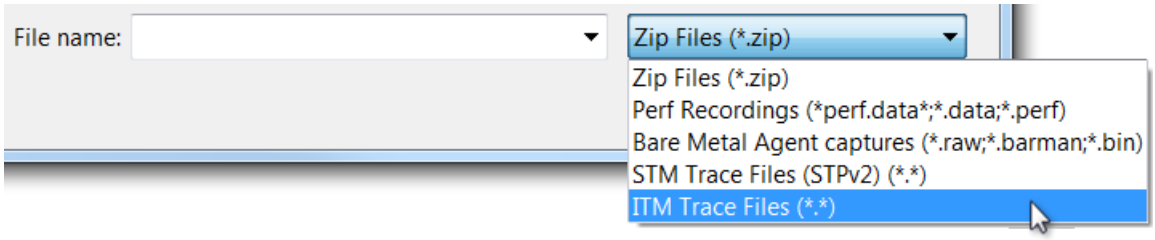
Import ITM trace files into Streamline for analysis.

Note

You can also import the ITM trace from the command line.

Procedure

1. Click **Import Capture File(s)…** in the **Streamline Data** view.
2. Select the import file type **ITM Trace Files**.



3. Select the trace file to import.
4. Click **Open** and a new dialog box opens.
5. Enter the location of the `barman.xml` file that the **Barman Generator Wizard** produced.

   This file contains information about how to find relevant data in the trace file. For example, the channel numbers used.
6. Click **OK**.

Streamline then reformats the data, and converts the ITM trace file into a Barman agent raw file.

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.6 Profiling with Embedded Trace Macrocell

This section describes the collection of profiling data using Embedded Trace Macrocell (ETM).

2.6.1 ETM workflow

The bare-metal agent can use the Data Trace feature that is provided with the ETM for streaming data from an R-class device in Arm Development Studio.

Note

The ETM interface requires ETM 3 or ETM 4 with data address tracing enabled.

To use ETM, generate the bare-metal agent as follows:

1. Open the **Barman Generator Wizard** by selecting **Streamline** > **Generate Barman sources**.
2. Select **ETM interface** as the data storage backend.
3. Click Finish.

To trace the device in Arm Development Studio, enable data address tracing from within the DTSL settings of your debug connection. For ETM 3, Streamline supports data-only mode. You can enable data value tracing, however it is not required, and it slows execution and greatly increases the trace size.

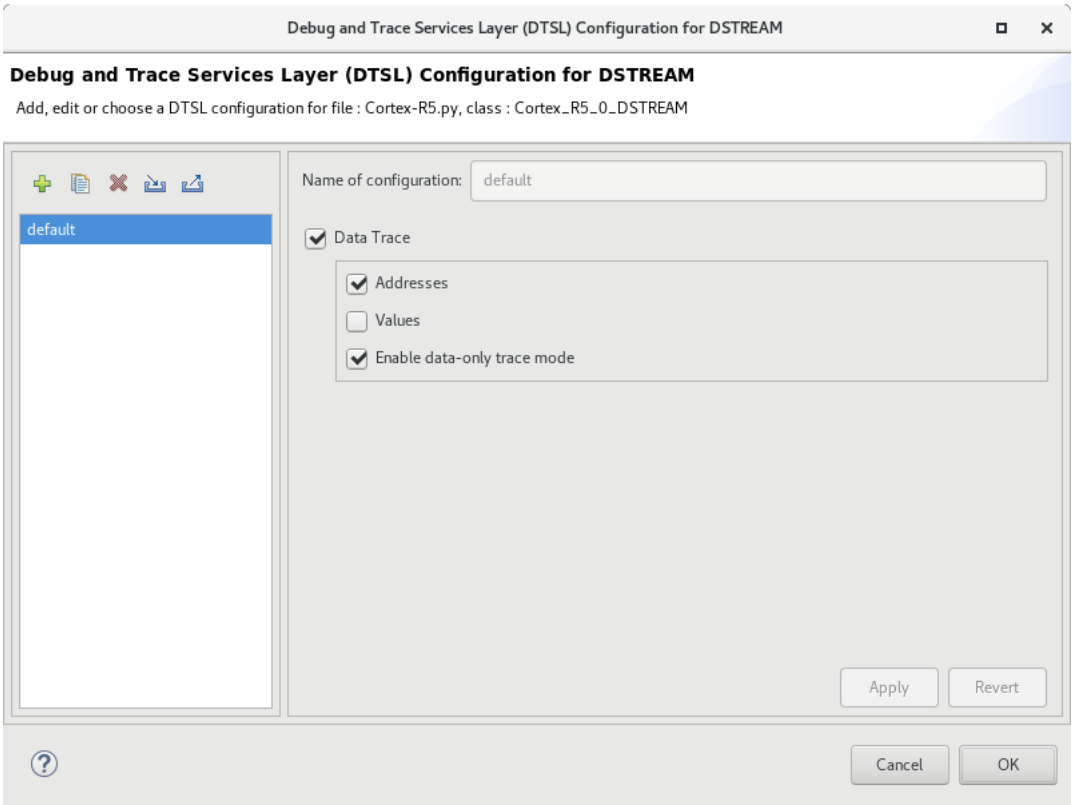


Figure 2-1 ETM3 configuration

Note

Different devices have different sets of options so this dialog might vary from the image shown.

You must use either DSTREAM or a large in-memory trace buffer for storing the trace. If you select **System Memory Trace Buffer (ETR/TMC)**, configure the in-memory trace buffer in the **ETR** tab. It is technically possible to use an on chip ETB for storing the trace, but the limited size, often less than 1KB, is not sufficient to store the capture.

