Arm® Streamline

Version 9.0

Target Setup Guide for Bare-metal Applications

Non-Confidential

Issue 00

101815_0900_00_en

Arm® Streamline

Target Setup Guide for Bare-metal Applications

Release information

Document history

Issue	Date	Confidentiality	Change
0708-00	20 August 2021	Non-Confidential	New document for v7.8
0709-00	18 November 2021	Non-Confidential	New document for v7.9
0800-00	18 February 2022	Non-Confidential	New document for v8.0
0801-00	20 May 2022	Non-Confidential	New document for v8.1
0802-00	19 August 2022	Non-Confidential	New document for v8.2
0803-00	17 November 2022	Non-Confidential	New document for v8.3
0804-00	14 February 2023	Non-Confidential	New document for v8.4
0805-00	20 April 2023	Non-Confidential	New document for v8.5
0806-00	8 June 2023	Non-Confidential	New document for v8.6
0807-00	3 August 2023	Non-Confidential	New document for v8.7
0808-00	28 September 2023	Non-Confidential	New document for v8.8
0809-00	23 November 2023	Non-Confidential	New document for v8.9
0900-00	15 February 2024	Non-Confidential	New document for v9.0

Proprietary Notice

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.

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Introduction......7
1.1 Conventions......7
1.2 Useful resources......8
1.3 Other information......9

2. Bare-metal Support......10
2.1 Bare-metal support overview......10

3. Profiling with the bare-metal agent......11
3.1 Profiling with Barman......11
3.2 Data synchronization......12
3.3 Data storage......12
3.4 Profiling with on-target RAM buffer......13
3.4.1 Configuring Barman......13
3.4.2 Extracting and importing data......19
3.4.3 Barman use case script......20
3.5 Profiling with System Trace Macrocell......21
3.5.1 STM workflow......21
3.5.2 Importing an STM trace......23
3.6 Profiling with Instrumentation Trace Macrocell......24
3.6.1 ITM workflow......24
3.6.2 Importing an ITM trace......27
3.7 Interfacing with Barman......27
3.7.1 Configuration #defines......28
3.7.2 Annotation #defines......29
3.7.3 Barman public API......29
3.7.4 External functions to implement......39
3.7.5 Write barman profile data to your own data storage mechanism......43
3.7.6 Write barman profile data in the memory buffer to custom storage......44
3.7.7 Write barman profile data to custom storage......45
3.8 Custom counters......48
3.8.1 Configuring custom counters......48
3.8.2 Sampled and nonsampled counters......49

3.9 Using the bare-metal generation mechanism from the command line.. 50

4. Examples.. 52

4.1 Examples using Barman.. 52

1. Introduction

Learn how to use Streamline to set up your bare-metal device for debugging and profiling.

1.1 Conventions

The following subsections describe conventions used in Arm documents.

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: developer.arm.com/glossary.

Typographic conventions

Arm documentation uses typographical conventions to convey specific meaning.

Convention	Use
italic	Citations.
bold	Interface elements, such as menu names. Terms in descriptive lists, where appropriate.
`monospace`	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace underline`	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`<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	Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Caution

We recommend the following. If you do not follow these recommendations your system might not work.

Warning

Your system requires the following. If you do not follow these requirements your system will not work.

Danger	You are at risk of causing permanent damage to your system or your equipment, or of harming yourself.

Note	This is important information and needs your attention.

Tip	A useful tip that might make it easier, better or faster to perform a task.

Remember	A reminder of something important that relates to the information you are reading.

1.2 Useful resources

This document contains information that is specific to this product. See the following resources for other useful information.

Access to Arm documents depends on their confidentiality:

- Non-Confidential documents are available at developer.arm.com/documentation. Each document link in the following tables goes to the online version of the document.
- Confidential documents are available to licensees only through the product package.

Arm product resources	Document ID	Confidentiality
Arm Development Studio Getting Started Guide	101469	Non-Confidential
Arm Development Studio User Guide	101470	Non-Confidential
Arm Streamline User Guide	101816	Non-Confidential

Non-Arm resources	Document ID	Organization
Sources for Arm® Streamline Performance Analyzer's gator daemon	-	GitHub

1.3 Other information

See the Arm website for other relevant information.

- Arm® Developer.
- Arm® Documentation.
- Technical Support.
- Arm® Glossary.

2. Bare-metal Support

Describes the bare-metal support available in Streamline.

2.1 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 the agent Barman.

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.

Related information

Profiling with Barman on page 11

3. Profiling with the bare-metal agent

This section explains how to profile your application with the bare-metal agent (Barman) with different data storage modes.

3.1 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
- Armv9-A

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 information

Data storage on page 12
Profiling with on-target RAM buffer on page 12
Profiling with System Trace Macrocell on page 21
Profiling with Instrumentation Trace Macrocell on page 24
Interfacing with Barman on page 27

3.2 Data synchronization

On Cortex®-A and Cortex-R systems Barman uses load/store exclusive operations to synchronize processor access to shared state and data storage. The memory used for Barman program data must be backed by a memory pages that support exclusive operations on the target platform.

See the Arm Architecture Reference Manual *Synchronization and Semaphore* section for the memory requirements for exclusive operations.

3.3 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.

3.4 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.

3.4.1 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.

