arm

Learn the architecture - Understanding trace
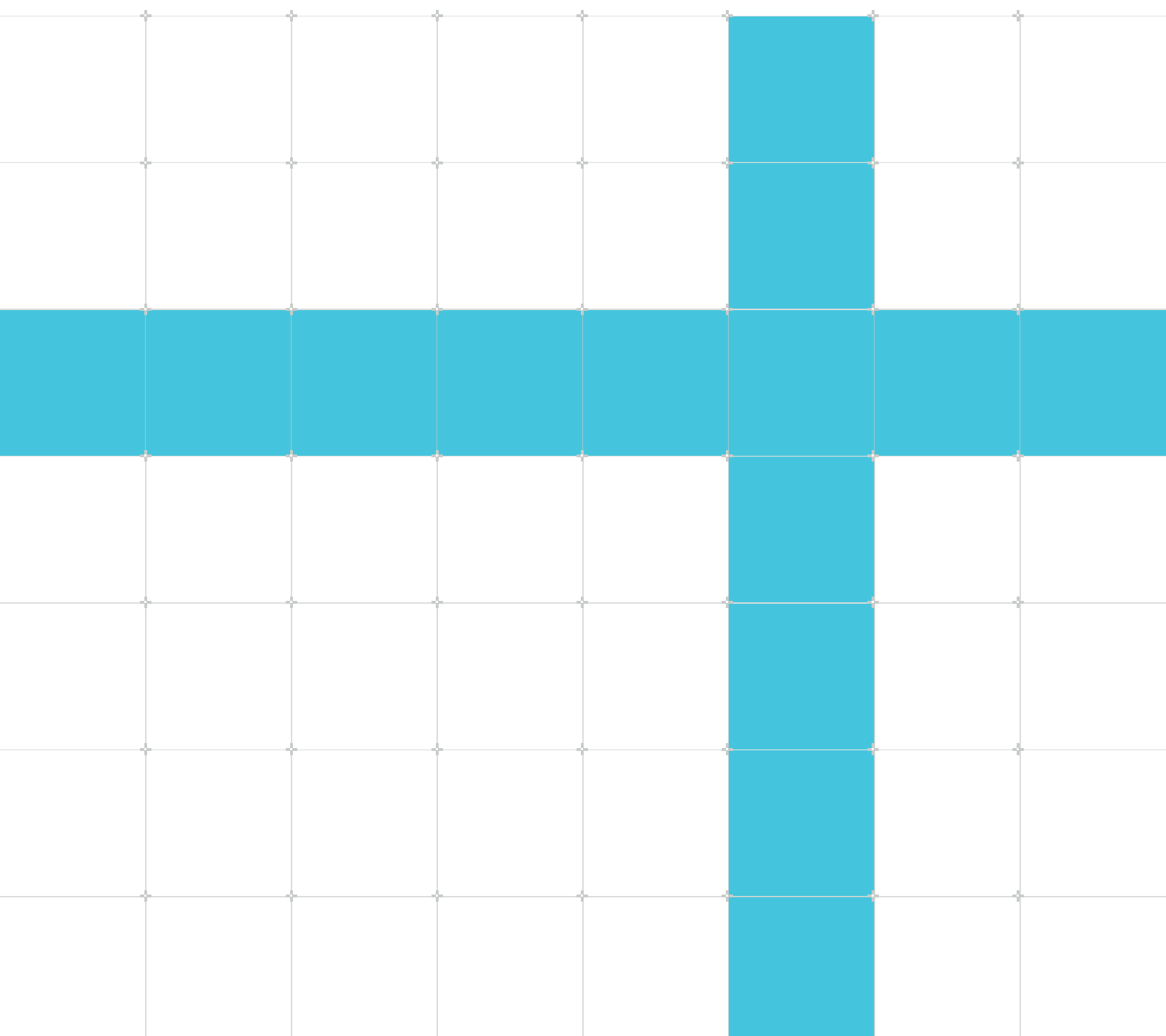
Version 2.0

Learn the architecture - Understanding trace

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	24 March 2020	Non-Confidential	First release
0200-02	14 May 2020	Non-Confidential	Updated the images

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. Overview

This guide focuses on a high-level view of trace in Armv7 systems, and Armv8 systems up to version Armv8.4.

The guide covers:

- What trace is and how it is used

- How the trace architecture is defined and how it maps on to the different trace component implementations

- What trace components are seen in Arm systems

- Examples of some trace systems

The information in this guide is tools agnostic. Screenshots of the Arm Debugger Trace view are used to illustrate points that are made in the guide.

This guide does not cover low-level trace details like trace protocols, trace capture mechanisms, trace data decode, or Tarmac trace.

Before you begin

We assume that you are familiar with CoreSight. If you are not, read our CoreSight guide (coming soon).

2. What is trace?

Trace refers to the process of capturing data that illustrates how the components in a design are operating, executing, and performing.

The ability to trace a target depends on what trace facilities the target offers. For example, you can only store trace data on a target for later analysis if the target offers the facilities to do so. If you have questions about the trace capabilities of your target, refer to the target manufacturer, target designer, and target documentation.

The Arm approach to trace usually involves a separate trace generation component for each type of trace that is performed. For example, different trace sources produce processor trace and bus trace.
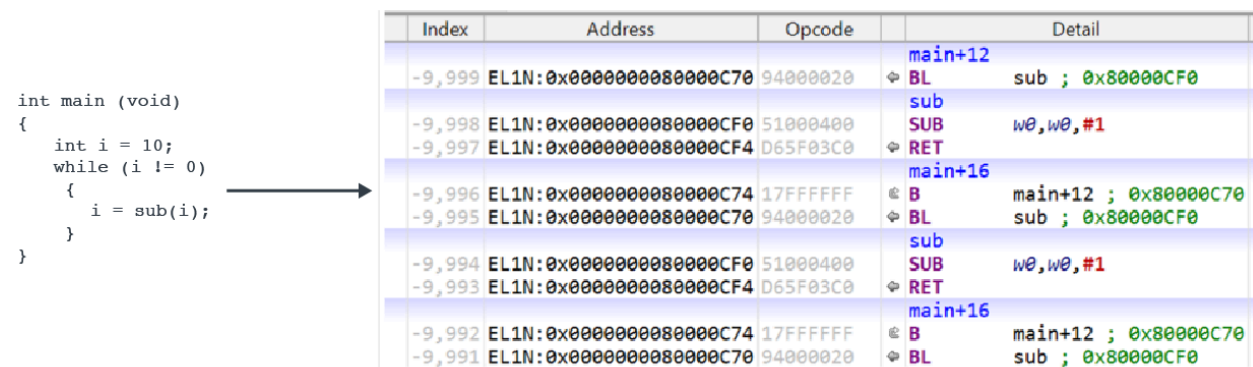
In this section of the guide, we look at the different kinds of trace and the components that produce them.