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

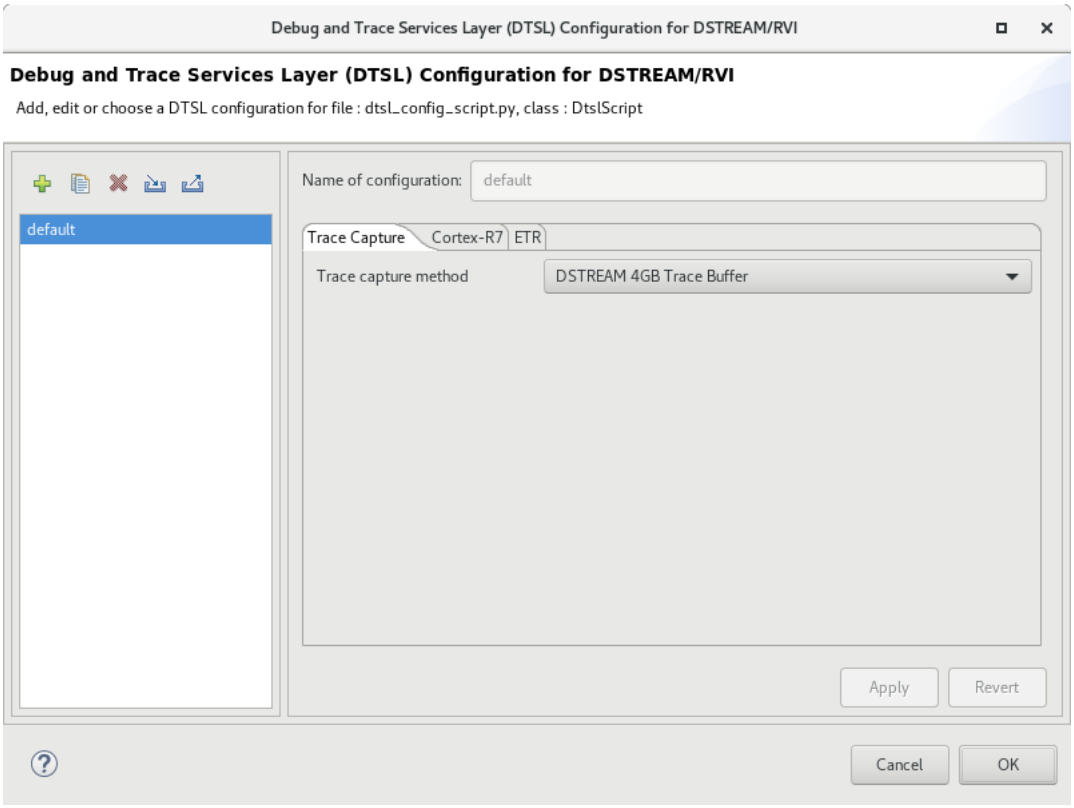


Figure 2-2 Select the Trace capture method in the DTSL configuration dialog.

For ETM4, you must enable ETM tracing for all cores you are interested in collecting data from. On this tab, you can also enable data address tracing. If the **Enable ETM Timestamps** option is available, select it.

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

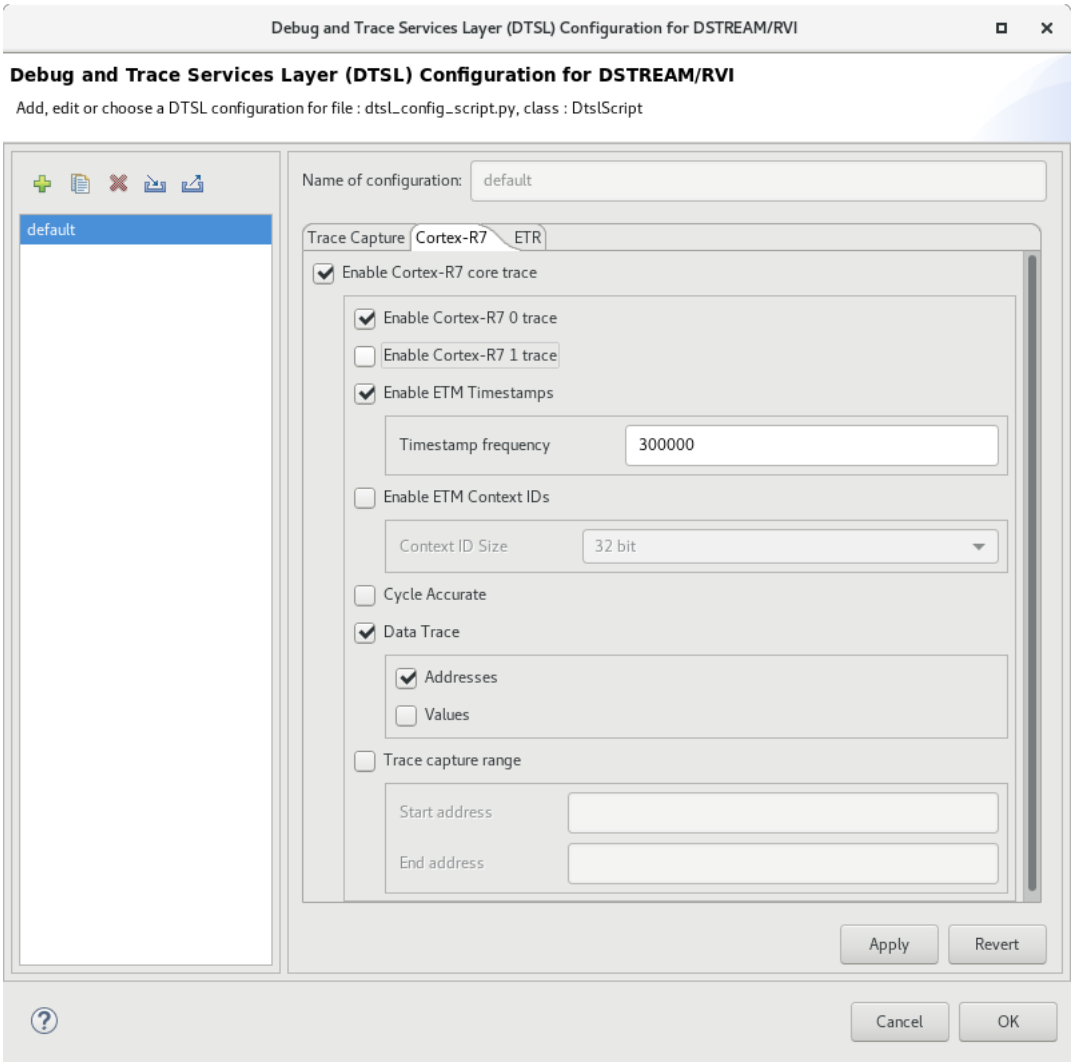


Figure 2-3 ETM4 configuration

You can run the provided python script, `barman_etm_filter_script.py`, as part of the debug configuration. The **Barman Generator Wizard** outputs this script alongside the generated source files. Set it as the debug initialization debugger script in your target debug configuration in Arm Development Studio. In the **Debug Configurations** dialog, click the **Debugger** tab, and enter the location of the script in the **Run debug initialization debugger script** field. This script limits tracing to the part of the bare-metal agent that sends the data. This limitation prevents you from getting the instruction trace, but reducing the amount of trace data in this way leads to smaller captures, faster imports, and the possibility to capture traces for longer. If you use ETM 4, Arm strongly recommends that you run this script. ETM 4 imports are much slower than ETM 3 imports. Limiting the trace data reduces the import time.

Use the following command to dump the ETM trace data:

```
trace dump <OUTPUT> <ETM_SOURCES>
```

Import this trace data into Streamline using the import button, or using the `-import-etm-dt` command-line option, in the same way as for STM and ITM.

Related information

Arm Debug and Trace Architecture
Embedded Trace Macrocell Architecture Specification ETMv1.0 to ETMv3.5
Embedded Trace Macrocell Architecture Specification ETMv4.0 to ETMv4.5
Debug and Trace Services Layer (DTSL)

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DTSL Configuration Editor dialog box

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.7 Interfacing with Barman

When Barman is linked into your executable code, the code must call the following functions:

1. `barman_initialize` to initialize Barman.
2. `barman_enable_sampling` to enable sampling.
3. The appropriate sample function, `barman_sample_counters` or `barman_sample_counters_with_program_counter`, to periodically collect data.

In a multiprocessor system, a call to one of the sampling functions only reads the counters for the processor element the code is executing on.

If you are running a preemptive kernel, RTOS, or similar, you must ensure that the thread running a call to a sampling function is not migrated from one processor element to another during the execution of the call.