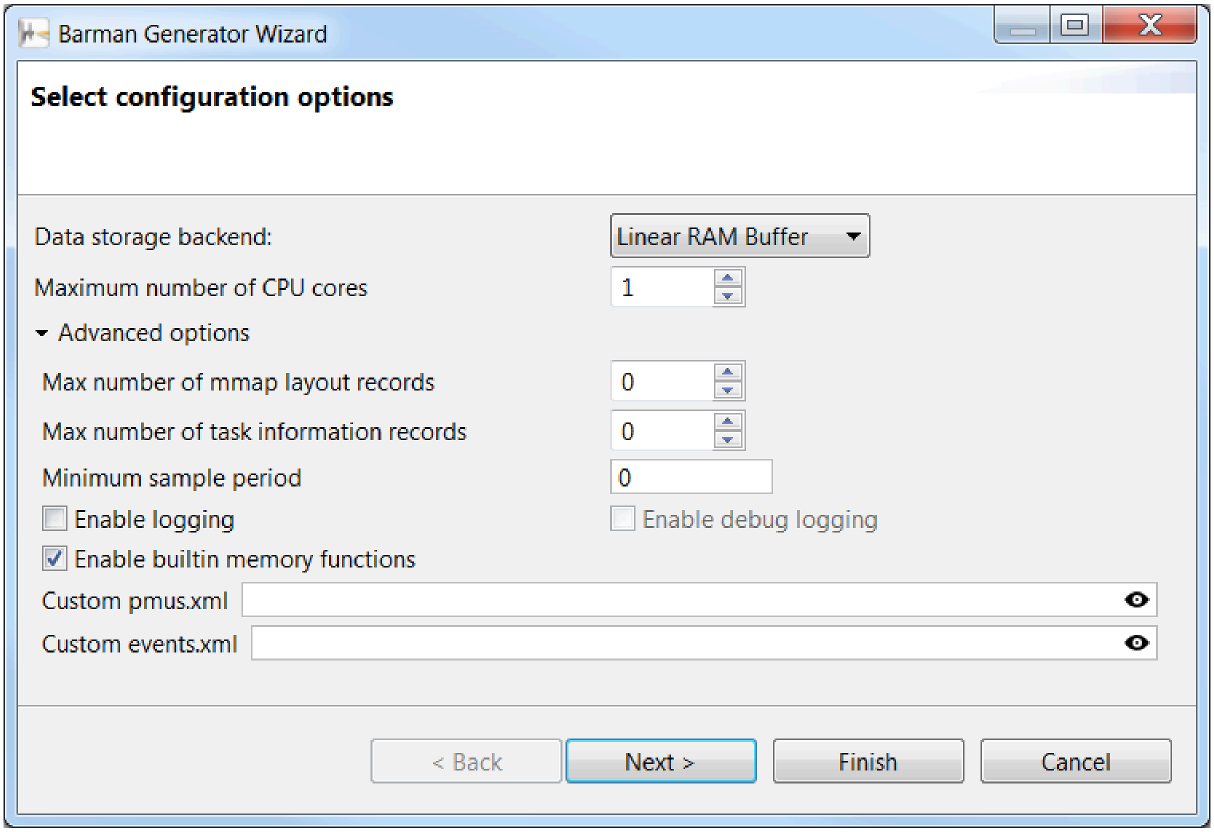
About this task

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).

Figure 3-1: Select configuration options dialog.



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.

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 Profiling with System Trace Macrocell for information about using the **STM Interface** data storage backend.

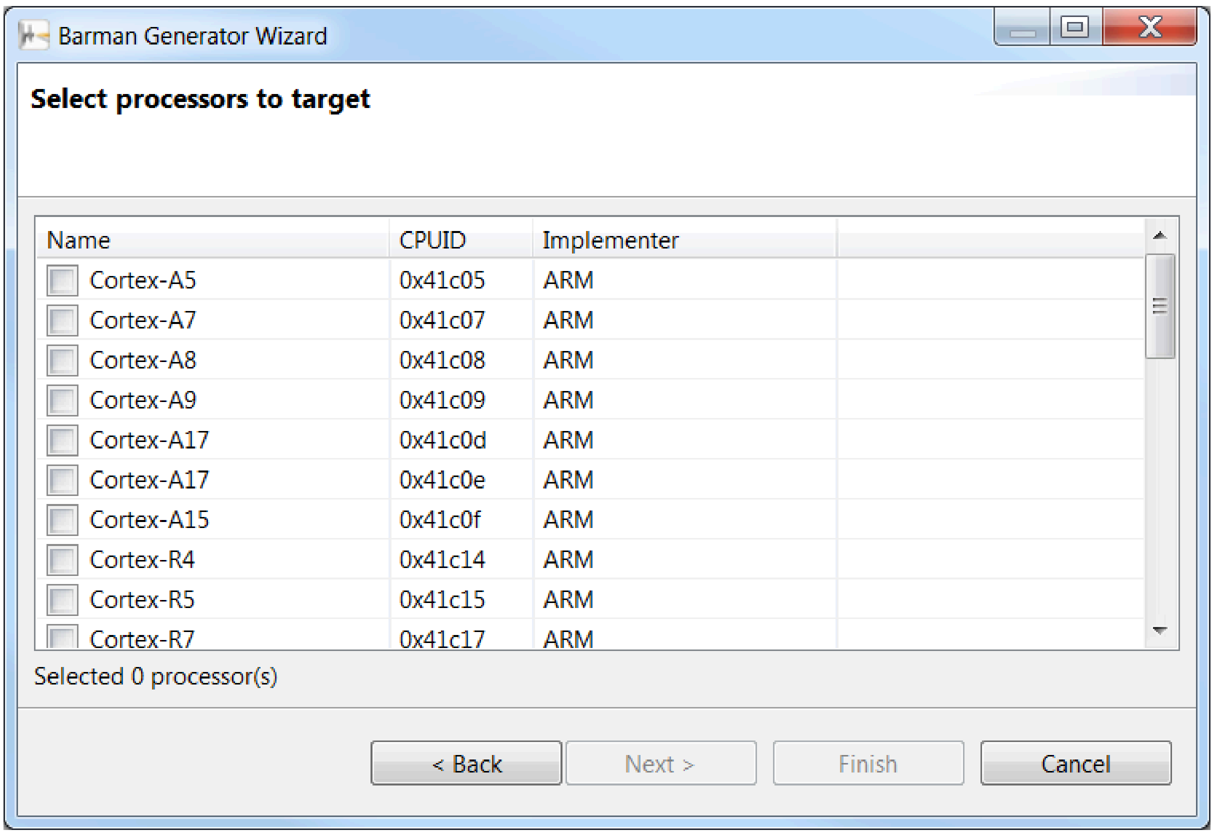
See Profiling with Instrumentation Trace Macrocell for information about using the **ITM Interface** data storage backend.