Instruction trace

Instruction trace generates information about the instruction execution of a core or processor. In a simple example, if a core executes a loop ten times and instruction trace is enabled, the decoded instruction trace data shows the associated loop code ten times.

The following diagram shows two iterations of a simple loop example through instruction trace:

Figure 2-1: Instruction trace

The trace capabilities of your target might allow additional information, for example cycle counter numbers or timestamps, to be captured alongside the trace data. You can choose the additional information that best fits your tracing requirements.
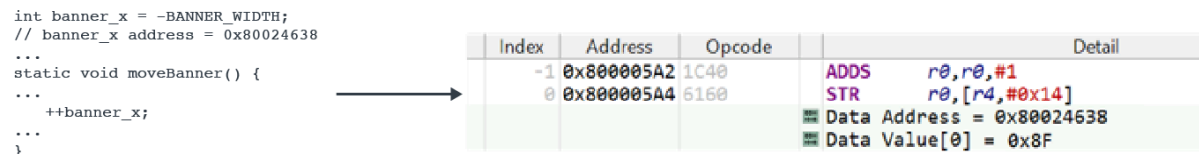
The Arm trace sources that generate instruction trace are the Embedded Trace Macrocell (ETM) and the Program Trace Macrocell (PTM). Whether a target includes an ETM or a PTM depends on the processor that is in the design. Armv8 designs have an ETM. Most Armv7 designs have a PTM.

Data trace

Data trace generates information about the data accesses of a core or processor. For example, if a memory load instruction is executed and data trace is enabled, the data trace shows a load instruction with the associated load address and value.

The following diagram shows how a data access appears through data trace:

Figure 2-2: Data trace

The Embedded Trace Macrocell (ETM) is the Arm trace source that captures data trace. ETM data trace capability is an optional feature in the ETM architecture. ETMv4 does not support data trace on Armv8 or Armv7 processors.

Instrumentation trace

Instrumentation trace outputs Operating System (OS) and application events and system information. For example, if an event occurs when an application runs and instrumentation trace is enabled, the environment pushes useful runtime information to the instrumentation trace source to analyze later.

The following diagram shows how Data Watchpoint and Trace unit (DWT) values that are written to an ITM appear through instrumentation trace:

Figure 2-3: Instrumentation trace

Instrumentation trace is versatile, but its capabilities depend on how it is implemented in the target design. Refer to your target manufacturer, target designer, and target documentation for information on the instrumentation trace capability of your target.

Arm designs use an Instrumentation Trace Macrocell (ITM) to capture instrumentation trace data.

System trace

System trace outputs data about components across the system. For example, the STM supports both target hardware and software event generation. System trace components have a superset of the functionality of an instrumentation trace component. This means that there are many similarities between the two components.

The following diagram shows how outputting an application text string and application-generated numbers to an STM appear through system trace:

Figure 2-4: System trace

Master 64 and all channels enabled. (DSTREAM: STM) (page 3 of 3+ from end of buffer)

Buffer Used: 5.6 MB

Mstr	Chnl	TS	Size	Data
64	0	03:49.880217	8	ambridge
64	0	03:49.881085	1	C
64	0	03:49.881085	8	ambridge
64	0	03:49.881890	1	C
64	0	03:49.881890	8	ambridge
64	0	03:49.882973	1	C
64	0	03:49.882973	8	ambridge
64	0	03:49.883745	1	C
64	0	03:49.883745	8	ambridge
64	0	03:49.884524	1	C
64	0	03:49.884524	8	ambridge
64	0	03:49.885415	1	C
64	0	03:49.885415	8	ambridge
64	0	03:49.886302	1	C
64	0	03:49.886302	8	ambridge
64	0	03:49.887270	1	C
64	0	03:49.887270	8	ambridge
64	0	03:49.888164	1	C
64	0	03:49.888164	8	ambridge
64	0	03:49.889048	1	C
64	0	03:49.889048	8	ambridge
64	0	03:49.890053	1	C
64	0	03:49.890053	8	ambridge
64	0	03:49.890907	1	C
64	0	03:49.890907	8	ambridge
64	0	03:49.891688	1	C
64	0	03:49.891688	8	ambridge
64	1	03:49.892830	1	0x29
64	1	03:49.894497	1	0xD
64	1	03:49.894497	1	0x2B
64	1	03:49.895226	1	0xE
64	1	03:49.895226	1	0x2F
64	1	03:49.896147	1	0xF

Outputting a text string and numbers from an application to STM

System trace is versatile, but its capabilities depend on how it is implemented in the target design. Refer to your target manufacturer, target designer, and target documentation for information on the system trace capability of your target.

Arm designs use a System Trace Macrocell (STM) to capture system trace data. The two variants are the STM and the STM-500.

Learn the architecture - Understanding trace

Document ID: 102119_0200_02_en
Version 2.0
What is trace used for?

3. What is trace used for?

As shown in What is trace?, trace in Arm systems captures various different target operations and information. This means that trace can help:

- Diagnose problems during runtime:
    - Instruction trace shows the core execution history and places where execution might behave unexpectedly.
    - Data trace shows what memory addresses were accessed and if the accesses completed successfully.
    - Instrumentation trace can output printf-style debugging. On a Cortex-M processor, instrumentation trace can dump useful core information.
    - Program system trace to monitor signals within the system that might show abnormal behavior.
- Measure performance:
    - Instruction trace shows cycle count information and timestamps alongside the instruction execution history.
    - Instrumentation and system trace output profiling information like performance register values or timestamps.
- View operation on a system-level:
    - System trace monitors signals outside the processor or core to show a wider scope of target activity.