In a multiprocessor system, if you are using periodic sampling (for example with a timer interrupt), you must provide a mechanism to call the sampling function for each processor element. In other words, to capture the counters of each processor element, there must be a timer interrupt or thread that is run separately on each processor element.

2.7.1 Configuration #defines

The configuration UI configures the following defines, which are stored in `barman.h`. They can be overridden at compile time as compiler parameters.

Define	Description
`BM_CONFIG_ENABLE_LOGGING`	Enables logging of messages when set to true.
`BM_CONFIG_ENABLE_DEBUG_LOGGING`	If `BM_CONFIG_ENABLE_LOGGING` is true, enables debug messages when set to true.
`BM_CONFIG_ENABLE_BUILTIN_MEMFUNCS`	Enables the use of built-in memory functions such as `__builtin_memset` and `__builtin_memcpy` when set to true.
`BM_CONFIG_MAX_CORES`	The maximum number of processor elements supported.
`BM_CONFIG_MAX_MMAP_LAYOUTS`	The maximum number of `mmap` layout entries to be stored in the data header. Configure to reflect the number of sections to be mapped for any process images.
`BM_CONFIG_MAX_TASK_INFOS`	The maximum number of distinct task entries that will be stored in the data. For single-threaded applications, this number can be zero, indicating that no information is provided. For multi-threaded applications or RTOS, this value indicates the maximum number of entries to store in the data header for describing processes, threads, and tasks.
`BM_CONFIG_MIN_SAMPLE_PERIOD`	The minimum period between samples in nanoseconds. If this value is greater than zero, calls to sampling functions are rate limited to ensure that there is a minimum interval of nanoseconds between samples.
`BARMAN_DISABLED`	Disables the Barman entry points at compile time when defined to a nonzero value. Use to conditionally disable calls to Barman, for example in production code.

2.7.2 Annotation #defines

Color macros to use for annotations.

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Define	Description
BM_ANNOTATE_COLOR_<color_name>	Named annotation color, where <color_name> is one of the following colors: RED BLUE GREEN PURPLE YELLOW CYAN WHITE LTGRAY DKGRAY BLACK
BM_ANNOTATE_COLOR_CYCLIC	Annotation color that cycles through a predefined set.
BM_ANNOTATE_COLOR_RGB(<R>, <G>, <B>)	Create an annotation color from its components, where <R>, <G>, and <B> are defined as follows: R The red component, where $0 \leq R \leq 255$. B The blue component, where $0 \leq B \leq 255$. G The green component, where $0 \leq G \leq 255$.

2.7.3 Barman public API

Use the bare-metal agent by calling the following public API functions.

barman_initialize

The prototype of barman_initialize varies depending on the datastore chosen.

When using the linear or circular RAM buffer:

```
BM_NONNULL((1, 3, 4))
bm_bool barman_initialize(bm_uint8 * buffer, bm_uintptr buffer_length,
```

When using STM:

```
BM_NONNULL((2, 3, 4))
bm_bool barman_initialize_with_stm_interface(void * stm_configuration_registers,
void * stm_extended_stimulus_ports,
```

When using ITM on Arm M-profile architectures:

```
BM_NONNULL((1, 2))
bm_bool barman_initialize_with_itm_interface(
```

When using ITM on Arm A- or R-profile architectures:

```
BM_NONNULL((1, 2, 3))
bm_bool barman_initialize_with_itm_interface(void * itm_registers,
```

The remaining parameters for each datastore are the same:

```
    const char * target_name,
    const struct bm_protocol_clock_info * clock_info,
#if BM_CONFIG_MAX_TASK_INFOS > 0
    bm_uint32 num_task_entries,
    const struct bm_protocol_task_info * task_entries,
#endif
#if BM_CONFIG_MAX_MMAP_LAYOUTS > 0
    bm_uint32 num_mmap_entries,
    const struct bm_protocol_mmap_layout * mmap_entries,
#endif
    bm_uint32 timer_sample_rate);
```


Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Description	Initialize Barman.

Parameters	`buffer`
	Pointer to in memory buffer.
	`buffer_length`
	Length of the in memory buffer.
	`stm_configuration_registers`
	Base address of the STM configuration registers. This parameter can be NULL if it is initialized elsewhere, for example by the debugger.
	`stm_extended_stimulus_ports`
	Base address of the STM extended stimulus ports.
	`itm_registers`
	Base address of the ITM registers.
	`datastore_config`
	Pointer to the configuration to pass to `barman_ext_datastore_initialize`.
	`target_name`
	Name of the target device.
	`clock_info`
	Information about the monotonic clock used for timestamps.
	`num_task_entries`
	Length of the array of task entries in `task_entries`. If this value is greater than `BM_CONFIG_MAX_TASK_INFOS`, it is truncated.
	`task_entries`
	The task information descriptors. Can be NULL.
	`num_mmap_entries`
	The length of the array of mmap entries in `mmap_entries`. If this value is greater than `BM_CONFIG_MAX_MMAP_LAYOUT`, it is truncated.
	`mmap_entries`
	The mmap image layout descriptors. Can be NULL.
	`timer_sample_rate`
	Timer-based sampling rate in Hz. Zero indicates no timer-based sampling (assumes max. 4GHz sample rate). This value is informative only, and is used for reporting the timer frequency in the Streamline UI.

Return value	`BM_TRUE`
	On success.
	`BM_FALSE`
	On failure.

Note

If `BM_CONFIG_MAX_TASK_INFOS` ≤ 0, `num_task_entries` and `task_entries` are not present.

If `BM_CONFIG_MAX_MMAP_LAYOUTS` ≤ 0, `num_mmap_entries` and `mmap_entries` are not present.

`barman_enable_sampling`

```
void barman_enable_sampling(void);
```

Description	Enables sampling. Call once all PMUs are enabled and the data store is configured.

`barman_disable_sampling`

```
void barman_disable_sampling(void);
```

Description	Disables sampling without reconfiguring the PMU. To resume sampling, call `barman_enable_sampling`.

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

barman_sample_counters

```
void barman_sample_counters(bm_bool sample_return_address);
```

Description	Reads the configured PMU counters for the current processing element and inserts them into the data store. Can also insert a Program Counter record using the return address as the PC sample.
Parameter	`sample_return_address` `BM_TRUE` to sample the return address as PC, `BM_FALSE` to ignore.

Note

- The **Call Paths** view displays the PC values. This view is blank if the application does not call `barman_sample_counters` with `sample_return_address == BM_TRUE`, or `barman_sample_counters_with_program_counter` with `pc != BM_NULL`.
- Application code that is not doing periodic sampling typically calls this function with `sample_return_address == BM_TRUE`.
- This function must be run on the processing element for the PMU that it intends to sample from. It must not be migrated to another processing element for the duration of the call. This is necessary as it needs to program the per processing element PMU registers.

barman_sample_counters_with_program_counter

```
void barman_sample_counters_with_program_counter(const void * pc);
```

Description	Reads the configured PMU counters for the current processing element and inserts them into the data store.
Parameter	`pc` The PC value to record. The PC entry is not inserted if `pc == BM_NULL`.