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

---

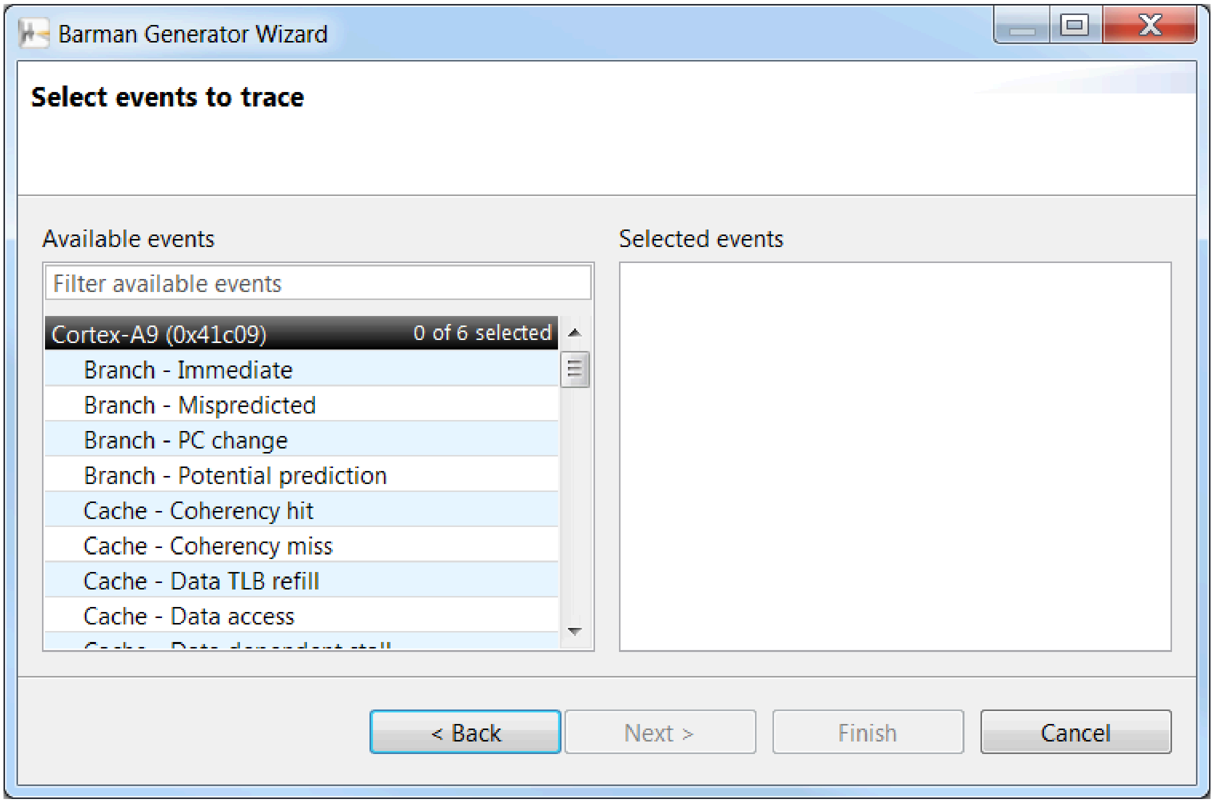
3. Select the target processor from the pre-defined list.

Figure 3-2: Select processors to target dialog.



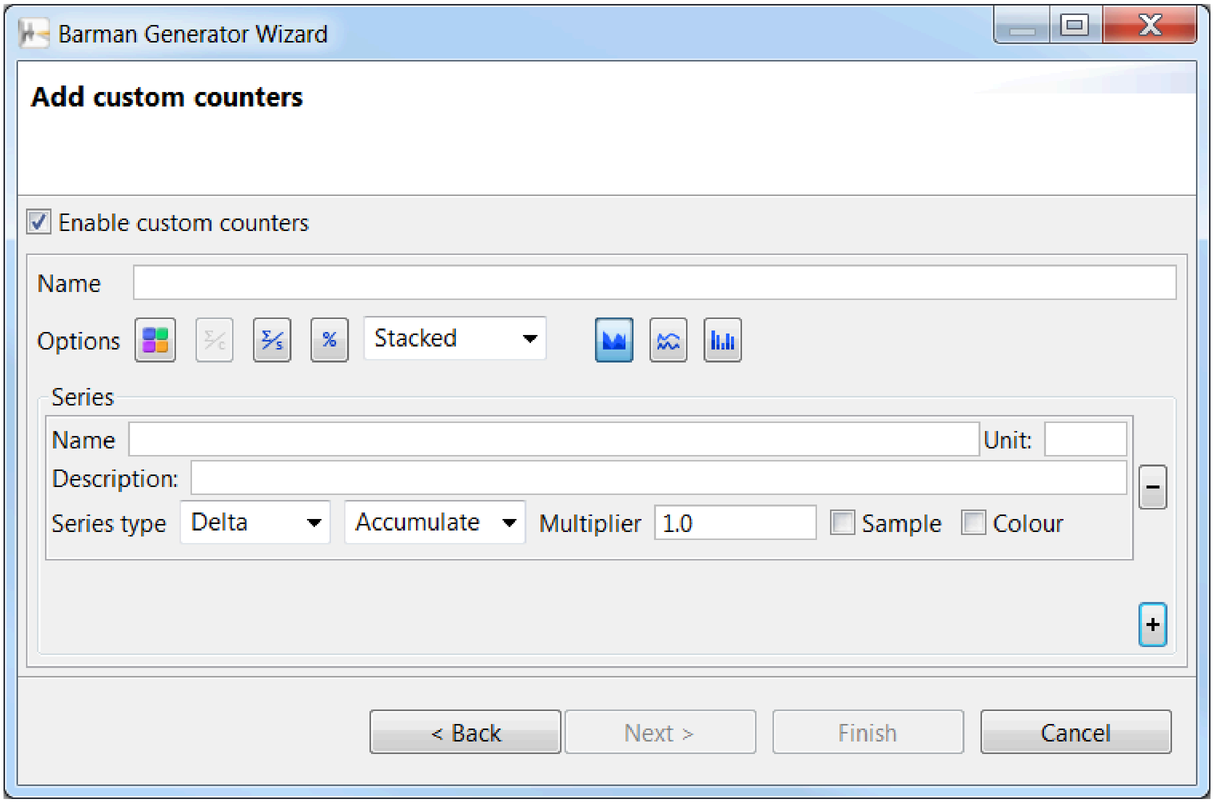
4. Select the PMU counters to collect during the capture session by double-clicking on them in the **Available events** list. Alternatively you can drag and drop the events into the **Selected events** list. To deselect events, drag and drop them back into the **Available events** list.

Figure 3-3: Select events to trace dialog.



