ARM® Debug and Trace

Configuration and Usage Models

Document number: ARM DEN 0034A

ARM®

ARM® Debug and Trace
Configuration and Usage Models

Release information

The following table lists the changes made to this document.

Change history

Date	Issue	Change
13 September 2013	A	First release

Non-Confidential Proprietary Notice

ARM Limited. Company 02557590 registered in England.
110 Fulbourn Road, Cambridge, England CB1 9NJ.

Table of Contents

1 Preface

1.1 About this document

This document describes usage models for ARM® debug and trace. It is organized into the following sections:

- Chapter 2 *Introduction* describes the basic concepts of debug and trace, the different usage models and the basics of the debug authentication interfaces and power management.
- Chapter 3 *Standard Usage Models for External Debug and Trace* describes the processing element configurations and software actions required to support debug and trace using a debugger that is external to the system being debugged.
- Chapter 4 *Standard Usage Models for Self-hosted Debug and Trace* describes the processing element configurations and software actions required to support debug and trace using a debugger that is executing on the system being debugged.

1.2 Additional reading

This section lists publications by ARM and by third parties.

See the Infocenter, http://infocenter.arm.com, for access to ARM documentation.

1.2.1 ARM publications

The following documents contain information relevant to this document:

- *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition* (ARM DDI 0406)
- *ARM Architecture Reference Manual ARMv7-M edition* (ARM DDI 0403)
- *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile* (ARM DDI 0487)
- *ARM Architecture Standard Configurations* (ARM DEN 0016)
- *ARM Debug Interface Architecture Specification ADIv5.0 to ADIv5.2* (ARM IHI 0031)
- *ARM System Memory Management Unit Architecture Specification* (ARM IHI 0062)
- *CoreSight Architecture Specification v2* (ARM IHI 0029)
- *CoreSight SoC Technical Reference Manual* (ARM DDI 0480)
- *CoreSight Trace Memory Controller Technical Reference Manual* (ARM DDI 0461)
- *Embedded Trace Macrocell Architecture Specification ETMv1.0 to ETMv3.5* (ARM IHI 0014)
- *Embedded Trace Macrocell Architecture Specification ETMv4* (ARM IHI 0064)
- *Power State Coordination Interface* (ARM DEN 0022)
- *Program Flow Trace Architecture Specification* (ARM IHI 0035)
- *STM Programmers' Model Architecture* (ARM IHI 0054)

2 Introduction

Debug enables a platform's software developers to create applications, middleware and platform software that meets the three key criteria of high performance, lower power consumption, and reliability.

External debug features were first introduced by the ARMv4 architecture to support developers using embedded and deeply-embedded processors, and has evolved into a broad portfolio of debug and trace features.

Support for rich application software platforms, in particular support for *self-hosted debug* and *performance profiling*, has been a more recent addition, in ARMv6 and ARMv7.

This document describes the usage models for these types of debug, and describes the responsibilities of the hardware and software when implementing these usage models.

Self-hosted debug and trace

To enable a mass-market of developers creating rich applications, a platform requires development tools that often run (at least in part) on the application processor itself rather than requiring expensive interface hardware to connect a second "host" computer. v8-A refines the architecture's support for this *self-hosted* form of debug. On existing desktop platforms, self-hosting is the prevalent method for software development.

Semi-hosting is a variant of self-hosting where a host computer is used to offload much of the work of a tool (such as the user interface, debug illusion, symbol management, etc.) from the target, usually using a low-cost interface (such as USB or Ethernet) to connect to the target. The architecture has no need to distinguish between semi- and self-hosting.

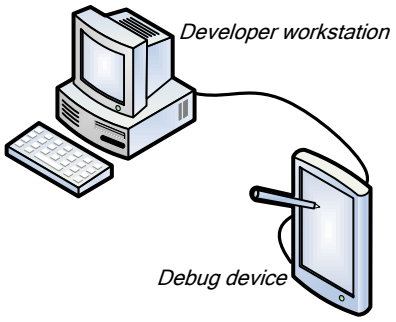


Figure 1 Semi-hosted debug environment

Note: *Self-hosted* debug is also known as *monitor mode* or *foreground* debug.

External debug and trace

However, often a complex system requires much of its hardware and software to be functional before any standard interfaces can be used for debug. It is very important to be able to get debug a system without relying on the system being debugged. This needs reliable *external debug* – that is, hardware-assisted, run-control debug and trace features, all of which can be controlled without the need for software operating on the platform – often very early in the product design cycle.

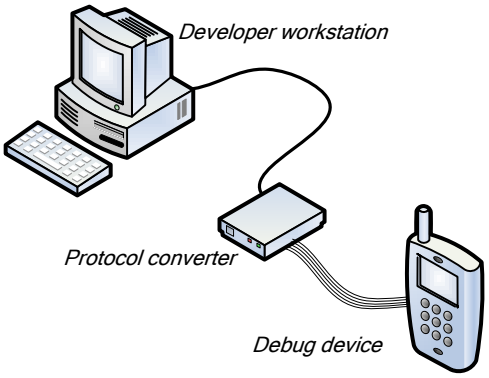


Figure 2 External debug environment

External debug usually means the debugger is executing on a separate machine connected via a debug interface such as JTAG or *Serial Wire Debug* (SWD). However, since ARMv7 external debug has also supported one processor debugging another within the same system.

Note: This is referred to as external debug even though the debugger and target are part of the same system.

Self-hosted tools usually require layers of software support, making it difficult to debug parts of the software, or making debugging too invasive for diagnosing some kinds of bug. Low-cost external debug interfaces such as SWD also help extend the range of applications where external debug is attractive.

Note: *External* debug is also known as *halting mode* or *background* debug.

Performance profiling

Self-hosted and external debug help a developer improve the reliability of the system. Key aspects of high performance and lower power consumption can be addressed using *performance profiling*. For many applications developers, the scope for performance optimization is somewhat limited, as they rely on middleware layers delivered by the platform. For those developing middleware platforms for the ARM architecture, performance optimization is critical. For a complex *System-on-chip* (SoC), profiling must not measure only the processor but the whole platform.