Note

- The **Call Paths** view displays the PC values. This view is blank if the application does not call `barman_sample_counters_with_program_counter` with `pc != BM_NULL`, or `barman_sample_counters` with `sample_return_address == BM_TRUE`.
- A periodic interrupt handler typically calls this function, with `pc == <the_exception_return_address>`.
- This function must be run on the processing element for the PMU that it intends to sample from. It must not be migrated to another processing element for the duration of the call. This is necessary as it needs to program the per processing element PMU registers.

The following functions are available if `BM_CONFIG_MAX_TASK_INFOS > 0`:

barman_add_task_record

```
bm_bool barman_add_task_record(bm_uint64 timestamp, const struct
bm_protocol_task_info * task_entry);
```

Description	Adds a task information record.

Parameters	`timestamp`
	The timestamp at which the record is inserted.
	`task_entry`
	The new task entry.

Return value	`BM_TRUE`
	On success.
	`BM_FALSE`
	On failure.

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

barman_record_task_switch

```
void barman_record_task_switch(enum bm_task_switch_reason reason);
```

Description	Records that a task switch has occurred. Call this function after the new task is made the current task, so a call to `barman_ext_get_current_task_id` returns the new task ID. For example, insert it into the scheduler of an RTOS just after the new task is selected to record the task switch.
Parameter	`reason` Reason for the task switch.

Note

Call after the task switch has occurred so that `bm_ext_get_current_task` returns the `task_id` of the switched to task.

The following function is available if `BM_CONFIG_MAX_MMAP_LAYOUTS > 0`:

barman_add_mmap_record

```
bm_bool barman_add_mmap_record(bm_uint64 timestamp, const struct
bm_protocol_mmap_layout * mmap_entry);
```

Description	Adds a `mmap` information record.

Parameters	`timestamp`
	The timestamp at which the record is inserted.
	`mmap_entry`
	The new `mmap` entry.

Return value	`BM_TRUE`
	On success.
	`BM_FALSE`
	On failure.

Data types associated with the public API functions:

bm_protocol_clock_info

```
struct bm_protocol_clock_info
{
    bm_uint64 timestamp_base;
    bm_uint64 timestamp_multiplier;
    bm_uint64 timestamp_divisor;
    bm_uint64 unix_base_ns;
};
```


Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Description	Defines information about the monotonic clock used in the trace. Timestamp information is stored in arbitrary units within samples. Arbitrary units reduce the overhead of making the trace by removing the need to transform the timestamp into nanoseconds at the point the sample is recorded. The host expects timestamps to be in nanoseconds. The arbitrary timestamp information is transformed to nanoseconds according to the following formula: `bm_uint64 nanoseconds = (((timestamp - timestamp_base) * timestamp_multiplier) / timestamp_divisor;` Therefore for a clock that already returns time in nanoseconds, `timestamp_multiplier` and `timestamp_divisor` should be configured as `1` and `1`. If the clock counts in microseconds then the multiplier and divisor should be set to `1000` and `1`. If the clock counts at a rate of n Hertz, then the multiplier should be set to `1000000000` and the divisor to n.

Members	`timestamp_base`
	The base value of the timestamp so that this value is zero in the trace.
	`timestamp_multiplier`
	The clock rate ratio multiplier.
	`timestamp_divisor`
	The clock rate ratio divisor
	`unix_base_ns`
	The Unix timestamp base value, in nanoseconds, so a `timestamp_base` maps to a `unix_base` Unix time value.

`bm_protocol_task_info`

```
struct bm_protocol_task_info
{
    bm_task_id_t task_id;
    const char * task_name;
};
```

Description	A task information record. Describes information about a unique task within the system.

Members	`task_id`
	The task ID.
	`task_name`
	The name of the task.

`bm_protocol_mmap_layout`

```
struct bm_protocol_mmap_layout
{
#if BM_CONFIG_MAX_TASK_INFOS > 0
    bm_task_id_t task_id;
#endif
    bm_uintptr base_address;
    bm_uintptr length;
    bm_uintptr image_offset;
    const char * image_name;
};
```

Description	An MMAP layout record. Describes the position of an executable image (or section thereof) in memory, allowing the host to map PC values to the appropriate executable image.

Members	`task_id`
	The task ID to associate with the map.
	`base_address`
	The base address of the image, or image section.
	`length`
	The length of the image, or image section.
	`image_offset`
	The image section offset.
	`image_name`
	The name of the image.

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

bm_task_switch_reason

```
enum bm_task_switch_reason
{
    BM_TASK_SWITCH_REASON_PREEMPTED = 0,
    BM_TASK_SWITCH_REASON_WAIT = 1
};
```

Description	Reason for a task switch.

Members	`BM_TASK_SWITCH_REASON_PREEMPTED`
	Thread is preempted.
	`BM_TASK_SWITCH_REASON_WAIT`
	Thread is blocked waiting, for example on I/O.

WFI and WFE event handling functions:

barman_wfi

```
void barman_wfi(void);
```

Description	Wraps WFI instruction and sends events before and after the WFI to log the time in WFI. This function is safe to use in place of the usual WFI `asm` instruction, as it degenerates to just a WFI instruction when Barman is disabled.

barman_wfe

```
void barman_wfe(void);
```

Description	Wraps WFE instruction and sends events before and after the WFE to log the time in WFE. This function is safe to use in place of the usual WFE `asm` instruction as it degenerates to just a WFE instruction when Barman is disabled.

barman_before_idle

```
void barman_before_idle(void);
```

Description	Call before a WFI or WFE, or other similar halting event, to log entry into the paused state. Can be used in situations where `barman_wfi()`/`barman_wfe()` is not suitable.

Note

- You must use `barman_before_idle` in a pair with `barman_after_idle()`.
- Using `barman_wfi()` or `barman_wfe()` is usually preferred, as it takes care of calling the before and after functions.

barman_after_idle

```
void barman_after_idle(void);
```

Description	Call after a WFI or WFE, or other similar halting event, to log exit from the paused state. Can be used in situations where `barman_wfi()` or `barman_wfe()` is not suitable.

Note

- You must use `barman_after_idle` in a pair with `barman_before_idle()`.
- Using `barman_wfi()` or `barman_wfe()` is usually preferred, as it takes care of calling the before and after functions.

Functions for recording textual annotations:

barman_annotate_channel

```
void barman_annotate_channel(bm_uint32 channel, bm_uint32 color, const char * string)
```


Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Description	Adds a string annotation with a display color, and assigns it to a channel.

Parameters	`channel`
	The channel number.
	`color`
	The annotation color from `bm_annotation_colors`.
	`text`
	The annotation text, or null to end the previous annotation.

Note

Annotation channels and groups are used to organize annotations within the threads and processes section of the **Timeline** view. Each annotation channel appears in its own row under the thread. Channels can also be grouped and displayed under a group name, using the `barman_annotate_name_group` function.

`barman_annotate_name_channel`

```
void barman_annotate_name_channel(bm_uint32 channel, bm_uint32 group, const char *
name)
```

Description	Defines a channel and attaches it to an existing group.