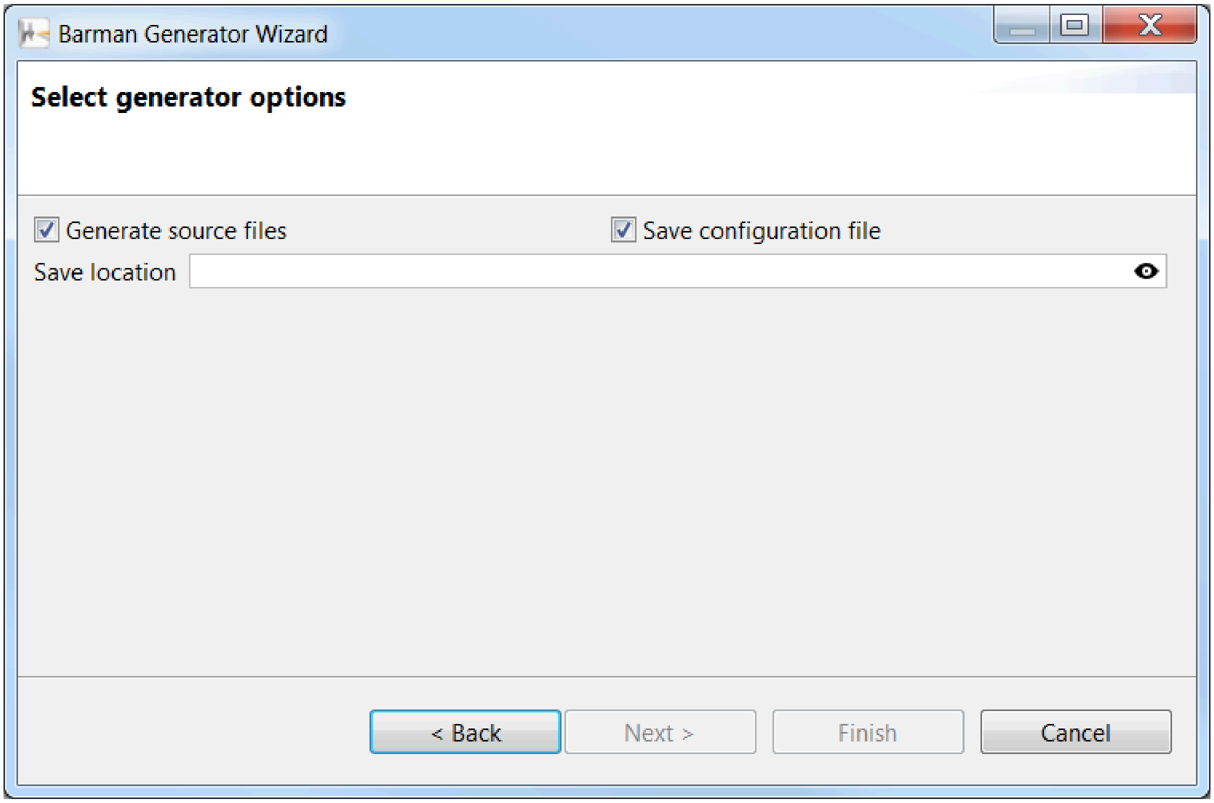
5. Add custom counters.

Figure 3-4: Add custom counters dialog.



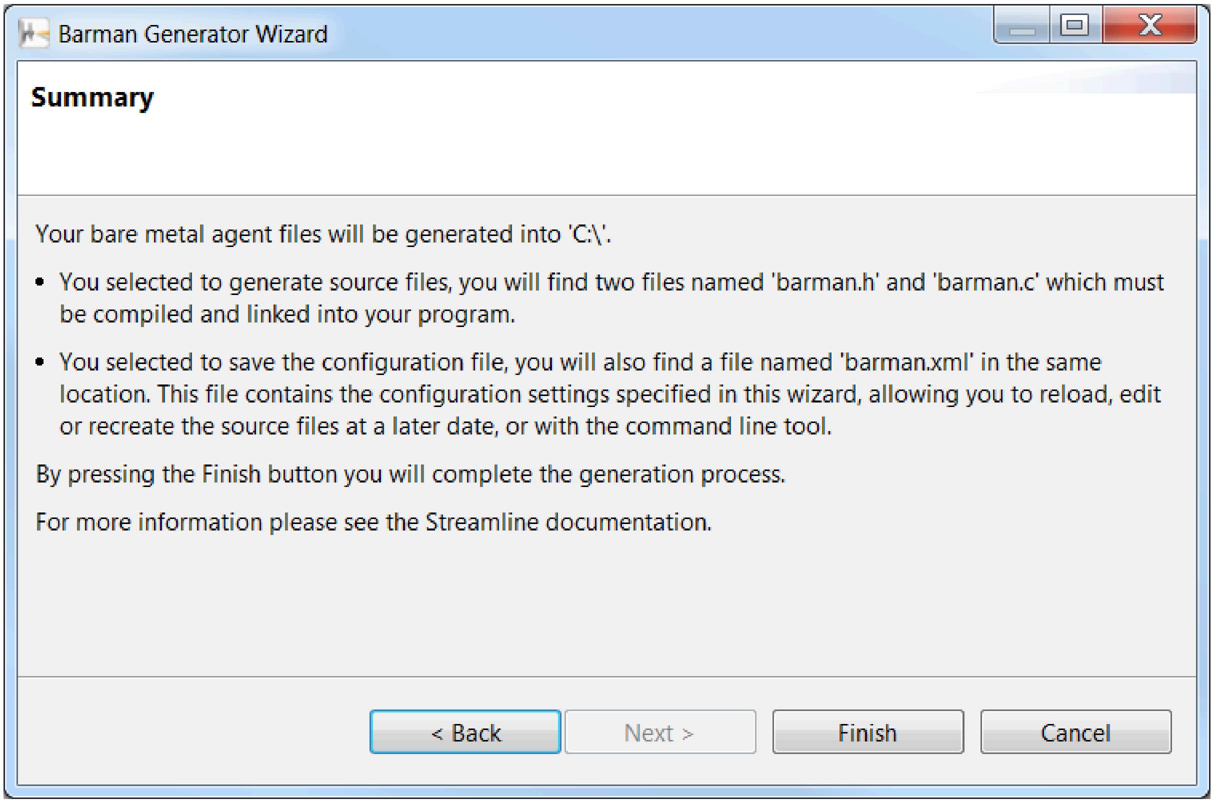
6. Select generator options.