Note

Target designers try to keep trace functionality available throughout the development life cycle of their target. However, trace capability might be phased out or restricted during the development cycle of some targets. Refer to your target manufacturer, target designer, and target documentation for the trace capability of the target design.

---

4. Trace output

Most trace sources generate packetized data that might be further formatted by the trace infrastructure. This means that the captured trace data is not in a human-readable format. The trace data must go through extraction, decompression, decode, and processing stages to become human-readable. Native or self-hosted trace software or a trace-capable debugger implement these steps.

The following diagram shows the trace processing stages.

Figure 4-1: Trace processing stages

There are exceptions to the trace data not being in human-readable format. For example, the ITM can generate output in UART encoding over Serial Wire Output (SWO). This means that if you look at the trace data in an ASCII hexadecimal editor, you can read some of the trace data.

What happens to the trace data when it is generated depends on your debugging environment and the capabilities of your target.

Most external debug environments pull the trace data into the debugger for analysis. In this situation, the trace data is stored on the target, or exported from the target, using either on-chip capture or off-chip capture. Let's look at these methods in detail for an external debugger environment.

On-chip capture

The trace data is on chip and is exported to the external debugger:

- On-chip trace data is usually stored in a small hardware buffer, or is routed to a higher capacity area of the system memory by the Trace Memory Controller (TMC). The applicable TMC configurations are Embedded Trace Buffer (ETB), Embedded Trace FIFO (ETF), or Embedded Trace Router (ETR).

- At particular points during debugging, the external debugger performs operations to extract the on-chip trace data for analysis purposes.

The following diagram shows on-chip capture:

Figure 4-2: On-chip capture
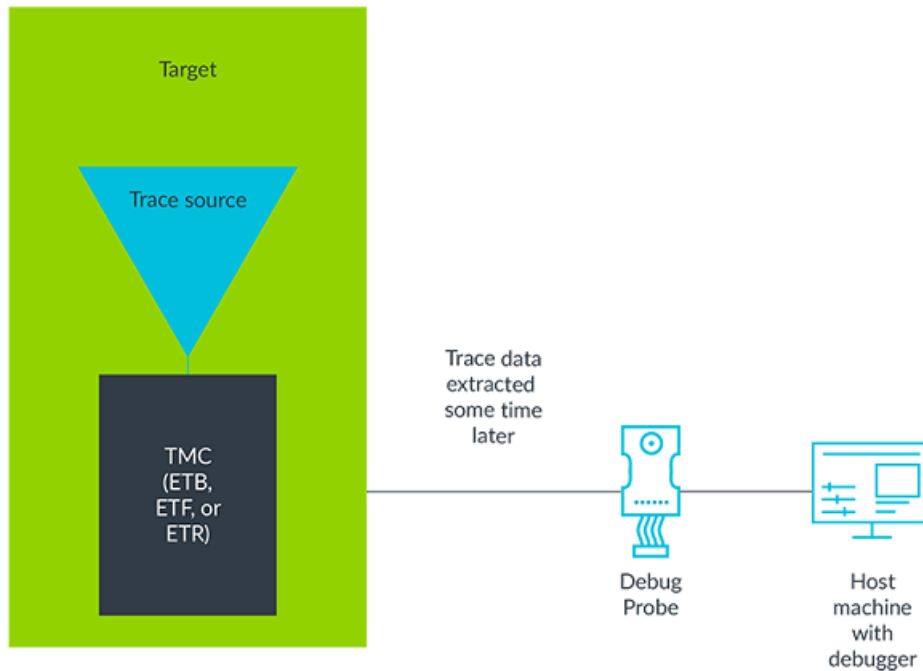
	When using on-chip capture, the trace data that is captured is heavily compressed. This heavy compression means that the trace decompression relies on the user having access to the exact image code that was executed by the processor. Any discrepancy between the image code that is used for decompression and the code that was run on the processor leads to decompression corruption.
Note	

Off-chip capture

The trace data is output from the target to a debug probe or directly to the external debugger:

- Trace data is output by the Trace Port Interface Unit (TPIU), Embedded Trace Streamer (ETS), or Serial Wire Output (SWO) that is on the target to an external debugger. The TPIU and ETS output trace data to a debug probe and the debug probe then passes the trace data onto the external debugger. The SWO passes the trace data directly to an external debugger.

- When an STM or an ITM generates the trace data, if the debugger and debug probe support real-time trace processing, the captured trace data is analyzed and displayed while it is captured. Real-time trace processing is often called streaming trace.

- Here is an alternative to streaming trace: At particular points during debugging, the external debugger can perform operations to dump trace data that is stored in the debug probe for later trace analysis.

The following diagram shows off-chip capture:

Figure 4-3: Off-chip capture
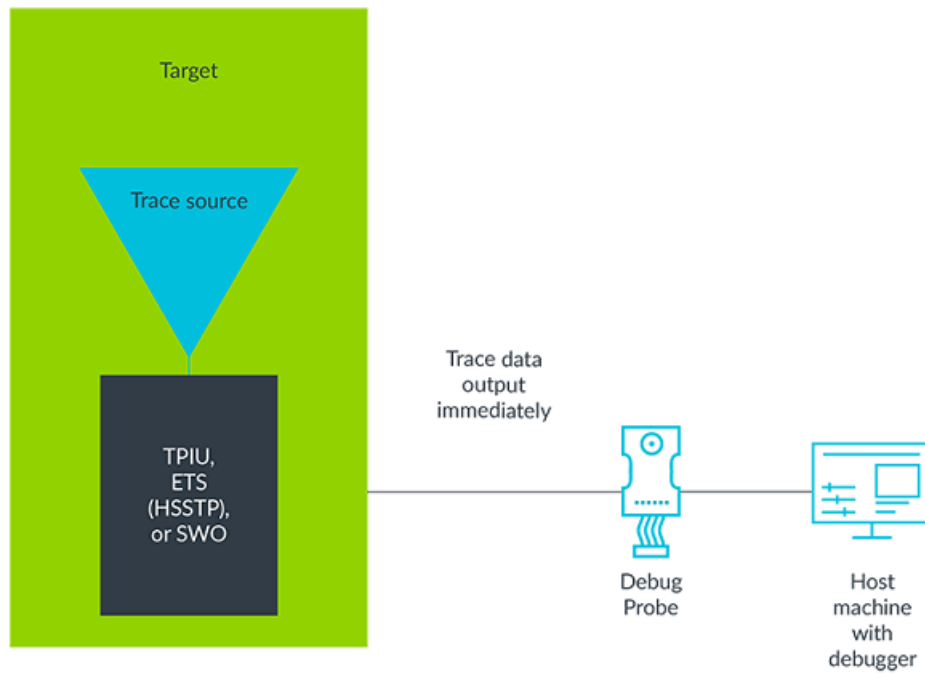
5. Trace components