Parameters	`channel`
	The channel number.
	`group`
	The group number.
	`name`
	The name of the channel.

Note

The channel number must be unique within the task.

`barman_annotate_name_group`

```
void barman_annotate_name_group(bm_uint32 group, const char * name)
```

Description	Defines an annotation group.

Parameters	`group`
	The group number.
	`name`
	The name of the group.

Note

The group identifier, `group`, must be unique within the task.

`barman_annotate_marker`

```
void barman_annotate_marker(bm_uint32 color, const char * text)
```


Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Description	Adds a bookmark with a string and a color to the **Timeline** view and **Log** view. The string is displayed in the **Timeline** view when you hover over the bookmark, and in the **Message** column in the **Log** view.

Parameters	`color`
	The marker color from `bm_annotation_colors`.
	`text`
	The marker text, or null for no text.

`bm_annotation_colors`

Description	Color macros for annotations. See *2.7.2 Annotation #defines* on page 2-29.

2.7.4 External functions to implement

You must provide the following external functions.

`barman_ext_get_timestamp`

```
extern bm_uint64 barman_ext_get_timestamp(void);
```

Description	Reads the current sample timestamp value, which must be provided for the time at the point of the call. The timer must provide monotonically incrementing values from an implementation defined start point. The counter must not overflow during the period that it is used. The counter is in arbitrary units. The mechanism for converting those units to nanoseconds is described as part of the protocol data header.
Return value	The timestamp value in arbitrary units.

The following functions have weak linkage implementations that can be overridden if necessary:

`barman_ext_disable_interrupts_local`

```
extern bm_uintptr barman_ext_disable_interrupts_local(void);
```

Description	Disables interrupts on the local processor only. Used to allow atomic accesses to certain resources, for example PMU counters.
Return value	The current interrupt enablement status value. This value must be preserved and passed to `barman_ext_enable_interrupts_local` to restore the previous state.

Note

A weak implementation of this function is provided that modifies DAIF on AArch64, or CPSR on AArch32.

`barman_ext_enable_interrupts_local`

```
extern void barman_ext_enable_interrupts_local(bm_uintptr previous_state);
```

Description	Enables interrupts on the local processor only.
Parameter	`previous_state` The value that was previously returned from `barman_ext_disable_interrupts_local`

Note

A weak implementation of this function is provided that modifies DAIF on AArch64, or CPSR on AArch32.

The following functions must be defined if `BM_CONFIG_MAX_CORES > 1`:

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

barman_ext_map_multiprocessor_affinity_to_core_no

```
extern bm_uint32 barman_ext_map_multiprocessor_affinity_to_core_no(bm_uintptr mpidr);
```

Description	Given the MPIDR register, returns a unique processor number. The implementation must return a value between 0 and N, where N is the maximum number of processors in the system. For any valid permutation of the arguments, a unique value must be returned. This value must not change between successive calls to this function for the same argument values.
	`//`
	`// Example implementation where processors are arranged as follows:`
	`//`
	`// aff2 | aff1 | aff0 | cpuno`
	`// -----+------+------+------`
	`//  0 | 0 | 0 | 0`
	`//  0 | 0 | 1 | 1`
	`//  0 | 0 | 2 | 2`
	`//  0 | 0 | 3 | 3`
	`//  0 | 1 | 0 | 4`
	`//  0 | 1 | 1 | 5`
	`//`
	`bm_uint32 barman_ext_map_multiprocessor_affinity_to_core_no(bm_uintptr mpidr)`
	`{`
	`  return (mpidr & 0x03) + ((mpidr >> 6) & 0x4);`
	`}`

Parameter	`mpidr` The value of the MPIDR register.
Return value	The processor number.

Note

This function only needs defining when `BM_CONFIG_MAX_CORES > 1`.

barman_ext_map_multiprocessor_affinity_to_cluster_no

```
extern bm_uint32 barman_ext_map_multiprocessor_affinity_to_cluster_no(bm_uintptr
mpidr);
```

Description	Given the MPIDR register, return the appropriate cluster number. Cluster IDs should be numbered from 0 to N, where N is the number of clusters in the system.
	`//`
	`// Example implementation which is compatible with the example implementation of`
	`// barman_ext_map_multiprocessor_affinity_to_core_no given above.`
	`//`
	`bm_uint32 barman_ext_map_multiprocessor_affinity_to_cluster_no(bm_uintptr mpidr)`
	`{`
	`  return ((mpidr >> 8) & 0x1);`
	`}`

Parameter	`mpidr` The value of the MPIDR register.
Return value	The cluster number.

Note

This function only needs defining when `BM_CONFIG_MAX_CORES > 1`.

The following function must be defined if `BM_CONFIG_MAX_TASK_INFOS > 0`:

barman_ext_get_current_task_id

```
extern bm_task_id_t barman_ext_get_current_task_id(void);
```

Description	Returns the current task ID.

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The following functions must be defined if `BM_CONFIG_ENABLE_LOGGING != 0`:

`barman_ext_log_info`

```
void barman_ext_log_info(const char * message, ...);
```

Description	Prints an info message.
Parameter	`message`

`barman_ext_log_warning`

```
void barman_ext_log_warning(const char * message, ...);
```

Description	Prints a warning message.
Parameter	`message`

`barman_ext_log_error`

```
void barman_ext_log_error(const char * message, ...);
```

Description	Prints an error message.
Parameter	`message`

The following function must be defined if `BM_CONFIG_ENABLE_DEBUG_LOGGING != 0`:

`barman_ext_log_debug`

```
void barman_ext_log_debug(const char * message, ...);
```

Description	Prints a debug message.
Parameter	`message`

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.8 Custom counters

You can configure one custom chart, with one or more series, in the configuration wizard.

2.8.1 Configuring custom counters

You can configure chart properties for custom counters.

The following chart properties can be configured:

Name
Human readable name for the chart.

Series Composition
Defines how to arrange series on the chart (stacked, overlay, or logarithmic).

Rendering Type
Defines how to render series on the chart (filled, line, or bar).

Per Processor
Indicates whether the data in the chart is per processor.

Average Selection
Sets whether the cross-section marker in Streamline displays average values.

Average Cores
Sets whether Streamline averages the values of multiple cores when viewing the aggregate data of a per processor chart.

Percentage
Sets whether to display data as a percentage of the maximum value in the chart.

The following series properties can be configured:

Name
Human readable name for the series.

Units
Defines the unit type to display in Streamline.

Sampled
When set to true, the value for this counter is sampled along with the PMU counters. When false, you must call a function to update the counter value.

Multiplier
Number to multiply by for fixed-point math. As the data sent from the agent is int64, it must be scaled. For example, the value 123 with a multiplier of 0.01 can represent the value 1.23.

Class
Specifies the nature of the data that is fed into the chart as follows:

delta
Used for values that increment or are accumulated over time, such as hardware performance counters. The exact time when the data occurs is unknown and therefore the data is interpolated between timestamps.

incident
The same as delta, except the exact time is known so no interpolation is calculated. Used for counters such as software trace.

absolute
Used for singular or impulse values, such as system memory used.