Figure 3-5: Select generator options dialog.



7. Finish.

Figure 3-6: Summary information.



Results

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 information

Barman use case script on page 20

3.4.2 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.

3.4.3 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
```

```
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 information

Configuring Barman on page 13
Use case scripts

3.5 Profiling with System Trace Macrocell

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

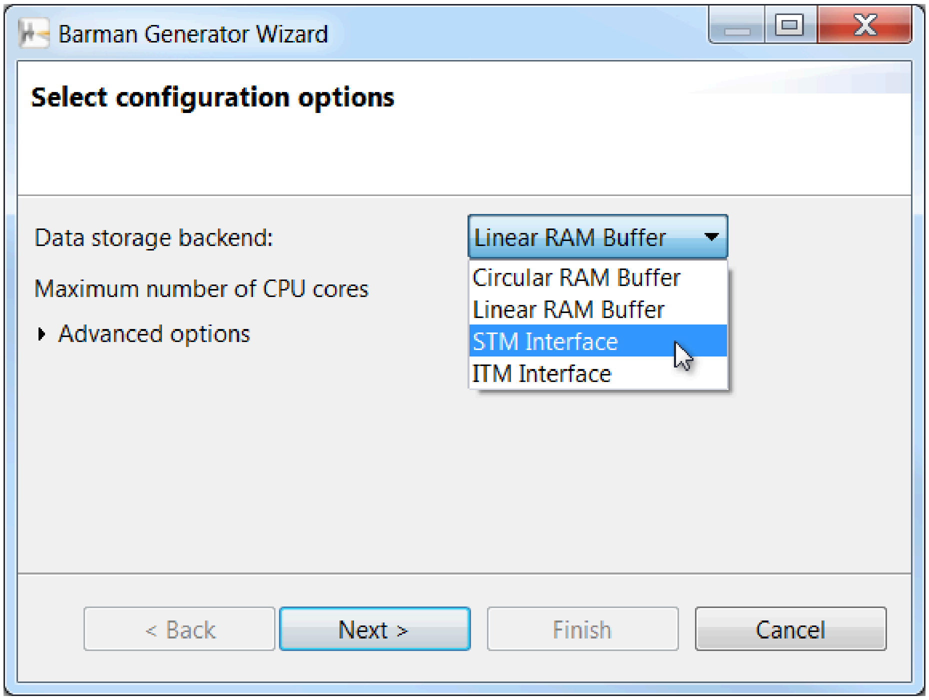
Further information about STM, including the Technical Reference Manual, can be found on System Trace Macrocell Arm Developer documentation.

3.5.1 STM workflow

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

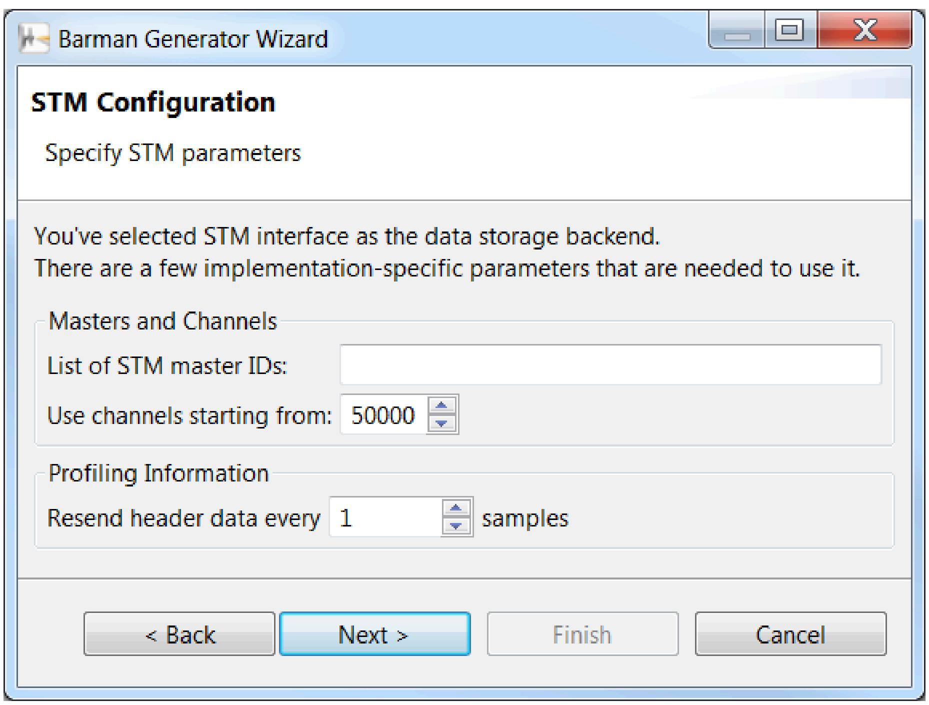
1. Generate Barman agent code for STM using the **Barman Generator Wizard** dialog in Streamline.
   a. Select **STM Interface** as the data storage backend.

Figure 3-7: Select STM backend.



b. Specify the STM parameters for your project.

Figure 3-8: STM configuration.



> **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 information

Configuring Barman on page 13

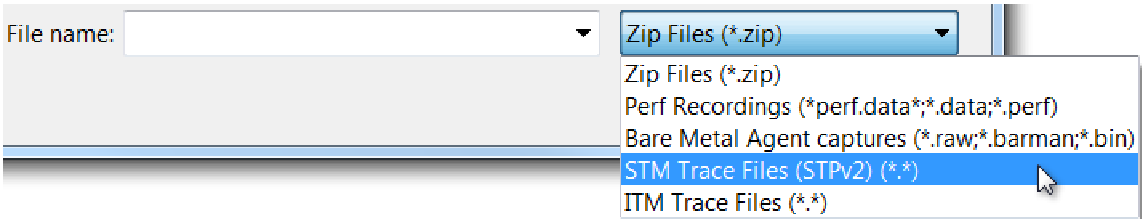
3.5.2 Importing an STM trace

Import STM trace files into Streamline for analysis.

Procedure

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

Figure 3-9: Selecting the STM file type.



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**.