There are many different types of trace components. Most trace components fall into three categories:

- Trace source - a trace component which generates trace data

- Trace sink - a trace component which stores or outputs trace data

- Trace link - a component which links trace or non-trace components together

In this section of the guide, we define the different types of trace components, describe their function, and provide the location of their implementation details. Specifications includes the trace component architecture specification documents.

> **Note**
>
> Debug subsystem design is highly configurable. It is the role of the target designer to create a debug subsystem design that is suitable for the target. This means that your target design might only implement a subset of the components that are described in this section.

5.1 Embedded Trace Macrocell Trace source

Depending on the processor, Armv8 systems have an ETMv4 or an ETMv3.x. For example, Armv8-A systems are ETMv4-only and Armv8-M might have ETMv4 or ETMv3.x.

The Embedded Trace Macrocell (ETM) architectures permit instruction and data trace. As we mentioned in What is trace?, a particular ETM implementation might not include data trace support. For example, the Armv8-A ETM is instruction trace only. Refer to your target documentation to learn whether the implemented ETMs support data tracing.

Because ETM trace data is packetized, data is decompressed and decoded before being analyzed.

Triggers and filters control ETM trace data generation. Filters allow you to control how much trace data is generated. This is useful if:

- The code or data being traced is large and can cause bandwidth issues on the target

- There are only certain points where having trace data is useful

Both triggers and filters are typically set using native or self-hosted trace software or an external debugger.

Triggers act like a start point to trace a region of interest. When a trigger is set, the ETM only generates trace data around the trigger point.

Filtering works for instruction and data trace by enabling trace generation between two filter points: a start point and end point. ETMv4 supports richer filters, for example filtering on context ID, Security state, and Exception level, than earlier ETM versions.

Multiple cores or processors can share a single ETM. If multiple cores share an ETM, trace data is generated for only one core at a time. This prevents the user from observing trace data from both cores concurrently. ETM sharing is not allowed for Armv8-A cores.

An ETM can generate cycle-accurate trace and can insert timestamps into the trace data. Cycle-accurate trace is useful for determining which code or functions are consuming the most execution time. Timestamps are useful when calculating how long a code or function takes to execute and for correlating trace data from different sources.

	Including cycle-accurate and timestamping information in the trace data increases the overall trace data size. This might be a problem for systems with limited trace data storage or small off-chip trace port sizes. Consider using triggers and filters to limit the amount of trace data that is generated when using cycle-accurate trace or timestamps.
Note	

ETM implementation details are either described in an ETM section of the Technical Reference Manual (TRM) for a processor, or in a separate TRM for that ETM. First, check whether the TRM for the processor includes an ETM section. If the TRM does not include this section, search for a separate TRM document. Separate TRM document names usually follow the format:

CoreSight ETM-<processor name> Technical Reference Manual

For example, the TRM for the Cortex-R7 ETM is CoreSight ETM-R7 Technical Reference Manual.

5.2 Program Trace Macrocell Trace source

Program Trace Macrocell (PTM) is only found in systems before Armv8. PTMs perform instruction trace only.

Because PTM trace data is packetized, data is decompressed and decoded before being analyzed.

PTM can provide triggering and filtering capabilities. The PTM implementation determines whether these capabilities are present and the number of capabilities that are available. The PTM trigger and filter capabilities work the same way that they work for an ETM.

A PTM can generate cycle-accurate trace and can insert timestamps into the trace data. These features work the same way that they work for an ETM.

PTM implementation details are either described in a PTM section of the TRM for a processor, or in a separate TRM for that PTM. Start by checking whether the TRM for the processor includes a PTM section. If the TRM does not include this section, search for a separate TRM document. Separate TRM document names usually follow the format:

CoreSight PTM-<processor name> Technical Reference Manual

For example, the TRM for the Cortex-A9 PTM is CoreSight PTM-A9 Technical Reference Manual.

5.3 Instrumentation Trace Macrocell Trace source

The Instrumentation Trace Macrocell (ITM) is a low-bandwidth, application-driven trace source. The ITM is mainly used to:

- Support `printf`-style debugging

- Trace OS and application events

- Output diagnostic system information

The ITM outputs trace data as packets. The four sources for the packets are:

- Software trace. Software can write directly to the ITM stimulus registers to generate packets.

- Hardware trace. The debug logic generates these packets, and the ITM outputs them. This is for Cortex-M processors only.

- Time stamping

- Global system timestamping. This is for Cortex-M processors only.

The ITM is programmed to control what information is traced.

Cortex-M ITM implementation details are found in the ITM section in the TRM for the Cortex-M processor. For example, the implementation details for the ITM for the Cortex-M4 are in the Instrumentation Trace Macrocell Unit section of the Arm Cortex-M4 Processor Technical Reference Manual.

General CoreSight ITM implementation details are found in the CoreSight Components Technical Reference Manual.

5.4 System Trace Macrocell Trace source

The System Trace Macrocell (STM) is a trace source that is designed to provide system trace and instrumentation information. This information includes:

- Memory-mapped writes to the STM Advanced eXtensible Interface (AXI) slave that carry information about the behavior of the software

- A hardware event interface to signify certain events that are happening in the system

The STM supports timestamps. These timestamps allow correlation with other timestamping trace sources in the CoreSight system, for example instruction trace.