Display

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The display value determines how to calculate the data when zooming out for each time bin as follows:

accumulate
Sum up the data (valid only for delta and incident class counters).

hertz
Does the same as accumulate then normalizes the value to one second (valid only for delta and incident class counters).

minimum
Display the smallest value encountered (valid only for absolute class counters).

maximum
Display the largest value encountered (valid only for absolute class counters).

average
Display the average (valid only for absolute class counters).

Color
The color to display the series in. If not set, Streamline selects a color.

Description
Human readable description for the series. This description becomes the tooltip when hovering over the series in Streamline.

2.8.2 Sampled and nonsampled counters

Sampled counters are polled when the PMU counter values are read.

For each sampled custom counter, a function prototype is generated of the following form:

```
extern bm_bool barman_cc_<chart_name>_<series_name>_sample_now(bm_uint64 * value_out);
```

For example:

```
extern bm_bool barman_cc_interrupts_fiq_sample_now(bm_uint64 * value_out);
```

You must implement this function to set the value of the `uint64` at `*value_out` to the value of the counter, then return `BM_TRUE`. If the counter value cannot be sampled, for example due to another thread accessing the hardware, the function can return `BM_FALSE` and be skipped.

You are responsible for writing nonsampled counters to the capture. For each nonsampled series, the following two functions are declared:

```
bm_bool barman_cc_<chart_name>_<series_name>_update_value(bm_uint64 timestamp, bm_uint32
core, bm_uint64 value);
bm_bool barman_cc_<chart_name>_<series_name>_update_value_now(bm_uint64 value);
```

For example:

```
bm_bool barman_cc_interrupts_fiq_update_value(bm_uint64 timestamp, bm_uint32 core, bm_uint64
value);
bm_bool barman_cc_interrupts_fiq_update_value_now(bm_uint64 value);
```

The second function is a shorthand for the first that passes the current timestamp and core number to the appropriate arguments.

When you call these functions, the value for the counter is stored to the capture.

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.9 Using the bare-metal generation mechanism from the command line

You can pass the configured, and optionally modified, XML file produced in the bare-metal configuration process to the command line. The generator then outputs the source and header files.

Enter `streamline -generate-bare-metal-agent <options>`

The following command-line arguments are available:

`-c, -config <config.xml>`
The configuration file to use to generate the bare-metal agent.

`-p, -pmus <pmus.xml>`
Specify the path to your `pmus.xml` file.

`-e, -events <events.xml>`
Specify the path to your `events.xml` file.

`-o, -output <output_path>`
Specify the output path to where the generated files will be written.

Related information

Streamline command-line options

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter 3
Profiling with Instruction Trace

This section describes how to import an instruction trace, and restrictions related to this task.

It contains the following sections:

- *3.1 Importing instruction trace* on page 3-44.
- *3.2 Instruction trace notes and restrictions* on page 3-47.

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.1 Importing instruction trace

Import a trace of the instructions that your application executed for Streamline to analyze. The supported forms of instruction trace are PTM 1.0-1.1, ETM 3.0-3.5, and ETM 4.0-4.2.

Prerequisites

1. Collect the instruction trace for your application using Arm DS Debugger. See the *Arm Development Studio User Guide* for instructions.
2. Export the instruction trace using the following command:

```
trace dump <output> <instruction_trace_sources>
```

Note

`<instruction_trace_sources>` must only specify valid ETM sources. For example, CSETM_APP_0.

Procedure

1. Click **Import Capture File(s)…** .
2. Navigate to the directory that contains the trace dump.
3. Select any of the files in the directory.

   A wizard opens, enabling you to choose what to do with the import.
4. Select **Instruction Trace**.

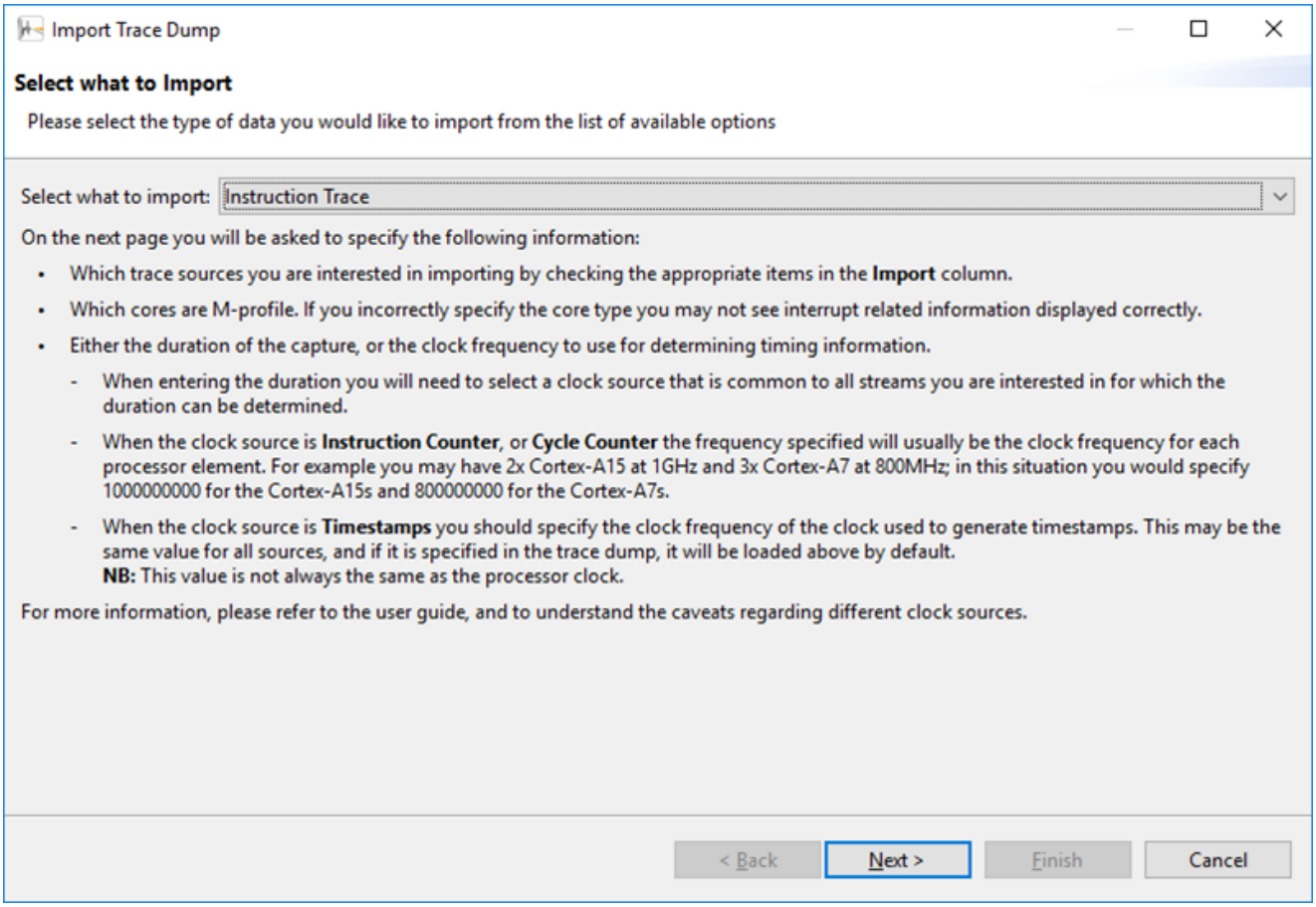


Figure 3-1 Select Instruction Trace.