Results

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

Related information

Import an STM trace from the command line

3.6 Profiling with Instrumentation Trace Macrocell

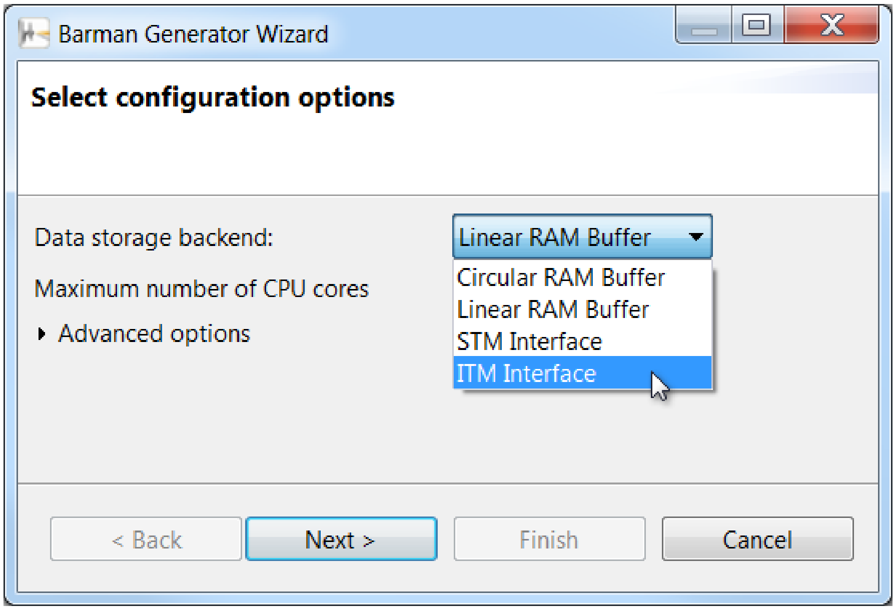
This section describes the collection of profiling data using Instrumentation Trace Macrocell (ITM).

3.6.1 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.

Figure 3-10: Select ITM backend.

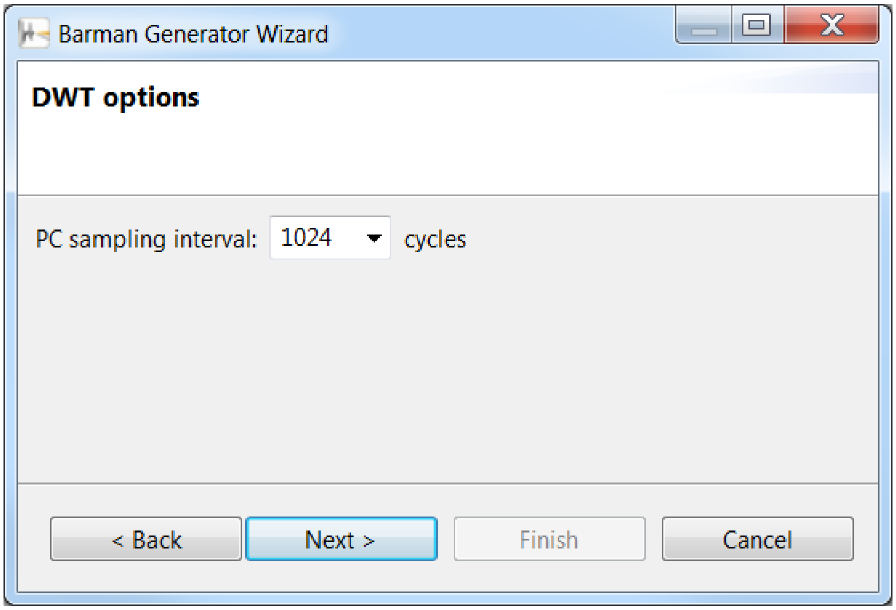


Note: Barman uses ports 16-19 for ITM.

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**.

Figure 3-11: Select PC sampling interval.



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 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 information

Configuring Barman on page 13

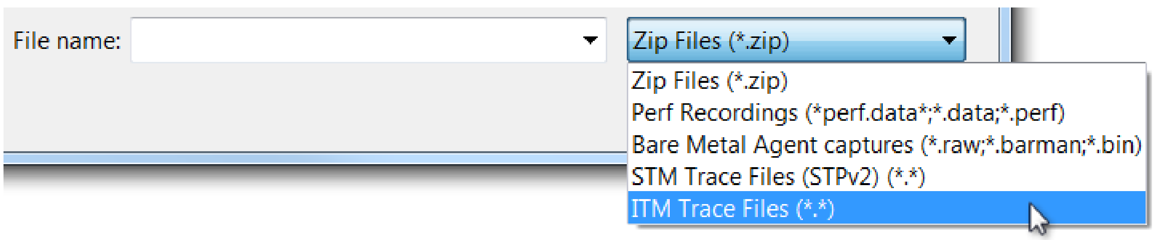
3.6.2 Importing an ITM trace

Import ITM trace files into Streamline for analysis.

Procedure

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

   Figure 3-12: Selecting the ITM file type.



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**.

Results

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

Related information

Import an ITM trace from the command line

3.7 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.

Note

To sample code running at EL3 using Barman, some additional configuration may be required. By default, counting of events is prohibited when the processor is executing at EL3. To change this behavior:

- When EL3 is using Aarch64, the register field `MDCR_EL3.SPME` must be **1**
- When EL3 is using Aarch32, the register field `SDCR.SPME` must be **1**

See the Arm Architecture Reference Manual *Prohibiting event and cycle counting* section for more information.

3.7.1 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.

Table 3-1: Defines available for configuration

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.

3.7.2 Annotation #defines

Color macros to use for annotations.

Table 3-2: Color macros to use for annotations

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 ≤ `R` ≤ 255. **B** The blue component, where 0 ≤ `B` ≤ 255. **G** The green component, where 0 ≤ `G` ≤ 255.