For implementation details, the STM and STM-500 each have their own TRM:

Embedded Trace Buffer

The Embedded Trace Buffer (ETB) contains a dedicated SRAM that stores generated trace data on-chip for later retrieval. The SRAM acts like a circular buffer that wraps when the buffer size limit is reached. Buffer wrapping works by replacing the oldest trace data with the newest data.

A single ETB can store multiple ETM and PTM trace streams. Normally, the buffer is small, typically from 4KB to 64KB, so the amount of trace data that can be captured is limited. It is usually necessary to use trace source triggering and filtering capabilities to limit the amount of trace data that is captured to ensure that important trace data is not lost due to buffer wrapping.

Embedded Trace FIFO

The Embedded Trace FIFO (ETF) contains a dedicated SRAM that can be used as either a circular buffer, a hardware FIFO, or a software FIFO. In Circular Buffer mode, the ETF has the same functionality as the ETB. In Hardware FIFO mode, ETF is typically used to smooth out fluctuations in the trace data. In Software FIFO mode, on-chip software uses the ETF to read out data over the debug AMBA Peripheral Bus (APB) interface. Configure the ETF mode at runtime.

Embedded Trace Router

With the Embedded Trace Router (ETR), trace can be routed over an AXI interface to the system memory, or to any other AXI slave. An ETR allows larger amounts of trace data to be stored on-chip than an ETB or ETF allows. Like the ETF, the ETR has Circular Buffer and Software FIFO modes. The ETR programmer decides where to store the trace data in memory. Refer to your target documentation or target designer for the best place to store ETR trace data on your target, so that used memory is not overwritten.

Embedded Trace Streamer

The Embedded Trace Streamer (ETS) routes trace data over an AXI4-Stream interface to a streaming device, for example an HSSTP link layer, either directly or through an AXI4-Stream interconnect. The ETS behaves in a similar way to the ETR, by keeping the same baseline functionality. However, the ETS does not include features that are not applicable to trace data streaming, for example incrementing address support.

5.7 Trace Port Interface Unit Trace sink

The Trace Port Interface Unit (TPIU) drives trace data to external pins on a target, so that the Trace Port Analyzer (TPA), which is often part of a debug probe, can capture the trace data. The TPIU:

- Coordinates the stopping of trace capture when it receives a trigger

- Inserts source identification information into the trace stream so that trace data can be re-associated with its trace source

- Outputs the trace data over trace port pins

- Outputs patterns over the trace port. This pattern output is often referred to as TPIU pattern generation. This allows a TPA to tune its capture logic to the trace port, which maximizes the trace data output frequency on the trace port.

TPIU implementation details are found in either the Arm CoreSight System-on-Chip SoC-600 Technical Reference Manual or the Arm CoreSight SoC-400 Technical Reference Manual.

CoreSight SoC-600 implements the Arm Debug Interface Architecture Specification ADIv6. CoreSight SoC-400 implements the Arm Debug Interface Architecture Specification ADIv5. Consult your target designer if are unsure which of these TRMs apply to your target.

5.8 Funnel Trace link

The funnel, also called an ATB funnel, merges multiple ATBs into a single ATB. Typically, the single ATB is then routed to a trace component, for example another funnel, an ETB, an ETR, or a TPIU. The funnel comes in programmable or non-programmable configurations. With the programmable configuration, the funnel priority setting is configurable.

Funnel implementation details are found in either the Arm CoreSight System-on-Chip SoC-600 Technical Reference Manual or the Arm CoreSight SoC-400 Technical Reference Manual.

CoreSight SoC-600 implements the Arm Debug Interface Architecture Specification ADIv6. CoreSight SoC-400 implements the Arm Debug Interface Architecture Specification ADIv5. Consult your target designer if are unsure which of these TRMs apply to your target.

5.9 Cross trigger network Trace link

The cross-trigger network consists of Cross Trigger Interfaces (CTIs) and Cross Trigger Matrices (CTMs). CTIs enable the distribution of events to and from sources and destinations in the system. CTIs are connected to each other using one or more CTMs through channel interfaces.

CTIs are software-configurable, which allows the user to program the trigger to channel and channel to trigger mappings. When a trigger event occurs on a mapped channel, the event is broadcast on that channel to all other CTIs in the system.

In the context of trace, CTIs communicate events between the different trace components and other CoreSight components. For example, the cross-trigger network allows triggers to be routed from trace sources like cores to trace sinks like an ETR.

CTI and CTM implementation details are found in either the Arm CoreSight System-on-Chip SoC-600 Technical Reference Manual or the Arm CoreSight SoC-400 Technical Reference Manual.

CoreSight SoC-600 implements the Arm Debug Interface Architecture Specification ADIv6. CoreSight SoC-400 implements the Arm Debug Interface Architecture Specification ADIv5. Consult your target designer if are unsure which of these TRMs apply to your target.

5.10 Timestamp generator

The timestamp generator generates 64-bit rolling time for distribution to other CoreSight components, which allows later alignment of trace information. In the Arm implementation, the timestamp generator runs at a constant clock frequency, regardless of the power and clocking state of the processor that uses it. The timestamp generator has two APB interfaces: a read-only interface to read the counter value and management registers, and a programming interface. In many Cortex-A processor systems, the processors and the CoreSight infrastructure use the same source of time.

Timestamp generator implementation details are found in either the Arm CoreSight System-on-Chip SoC-600 Technical Reference Manual or the Arm CoreSight SoC-400 Technical Reference Manual.

CoreSight SoC-600 implements the Arm Debug Interface Architecture Specification ADIv6. CoreSight SoC-400 implements the Arm Debug Interface Architecture Specification ADIv5. Consult your target designer if are unsure which of these TRMs apply to your target.

6. Trace infrastructure examples

In this section of the guide, we provide some trace infrastructure examples. These examples illustrate:

- The general flow of trace data from trace source to trace sink

- The interaction of non-trace components with the trace infrastructure

- That trace infrastructures can have varying levels of complexity

Refer to your target designer and target documentation to learn how your target implements its trace infrastructure.

Note The trace infrastructure examples use ETMs as the processor instruction and data trace source. This is because most Arm-based systems use ETMs. Other core trace sources, for example PTMs, could be used in the same way.