5. Select the sources to import, and specify the timing information for them.

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

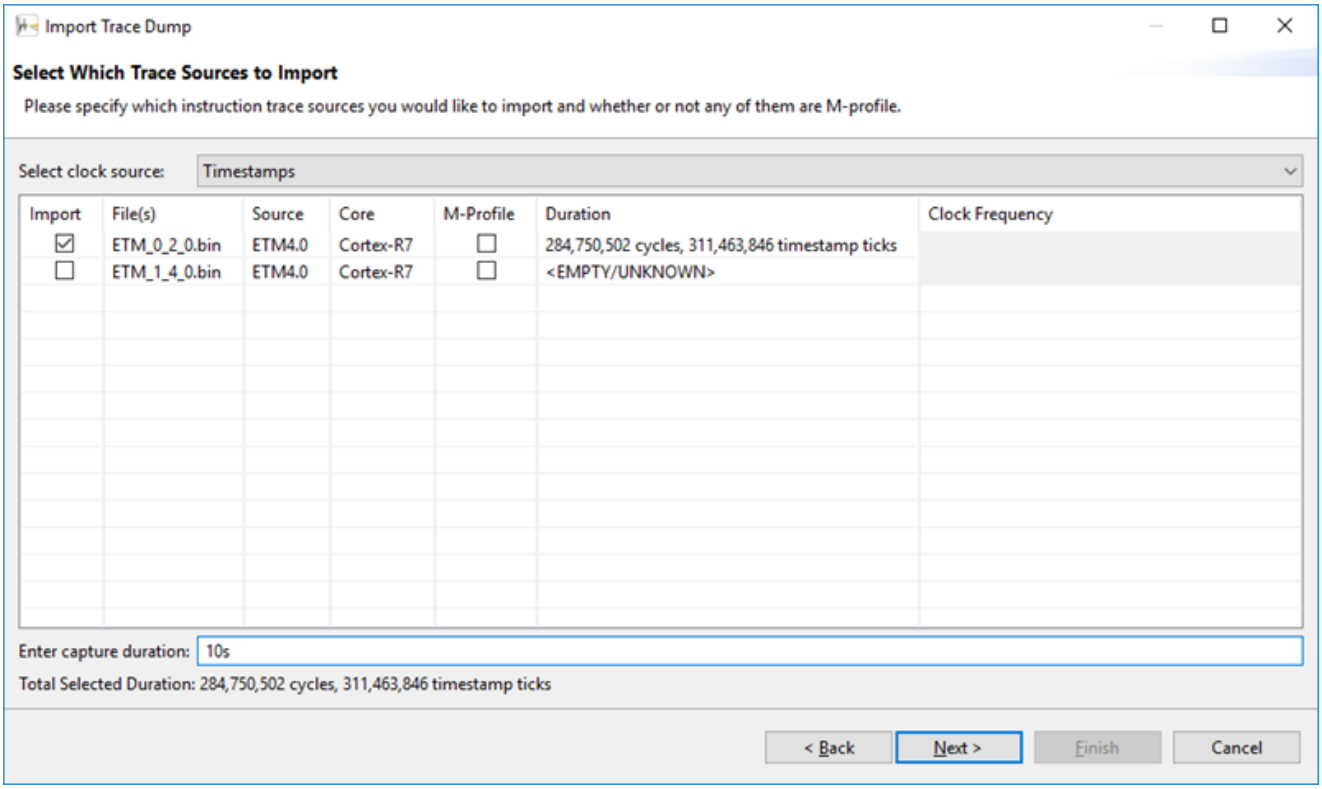


Figure 3-2 Select trace sources to import.

6. Select the executable images that were run.

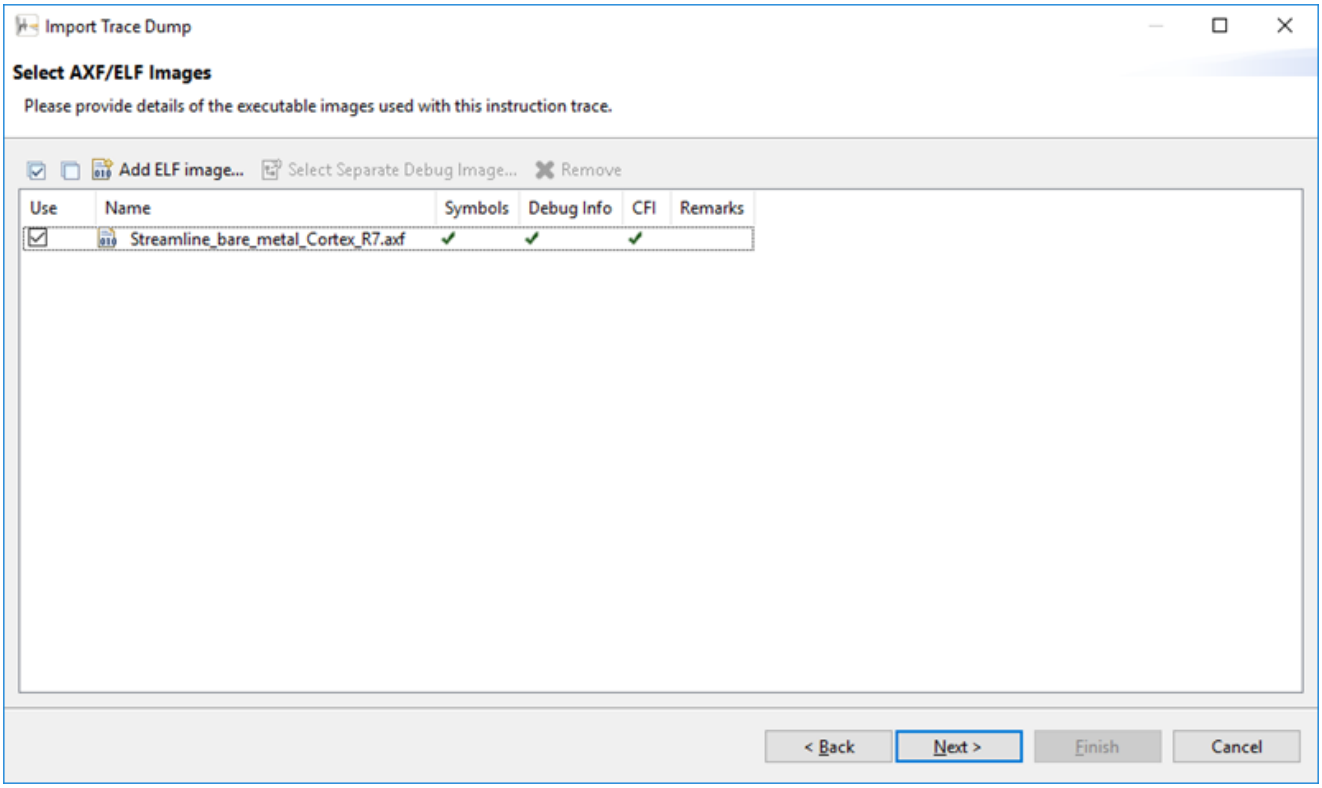


Figure 3-3 Select executable images.