3.7.3 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);
```

Table 3-3: barman_initialize function information

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 Hertz. Zero indicates no timer-based sampling (assumes a maximum 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);
```

Table 3-4: barman_enable_sampling function information

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

barman_disable_sampling

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

Table 3-5: barman_disable_sampling function information

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

barman_sample_counters

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

Table 3-6: barman_sample_counters function information

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);
```

Table 3-7: barman_sample_counters_with_program_counter function information

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);
```

Table 3-8: barman_add_task_record function information

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.

barman_record_task_switch

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

Table 3-9: barman_record_task_switch function information

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);
```

Table 3-10: barman_add_mmap_record function information

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;
};
```

Table 3-11: bm_protocol_clock_info function information

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;
};
```

Table 3-12: bm_protocol_task_info function information

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;
};
```

Table 3-13: bm_protocol_mmap_layout function information

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.

bm_task_switch_reason

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

Table 3-14: bm_task_switch_reason function information

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);
```

Table 3-15: barman_wfi function information

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);
```

Table 3-16: barman_wfe function information

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);
```

Table 3-17: barman_before_idle function information

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()` or `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);
```

Table 3-18: barman_after_idle function information

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)
```

Table 3-19: barman_annotate_channel function information

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)
```

Table 3-20: barman_annotate_name_channel function information

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)
```

Table 3-21: barman_annotate_name_group function information

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)
```

Table 3-22: barman_annotate_marker function information

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

Table 3-23: bm_annotation_colors function information

Description	Color macros for annotations. See Annotation #defines .

3.7.4 External functions to implement

You must provide the following external functions.

barman_ext_get_timestamp

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

Table 3-24: barman_ext_get_timestamp function information

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);
```

Table 3-25: barman_ext_disable_interrupts_local function information

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);
```

Table 3-26: barman_ext_enable_interrupts_local function information

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`:

barman_ext_map_multiprocessor_affinity_to_core_no

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

Table 3-27: barman_ext_map_multiprocessor_affinity_to_core_no function information

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. See the following example.
Parameter	**`mpidr`** The value of the MPIDR register.
Return value	The processor number.

Example

```
//
// 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);
}
```

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);
```

Table 3-28: barman_ext_map_multiprocessor_affinity_to_cluster_no function information

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. See the following example.
Parameter	**`mpidr`** The value of the MPIDR register.
Return value	The cluster number.

Example

```
//
// 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);
}
```

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);
```

Table 3-29: barman_ext_get_current_task_id function information

Description	Returns the current task ID.

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

barman_ext_log_info

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

Table 3-30: barman_ext_log_info function information

Description	Prints an info message.
Parameter	`message`

barman_ext_log_warning

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

Table 3-31: barman_ext_log_warning function information

Description	Prints a warning message.
Parameter	`message`

barman_ext_log_error

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

Table 3-32: barman_ext_log_error information

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, ...);
```

Table 3-33: barman_ext_log_debug function information

Description	Prints a debug message.

Parameter	`message`

3.7.5 Write barman profile data to your own data storage mechanism

You can interface with barman and instruct it to store capture data in a RAM buffer, or to stream the data, and write it to an external data storage backend of your choice.

> **Note**
> Collecting profiling data with barman is not supported for Cortex®-M targets.

Barman allows you to collect capture data in the RAM buffer on your target, or to stream the data using a trace probe (such as a DSTREAM). The captured or streamed data can then be:

- Processed and saved by a debugger:
  - To learn about processing profile data stored on an on-target RAM buffer, see: Profiling with on-target RAM buffer.
  - To learn about processing profile data streaming from your target, see: Profiling with System Trace Macrocell and Profiling with Instrumentation Trace Macrocell.
- Processed or streamed directly to your own data handler or storage:
  - To learn about writing data from the on-target RAM buffer to your own storage location, see: Write barman profile data in the memory buffer to custom storage.

    > **Note**
    > When processing data stored in a RAM buffer, your in-memory buffer must be large enough to hold this data until the end of the capture when that data can be written out.

  - To learn about writing data to your own storage location instead of the on-target RAM buffer, see: Write barman profile data to custom storage.

    When data is streamed to your own storage location, you do not need to provide as much memory in advance, and can capture a lot more data over a longer period of time.

    > **Note**
    > - You can also use this mechanism to stream data off the target, as your capture continues to run, either through your network or through physical device connection cables (for example, USB).
    > - This method is only supported in Arm® Streamline Performance Analyzer version 8.7, and later versions.

Related information

Profiling with System Trace Macrocell on page 21
Profiling with Instrumentation Trace Macrocell on page 24

Write barman profile data in the memory buffer to custom storage on page 44
Write barman profile data to custom storage on page 44

3.7.6 Write barman profile data in the memory buffer to custom storage

This topic describes how to interface, and instruct, barman to store capture data in a RAM buffer then write out the data to a storage of your choice.

Procedure

1. Follow the steps in Configuring Barman to configure barman to use an on-target RAM buffer. As part of the integration process, you must initialize the barman agent with a user-provided memory buffer. Because you are providing the memory buffer, you can choose how to save its contents when the profiled code has finished running.
2. After the sampling has stopped, store the contents of the memory buffer using your chosen mechanism. You can, for example, save to file or send over a network connection if your platform supports these options.
   The following example stops the sampling by calling `barman_disable_sampling()`, and stores the memory buffer contents to a file called `barman.raw`:

```
/* Define the RAM buffer used to store the capture data.
 */
#define SIZE  4096*4096
bm_uint8 data[SIZE] __attribute__((aligned(8)));

/* Application initializes Barman.
 */
void my_app_profiling_setup_code()
{
    barman_initialize(data, SIZE, "barman-example", &clock_info, 0);
    barman_enable_sampling();
}

/* Application provides a mechanism to stop profiling and save the buffer.
 */
void my_app_profiling_tear_down_code()
{
    barman_disable_sampling();
    /* This function uses the API that your embedded software or RTOS
     * provides to store the buffer contents to a file.
     */
    write_byte_buffer_to_file("barman.raw", data, SIZE);
}
```

Related information

Write barman profile data to custom storage on page 44
Write barman profile data to your own data storage mechanism on page 43

3.7.7 Write barman profile data to custom storage

This task describes how to interface with, and instruct, barman to store (or stream) captured data to storage of your choice.

Procedure

1. To configure barman to use an in-memory buffer, follow the steps in Configuring Barman .
   This step generates the `barman.c` and `barman.h` files. The following steps override the backend selected in the configuration tool.

   When initializing the barman library, use the `barman_initialize_with_user_supplied` function instead of `barman_initialize`.

   The first argument passed to `barman_initialize_with_user_supplied` gets passed directly to the `barman_ext_streaming_backend_init` function shown in the following example.
2. Prepare your code by implementing the functions described in the `barman-ext-streaming-backend.h` header file that is available on GitHub:
   For example, you can implement the functions as:

```
/**
 * Initialize the backend.
 *
 * @param config    Pointer to some configuration data.
 * @return          True if successful.
 */
 bm_bool barman_ext_streaming_backend_init(void * config);

/**
 * Write data as a frame.
 *
 * @param data    Data to write in the frame.
 * @param length  Length of the frame.
 * @param channel Channel to write the frame on.
 * @param flush_header   Set to BM_TRUE when the frame contains an update.
 * to the header. Indicates to flush the channel after writing the frame.
 */
 void barman_ext_streaming_backend_write_frame(const bm_uint8 * data, bm_uintptr
 length, bm_uint16 channel, bm_bool flush_header);

/**
 * Shutdown the backend.
 */
 void barman_ext_streaming_backend_close(void);

/**
 * Get the channel bank.
 *
 * If banked by a core this is just the core number.
 * If not banked by a core, this should always be 0.
 *
 * @return The bank
 */
 bm_uint32 barman_ext_streaming_backend_get_bank(void);
```

- The function `barman_ext_streaming_backend_init` must perform whatever necessary setup (for example, opening the data file), and returns `BM_TRUE` on success, or `BM_FALSE` on failure.

- The function `barman_ext_streaming_backend_close` must be called when the capture is disabled, and used to close any relevant storage (for example, by closing the data file).
- The function `barman_ext_streaming_backend_get_bank` must return `barman_get_core_no()` in a multicore system, or return `0`:

```
bm_uint32 barman_ext_streaming_backend_get_bank(void)
{
    return barman_get_core_no();
}
```

- The function `barman_ext_streaming_backend_write_frame` must be called for each data record, and store the records to the data store according to the following pseudocode:

```
{
   /* Note: When the host has multiple cores or threads, so that it is
    * possible for multiple cores to call this function in parallel,
    * then you must ensure to synchronize here.
    */

    if (flush_header) {
        assert(data_store_is_empty || (previous_header_length == length));
       /** Write (or overwrite) the existing header at the start of the
         * data store.
        */
        write_blob_to_data_store_at_offset_zero(data, length);
    } else {
        assert(received_header);

       /** You must prefix the data record with a length field. Note:
        * This following example code produces a little-endian length,
        * but for big-endian targets this must be changed.
        */
        bm_uint8 length_header[8] = {
            length >> (0 * 8),
            length >> (1 * 8),
            length >> (2 * 8),
            length >> (3 * 8),
            length >> (4 * 8),
            length >> (5 * 8),
            length >> (6 * 8),
            length >> (7 * 8),
        };

       /** Append the size, which depends on if the target is 32-bit
        * (4-bytes) or 64-bit (8-bytes), to the end of the data store.
        */
        write_blob_to_end_of_data_store(length_header, sizeof(bm_uintptr));

       /** Now append the data to the end of the data store.
        */
        write_blob_to_end_of_data_store(data, length);
    }
}
```

For example, on a POSIX-like API, you could implement `write_blob_to_data_store_at_offset_zero` and `write_blob_to_end_of_data_store` as:

```
static int file_handle;
static bm_uintptr header_length = 0;

static void write_blob(const bm_uint8 * data, bm_uintptr length)
{
```

```
    while (length > 0)
    {
        int result = write(file_handle, data, length);

        assert(result > 0);
        length -= result;
        data += result;
    }
}

static void write_blob_to_end_of_data_store(const bm_uint8 * data, bm_uintptr
 length)
{
    off_t new_offset = lseek(file_handle, 0, SEEK_END);
    assert((new_offset > 0) && (new_offset >= header_length));
    write_blob(data, length);
}

static void write_blob_to_data_store_at_offset_zero(const bm_uint8 * data,
 bm_uintptr length)
{
    assert((header_length == 0) || (header_length == length));
    header_length = length;
    off_t new_offset = lseek(file_handle, 0, SEEK_SET);

    assert(new_offset == 0);

    write_blob(data, length);
    new_offset = lseek(file_handle, 0, SEEK_END);

    assert(new_offset >= length);
}
```

3. Compile your application with the following options:
   - `-DBM_CONFIG_USE_DATASTORE=BM_CONFIG_USE_DATASTORE_STREAMING_USER_SUPPLIED` - Sets a user-supplied datastore.
   - `-DBM_CONFIG_STREAMING_DATASTORE_USER_SUPPLIED_NUMBER_OF_BANKS=<n>` - Passes the number of banks in the user-supplied datastore. `<n>` is the maximum number of CPUs on the system. In other words, one greater than the largest value returned by `barman_get_core_no()`.
   - `-DBM_CONFIG_STREAMING_DATASTORE_USER_SUPPLIED_NUMBER_OF_CHANNELS=<n>` - Passes the number of channels in the user-supplied datastore. `<n>` is a number greater than, or equal to, 1, and is the number of buffers per CPU to use for temporary storage.

```
./<compiler-command> <options> -
DBM_CONFIG_USE_DATASTORE=BM_CONFIG_USE_DATASTORE_STREAMING_USER_SUPPLIED
 -DBM_CONFIG_STREAMING_DATASTORE_USER_SUPPLIED_NUMBER_OF_BANKS=<n> -
DBM_CONFIG_STREAMING_DATASTORE_USER_SUPPLIED_NUMBER_OF_CHANNELS=<n> <source>
```

Related information

Write barman profile data in the memory buffer to custom storage on page 44
Write barman profile data to your own data storage mechanism on page 43
GitHub

3.8 Custom counters

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

3.8.1 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

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.

3.8.2 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.

3.9 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

4. Examples

This section contains information about the bare-metal examples that are supplied with Streamline.

4.1 Examples using Barman

Streamline includes several examples of how to use Barman.

You can find these examples in <install_directory>/sw/streamline/examples/barman.

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.

RTX5_Cortex-A9_Blinky_Streamline A demonstration of how to use Barman with the CMSIS RTX5 RTOS on a Cortex-A9 processor, collection of profiling information from RAM with DSTREAM, and analysis in Streamline.

RTX5_Cortex-M33_Blinky_Streamline A demonstration of how to use Barman with the CMSIS RTX5 RTOS on a Cortex-M33 processor, collection of profiling information via ITM with DSTREAM, and analysis in Streamline.