ETM trace infrastructure examples

The following diagrams show two similar trace infrastructure examples that are centered around ETM trace generation:

Figure 6-1: ETM trace infrastructure example 1
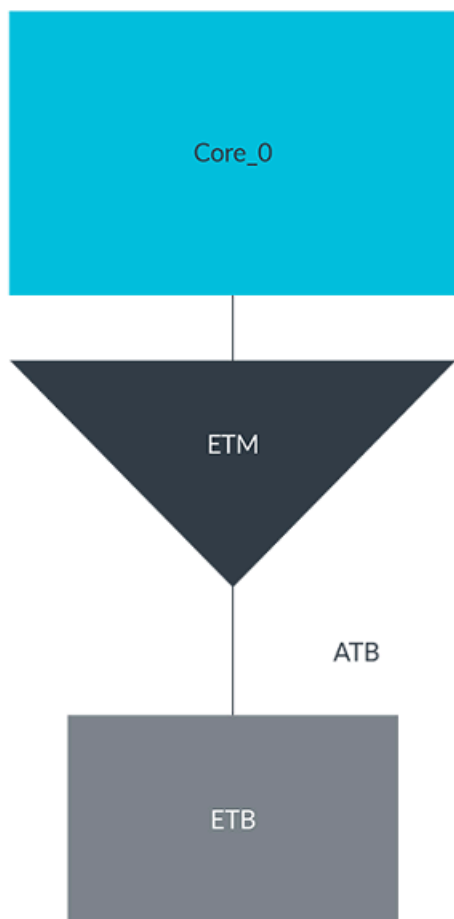
Figure 6-2: ETM trace infrastructure example 2
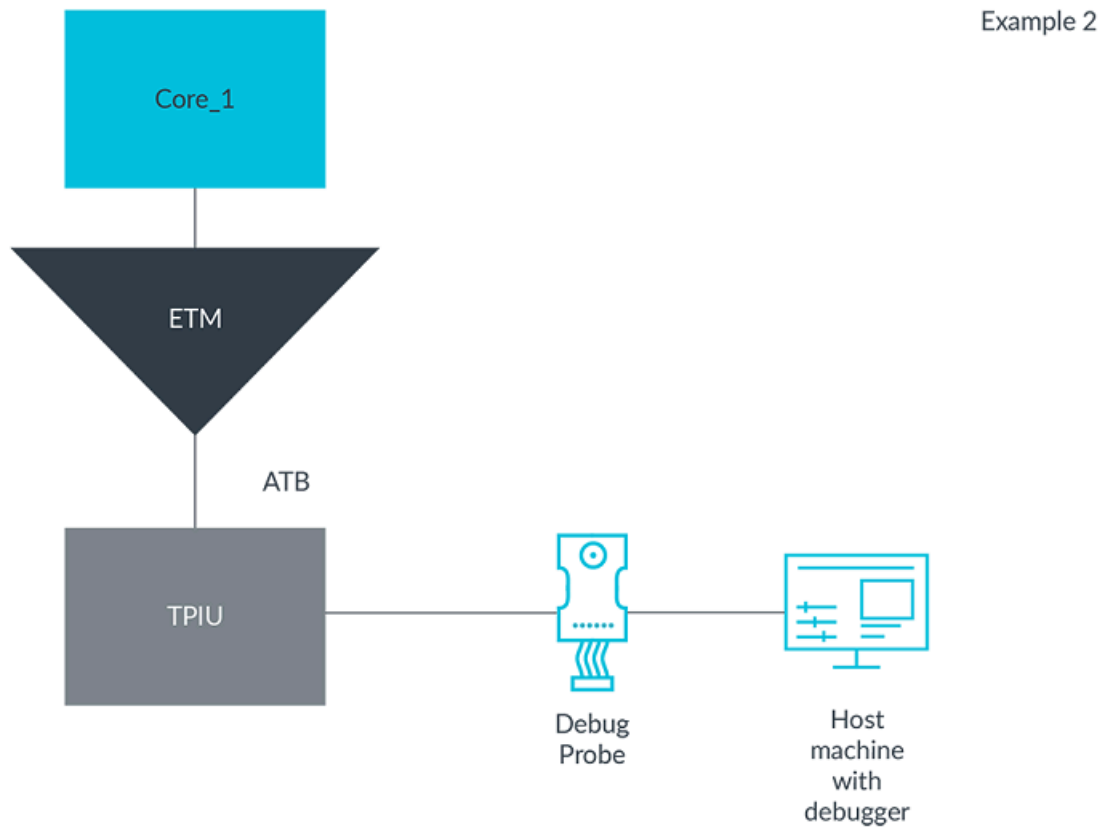
In both examples, an ETM is used like a trace source to generate trace data for an Arm core. The ETMs generate instruction trace, and potentially data trace, for the cores that they are connected to. The ETM-generated trace data is passed to a trace sink over the ATB. Example 1 uses an ETB for a trace sink, and Example 2 uses a TPIU.

In systems where the trace sink is an ETB or an ETR, the trace data is stored on-chip for later retrieval by a debugger. If the trace sink is a TPIU, a debug probe is required to capture the trace data that is output by the TPIU.

Multiple core trace infrastructure example

The following diagram shows a trace infrastructure example in which more than one core is traced:

Figure 6-3: Multiple core trace infrastructure example
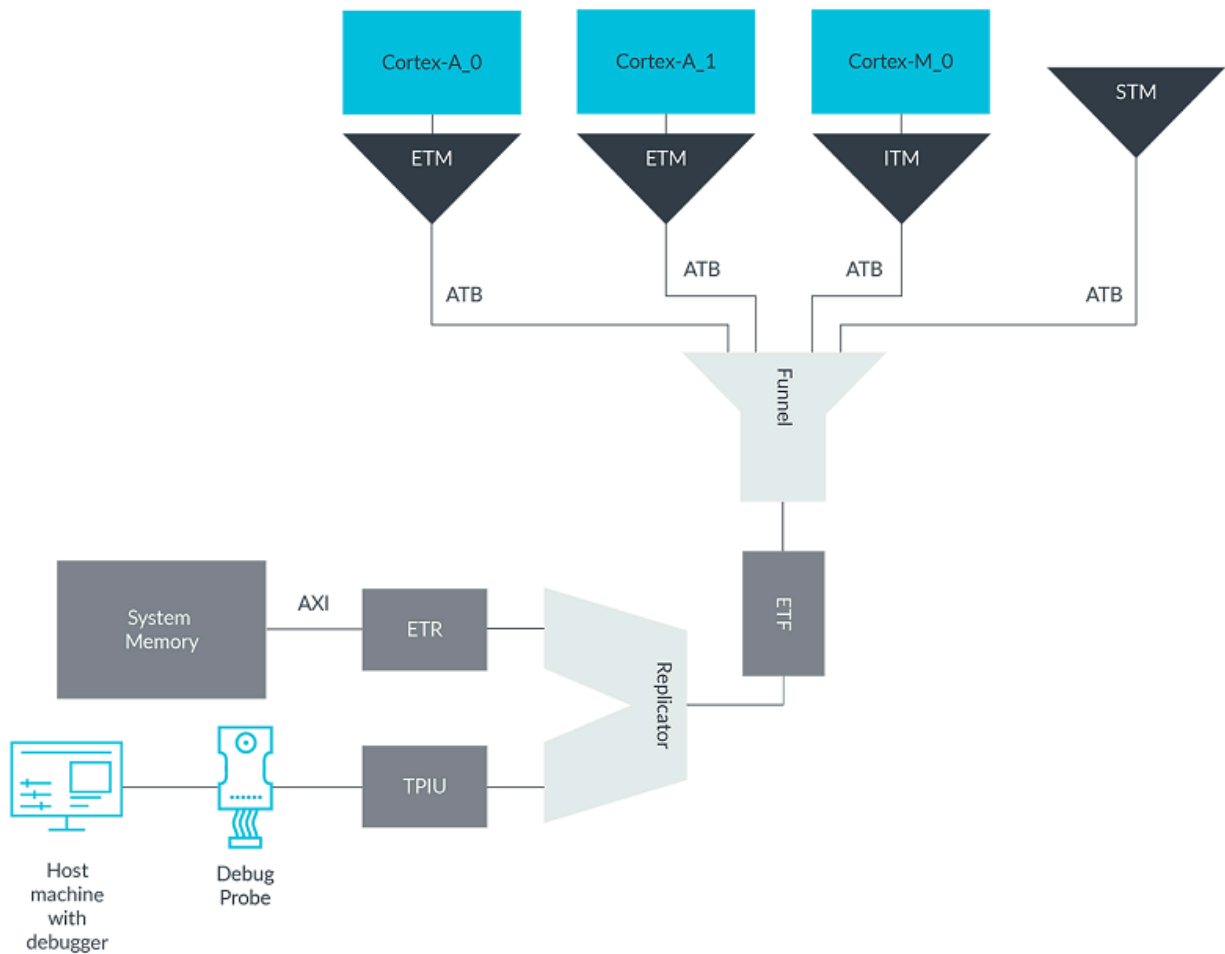
This example shows a mix of different types of cores: two Cortex-A class cores and one Cortex-M class core. This type of configuration is referred to as a heterogeneous system. The Cortex-A class cores each have an ETM as their trace source to generate core trace data. The ETMs generate instruction trace, and potentially data trace, for the cores that they are connected to. The Cortex-M class core is connected to an ITM to collect instrumentation trace from that core. There is also an STM to collect system trace for the target.

All the trace sources pass trace data over the ATB to a trace funnel. This means that all the ATBs can be combined before they are passed to the trace sinks. There are three trace sinks in the system: ETF, ETR, and TPIU.

The ETF can be configured either as a circular buffer, to capture trace data, or as a FIFO, to smooth out trace data fluctuations.

From the ETF, the replicator duplicates the ATB before it reaches the ETR and the TPIU. The ETR routes the combined ATB to system memory through AXI, the system interconnect. The trace data that is stored in the system memory can later be retrieved by a debugger for analysis. In the case

of self-hosted debug, the trace data in the system memory could be consumed by another core on the target. The TPIU operates in the same way as in the ETM trace infrastructure example.

CTI trace infrastructure example

The following diagram shows a trace infrastructure example with CTI connections:

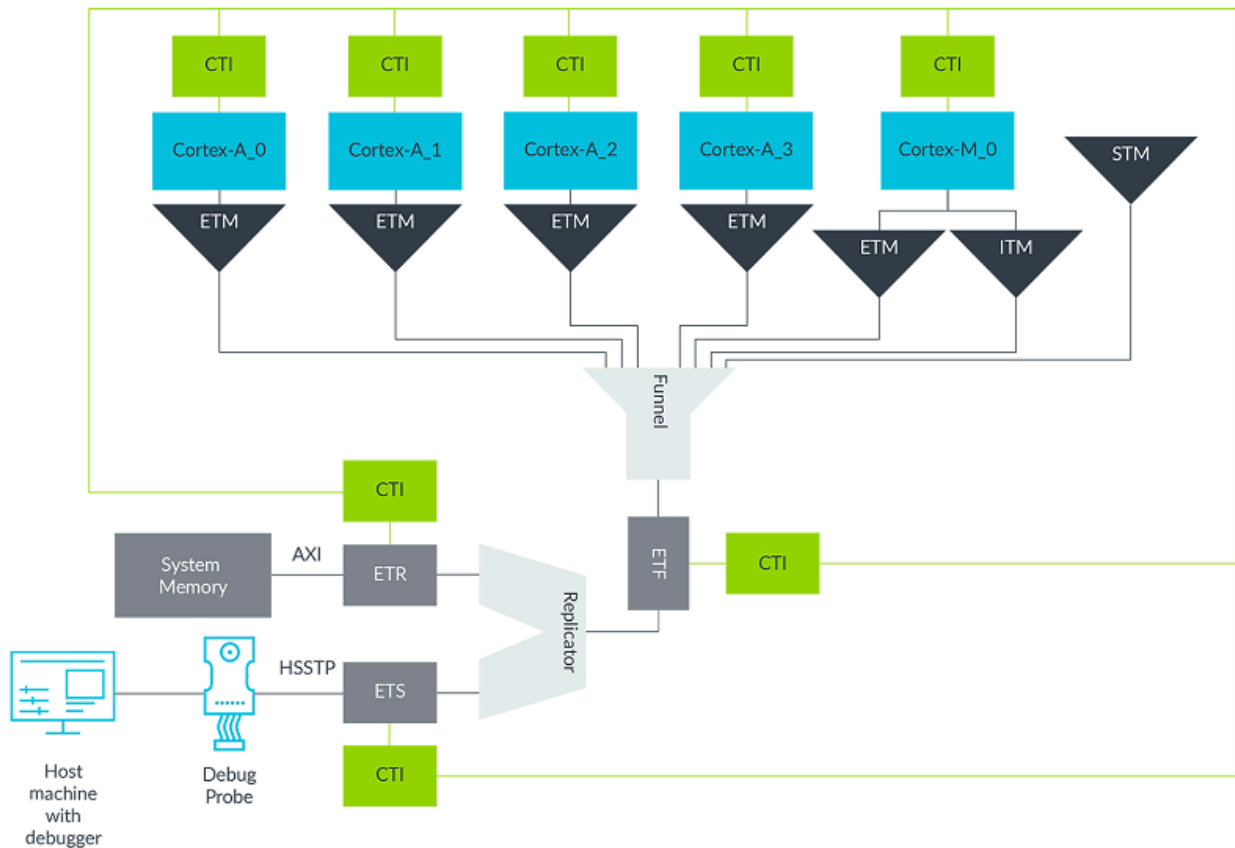**Figure 6-4: CTI trace infrastructure example**