Streamline uses these files to decode the trace.

7. Configure parameters for the capture that Streamline generates from the trace.

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

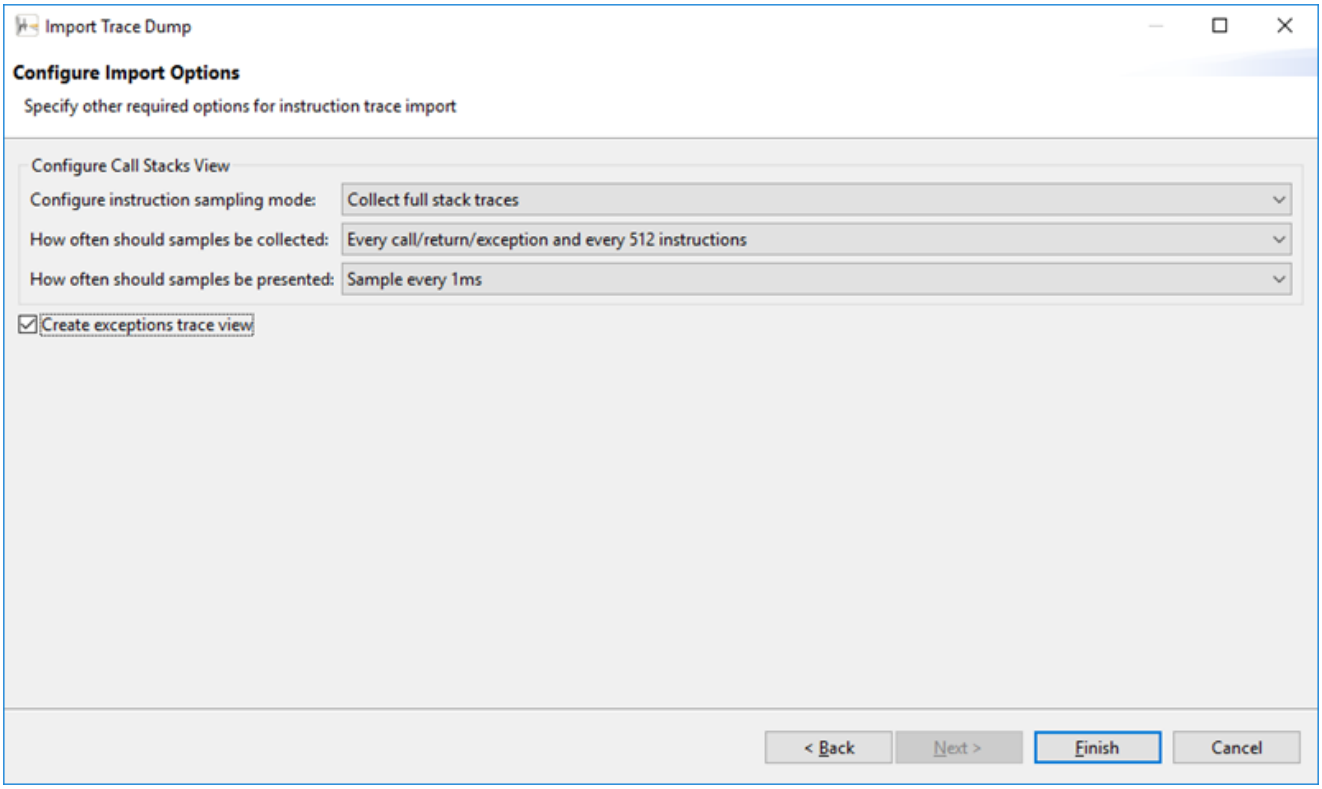


Figure 3-4 Configure import options.

Streamline reads the instruction trace and generates a capture from it. The generated capture contains the following:

- Charts that give approximate information about branching, instructions, load/stores, and exceptions.
- The **Call Paths** view and **Functions** view that Streamline derives from the instruction trace.
- Optionally, the **Exceptions** view, which shows the exceptions that were taken.

Related information

Configuring trace for bare-metal or Linux kernel targets

Capturing trace data using the command-line debugger

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.2 Instruction trace notes and restrictions

There are some restrictions to instruction traces.

- Dynamic compilation and code modification is not supported. If the modification happens before the code is traced, and you supply an ELF image containing the executable code after modification, Streamline can support one-off runtime code modification. You can use the Arm Development Studio memory dump command, specifying the ELF output format.
- You must specify whether the imported files are M-profile, otherwise interrupts are not shown correctly. This restriction does not usually affect the import of the trace, just the output.
- The **Call Paths** view tracks function entry and exit instructions, and attempts to track state across interrupts even when simple context switching is used. This functionality might not be compatible with a more complex OS.

Streamline generates timing information from the trace in one of the following ways:

- Using timestamps, if they are included in all the relevant trace streams.
  - Use this method if it is available, especially if you are importing streams from more than one processing element.
  - Timestamps give a near-accurate representation of the relative times of different events. The accuracy of the timing information depends on the relative frequency of timestamp packets within each trace stream.
  - Timestamps allow showing periods of inactivity such as during WFE or WFI.
  - Timestamps are usually clocked using a separate clock to the processor. Arm recommends that you enter the duration of the capture rather than the clock frequency for the timestamp clock, unless you know the exact frequency.
- Using cycle counts, if cycle-accurate mode is enabled for all relevant trace streams.
  - Use this mode if timestamps are not available as it gives better relative timing information between different instructions in the trace.
  - Timing information is not synchronized across different processing elements.
  - Unless the cycle counter increments during WFE or WFI instructions, it is not possible to see how long the processor waited for.
  - Systems using dynamic frequency scaling are not clocked correctly because the cycle count for individual instructions does not change with the clock frequency.
- Using the instruction counter.
  - The only mode that is universally available.
  - This mode has all the same limitations as the cycle counter mode.
  - This mode cannot display relative timing information for different instructions.

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter 4
Examples

It contains the following section:

- *4.1 Examples using Barman* on page 4-49.

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.1 Examples using Barman

See `<install_directory>/sw/streamline/examples/barman` for the following Barman examples:

`Streamline_bare_metal_ARMv8_AArch64`
A demonstration of how to use Barman with AArch64, from configuring the bare-metal agent to analyzing the results.

`Streamline_bare_metal_Cortex_R5`
A demonstration of how to use Barman with Arm Cortex-R5, from configuring the bare-metal agent to analyzing the results.

`Streamline_bare_metal_M_profile`
A demonstration of how to use Barman with Armv7-M and Armv8-M, from configuring the bare-metal agent to analyzing the results.

`u-boot-instrumentation`
An example of how to modify U-Boot to allow it to be profiled using Barman.

Copyright © 2019, 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential