



Learn the architecture - Understanding Armv9-A trace

Version 1.0

guide

Non-Confidential

Copyright © 2022 Arm Limited (or its affiliates).
All rights reserved.

Issue 01

102856_0100_01_en



Learn the architecture - Understanding Armv9-A trace guide

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	4 March 2022	Non-Confidential	Initial release

Proprietary Notice

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

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

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

This document may include technical inaccuracies or typographical errors.

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

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws

and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

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

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

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.....	6
1.1 Before you begin.....	6
2. Embedded Trace Extension.....	8
2.1 Reset.....	9
2.2 Prohibited regions.....	9
2.3 Power states and domains.....	10
2.4 PE in Debug state.....	11
2.5 Filtering trace generation.....	11
3. Trace Buffer Extension.....	13
3.1 Trace buffer writes.....	13
3.2 Owning translation regime.....	14
3.3 Prohibited regions.....	14
3.4 Operation.....	15
3.5 Triggers.....	16
4. Example sytem.....	17
5. Trace capture examples.....	20
5.1 el1_self_hosted_trace example.....	21
5.2 el2_self_hosted_trace example.....	25
5.3 el2_el1_self_hosted_trace example.....	26
5.4 el2_controlled_self_hosted_trace example.....	27
6. Related information.....	31
7. Next steps.....	32

1. Overview

This guide focuses on the Embedded Trace Extension (ETE) and the Trace Buffer Extension (TRBE). ETE and TRBE are trace components introduced in Armv9-A. The ETE and TRBE are documented in the [Arm Architecture Reference Manual Supplement Armv9, for Armv9-A architecture profile \(DDI0608\)](#).

The ETE provides details about the software control flow running on a Processing Element (PE). These details help you to debug or optimize the software on your target. Capturing the software control flow is referred to as trace.

Self-hosted trace means software running on the target sets up and runs the trace capture of a PE or PEs. The ETE is oriented towards self-hosted trace and self-hosted analysis environments. Using self-hosted trace enables debugging of software executing on PEs that are deeply embedded within a System on Chip (SoC). Self-hosted analysis means that analysis of the captured trace data can happen on the target.

Generally, in documentation, ETE refers to the trace architecture used, and trace unit refers to the physical implementation of the ETE trace architecture.

The TRBE takes program trace generated by a trace unit, like an ETE, and writes it directly to memory. Like the ETE, the TRBE is oriented towards self-hosted trace environments.

On an Armv9-A target, ETE and TRBE together enable capture and storage of trace data on an SoC for on-target or off-target analysis.

The Embedded Trace Extension feature is identified as FEAT_ETE. The Trace Buffer Extension feature is identified as FEAT_TRBE. Targets that implement FEAT_TRBE must also implement the Self-hosted Trace Extension, FEAT_TRF. FEAT_TRF is defined in the [Arm Architecture Reference Manual for Armv8-A architecture profile](#).

This guide describes ETE and TRBE and provide steps to capture and store trace data with ETE and TRBE. This guide also explains what an Armv9-A trace system looks like and explains the ETE and TRBE example projects included with the guide.

This guide focuses on topics that are relevant when interacting with the ETE and the TRBE with software or an external debugger. It does not focus on hardware implementation or micro-architectural details.

1.1 Before you begin

This guide assumes that you are familiar with Arm debug and trace, Exception levels, memory management, and Security states. If you are unfamiliar with any of the previous topics, read the following guides before continuing with this guide:

- [AArch64 self-hosted debug](#): Introduces the Armv8-A AArch64 Debug architecture and describes self-hosted debug features.

- [Introducing CoreSight: Debug and trace infrastructure](#): Introduces Arm CoreSight Architecture for debug and trace on Arm A-profile processors.
- [Understanding trace](#): Gives a high-level view of trace on Armv7 and Armv8 systems.
- [AArch64 exception model](#): Introduces the exception and privilege model in AArch64.
- [AArch64 memory management](#): Introduces the MMU, which is used to control virtual to physical address translation.
- [Introduction to security](#): Introduces some generic concepts about security on Arm platforms.

2. Embedded Trace Extension

ETE is the trace architecture for Armv9-A PEs. ETE has many similarities with the Arm ETMv4 architecture. These similarities create a trace programming and decode environment that is compatible between ETE and ETMv4 implementations.

Like ETMv4, the ETE traces the following software control flow elements:

- Direct and indirect branches
- ISB
- WFE and WFI
- TSTART
- Exceptions

Some elements are optionally traced like **IMPLEMENTATION DEFINED** exceptions. Examples of **IMPLEMENTATION DEFINED** exceptions are Error Correcting Code (ECC) error correction and generic replay of program execution.

ETE does not support all the features of ETMv4. For example, you cannot use one ETE trace unit to trace multiple PEs, but you can with ETMv4. Also, ETE does not support data trace.

To help with debugging, the architecture has many features to modify the trace stream to provide additional information. This table describes features that modify the trace stream:

Feature	Description
Branch broadcasting	Causes the trace unit to explicitly output the target address of any taken, direct branch instructions executed.
Context identifier tracing	Includes the Context identifier, CONTEXTIDR_EL1, that contains the current PE execution context.
Virtual context identifier tracing	If EL2 is implemented, includes CONTEXTIDR_EL2.PROCID as the Virtual context identifier.
Event trace (ETEEEvents)	Adds selected resource information. The ETE supports up to 4 ETEEEvents. An ETEEEvent might have an associated external output that can trigger other debug events in the system.
No overflow	A feature that prevents trace unit overflow. Might have a significant impact on the performance of the PE.
PE stalling and overflow avoidance	Reduces the chance of trace unit buffer overflow by slowing the execution of the PE. Trace unit buffer overflows cause loss of trace data. Stalling the PE impacts the performance of the PE.
Timestamping	<p>Inserts timestamps containing the common global time value. Used to correlate between multiple trace streams, determine the passage of time, or for analyzing performance.</p> <p>Time value options are:</p> <ul style="list-style-type: none"> • Physical time, as seen by the generic timers in the PE • Virtual time, as seen by the generic timers in the PE • An IMPLEMENTATION DEFINED time value, often supplied by a CoreSight system

Registers TRCIDR0 to TRCIDR13 indicate which features are implemented. The following sections cover ETE topics that are relevant from the perspective of software or an external debugger.

2.1 Reset

A trace unit reset sets all the trace and management registers to the individual register description values. When the trace unit core power domain is powered up, a reset is applied. It is **IMPLEMENTATION DEFINED** whether the system has a mechanism to initiate a trace unit reset on demand. Unlike previous trace architectures, ETE does not support multiple types of reset.

A PE Warm reset does not cause a trace unit reset. If the target has an Advanced Trace Bus (ATB) or other interface for trace output, we recommend implementing the output path so that PE Warm reset does not affect it. This lack of effect ensures that tracing is possible through a PE Warm reset to help with low level debugging scenarios.

In a PE with FEAT_TRF, when a PE Cold reset occurs, EDSCR.TFO is reset to 0b0. This EDSCR.TFO reset prohibits trace until the software explicitly permits it. If tracing from the start of PE execution is required, before execution begins, the debugger must ensure any relevant controls are programmed to permit tracing. The debugger might put the PE into Debug state to ensure the registers are programmed before the PE starts executing instructions.

2.2 Prohibited regions

An executable program might contain regions of code that are prohibited to trace. These prohibited regions might be associated with a different Security state, or with the PE entering a privileged mode to execute certain code. If the trace unit is enabled when in a prohibited region, the trace unit generates no trace until the PE leaves the prohibited region.

Prohibited regions are set using the FEAT_TRF system registers or authentication interface or interfaces. The FEAT_TRF defines system registers accessible with MRS and MSR instructions to program the trace setup. The authentication interface uses hardware signals to define what can be debugged and traced on the target. The trace unit might implement optional authentication interfaces like the CoreSight Authentication interface defined in the [Arm CoreSight Architecture Specification](#).

The prohibited regions might change during execution. For example, an authentication interface might change, prohibiting trace in regions where it was possible before. For example, if the authentication interface changes so that Secure state is prohibited, Secure state can no longer be traced.

When the PE is executing from a prohibited region, the trace unit does not do the following:

- Trace instructions
- Trace exceptional occurrences like PE reset and System Error
- Match the instruction Address Comparators with any instructions in the prohibited region
- Output any trace that provides information about execution in the prohibited region. Examples of information include:
 - Context of execution

- Instruction addresses
- Address of the first instruction in the prohibited region

If cycle counting is enabled when the PE executes in a prohibited region, the cycle counter continues to count. When the PE leaves a prohibited region, the cycles counted are included in the general cycle count value.

A PE leaves a prohibited region using one of the following methods:

- Context synchronization event, such as the PE taking or returning from an exception
- Authentication interface changes

2.3 Power states and domains

The trace unit might implement different power states to improve the energy efficiency of the target. Four possible power states are defined in the [Arm Architecture Reference Manual for ARMv8-A architecture profile](#): Normal, Standby, Retention, and Powerdown.

The trace unit might implement a low-power state that is equivalent to Standby state. In low-power state, the trace unit core power domain is on, but energy consumption is reduced. When in low-power state, the trace unit is accessible from software and an external debugger.

While the trace unit is in the low-power state, the trace unit performs the following actions:

- No trace is generated. This lack of trace generation includes ETEEvent trace.
- Trace resources, like counters and single-shot comparators controls, remain in the same state as before low-power state was entered.
- All external outputs are driven low.
- Recognition of external events, like external inputs, might not occur.
- Timestamp requests might be ignored.
- Cycle counter might continue to count depending on the **IMPLEMENTATION DEFINED** setting.

Lower-power state is entered for the following reasons:

- The PE is in a low-power state because of the Wait for Event (WFE) mechanism.
- The PE is in a low-power state because of the Wait for Interrupt (WFI) mechanism.
- The PE is in Debug state.
- The trace unit is disabled.

While the PE is in low-power state, the trace unit might intermittently leave and reenter the low-power state. If this reentry happens, resources and trace generation might intermittently become active. When low-power state is intermittent, the behavior of the trace unit is **IMPLEMENTATION DEFINED**.

You can prevent the trace unit from entering low-power state by enabling `TRCEVENTCTL1R.LPOVERRIDE`. ETE Low-power Override Mode does not affect the PE. Because the mode does not affect the PE, the PE can enter low-power mode. If the PE enters low-power mode, ETE trace is affected.

Other possible power states implemented by the trace unit are Retention and Powerdown state. In Retention and Powerdown power state, the trace unit power domain is powered down. In both power states, no trace is generated, and an external debugger cannot access the trace unit.

To determine the power state of the trace unit, read `TRCPDSR`.

Previous Arm trace architectures supported multiple power domains in the ETE. ETE only supports a single power domain and therefore `TRCPDSR.POWER` is always `0b1`.

The trace unit and PE might reside in separate power domains. If the trace unit is in a separate power domain, powering down the trace unit might not power down the PE. If the PE is powered down, so is the trace unit.

If a trace unit is associated with a disabled PE, the core power domain of the trace unit is powered down or held in reset.

2.4 PE in Debug state

An external debugger might put the PE into Debug state to check the state of the PE and perform analysis of program execution.

When the PE enters Debug state, the trace unit has the following operations:

- Instructions are not traced.
- Trace filtering is inactive.
- Trace filtering state is maintained.

If the PE resets while in Debug state, the PE leaves Debug state. Based on the trace unit setup, the trace unit might resume tracing after the PE exits Debug state.

2.5 Filtering trace generation

A significant amount of trace can be generated when capturing the execution of a PE. ETE provides filtering functionality that allows you to capture only the code or execution periods of interest. The filtering reduces the amount of trace data generated. This reduction decreases the amount of trace bandwidth and storage necessary to trace the PE. When trace filtering is not used, the trace unit traces all instructions.

This table describes the types of trace filtering that ETE supports:

Trace filter	Decription
ViewInst start/stop function	<p>Used to trace a particular piece of code with all the functions that the piece of code calls. Uses the Single Address Comparators and the PE Comparator Inputs to define trace start points and stop points.</p> <p>Set start points using TRCVISSCTLR.START and TRCVIPCSSCTLR.START.</p> <p>Set stop points using TRCVISSCTLR.STOP and TRCVIPCSSCTLR.STOP.</p> <p>Other relevant register values are TRCVICTLR.SSSTATUS and TRCIDR4.NUMPC.</p>
ViewInst include/exclude function	<p>Used to include or exclude instruction ranges in the trace data. Uses the Address Range Comparators, TRCVIIECTLR.INCLUDE, and TRCVIIECTLR.EXCLUDE.</p> <p>Differs from the ViewInst start/stop function in the following ways:</p> <ul style="list-style-type: none"> Traces any instruction that branches or jumps from within the specified address range. Does not trace any functions called by the instructions in the address range.
Exception level filtering	<p>Filter based on Exception level. To select which Exception levels are traced or not, configure:</p> <ul style="list-style-type: none"> TRCVICTLR.EXLEVEL_S_ELO for Secure ELO TRCVICTLR.EXLEVEL_S_EL1 for Secure EL1 TRCVICTLR.EXLEVEL_S_EL2 for Secure EL2 TRCVICTLR.EXLEVEL_S_EL3 for Secure EL3 TRCVICTLR.EXLEVEL_NS_ELO for Non-secure ELO TRCVICTLR.EXLEVEL_NS_EL1 for Non-secure EL1 TRCVICTLR.EXLEVEL_NS_EL2 for Non-secure EL2 <p>Note: If FEAT_TRF is implemented, the TRFCR settings can override this filtering.</p>
Resource event based filtering	<p>Filter on system conditions or on trace unit resources. The following are examples of resources to filter on:</p> <ul style="list-style-type: none"> Sampling based on cycle counts. Activating tracing on the nth function call. Performance Monitoring Unit (PMU) events. <p>To select the combination of resource selectors to filter with, set TRCVICTLR.EVENT.SEL and TRCVICTLR.EVENT.TYPE.</p>

3. Trace Buffer Extension

The TRBE is a trace buffer unit that stores trace generated by a trace unit into system memory. For example, a TRBE could capture trace generated by a trace unit for a particular PE.

The trace buffer has the following characteristics:

- Can be physically or virtually addressed (normally virtually addressed).
- Has an owning translation regime.
- Is defined by pointer registers.
- Its operation is affected by external events like triggers.

The following sections further describe the characteristics of the TRBE.

3.1 Trace buffer writes

The trace buffer can be either virtually or physically addressed. The trace buffer is usually virtually addressed. The buffer is specified as a single contiguous region in the specified address space. When virtual addressing is used, the buffer must be contiguous in the virtual address space but might not be contiguous in physical memory. Physically addressing the trace buffer can aid with debugging software that changes the virtual address mappings.

Three buffer pointer registers define the trace buffer:

- The Base pointer: the start address of the trace buffer.
- The Limit pointer: the end address of the trace buffer.
- The current write pointer: the address where the next trace data values are written to memory.

You must align the Base pointer and Limit pointer to at least 4KBs. This alignment requirement means the buffer is at least one full virtual page in size. The required alignment of the current write pointer is **IMPLEMENTATION DEFINED**.

TRBLIMITR_EL1.nVM determines whether the trace buffer pointers are virtually or physically addressed. When the pointers are virtual addresses, all the virtual addresses conform to the owning translation regime. The owning translation regime is described in the following section. The Translation Lookaside Buffer (TLB) might cache translations used by the TRBE. Any TLB maintenance operations executed by a PE affect the TLB translations cached for the TRBE. Also, any cache maintenance operations executed by the PE affect the data caching of the TRBE. If the Memory Tagging Extension, FEAT_MTE, is implemented, the trace buffer unit generates an Unchecked access for each access to the trace buffer. This access type is true even when a Tagged Normal memory type is accessed.

3.2 Owning translation regime

The owning Security state and the owning Exception level define the owning translation regime. The owning translation regime uses its address translation table data to determine the properties of the trace data transactions written to system memory.

In practice, the owning translation regime maps onto the Security state and Exception level where the software managing the TRBE is executing. For example, if a rich OS configures the TRBE, the owning Security state and Exception level might be Non-secure EL1.

One of the following defines the owning Security state:

- Whether EL3 is implemented and the executing Security state of the PE, or
- MDCR_EL3.NSTB

Possible owning Security states are Secure or Non-secure. If the Realm Management Extension (RME) is implemented, another possible owning Security state is Realm.

One of the following defines the owning Exception level:

- Whether EL2 is implemented and enabled for the owning Security state Or
- MDCR_EL2.E2TB

Possible owning Exception levels are EL2 or EL1.

When the trace buffer is virtually addressed, writes to the trace buffer use the translation table entries of the owning Exception level. For example, if the owning Exception level is EL2, when tracing from EL1, trace buffer writes use the EL2 translation tables.

If the Memory System Resource Partitioning and Monitoring (MPAM) Extension, FEAT_MPAM, is implemented, trace buffer accesses use the MPAM values of the owning Exception level and owning Security state.

3.3 Prohibited regions

Trace is prohibited if the owning translation regime is not in context. This prohibition occurs when:

- The PE is executing at a higher Exception level than the owning Exception level.
- The PE is not executing in the owning Security state.

For example, if the PE is executing in EL2 and the owning Exception level is EL1, EL2 trace is prohibited. If a region is prohibited, no trace is captured.

The following table summarizes the relationship between the owning Exception level, the owning Security state, the owning translation regime, and prohibited trace regions:

Owning Exception level	Owning Security state	Owning translation regime	Prohibited trace regions
EL1		EL1 and EL0	<ul style="list-style-type: none"> EL3 EL2 EL0, if EL2 is implemented and enabled and HCR_EL2.TGE is 0b1
EL2		<ul style="list-style-type: none"> EL2, if HCR_EL2.E2H is 0b0 EL2 and EL0, if HCR_EL2.E2H is 0b1 	EL3
	Non-secure		Secure state
	Secure state		Non-secure state

The Self-hosted Trace Extension, FEAT_TRF, also defines when trace is prohibited. The following register bits define the FEAT_TRF prohibited regions:

- SCR_EL3.NS
- MDCR_EL3.STE
- MDCR_EL3.NSTB
- MDCR_EL2.E2TB
- SCR_EL3.EEL2
- HCR_EL2.TGE
- TRFCR_EL2.E2TRE
- TRFCR_EL1.E1TRE
- TRFCR_EL2.E0HTRE
- TRFCR_EL1.E0TRE

Prohibitions created by the owning Exception level, owning Security state, and FEAT_TRF add to the prohibited regions defined by the trace unit.

3.4 Operation

The TRBE is enabled when self-hosted trace is enabled and TRBLIMITR_EL1.E is 0b1. If the TRBE is disabled, it does not collect trace data from the trace unit.

The TRBE is running when it is enabled and TRBSR_EL1.S is 0b0. TRBE trace collection stops when the TRBE is enabled and TRBSR_EL1.S is 0b1.

When the TRBE is running, trace data is collected from the trace unit. The TRBE either accepts the data and writes the data to system memory or does not accept the data. If the data is not accepted, the trace unit holds onto the data. If the trace unit buffer space runs out before the TRBE can accept the data, the buffer overflows and data is lost. One possible reason for the TRBE not to accept trace data is its internal buffers are full.

The TRBE supports the following trace buffer modes:

- Circular Buffer mode: When the current write pointer reaches the Limit pointer, the buffer is wrapped by setting the current write pointer to the Base pointer. This wrapping means the latest trace data overwrites the oldest trace data.
- Wrap mode: Same as Circular Buffer mode, except that an interrupt request is generated when the current write pointer is wrapped.
- Fill mode: Same as Wrap mode, except that trace collection stops when the current write pointer is wrapped.

TRBLIMITR_EL1.FM controls the trace buffer mode.

In software, you can configure the Performance Monitoring Unit (PMU) to count the number of trace buffer wraps.

3.5 Triggers

TRBE supports detecting triggers generated by the trace unit. Triggers are used to capture trace around points of interest.

A trigger counter is used to collect trace before, around, or after a detected trigger. After a detected trigger, when the programmed trigger counter value is hit, a trigger event occurs. The trigger counter value is programmed in software. If the TRBE is disabled, trigger events are not generated.

The TRBE supports the following trigger modes:

- Stop on trigger: Trace collection is stopped and an interrupt request is generated after a trigger event.
- IRQ on trigger: An interrupt request is generated after a trigger event.
- Ignore trigger: The TBRE ignores the trigger.

4. Example sytem

This section provides information about what an Armv9-A trace system looks like. To learn how the trace system of your target is implemented, read your target implementation documentation.

An Armv9-A trace system must adhere to the following rules:

- If an ETE trace unit is implemented, then TRBE is implemented.
- If TRBE is implemented, then a trace unit that implements ETE is used.
- If TRBE is implemented, then FEAT_TRF is implemented.
- If the TRBE is enabled, the trace unit only sends trace data to the TRBE.
- When TRBE is disabled, the trace unit might send the trace data to one or more **IMPLEMENTATION DEFINED** trace output interfaces. A common trace output interface is an AMBA ATB interface, for exporting the trace to a CoreSight trace subsystem.
- There is one ETE trace unit for each PE in the processor.
- There is one TRBE for each PE in the processor.
- PEs do not share an ETE trace unit. Previous trace architectures allowed trace unit sharing.