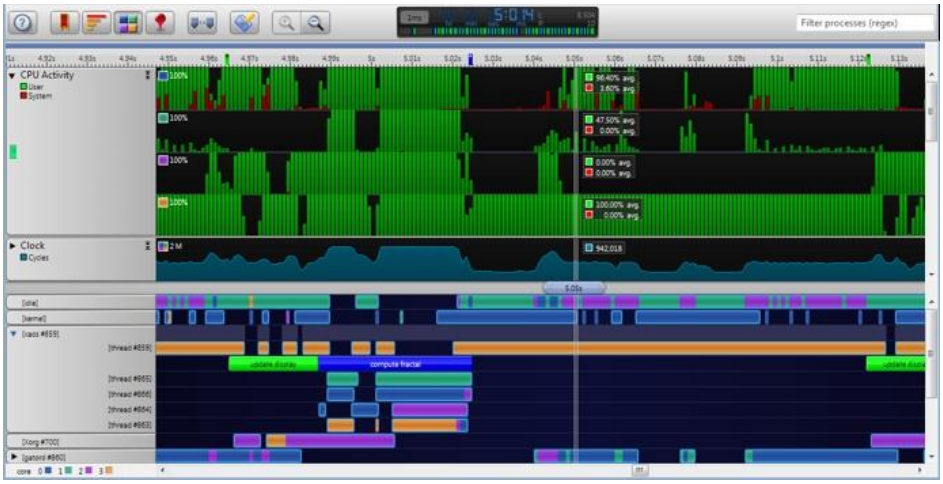


Figure 3 Screenshot from a performance profiling tool

2.1 ARM architecture privilege model and software views

The ARM architecture supports several software layers, each executing at a different execution or *exception level*. There are four exception levels, from the exception level with the lowest privilege, EL0, to the highest privilege, EL3. These exception levels are shown hierarchically in Figure 4.

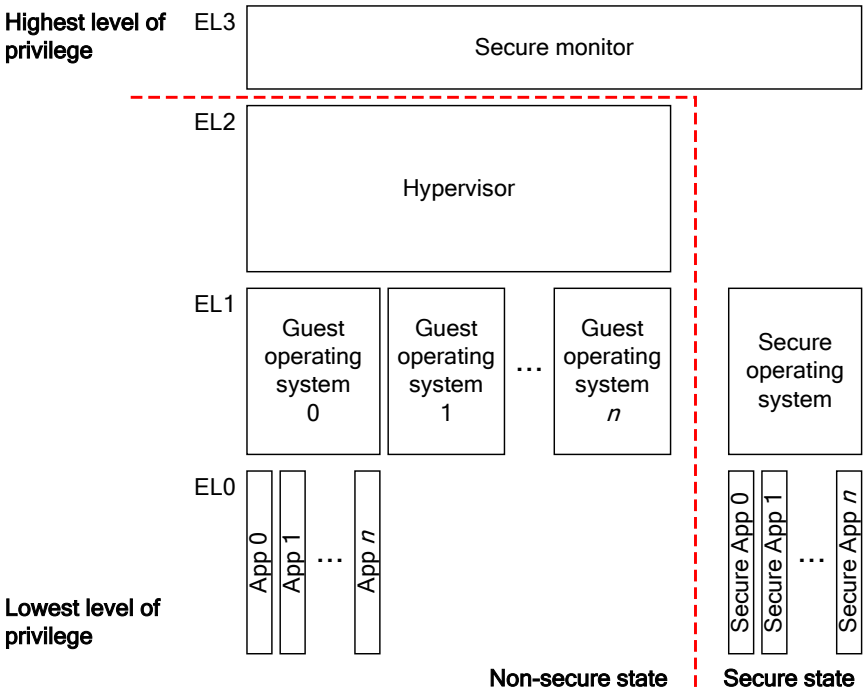


Figure 4 Exception Levels, EL3 using AArch64

Not all processors implement all exception levels. EL2 and EL3 are optional. If EL3 is not implemented then:

- For an ARMv7 processor, only the Secure state is implemented.
- For an ARMv8 processor:
  - If EL2 is implemented then only the Non-secure state is implemented.
  - Otherwise it is IMPLEMENTATION DEFINED which security state is implemented.

Secure EL1 is not supported if the highest level of privilege (EL3) is using AArch32, which:

- Is always true for an ARMv7 processor.
- Is optional for ARMv8.

The *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition* uses a different concept of *privilege levels* for each processor mode:

PL0 Privilege Level 0 describes modes at EL0, that is, User mode.

PL1 Privilege Level 1 describes modes at Non-secure EL1 and Secure EL3, that is, all modes other than User mode and Hyp mode.

PL2 Privilege Level 2 describes modes at EL2, that is, Hyp mode.

This is shown in Figure 5.

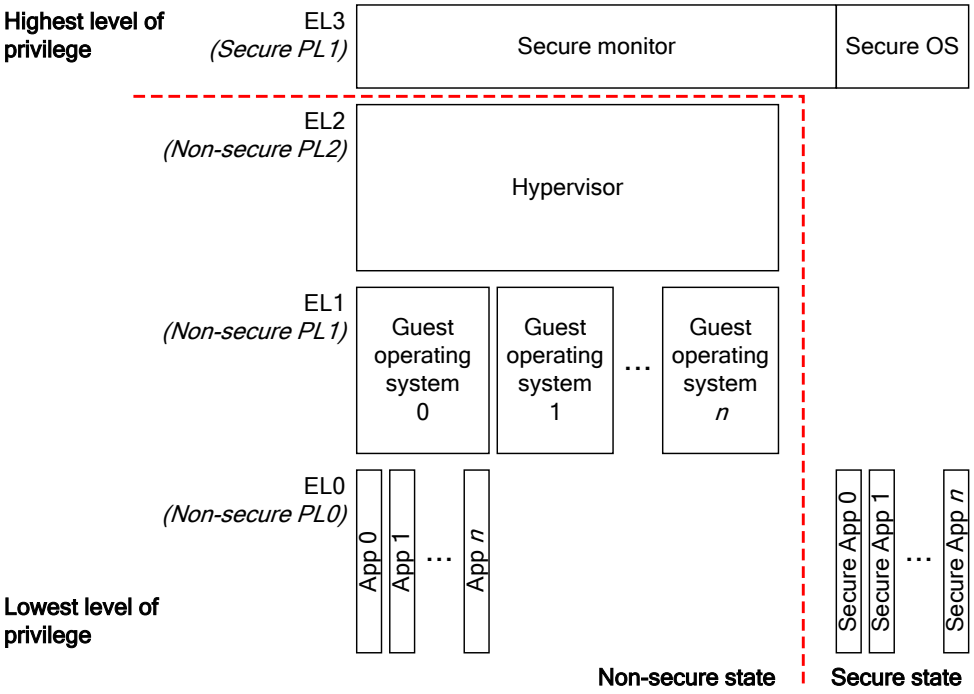


Figure 5 Exception and Privilege Levels, EL3 using AArch32

Associated with the exception levels are *software views*:

- The *Hardware View*, which supports running a Secure monitor for switching between Secure and Non-secure states.
- The *Virtualizer View*, which supports running a Hypervisor for switching between multiple Guest operating systems. The Virtualizer View is only available in Non-secure state.

  Note: The Virtualizer View is indistinguishable from the Hardware View in Non-secure state.
- The *Single Machine View*, which supports running an operating system kernel:
  - *Secure Machine View* is the variant of this view in Secure state

    Note: If EL3 is using AArch32 the Secure Machine View is indistinguishable from the Hardware View in Secure state.
  - *Multiple Machine View* refers to the union of several Single Machine Views under a hypervisor.
- The *Application View*, which is unprivileged and supports application code:
  - *Single Application View* and *Multiple Application View* refer to views of one or more applications
  - *Single Secure Application View* and *Multiple Secure Application View* are variants of these views in Secure state.

These views are shown hierarchically in Figure 6.

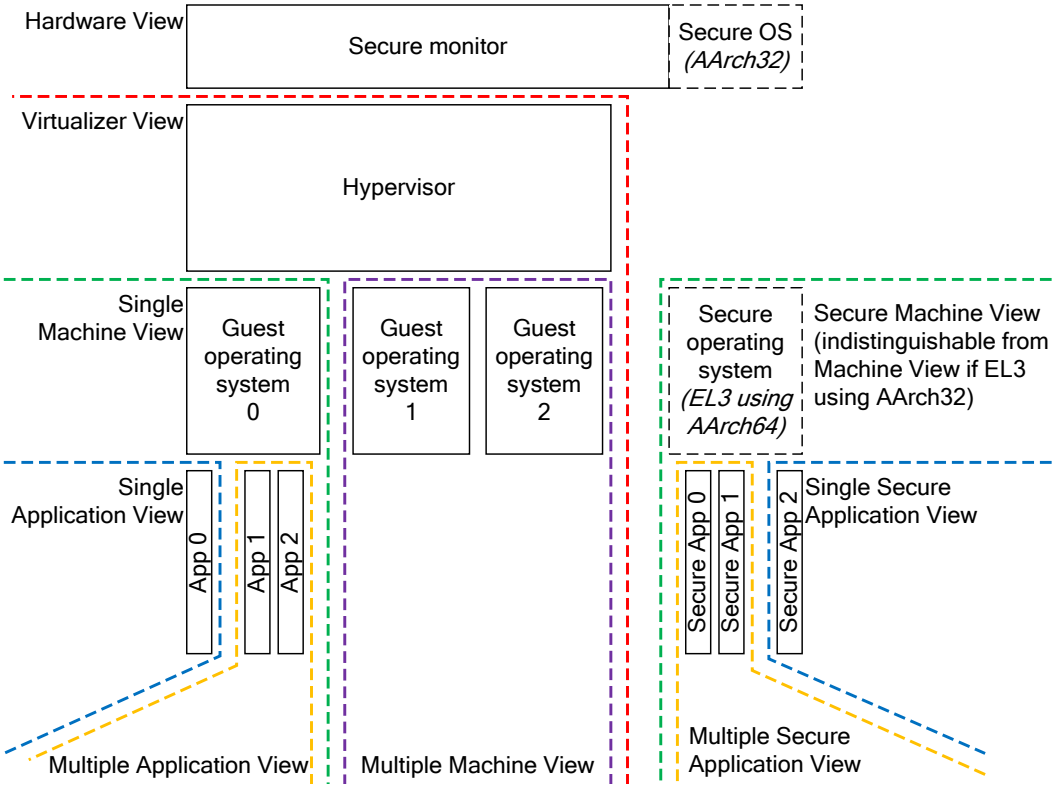


Figure 6 Machine Views

In the ARM debug architecture, each software layer can enable either:

- That view to be debugged.
  - For example, operating system kernel debugging.
- The views below it to be debugged.
  - For example, an operating system providing debug services to applications, or a hypervisor managing one or more guest operating system contexts.

In both cases, the software component may be required to prevent:

- Debug visibility of more privileged views from less privileged views.
- Malicious or accidental corruption of hardware debug by less privileged views.

For some types of debug visibility and some views, this can be done by a combination of hardware and software support.

2.2 CoreSight system debug

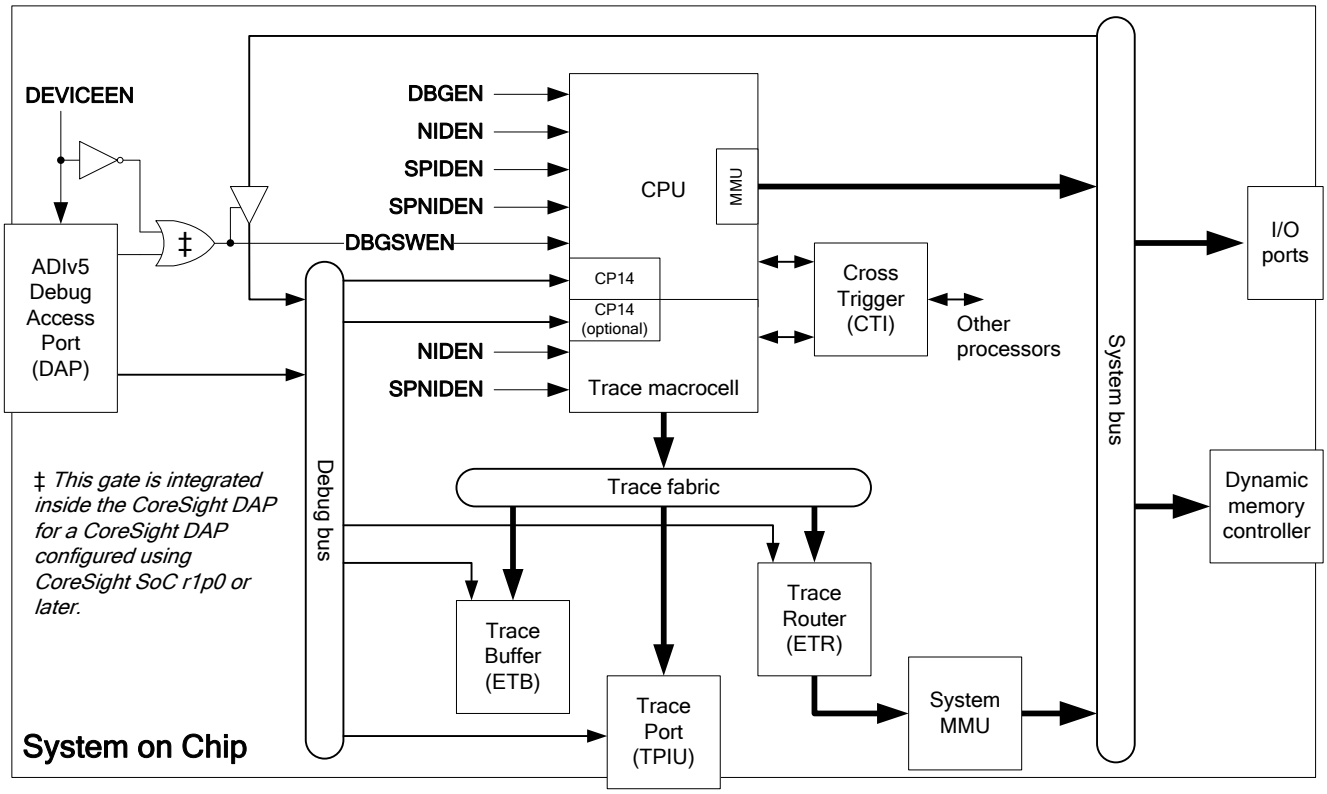


Figure 7 Example SoC with CoreSight interfaces

Figure 7 shows an SoC with CoreSight™ interfaces. With respect to the components shown:

- The processor:
  - Can drive transactions onto the system bus to access bulk memory, and debug components through a bridge to the debug bus.
  - Has five *debug authentication* signals:
    - **DBGEN**
    - **NIDEN**
    - **SPIDEN**
    - **SPNIDEN**
    - **DBGSWEN** (ARMv7 only).
  - Is programmable by target software:
    - If an ARMv7 processor, via the CP14 register interface.
    - If an ARMv8 processor, via the system register interface.
  - Has a slave interface from the debug bus.
- The trace macrocell:
  - Can generate trace data.
  - Has two debug authentication signals:
    - **NIDEN**
    - **SPNIDEN**
  - Might be programmable by target software:
    - If an ARMv7 processor, via the CP14 register interface.

        - If an ARMv8 processor, via the system register interface.

      These interfaces are not implemented on ARM Cortex processors. Target software must program the trace macrocell using the debug bus.

    - Has a slave interface from the debug bus.

- The trace fabric, consists of a number of components such as:
    - Funnels, which collects trace data from multiple slave interfaces onto a single master interface.
    - Replicators, which splits trace data from a single slave interface to multiple master interfaces.
    - *Embedded Trace FIFOs* (ETFs) which have a single slave and master interfaces.

  Some of these components have slave interfaces from the debug bus.

- The trace sinks, such as an *Embedded Trace Router* (ETR), *Parallel Trace Port Unit* (TPIU), or *Embedded Trace Buffer* (ETB) each of which:
    - Has a slave interface from the trace fabric.
    - Has slave interfaces from the debug bus.

  The ETR has a master interface to write trace to the system bus via a System MMU to provide a translation context. An ETF can be programmed to behave as an ETB trace sink.

- The *Cross Trigger Interface* (CTI) routes events between components.
- The ADIv5 *Debug Access Port* (DAP):
    - Connects to an external debugger via JTAG or SWD, and is ADIv5 compliant.
    - Contains a *debug memory access port* which is a master of the debug bus from a *debug memory*.
    - Optionally contains a *system memory access port* which is a master interface to the system bus (not shown).
    - Generates the ARMv7 **DBGSWEN** signal.

      Note: **DBGSWEN** is sometimes referred to as **DBGSWENABLE**. The debugger should not normally de-assert **DBGSWEN**. **DBGSWEN** is not required by ARMv8 processors.

    - Has three debug authentication signals:
        - **DEVICEEN**
        - **DBGEN**
        - **SPIDEN**.

      If **DEVICEEN** is LOW, the DAP cannot generate debug bus transactions and **DBGSWEN** is forced HIGH

      Note: In a CoreSight DAP configured using CoreSight SoC revisions earlier than r0p1, or any CoreSight Design Kit that is not CoreSight SoC, **DBGSWEN** must be forced HIGH externally to the DAP.

      DBGEN and **SPIDEN** are used by optional system memory access port to control whether Non-secure and Secure system memory can be accessed.

Other interfaces and connections are not shown.

Note: The DAP is a second master for the debug bus. To properly secure the system there should be no other masters of the debug bus that are not secured, either by an MMU / System MMU or a debug authentication interface (i.e. **DEVICEEN**).

This document does not describe in detail the programmers' models for all these components. For more information on ADIv5 and CoreSight components see:

- *ARM Debug Interface Architecture Specification ADIv5.0 to ADIv5.2*
- *CoreSight Architecture Specification v2*
- *CoreSight SoC Technical Reference Manual*
- *CoreSight Trace Memory Controller Technical Reference Manual*.

For more information on processor debug see:

- *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition*
- *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile*

For more information on trace macrocells see:

- *Embedded Trace Macrocell Architecture Specification ETMv1.0 to ETMv3.5*
- *Embedded Trace Macrocell Architecture Specification ETMv4*
- *Program Flow Trace Architecture Specification*.
- *STM Programmers' Model Architecture*

For more information on the System MMU see *ARM System Memory Management Unit Architecture Specification*.

2.3 Summary of debug and trace types

The types of debug visibility supported are:

ED External Debug. Debug controlled from an external *host* connected to the *target* SoC by an ADIv5 interface.

ET External Trace. Trace controlled from an external host connected to the target by ADIv5 and (optionally) CoreSight trace port interfaces.

EP External Profiling. Profiling controlled from an external host connected to the target by an ADIv5 interface.

SD Self-hosted Debug. Debug tools that run, at least in part, on the target, and communicate with the user using only standard interfaces.

ST Self-hosted Trace. Trace tools that run, at least in part, on the target, and communicate with the user using only the standard interfaces.

SP Self-hosted Profiling. Profiling tools that run, at least in part, on the target, and communicate with the user using only the standard interfaces.

This document does not consider use of other trace components, such as a *System Trace Macrocell* (STM). However, many of the principles described extend to these other components. Software must take into account that these components might be dynamically shared and simultaneously used by multiple agents.

2.3.1 Self-hosted debug

Self-hosted debug uses the debug hardware to generate debug exceptions which are processed by target software. Self-hosted debug supports several usage models:

Application debugging

The debug hardware can be programmed to generate debug exceptions from an application (EL0) which are handled by an operating system (EL1). This allows an operating system to debug an application. The debug hardware must be programmed to prevent debug exceptions being generated from EL1.

Kernel debugging

The debug hardware can be programmed to generate debug exceptions from both EL0 and EL1 and taken to EL1. This allows a kernel debugger extension to an operating system to make use of hardware debug features.

ARMv8 adds features to prevent re-entrant exceptions within critical code regions, and ARMv8 kernel debugging must be enabled separately from application debugging.

Guest operating system debugging

In Non-secure state, debug exceptions from EL0 and EL1 can be routed to a hypervisor (EL2), as in the v7-A Virtualization Extensions. This allows the hypervisor to debug a guest operating system.

Hypervisor debugging

In ARMv8 only, the kernel debugging feature can be applied to EL2, allowing a kernel debugger extension to a hypervisor.

Note: This document uses the term *kernel debugging* to cover both debugging of an operating system at EL1 by a debugger at EL1 and of a hypervisor at EL2 by a debugger at EL2.

ARMv7 does not support self-hosted hypervisor debugging.

The ARM architecture does not support routing of debug events to a Secure monitor (EL3).

The virtualization extensions for debug also allow the hypervisor to share hardware debug resources between guest operating systems, and emulate self-hosted debug within a guest.

2.3.2 Self-hosted trace

Self-hosted trace uses trace hardware generate and collect trace data on the target. Trace data typically consists of time-based information about the execution flow of a program. Self-hosted trace requires target software to:

- Manage the trace macrocell, such as a:
  - *Embedded Trace Macrocell* (ETM), compliant to an ETM architecture.
  - *Program Trace Macrocell* (PTM), compliant to the *Program Flow Trace* (PFT) architecture.
  - *System Trace Macrocell* (STM).
- Manage the trace fabric.
- Manage the trace sink(s).
- Process the trace.

Some processing and visualization of the trace might be done on an external system.

Target software must configure the trace macrocell to generate trace for software being traced. If an ETR is being used, this must be configured to collect the trace in a buffer belonging to the target software. This might require use of a *System MMU* to provide stage 2 translations in a system using Virtualization, as accesses made by an ETR are not subject to the processor's translation regimes.

2.3.3 Self-hosted profiling

Self-hosted profiling uses profiling hardware to generate, collect and process profiling data on the target. Profiling data typically consists of counter values and sampled data relating to the performance of programs on the hardware. Self-hosted profiling requires target software to:

- Manage the profiling hardware, such as *Performance Monitoring Units* (PMUs).
- Process the profiling data.

Some processing and visualization of the profiling data might be done on an external system. Profiling data might be collected using the same trace fabric, but is typically collected by reading profiling registers.

Self-hosting profiling is not considered in detail by this document. The requirements are largely covered by the support described for self-hosted debug and self-hosted trace.

2.3.4 External debug

The basic principles of halting debug in the ARM architecture are:

- When configured for halting debug, a debug event causes entry to a special *Debug state*.
- In Debug state, the processor does not fetch instructions from memory, but from a special *Instruction Transfer Register*.
- *Data Transfer Registers* are used to move register and memory contents between host and target.

An important characteristic of an external debugger is that it is operating concurrently and (possibly) independently of the process or processor being debugged, and debugging must be possible out of device reset.

2.3.5 External trace

External trace uses trace hardware to generate trace data, and either:

- Collect it externally to the target. This is typically in some large off-chip buffer, from where it is transferred to the host. It might also be streamed to the host directly.
- Collect it on the target and later transfer it to the host.

External trace requires host software to:

- Manage the trace macrocell, such as ETM or PTM.
- Manage the trace fabric.
- Manage the trace sink(s).
- Process the trace.

2.3.6 External profiling

External profiling uses the profiling hardware to collect profiling data on the target and later transfer and process it on the host.

External profiling requires host software to:

- Manage the profiling hardware, such as PMUs.
- Collect and process the profiling data.

External profiling is not considered in detail by this document. The requirements are largely an overlap of external debug and external trace.

2.4 Debug authentication interfaces

The debug authentication interface is used to restrict the capabilities of a debugger. Table 1 below summarizes each of the debug authentication interface signals.

Table 1 Summary of authentication interface signals

Signal	Behavior at each type of component			
	ARMv7 processor	**ARMv8 processor**	**Trace macrocell (ARMv7 and ARMv8)**	**Debug access port (ARMv7 and ARMv8)**
DBGEN	Controls whether any invasive debug of Non-secure state is enabled. Affects both self-hosted and external debugging.	Controls whether any external invasive debug of Non-secure state is enabled. Does not affect self-hosted debug of Non-secure state.	-	If the DAP includes a system memory access port, controls whether access to Non-secure system memory is allowed.
NIDEN	Controls whether performance monitoring of Non-secure state is enabled. Affects both self-hosted and external use.	Controls whether external access to performance monitors is allowed. Does not affect self-hosted use of performance monitors.	Controls whether trace of Non-secure state is enabled.	-
SPIDEN	Controls whether any invasive debug of Secure state enabled. Affects both self-hosted and external debugging.	Controls whether any external invasive debug of Secure state is enabled. Affects self-hosted debug of AArch32 Secure state by default, but can be overridden by software. Does not affect self-hosted debug of AArch64 Secure state.	-	If the DAP includes a system memory access port, controls whether access to Secure system memory is allowed.
SPNIDEN	Controls whether performance monitoring of Secure state is enabled. Affects both self-hosted and external use.	Overrides self-hosted controls for performance monitoring of Secure state.	Controls whether trace of Secure state is enabled.	-
DBGSWEN	Controls whether self-hosted debug can access shared debug resources.	-	-	-
DEVICEEN	-	-	-	Controls whether access to on-chip debug resources is allowed.

These are only brief summaries. For more information see:

- *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition*
- *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile*
- *CoreSight Architecture Specification v2*
- *Embedded Trace Macrocell Architecture Specification ETMv1.0 to ETMv3.5*
- *Embedded Trace Macrocell Architecture Specification ETMv4*
- *Program Flow Trace Architecture Specification*.

2.4.1 Configuring the debug authentication interface for different views

Chapters 3 and 4 give detailed descriptions of the authentication settings for different use cases of debug and trace. To support all the use cases described in this document, the platform must support these configurations.

The method by which an external debugger configures the authentication settings within the platform is platform dependent. Typical examples include:

- The silicon provider or OEM fixes the configuration using fuses.
- Initial boot software writes to a secure peripheral that overrides the fixed configuration with configuration settings from a signed boot image.
- An external debugger writes a secret value to a JTAG scan chain to authenticate itself and override the fixed configuration. This secret value might vary from device to device.
- The external debugger interacts with an *Authentication Module* which issues a challenge and allows the debugger to change the settings if it provides a valid response. This might involve the debugger interacting with a secure server.

The authentication settings are summarized in:

- Table 2 for ARMv7 processors
- Table 3 for ARMv8 processors.

In these tables, an asterisk (*) next to a value means there is more information about this signal value in the relevant subsections of chapters 3 and 4.

Table 2 ARMv7 authentication summary

Software view(s)	Use case	NIDEN	SPNIDEN	DBGEN	SPIDEN	DEVICEEN
Hardware View	External debug	HIGH	HIGH	HIGH	HIGH	HIGH
	External trace	HIGH	HIGH	-	-	HIGH
	Self-hosted debug	-	-	HIGH	HIGH	-
	Self-hosted trace	HIGH	HIGH	-	-	-
Virtualizer View	External debug	HIGH	LOW	HIGH	LOW	HIGH
	External trace	HIGH	LOW	-	LOW	HIGH
	Self-hosted debug	HIGH	LOW	HIGH	LOW	-
	Self-hosted trace	HIGH	LOW	-	LOW	-
Single Machine View / Multiple Machine View / Single Application View / Multiple Application View	Self-hosted debug	HIGH	LOW	HIGH	LOW	-
	Self-hosted trace	HIGH	LOW	-	LOW	-
Secure Machine View	Self-hosted debug	HIGH	HIGH*	HIGH	HIGH*	-
	Self-hosted trace	HIGH	HIGH*	-	-	-
Disabled	External debug	-	-	-	-	LOW
	External trace	-	-	-	-	LOW

Table 3 ARMv8 authentication summary

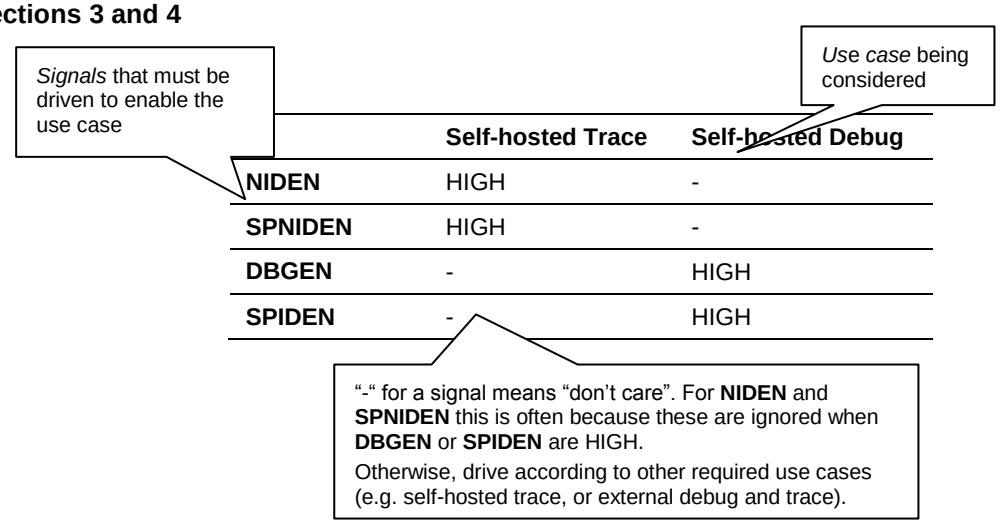
Software view(s)	Use case	NIDEN	SPNIDEN	DBGEN	SPIDEN	DEVICEEN
Hardware View	External debug	HIGH	HIGH	HIGH	HIGH	HIGH
	External trace	HIGH	HIGH	-	-	HIGH
	Self-hosted debug	-	-	-	-	-
	Self-hosted trace	HIGH	HIGH	-	-	-
Virtualizer View	External debug	HIGH	LOW	HIGH	LOW	HIGH
	External trace	HIGH	LOW	-	-	HIGH
	Self-hosted debug	-	-	-	-	-
	Self-hosted trace	HIGH	LOW	-	LOW	-
Single Machine View / Multiple Machine View / Single Application View / Multiple Application View	Self-hosted debug	-	-	-	-	-
	Self-hosted trace	HIGH	LOW	-	LOW	-
Secure Machine View	Self-hosted debug	-	-	-	-	-
	Self-hosted trace	HIGH	HIGH*	-	-	-
Disabled	External debug	-	-	LOW	LOW	LOW*
	External trace	-	-	LOW	LOW	LOW

2.4.2 Heterogeneous architecture systems

Because of the differences between ARMv8 and ARMv7 processors, it is recommended that systems employing a mix of ARMv7 and ARMv8 processors implement separate **DBGEN**, **NIDEN**, **SPIDEN** and **SPNIDEN** signals for each domain.

(More generally, systems should consider separate authentication signals for each application domain.)

2.4.3 Using the tables in sections 3 and 4



2.5 Power management and multiprocessor systems, including big.LITTLE

For multiprocessor (MP) systems, tasks may migrate between processors, and individual processors within the system can change their power states.

A software layer, called the *Operating System Power Management* (OSPM) software in this document, controls the operating conditions and schedules processes to maximize performance and efficiency by interrogating and controlling a power controller. The power controller is able to turn on and off components, control clock speeds, and so on. The nature of the power controller and granularity of control it provides is platform-specific and outside the scope of this document. It might be implemented in hardware or as a software service running on one of the processors or on a dedicated *system control processor* (SCP).

For more information on OSPM software, see *Power State Coordination Interface*.

The ARM architecture also supports heterogeneous MP systems comprising processors both high performance processors and high efficiency processors, arranged in separate clusters. That is, with different compute capacities but a common architecture. An example of this is referred to as a *big.LITTLE* system.

The OSPM can schedule software to take advantage of the heterogeneous system. It can use one of three common scheduling models for big.LITTLE systems:

- Cluster migration.
- CPU migration.
- Multiprocessor (MP).

big.LITTLE Cluster Migration

The hardware platform has two clusters (big and LITTLE), each with the same number of processors.

The OSPM considers factors including the overall load on the currently active cluster, and switches the complete cluster context between clusters if necessary. Only one cluster is active most of the time. (Both clusters will be active during switchover.)

big.LITTLE CPU Migration

The hardware platform has two clusters (big and LITTLE), each with the same number of processors. Each processor in one cluster is paired with a processor in the other cluster. Only one processor of each pair is active at any one time. Both clusters can be active at the same time.

The OSPM software considers factors including the load on each processor, and switches an individual processor context between clusters if necessary.

big.LITTLE MP

The hardware platform has multiple clusters with multiple processors. Any processor can be active at any time.

The SMP operating system operates across all processors in all clusters, and schedules processes onto processors according to their load requirements. The OSPM manages the power states of each processor according to the current load on that processor.

Microarchitectural differences in heterogeneous systems

ARM recommends that in a heterogeneous MP system, the different classes of processor implement at least:

- The same number of breakpoints and watchpoints.
- The same number of event counters.

However, there may be microarchitectural differences between processors that must be considered when migrating state between processors.

For example:

- Different trace architectures define different trace protocols, feature sets and programmers' models.
- The set of events implemented by a PMU is IMPLEMENTATION DEFINED, and the interpretation of events requires an understanding of the microarchitecture.

3 Standard Usage Models for External Debug and Trace

External Debug and External Trace only support debugging at two views, shown in Figure 8.

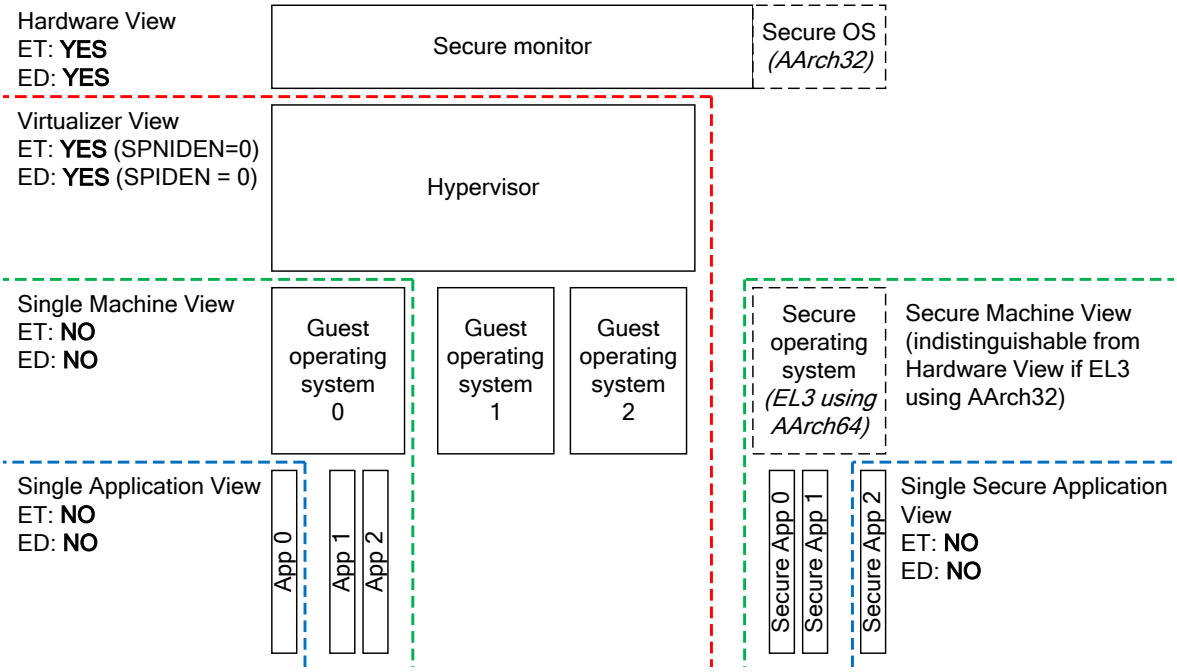


Figure 8 Debugging the software views: external debug and trace

Hardware View

External debug and trace of the *Hardware View* means an external debugger being able to debug and trace all software in both Secure and Non-secure states.

Virtualizer View

External debug and trace of the *Virtualizer View* means an external debugger being able to debug and trace all software in Non-secure state only, with no visibility of Secure state.

Single Machine, Single Application, Secure Machine and Secure Application Views

The ARM architecture fundamentally does not support restricting external debug and trace to the Single Machine, Single Application, Secure Machine and Secure Application Views.

See also *Configuring debug and trace by an external debugger* on page 23.

Disabled

The external debugger has no visibility over of the system.

3.1 Platform support for external debug and trace

In addition to implementing external debug interfaces as shown in Figure 7 on page 10, in order to support different views of external debug and trace, the platform must support different configurations of the authentication interface as described in the following sections.

3.1.1 Hardware View

Table 4 shows the authentication signal settings for enabling external debug and trace *Hardware View* in both ARMv7 and ARMv8.

Table 4 Enabling external debug and trace, Hardware View, ARMv7 and ARMv8

	External trace	External debug and trace
NIDEN	HIGH	HIGH
SPNIDEN	HIGH	HIGH
DBGEN	-	HIGH
SPIDEN	-	HIGH
DEVICEEN	HIGH	HIGH

3.1.2 Virtualizer View

Table 5 shows the authentication signal settings for enabling external debug and trace *Virtualizer View* in both ARMv7 and ARMv8. These settings also disable the *Secure Machine View*.

Table 5 Enabling external debug and trace, Virtualizer View, ARMv7 and ARMv8

	External trace	External debug and trace
NIDEN	HIGH	HIGH
SPNIDEN	LOW	LOW
DBGEN	-	HIGH
SPIDEN	LOW	LOW
DEVICEEN	HIGH	HIGH

3.1.3 Disabled

To completely disable all debugging of the Hardware View, **DBGEN** and **NIDEN** can be tied LOW. For ARMv7, this is not recommended, as it also disables self-hosted debug views, and **DEVICEEN** should be used instead. In ARMv8, **DBGEN** does not affect self-hosted debug, but **NIDEN** does control self-hosted trace.

Table 6 Disabling external debug and trace, ARMv7 (with self-hosted *Secure Machine View* disabled)

	External trace	External debug and trace
NIDEN	-	-
SPNIDEN	-	-
DBGEN	-	-
SPIDEN	-	-
DEVICEEN	LOW	LOW

Driving **DEVICEEN** LOW disables all external debug and trace access. Without external access, an external debugger cannot program the device, meaning **DBGEN**, **NIDEN**, **SPIDEN** and **SPNIDEN** can be driven to allow self-hosted debugging.

Using **DEVICEEN** to remove external debug access to the device does not prevent the processor from entering Debug state if **DBGEN** is HIGH. For example, a `BKPT` instruction executed with DBGDSCR.HDBGen=1 will cause entry to Debug state. DBGDSCR.HDBGen can be programmed when the OS Lock is locked. Hypervisors can use HDCR.TDOSA to prevent guests accessing the OS Lock.

In Debug state, a would-be attacker has a full Virtualizer View (if **SPIDEN** is LOW) or Hardware View (if **SPIDEN** is HIGH), if it has access to the debug registers on the Debug bus. For this reason, even when self-hosted trace is being used, it is essential that software controls access to the Debug bus; for example, by an MMU, System MMU, or *Address Space Controller* (ASC). (See also the comment on programming the ETM or PTM above.)

Without such controls, Debug state is a potential *denial of service* attack.

Table 7 Disabling external debug and trace, ARMv8 (with self-hosted *Secure Machine View* disabled)

	External trace	**External debug**
NIDEN	-	-
SPNIDEN	-	-
DBGEN	-	LOW
SPIDEN	-	LOW
DEVICEEN	LOW	LOW (see text)

As ARMv8 self-hosted debug does not rely on **DBGEN** and **SPIDEN**, these can be driven LOW to disable external debugging. If **DEVICEEN** is HIGH, the external debugger can still access the debug registers via the debug APB. A secure monitor can use MDCR_EL3.EDAD to prevent access whilst in Secure state. Thus in an ARMv8 system, external debug can be disabled without disabling external trace.

To disable both external debug and external trace, **DEVICEEN** should be tied LOW.

Driving **DEVICEEN** LOW disables all external debug and trace access. Without external access, an external debugger cannot program the device, meaning **DBGEN**, **NIDEN**, **SPIDEN** and **SPNIDEN** can be driven to allow self-hosted or on-chip debugging. ARM recommends that if **DEVICEEN** is LOW, **DBGEN** and **SPIDEN** should be driven LOW to any ARMv8 processors.

In a heterogeneous system comprising a mix of ARMv7 and ARMv8 processors, there is a single **DEVICEEN** and so external trace cannot be enabled when external is disabled.

3.1.4 Platform support for external debug and trace over power-down

The processor loses state if the processor Core power domain is powered down. To aid external debug and trace, ARM recommends that the processor implements separate Core and Debug power domains and that the platform supports powering down the Core power domain whilst leaving the Debug power domain powered up when an external debugger is connected.

Depending on the processor implementation, the system power controller should also support emulation of power-down based on the **DBGCOREPURQ**, **DBGNOPWRDWN** and **ETMNOPWRDWN** signals.

Emulating power-down states

How power-down is emulated is IMPLEMENTATION DEFINED. Depending on the approach taken, the ability of the debugger to access the state of the processor and the system may be limited.

In emulated power-down state, the debugger must be able to access all debug registers in both the Debug and Core power domains as if the Core power domain is on. That is, read and write such registers without receiving errors. This allows an external debugger to debug the power-up sequence.

Otherwise, the behavior of the processor in emulated power-down must be similar to that in a real power-down state. In particular, the processor must not respond to other system stimuli such as interrupts.

Two example approaches for emulating power-down are:

- The processor is held in a standby state isolated from system stimuli. It is IMPLEMENTATION DEFINED whether the processor can respond to debug stimuli such as an External Debug Request debug event.
- The processor held in Warm reset. This limits the ability of an external debugger to access the processor's resources. For example, the processor cannot be put into Debug state.