This example shows four Cortex-A class cores and one Cortex-M class core.

All the Cortex-A cores have their own ETM to generate core trace data. Unlike the multiple core trace infrastructure example, the Cortex-M core has both an ETM and an ITM. This means that core and instrumentation trace data are generated.

The configuration also includes an STM to collect system trace data for the target.

The output from the ETMs, ITM, and STM are routed to a trace funnel to combine into one ATB. The single ATB from the funnel is routed to an ETF. The ETF in this example operates like the ETF in the multiple core trace infrastructure example.

From the ETF, the ATB is routed to a replicator. This example includes an ETS that allows the trace data to be passed off-chip using an HSSTP. An HSSTP is useful if the target is running at high frequencies, or if the target has many trace sources that must be captured concurrently.

CTIs are connected to the ETF, ETR, and ETS. If the CTIs are programmed appropriately, they allow the ETF, ETR, and ETS to pass halt requests to the core CTIs. For example, the ETR can send a halt request through the associated CTIs to the cores when the circular memory buffer is full. The halt request halts the cores, which allows the debugger to collect the trace data before the ETR wraps. This means that no core trace data is lost.

Learn the architecture - Understanding trace

Document ID: 102119_0200_02_en
Version 2.0
Can trace capture affect a system?

7. Can trace capture affect a system?

Except for the power that is consumed by the system trace components, trace is almost entirely non-invasive. This means that performing trace generation and collection does not influence the wider system. There are exceptions to this.

For example, a target might be able to halt the cores or processors when the trace sink capturing the trace source data is full. Halting the cores or processors is an invasive action and does affect the system.

Also, executing instrumentation code might be required to generate output for any ITMs or STMs in the system. This is code that a core would not usually execute and therefore affects the operation of the system.

Using an ETR consumes some bandwidth on the system interconnect. This can affect system performance. Configuring the appropriate Quality-of-Service (QoS), or setting priorities on the system interconnect, can minimize this effect.

If the system that you are working with is sensitive to disturbances, consult your target designer or target documentation to learn how trace might affect your system.

8. Specifications

Here are the trace component architecture specifications that are referred to in this guide:

- Embedded Trace Macrocell (ETMv4) Architecture Specification: The architecture specification that is used by CoreSight-compatible Embedded Trace Macrocells (ETMs)

- Embedded Trace Macrocell Architecture Specification ETMv1.0 to ETMv3.5: The architecture specification that is used by CoreSight and non-CoreSight-compatible Embedded Trace Macrocells (ETMs)

- CoreSight Program Flow Trace Architecture Specification: The architecture specification that is used by CoreSight Program Trace Macrocells (PTMs)

- Armv8-M Architecture Reference Manual: The architecture specification that is used by the Armv8-M Instrumentation Trace Macrocells (ITMs)

- Armv7-M Architecture Reference Manual: The architecture specification that is used by the Armv7-M Instrumentation Trace Macrocells (ITMs)

- Arm System Trace Macrocell Programmers' Model Architecture Specification: The programmers' model architecture specification that is used by System Trace Macrocells (STMs)

- Arm Embedded Trace Router Architecture Specification: The architecture specification that is used by CoreSight-enabled Embedded Trace Routers (ETRs)

The STM and STM-500 conform to the MIPI Alliance System Trace Protocol (STP) specification. Request a copy of the specification from the MIPI System Trace Protocol (STP) download page.

9. Check your knowledge

The following questions will help you test you knowledge:

Enabling cycle-accurate trace increases the amount of ETM trace data that is captured. True or False?

True. Enabling any type of cycle-accurate trace increases the amount of ETM trace data collected. This increase in trace data might not be supportable if the trace bandwidth of the target is limited.

All target designs contain trace components. True or False?

False. Not all targets include trace components. The target designer decides whether trace components are included in the target or not.

Which TMC configurations can buffer trace data?

- ETB
- ETF
- ETR
- ETS
- All of the above

10. Related information

Here are some resources that are related to material in this guide:

- Arm Community - ask development questions, and find articles and blogs on specific topics from Arm experts
- Arm CoreSight Base System Architecture - Arm Platform Design Document - learn more about designing Armv8-A CoreSight systems
- Trace component architecture specifications referred to in this guide.

11. Next steps

This guide focused on providing a high-level view of trace in Armv8 and Armv7 systems. We have looked at what trace is, how it is useful, how it is output, and what components can make up the trace infrastructure.

This knowledge is useful for first-time trace users, or for users who want to expand their existing trace knowledge.

If you want to learn more about debug and trace, read these guides in our series:

- Before debugging on Armv8-A
- Debugger usage on Armv8-A