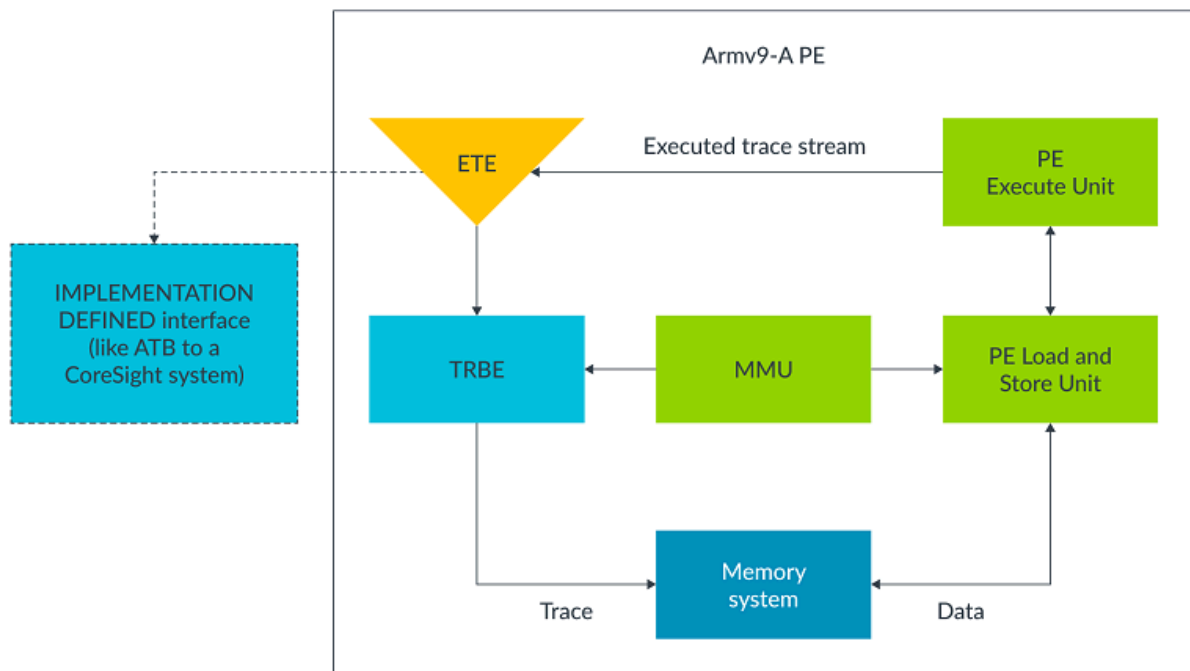
If a PE is permanently disabled, we recommend that the trace unit for the PE is removed from the target design. Alternatively, mark the trace unit as not present in the associated CoreSight ROM Table so the trace unit is not used.

Armv9-A targets can capture trace on-chip or off-chip. Capturing trace on-chip means storing the captured trace data on the target. The captured trace data is stored to the following:

- Main system memory through a buffer like with a TRBE.
- Shared system memory through the main system bus like with an ETR.
- Dedicated on-chip buffer like an ETB.

The captured on-chip trace data is either analyzed on target or pulled off the target later for analysis. Capturing trace off-chip means sending the trace data off-chip for analysis. The trace data is sent off-chip through an ATB and CoreSight trace subsystem or an **IMPLEMENTATION DEFINED** interface. An external debugger usually analyzes off-chip trace data.

The following diagram shows what an Armv9-A trace system with an ETE trace unit and TRBE might look like:



The ETE trace unit generates trace data based on the executed trace stream of the PE. The TRBE collects the trace data from the trace unit and stores the data to the memory system. Software can analyze the stored trace data on-chip or send it off-chip for later analysis.

If the TRBE pointers are virtual addresses, the TRBE uses the translation tables setup for the MMU of the owning translation regime. If the TRBE is disabled, if there is an **IMPLEMENTATION DEFINED** interface like an ATB to a CoreSight system, ETE trace data is sent to this interface.

You can flush the trace unit to make the trace stream visible to observers like other PEs on the SoC. Trace unit flush occurs when:

- Transitioning from an enabled to a disabled state.
- The trace capture infrastructure requests it.
- If the TRBE is implemented and enabled, executing a TSB CSYNC instruction.

TSB CSYNC is used with a DSB operation to flush trace to the TRBE. If FEAT_TRBE is implemented, it is possible to execute a TSB CSYNC instruction in Debug state.

Given the close connection between the PE and ETE trace unit, for some trace implementations, the trace unit might impact the performance of the PE.

You can program the ETE trace unit and TRBE to respond to certain inputs or drive output events to the wider system. The following are examples of input and output reactions:

- The trace unit takes in or generates PMU events.

- The trace unit reacts to events from the Cross Trigger Interface (CTI). An external debugger might use CTIs to control trace functionality.
- The TRBE sends an interrupt request through the Generic Interrupt Controller (GIC) to notify the PE of aborts or trace buffer filling.

5. Trace capture examples

This guide has four trace examples to show the basic programming necessary to create a minimal ETE trace unit and TRBE trace setup. Your trace environment might require more steps to achieve a working ETE and TRBE trace setup. To check for more steps, read your board documentation.

The examples are run using the following tools:

- [Arm AEMvA Fixed Virtual Platform \(FVP\)](#) model: Fast Models version 11.16 and later supports ETE and TRBE.
- [Arm Development Studio](#): Arm DS Platinum version 2021.a and later supports ETE.
- [OpenCSD](#): An open-source Arm trace decode library. This library requires a Linux host machine.

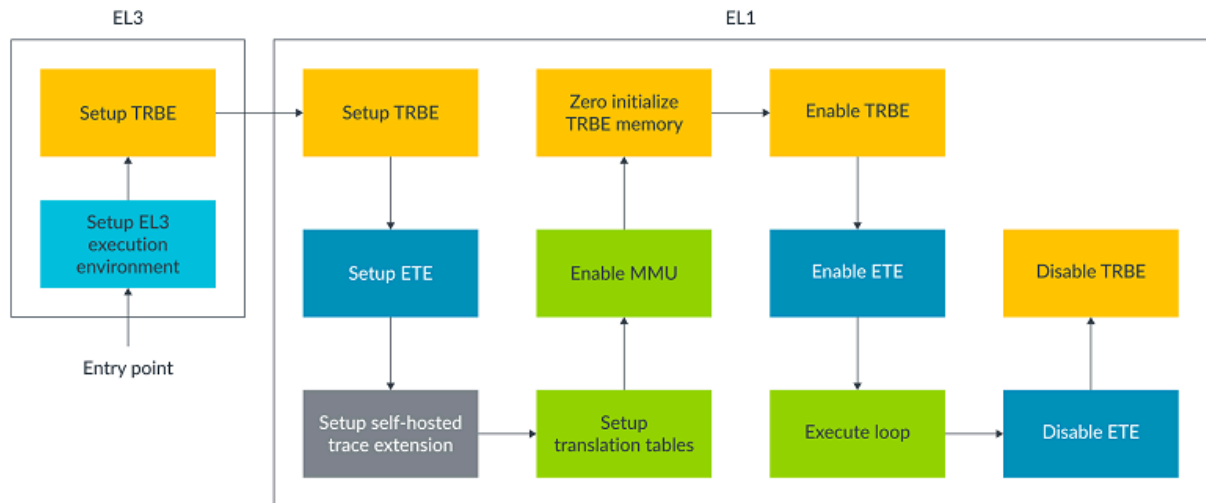
To download the four examples, click Download and select Armv-9A_trace_examples.zip. For more information about running the examples, read the included README.txt. The four examples are:

- `el1_self_hosted_trace`: Configure and trace EL1, Non-secure instruction execution.
- `el2_self_hosted_trace`: Configure and trace EL2, Non-secure instruction execution.
- `el2_el1_self_hosted_trace`: Configure and trace EL2 and EL1, Non-secure instruction execution. The owning Exception level for the TRBE is EL2.
- `el2_controlled_self_hosted_trace`: Configure and trace EL2 and EL1, Non-secure instruction execution. During execution, changes EL1 from a prohibited trace region to a permitted trace region. The owning Exception level for the TRBE is EL2.

The following sections explain the program flow, settings, and differences of each example.

5.1 el1_self_hosted_trace example

The following diagram shows the program flow of the el1_self_hosted_trace example:



In the program flow, the ETE trace unit and TRBE are configured before being enabled. Do not write to trace unit or TRBE registers unless the components are disabled and idle. When the components are enabled or not idle, writes to the registers are either ignored or cause **UNPREDICTABLE** behavior. TRCSTATR.IDLE indicates whether the trace unit is idle.

The TRBE is enabled before the trace unit to ensure the TRBE is ready to collect trace data from the trace unit. The trace unit is disabled before the TRBE to ensure the TRBE collects all the generated trace data.

In the examples, the TRBE uses virtual pointer addresses. Because virtual pointer addresses are used, the translation tables are configured and the MMU enabled before the TRBE is enabled.

The following tables explain the trace unit and TRBE operations and register settings in the example. The operations and settings are organized by Exception level.

In EL3, the following TRBE settings are used:

Register	Settings
MDCR_EL2	To set the TRBE owning Exception level to EL1, set E2TB to 0b11.
MDCR_EL3	To set the TRBE owning Security state to Non-secure, set NSTB to 0b11.

In EL1, the following ETE trace unit and TRBE operations and settings are performed:

Operation	Register	Settings
TRBE setup	TRBBASER_EL1	Set BASE to a writable memory address. address. In this example, 0x90000000 is DRAM.

Operation	Register	Settings
	TRBLIMITR_EL1	<ul style="list-style-type: none"> Set LIMIT to the end address of the trace buffer. In this example, the end address is 0xA0000000. To ignore triggers, set TM to 0b11. To use Circular Buffer mode, set FM to 0b11.
	TRBPTR_EL1	Set PTR to the same address as TRBBASER_EL1.BASE. This configuration sets the current write pointer to the start address of the trace buffer.
Setup ETE	OSLAR_EL1	To unlock the PE OS lock, clear OSLK. Unlike earlier trace architectures, ETE uses the OS lock of the PE. ETE does not have a separate OS lock.
	TRCCONFIGR	<ul style="list-style-type: none"> To enable the return stack, set RS to 1. To add virtual context identifiers to the trace stream, set VMID to 1. To add context identifiers to the trace stream, set CID to 1.
	TRCSYNCPR	To enable trace synchronization every 4096 bytes of trace, set PERIOD to 0xC.
	TRCTRACEIDR	To set the trace ID, set TRACEID to 0x2. The trace ID identifies which trace source the trace data belongs to when performing trace decode. If only TRBE trace capture occurs, the trace ID is not used.
	TRCVICTLR	<ul style="list-style-type: none"> To not generate trace for Non-secure EL2, set EXLEVEL_NS_EL2 to 1. To generate trace for Non-secure EL1, set EXLEVEL_NS_EL1 to 0. To not generate trace for Non-secure EL0, set EXLEVEL_NS_EL0 to 1. To not generate trace for Secure EL3, set EXLEVEL_S_EL3 to 1. To not generate trace for Secure EL2, set EXLEVEL_S_EL2 to 1. To not generate trace for Secure EL1, set EXLEVEL_S_EL1 to 1. To not generate trace for Secure EL0, set EXLEVEL_S_EL0 to 1. To set the ViewInst start/stop functions state to start, set SSSTATUS to 1. To select resource selector 1, set EVENT.SEL to 1.
	TRCCCCTLR	To set the threshold for instruction trace cycle counting to 0x16, set THRESHOLD to 0x16.
	TRCSTALLCTLR	To disable unused ETE functionality, set to zero.
	TRCTSCTLR	To disable unused ETE functionality, set to zero.
	TRCVIIECTLR	To disable unused ETE functionality, set to zero.
	TRCVISSCTLR	To disable unused ETE functionality, set to zero.
	TRCBBCTLR	To disable unused ETE functionality, set to zero.
	TRCRSR	To set the trace resource status to zero, clear this register.
Set up the Self-hosted Trace Extension	TRFCR_EL1	To enable EL0 and EL1 trace, set EOTRE and E1TRE to 1.
Enable TRBE	TRBSR_EL1	Clear this register.
	TRBLIMITR_EL1	Set E to 1.
Enable ETE	TRCPRGCTLR	Set EN to 1. To synchronize all the register updates, after setting TRCPRGCTLR.EN, you must execute an ISB instruction.
Disable ETE	TRFCR_EL1	Save the state of this register. After the ETE is disabled and idle, restore the state of TRFCR_EL1.
	TRFCR_EL1	Clear ExTRE to put the PE into a prohibited region to prevent further tracing. To synchronize the entry to a prohibited region, after clearing TRFCR_EL1.ExTRE, execute an ISB instruction. After the ISB instruction, to flush the trace data to the TRBE, execute a TSB CSYNC instruction.
	TRCPRGCTLR	To disable the ETE, set EN to 0.
	TRCSTATR	To check that the ETE is idle, poll IDLE.

Operation	Register	Settings
Disable TRBE	TRBLIMITR_EL1	To disable the TRBE, set E to 0.

After running the example to the WFI instruction in Arm DS, download and save the ETE trace data in the TRBE. The saved trace data is decoded using the OpenCSD trc_pkt_lister program. The OpenCSD trc_pkt_lister program decodes the trace packets that form the collected trace stream. For more information about trace packets, read the Protocol Description section of the [Arm® Architecture Reference Manual Supplement Armv9, for Armv9-A architecture profile](#).

With the trc_pkt_lister program, all the decoded trace data starts with `Idx:<n>; ID:<id>`. `Idx:<n>` shows the index of the trace packet in the trace stream. `<number>` gives the index number. `ID:<id>` shows the trace ID for the decoded trace packets. `<id>` is the trace ID number.

The following table shows the OpenCSD trc_pkt_lister program output for the trace data generated by the el1_self_hosted_trace example:



`Idx:<n>; ID:<id>` are removed to simplify the decode output.

OpenCSD trc_pkt_lister output	Meaning
CSD_GEN_TRC_ELEM_TRACE_ON([begin or filter])	Trace enabled.
OCSD_GEN_TRC_ELEM_PE_CONTEXT((ISA=A64) EL1N; 64-bit; VMID=0x0; CTXTID=0x0;)	<ul style="list-style-type: none"> Following trace captured at Non-secure EL1 (EL1N). Tracing AArch64 instructions (ISA=A64). Virtual Context identifier (VMID) is 0. Context identifier (CTXTID) is 0.
OCSD_GEN_TRC_ELEM_INSTR_RANGE(exec range=0x800002a4: [0x800002a8] num_i(1) last_sz(4) (ISA=A64) E ISB)	<p>At address 0x800002A4 (range=0x800002a4: [0x800002a8]), one instruction executed (num_i{1}).</p> <p>This instruction was an ISB instruction that was taken.</p>
OCSD_GEN_TRC_ELEM_INSTR_RANGE(exec range=0x80000290: [0x800002b8] num_i(10) last_sz(4) (ISA=A64) E BR <cond>)	<p>From address 0x80000290 to 0x800002B4, ten instructions executed.</p> <p>The last instruction was a BR <cond> instruction at address 0x800002B4 that was taken.</p>
OCSD_GEN_TRC_ELEM_INSTR_RANGE(exec range=0x800002a0: [0x800002b8] num_i(6) last_sz(4) (ISA=A64) E BR <cond>)	<p>From address 0x800002A0 to 0x800002B4, six instructions executed.</p> <p>The last instruction was a BR <cond> instruction at address 0x800002B4 that was taken.</p>
OCSD_GEN_TRC_ELEM_INSTR_RANGE(exec range=0x800002a0: [0x800002b8] num_i(6) last_sz(4) (ISA=A64) N BR <cond>)	<p>From address 0x800002A0 to 0x800002B4, six instructions executed.</p> <p>The last instruction was a BR <cond> instruction at address 0x800002B4 that was not taken.</p>

OpenCSD trc_pkt_lister output	Meaning
OCSD_GEN_TRC_ELEM_INSTR_RANGE(exec range=0x800002b8: [0x800002cc] num_i(5) last_sz(4) (ISA=A64) E iBR A64:eret)	<p>From address 0x800002B8 to 0x800002C8, five instructions executed.</p> <p>The last instruction was an ERET instruction at address 0x800002C8 that was taken.</p> <p>This instruction jumps from EL2 to EL1.</p>
OCSD_GEN_TRC_ELEM_EXCEPTION_RET()	An exception return occurred.
OCSD_GEN_TRC_ELEM_TRACE_ON([begin or filter])	<p>Because EL1 trace is prohibited, the first loop in EL1 is not traced.</p> <p>An HVC is executed to jump from EL1 to EL2 to enable EL1 trace.</p> <p>Trace is enabled on entry to EL2.</p>
OCSD_GEN_TRC_ELEM_PE_CONTEXT((ISA=A64) EL2N; 64-bit; VMID=0x0; CTXTID=0x0;)	<ul style="list-style-type: none"> Following trace captured at Non-secure EL2. Tracing AArch64 instructions. Virtual Context identifier is 0. Context identifier is 0.
OCSD_GEN_TRC_ELEM_INSTR_RANGE(exec range=0x80001400: [0x80001404] num_i(1) last_sz(4) (ISA=A64) E BR)	<p>At address 0x80001400, one instruction executed.</p> <p>This instruction was a BR instruction that was taken.</p> <p>This instruction is part of the EL1 trace enable code.</p>
OCSD_GEN_TRC_ELEM_INSTR_RANGE(exec range=0x800002cc: [0x800002dc] num_i(4) last_sz(4) (ISA=A64) N BR <cond>)	<p>From address 0x800002CC to 0x800002D8, four instructions executed.</p> <p>The last instruction was a BR <cond> instruction at address 0x800002D8 that was not taken.</p> <p>This instruction is part of the EL1 trace enable code.</p>
OCSD_GEN_TRC_ELEM_TRACE_ON([begin or filter])	<p>ETE trace unit and TRBE are disabled, EL1 trace is enabled, and ETE trace unit and TRBE are enabled.</p> <p>Trace shown as enabled after ETE is enabled in EL2.</p>
OCSD_GEN_TRC_ELEM_PE_CONTEXT((ISA=A64) EL2N; 64-bit; VMID=0x0; CTXTID=0x0;)	<ul style="list-style-type: none"> Following trace captured at Non-secure EL2. Tracing AArch64 instructions. Virtual Context identifier is 0. Context identifier is 0.
OCSD_GEN_TRC_ELEM_INSTR_RANGE(exec range=0x80000338: [0x8000033c] num_i(1) last_sz(4) (ISA=A64) E iBR A64:eret)	<p>At address 0x80000338, one instruction executed.</p> <p>This instruction was an ERET instruction that was taken.</p> <p>ERET instruction jumps from EL2 to EL1.</p>
OCSD_GEN_TRC_ELEM_EXCEPTION_RET()	An exception return occurred.

OpenCSD trc_pkt_lister output	Meaning
OCSD_GEN_TRC_ELEM_PE_CONTEXT((ISA=A64)EL1N; 64-bit; VMID=0x0; CTXTID=0x0;)	<ul style="list-style-type: none"> Following trace captured at Non-secure EL1. Tracing AArch64 instructions. Virtual Context identifier is 0. Context identifier is 0.
OCSD_GEN_TRC_ELEM_INSTR_RANGE(exec range=0x800003f4: [0x800003f8] num_i(1) last_sz(4) (ISA=A64) E ISB)	At address 0x800003F4, one instruction executed. This instruction was an ISB instruction that was taken.
OCSD_GEN_TRC_ELEM_INSTR_RANGE(exec range=0x800003f8: [0x80000420] num_i(10) last_sz(4) (ISA=A64) E BR <cond>)	From address 0x800003F8 to 0x8000041C, ten instructions executed. The last instruction was a BR <cond> instruction at address 0x8000041C that was taken.
OCSD_GEN_TRC_ELEM_INSTR_RANGE(exec range=0x80000408: [0x80000420] num_i(6) last_sz(4) (ISA=A64) E BR <cond>)	From address 0x80000408 to 0x8000041C, six instructions executed The last instruction was a BR <cond> instruction at address 0x8000041C that was taken.
OCSD_GEN_TRC_ELEM_INSTR_RANGE(exec range=0x80000408: [0x80000420] num_i(6) last_sz(4) (ISA=A64) E BR <cond>)	From address 0x80000408 to 0x8000041C, six instructions executed. The last instruction was a BR <cond> instruction at address 0x8000041C that was taken.
OCSD_GEN_TRC_ELEM_INSTR_RANGE(exec range=0x80000408: [0x80000420] num_i(6) last_sz(4) (ISA=A64) N BR <cond>)	From address 0x80000408 to 0x8000041C, six instructions executed. The last instruction was a BR <cond> instruction at address 0x8000041C that was not taken.

5.2 el2_self_hosted_trace example

The el2_self_hosted_trace example has the same program flow as the el1_self_hosted_trace example. The main difference between the two examples is the el1_self_hosted_trace example traces EL1 execution and the el2_self_hosted_trace example traces EL2 execution.

The following lists the differences between the el2_self_hosted_trace and el1_self_hosted_trace examples:

el2_self_hosted_trace example difference	el1_self_hosted_trace example difference
MDCR_EL2 is configured in EL2.	MDCR_EL2 is configured in EL3.
To set the TRBE owning Exception level to EL2, set MDCR_EL2.E2TB to 0b00.	To set the TRBE owning Exception level to EL1, set MDCR_EL2.E2TB to 0b11
At the end of the EL3 code, execution jumps to EL2.	At the end of the EL3 code, execution jumps to EL1.
For the trace unit to generate EL2 trace, TRCVICTLR.EXLEVEL_NS_EL2 is set to 0.	For the trace unit to generate EL1 trace, TRCVICTLR.EXLEVEL_NS_EL1 is set to 0.
To enable EL0 and EL2 trace, TRFCR_EL2.E0HTRE and TRFCR_EL2.E2TRE are set to 1.	To enable EL0 and EL1 trace, TRFCR_EL1.E0TRE and TRFCR_EL1.E1TRE are set to 1.

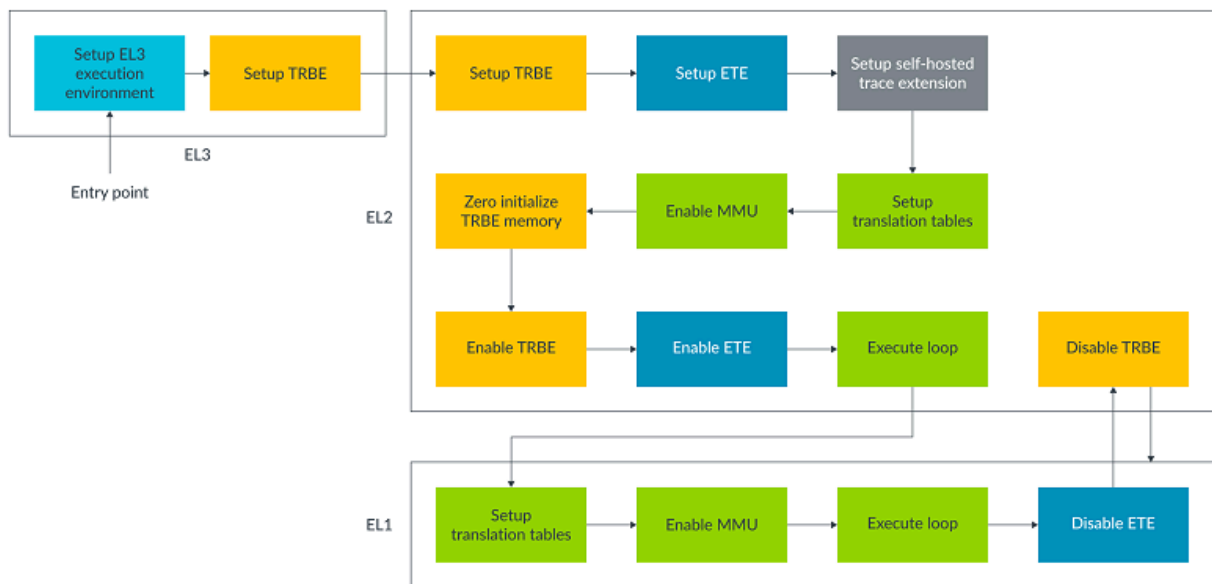
The decoded trace data for the `el2_self_hosted_trace` example is like the `el1_self_hosted_trace` example. The main difference between the two examples is the `el2_self_hosted_trace` example shows the trace data was generated when executing in EL2 rather than EL1.

5.3 el2_el1_self_hosted_trace example

Unlike the previous examples, the `el2_el1_self_hosted_trace` example traces two Exception levels, EL2 and EL1. To trace both EL2 and EL1, the `el2_el1_self_hosted_trace` example combines the ETE trace unit and TRBE register settings from the previous examples.

For instance, to enable EL2, EL1, and EL0 trace, `TRFCR_EL2.EOHTRE`, `TRFCR_EL2.E2TRE`, `TRFCR_EL1.E0TRE`, and `TRFCR_EL1.E1TRE` are all set to 1. Another instance is, to permit tracing Non-secure EL2 and EL1, `TRCVICTLR.EXLEVEL_NS_EL2` and `TRCVICTLR.EXLEVEL_NS_EL1` are set to 0.

The following diagrams shows the program flow of the `el2_el1_self_hosted_trace` example:



In this example, the owning Exception level for the TRBE is EL2. Because the owning Exception level is EL2, the following is true:

- The virtually addressed TRBE pointers are translated using the EL2 translation regime.
- The code must jump to EL2 before disabling the TRBE. The jump is performed by executing an HVC instruction.

The same loop is traced in both EL2 and EL1, so the trace data decode for the loop is the same for both Exception levels. The decoded trace data differs from the previous examples in the following ways:

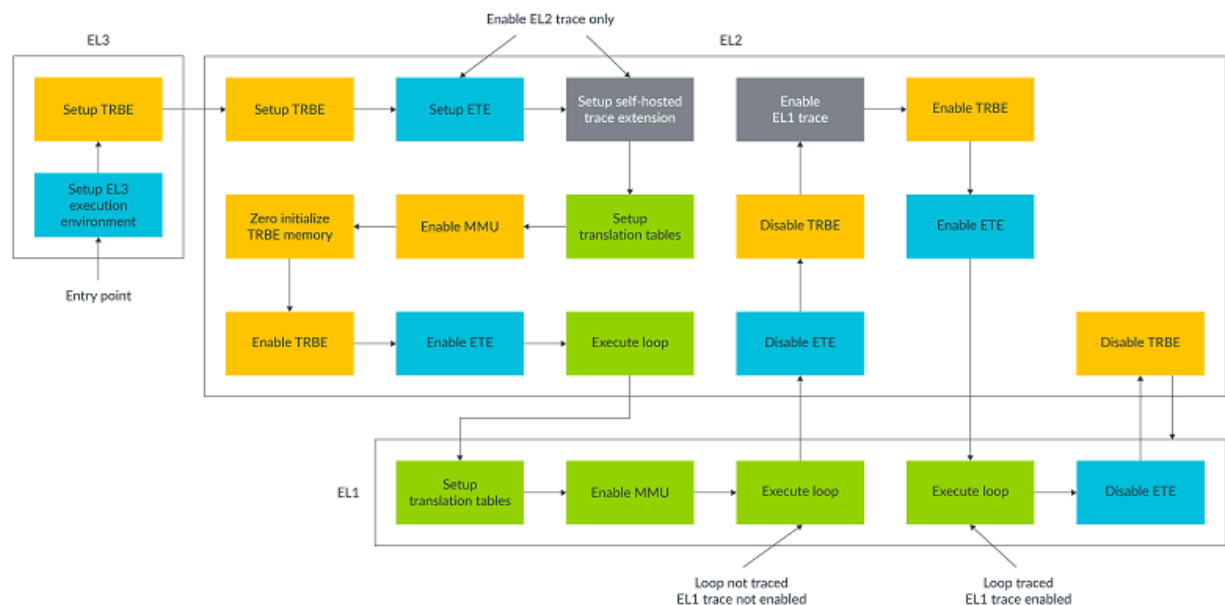
- There are two `OCSD_GEN_TRC_ELEM_PE_CONTEXT` messages: one message for when tracing in EL2 and another for when tracing in EL1.

- At the end of the EL2 trace decode section, the following message appears:
`OCSD_GEN_TRC_ELEM_INSTR_RANGE(exec_range=0x800002bc:[0x800002d0] num_i(5) last_sz(4) (ISA=A64) E iBR A64:eret` This message corresponds to five instructions being executed where the last instruction is an ERET instruction that jumps from EL2 to EL1.
- At the beginning of the EL1 trace decode section, four ISB instructions are shown as executed. These ISB instructions correspond to the ISB instructions executed during the EL1 translation table setup and MMU enable code.

5.4 el2_controlled_self_hosted_trace example

The `el2_controlled_self_hosted_trace` example is like the `el2_el1_self_hosted_trace` example in that two Exception levels are traced, EL2 and EL1. However, in the `el2_controlled_self_hosted_trace` example, the ETE trace unit and Self-hosted Trace Extension setup code in EL2 initially prohibits EL1 trace.

The following diagram shows the program flow of the `el2_controlled_self_hosted_trace` example:



In the program flow, the initial ETE trace unit and Self-hosted Trace Extension setup only enables EL2, not EL1. Because EL1 trace is prohibited, the first loop executed in EL1 is not traced. To enable EL1 trace, the code jumps back to EL2 by executing an HVC instruction.

Before EL1 trace is enabled, the trace unit and TRBE are disabled. These components are disabled to avoid the register writes not occurring or causing **UNPREDICTABLE** behavior.

Enable EL1 trace by setting:

- TRCVICTLR.EXLEVEL_NS_EL1 to 0
- TRFCR_EL1.E0TRE and TRFCR_EL1.E1TRE to 1

After EL1 trace is enabled, the trace unit and TRBE are enabled, and the code jumps back to EL1. A second loop is executed. This second loop is traced as EL1 trace is now enabled. After the second loop, the trace unit and TRBE are disabled. Like in the `el2_el1_self_hosted_trace` example, the code jumps to EL2 to disable the TRBE.

The following table shows the OpenCSD `trc_pkt_lister` program output for the trace data generated by the `el2_controlled_self_hosted_trace` example:



Idx:<n>; ID:<id> are removed to simplify the decode output.

OpenCSD <code>trc_pkt_lister</code> output	Meaning
<code>OCSD_GEN_TRC_ELEM_TRACE_ON([begin or filter])</code>	Trace enabled.
<code>OCSD_GEN_TRC_ELEM_PE_CONTEXT((ISA=A64) EL2N; 64-bit; VMID=0x0; CTXTID=0x0;)</code>	<ul style="list-style-type: none"> Following trace captured at Non-secure EL2 (EL2N). Tracing AArch64 instructions (ISA=A64). Virtual Context identifier (VMID) is 0. Context identifier (CTXTID) is 0.
<code>OCSD_GEN_TRC_ELEM_INSTR_RANGE(exec range=0x8000028c: [0x80000290] num_i(1) last_sz(4) (ISA=A64) E ISB)</code>	At address 0x8000028C, one instruction executed. This instruction was an ISB instruction that was taken.
<code>OCSD_GEN_TRC_ELEM_INSTR_RANGE(exec range=0x80000290: [0x800002b8] num_i(10) last_sz(4) (ISA=A64) E BR <cond>)</code>	From address 0x80000290 to 0x800002B4, ten instructions executed. The last instruction was a BR <cond> instruction at address 0x800002B4 that was taken.
<code>OCSD_GEN_TRC_ELEM_INSTR_RANGE(exec range=0x800002a0: [0x800002b8] num_i(6) last_sz(4) (ISA=A64) E BR <cond>)</code>	From address 0x800002A0 to 0x800002B4, six instructions executed. The last instruction was a BR <cond> instruction at address 0x800002B4 that was taken.
<code>OCSD_GEN_TRC_ELEM_INSTR_RANGE(exec range=0x800002a0: [0x800002b8] num_i(6) last_sz(4) (ISA=A64) E BR <cond>)</code>	From address 0x800002A0 to 0x800002B4, six instructions executed. The last instruction was a BR <cond> instruction at address 0x800002B4 was taken.
<code>OCSD_GEN_TRC_ELEM_INSTR_RANGE(exec range=0x800002a0: [0x800002b8] num_i(6) last_sz(4) (ISA=A64) N BR <cond>)</code>	From address 0x800002A0 to 0x800002B4, six instructions executed. The last instruction was a BR <cond> instruction at address 0x800002B4 that was not taken.
<code>OCSD_GEN_TRC_ELEM_INSTR_RANGE(exec range=0x800002b8: [0x800002cc] num_i(5) last_sz(4) (ISA=A64) E iBR A64:eret)</code>	From address 0x800002B8 to 0x800002C8, five instructions executed. The last instruction was an ERET instruction at address 0x800002C8 that was taken. This instruction jumps from EL2 to EL1.

OpenCSD trc_pkt_lister output	Meaning
OCSD_GEN_TRC_ELEM_EXCEPTION_RET()	An exception return occurred.
OCSD_GEN_TRC_ELEM_TRACE_ON([begin or filter])	<p>Because EL1 trace is prohibited, the first loop in EL1 is not traced.</p> <p>An HVC is executed to jump from EL1 to EL2 to enable EL1 trace.</p> <p>Trace is enabled on entry to EL2.</p>
OCSD_GEN_TRC_ELEM_PE_CONTEXT((ISA=A64) EL2N; 64-bit; VMID=0x0; CTXTID=0x0;)	<ul style="list-style-type: none"> Following trace captured at Non-secure EL2. Tracing AArch64 instructions. Virtual Context identifier is 0. Context identifier is 0.
OCSD_GEN_TRC_ELEM_INSTR_RANGE(exec range=0x80001400: [0x80001404] num_i(1) last_sz(4) (ISA=A64) E BR)	<p>At address 0x80001400, one instruction executed.</p> <p>This instruction was a BR instruction that was taken.</p> <p>This instruction is part of the EL1 trace enable code.</p>
OCSD_GEN_TRC_ELEM_INSTR_RANGE(exec range=0x800002cc: [0x800002dc] num_i(4) last_sz(4) (ISA=A64) N BR <cond>)	<p>From address 0x800002CC to 0x800002D8, four instructions executed.</p> <p>The last instruction was a BR <cond> instruction at address 0x800002D8 that was not taken.</p> <p>This instruction is part of the EL1 trace enable code.</p>
OCSD_GEN_TRC_ELEM_TRACE_ON([begin or filter])	<p>ETE trace unit and TRBE are disabled, EL1 trace is enabled, and ETE trace unit and TRBE are enabled.</p> <p>Trace shown as enabled after ETE is enabled in EL2.</p>
OCSD_GEN_TRC_ELEM_PE_CONTEXT((ISA=A64) EL2N; 64-bit; VMID=0x0; CTXTID=0x0;)	<ul style="list-style-type: none"> Following trace captured at Non-secure EL2. Tracing AArch64 instructions. Virtual Context identifier is 0. Context identifier is 0.
OCSD_GEN_TRC_ELEM_INSTR_RANGE(exec range=0x80000338: [0x8000033c] num_i(1) last_sz(4) (ISA=A64) E iBR A64:eret)	<p>At address 0x80000338, one instruction executed.</p> <p>This instruction was an ERET instruction that was taken.</p> <p>ERET instruction jumps from EL2 to EL1.</p>
OCSD_GEN_TRC_ELEM_EXCEPTION_RET()	An exception return occurred.
OCSD_GEN_TRC_ELEM_PE_CONTEXT((ISA=A64) EL1N; 64-bit; VMID=0x0; CTXTID=0x0;)	<ul style="list-style-type: none"> Following trace captured at Non-secure EL1. Tracing AArch64 instructions. Virtual Context identifier is 0. Context identifier is 0.
OCSD_GEN_TRC_ELEM_INSTR_RANGE(exec range=0x800003f4: [0x800003f8] num_i(1) last_sz(4) (ISA=A64) E ISB)	<p>At address 0x800003F4, one instruction executed.</p> <p>This instruction was an ISB instruction that was taken.</p>

OpenCSD trc_pkt_lister output	Meaning
OCSD_GEN_TRC_ELEM_INSTR_RANGE(exec range=0x800003f8: [0x80000420] num_i(10) last_sz(4) (ISA=A64) E BR <cond>)	From address 0x800003F8 to 0x8000041C, ten instructions executed. The last instruction was a BR <cond> instruction at address 0x8000041C that was taken.
OCSD_GEN_TRC_ELEM_INSTR_RANGE(exec range=0x80000408: [0x80000420] num_i(6) last_sz(4) (ISA=A64) E BR <cond>)	From address 0x80000408 to 0x8000041C, six instructions executed The last instruction was a BR <cond> instruction at address 0x8000041C that was taken.
OCSD_GEN_TRC_ELEM_INSTR_RANGE(exec range=0x80000408: [0x80000420] num_i(6) last_sz(4) (ISA=A64) E BR <cond>)	From address 0x80000408 to 0x8000041C, six instructions executed. The last instruction was a BR <cond> instruction at address 0x8000041C that was taken.
OCSD_GEN_TRC_ELEM_INSTR_RANGE(exec range=0x80000408: [0x80000420] num_i(6) last_sz(4) (ISA=A64) N BR <cond>)	From address 0x80000408 to 0x8000041C, six instructions executed. The last instruction was a BR <cond> instruction at address 0x8000041C that was not taken.

As shown by the decoded output, while EL1 trace is prohibited, trace is not captured for EL1. This lack of trace data demonstrates the control the ETE and Self-hosted Trace Extension has over what trace data is generated.

6. Related information

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

- [Arm Architecture Reference Manual for ARMv8-A architecture profile](#)
- [Arm® Architecture Reference Manual Supplement Armv9, for Armv9-A architecture profile](#)
- [Arm CoreSight Architecture Specification](#)
- [CPU architecture Exploration tools for the A-profile](#)
- [OpenCSD](#)

7. Next steps

This guide described ETE and TRBE and provided steps to capture and store trace data with ETE and TRBE. Through this guide you could learn what an Armv9-A trace system looks like and have an explanation through ETE and TRBE example projects.

To learn about the ETE and TRBE in more depth, read the [Arm® Architecture Reference Manual Supplement Armv9, for Armv9-A architecture profile](#). For a description of the ETE and TRBE registers, see the [CPU architecture Exploration tools for the A-profile](#). To learn more about the Arm A-profile architecture, read the [Learn the Architecture guides](#).