On exit from emulated power-down the processor is reset, but the debug registers that are only reset by a Cold reset must not be reset. Typically this means that a Warm reset is substituted for the Cold reset.

It must also be noted that:

- For an ARMv8 processor, Warm reset and Cold reset have different effects outside of resetting debug registers. In particular, the RMR_ELx register is reset by Cold reset and controls the reset state on Warm reset. This means that if Cold reset is substituted by a Warm reset, the behavior of the reset code may be different.
- The timing effects of power down, and voltage and clock stabilization on power-up, are typically not factored in the power-down emulation.

Emulation does not model state lost during power down, meaning it may mask errors in the state storage and recovery routines.

3.2 Software considerations for external debug and trace

3.2.1 Preventing conflict between self-hosted debug and an external debugger

ARM recommends that the CLAIM tag registers are used to claim coarse-grained ownership of shared debug resources. For a given debug component:

- An external debugger should set CLAIM[0] to indicate it owns the resource. Self-hosted debug software must check that CLAIM[0] is clear before using any shared resource.
- Self-hosted debug software must set CLAIM[1] to indicate it owns the resource. When an external debugger connects, it must check that CLAIM[1] is clear before using any shared resource.

Note: The CLAIM register is accessed using the CLAIMSET and CLAIMCLR registers. This allows for atomic updates to individual bits in the CLAIM register. To avoid a race, an agent must set its own bit before checking the other agent's bit.

For the processor debug component, CLAIM means the DBGCLAIM registers. For a processor trace macrocell, CLAIM means the ETMCLAIM registers.

For debug components without CLAIM tags, the DBGCLAIM tags of the associated processor should be used:

- for the PMU registers of a processor use DBGCLAIM[3:2]
- for the CTI registers of a processor use DBGCLAIM[5:4].

Other system-level debug components, such as the trace fabric and system-level performance monitors, should provide similar CLAIM registers and interfaces.

The means for an external debugger to request that self-hosted debug relinquish control of a resource is IMPLEMENTATION DEFINED. One possible implementation is to provide an API such as pseudo-filesystem that the user can write to from a console. Similarly, the means for an external debugger to signal that it has relinquished control is IMPLEMENTATION DEFINED, although in many scenarios this will be resetting the device.

The platform might also provide an API to allow finer-grained sharing of physical resources, such as breakpoints, watchpoints and performance monitor counters, between a self-hosted component and an external component.

See also *Target software support for external debug and trace over power-down* below.

3.2.2 Configuring debug and trace by an external debugger

As described in *Platform support for external debug and trace* on page 20, the platform only provides the Hardware and Virtualizer Views to an external debugger. However, an external debugger can use features of the debug and trace architectures to restrict debug operations to the Single Machine, Single Application, Secure Machine and Secure Application Views. See:

- *Configuring breakpoints and watchpoints for different views* on page 23
- *Configuring trace for different views* on page 23.

Configuring breakpoints and watchpoints for different views

The ARMv7 and ARMv8 breakpoints and watchpoints can be configured to only match within a certain software view. In order to prevent unwanted breakpoint and watchpoint matches, a debugger should limit its programming according to the view it requires.

This is controlled by the HMC and SSC fields of a breakpoint or watchpoint control register.

Table 8 Breakpoint and watchpoint view encodings

HMC	SSC	Non-secure view	Secure view	Notes
0	0b00	Single Machine View	Secure Machine View	-
0	0b01	Single Machine View	None	-
0	0b10	None	Secure Machine View	-
1	0bX1	Virtualizer View	Not applicable	-
1	0b00	Hardware View	Hardware View	-
1	0b10	None	Hardware View	ARMv8 only

Within each view the PMC or PAC field can be used to specify the modes to match in. Generally breakpoints and watchpoints should be configured to match only within a single translation regime.

Breakpoints and watchpoints can also be linked to CONTEXTIDR and VTTBR.VMID matching breakpoints to further restrict matching to a specific application or virtual machine.

For more information see:

- *Communicating CONTEXTIDR and VTTBR.VMID to a debugger* on page 25
- *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition*
- *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile*.

Configuring trace for different views

Trace can be configured to only trace specific exeception levels within a particular security state:

- In ETMv3 and PFTv1, using address range comparators
- In ETMv4, using either ViewInst exception level filtering or address range comparators.

- Address range comparators can also linked to CONTEXTIDR and VTTBR.VMID comparators to further restrict matching to a specific application or virtual machine.

See also:

- *Communicating CONTEXTIDR and VTTBR.VMID to a debugger* on page 25
- *Embedded Trace Macrocell Architecture Specification ETMv1.0 to ETMv3.5*
- *Embedded Trace Macrocell Architecture Specification ETMv4*
- *Program Flow Trace Architecture Specification*.

3.2.3 Target awareness in the external debugger

An external debugger does require some knowledge of the higher levels of software, and that software must co-operate with the tool:

- When in Debug state:
  - Accesses generated by instructions issued to the processor by the debugger can generate permission faults, for example:
    - Because of a stage 1 translation controlled by the operating system (Application View).
    - Because of a stage 2 translation controlled by the hypervisor (Machine View).
  - In ARMv8 accesses to system registers issued to the processor by the debugger can be trapped, if access is trapped by CPACR_EL1, CPTR_EL2 or CPTR_EL2.
  - In ARMv7:
    - Accesses to coprocessor registers issued to the processor by the debugger can generate Undefined Instruction exceptions, if access is not allowed by CPACR or NSACR.
    - However, Hyp Trap exceptions are ignored and the debugger must filter out such accesses to avoid returning stale data to the user.
- If the debugger is making use of CONTEXTIDR and/or VTTBR.VMID values to filter debug events and/or trace, then the operating system must set CONTEXTIDR and/or the hypervisor must set VTTBR.VMID to a unique value for each Application View and/or Machine View. See *Communicating CONTEXTIDR and VTTBR.VMID to a debugger* on page 25.

Emulating faulted memory and coprocessor accesses

If an access is faulted, or the debugger determines that an access is subject to a Hyp Trap, then this is because the operating system and/or hypervisor has to emulate the access. Example use cases include:

- Lazy context switching by an operating system and/or hypervisor. Registers are only swapped in on first use. If the debugger is the first user, then the values in the actual registers are stale.
- On demand paging of memory from a swap device by an operating system and/or hypervisor.
- On demand paging of a file from the file system by an operating system.
- Memory-mapped pseudo-files mapped into the address space using mmap().
- Virtualization of hardware such as an interrupt controller, GPU, NIC, UART, etc.

In some cases, the debugger can emulate the access itself. However, in other cases, the debugger must either:

- Report the fault to the user, and not provide the information requested.
- Emulate the access by getting the operating system and/or hypervisor to execute its emulation code before returning to the Application or Machine View being debugged.

Emulation of the access requires the debugger to have a detailed understanding of the operating system and/or hypervisor platform.

See also *Software considerations for external debug and trace* on page 22.

Communicating CONTEXTIDR and VTTBR.VMID to a debugger

ARM recommends that the debugger is able to set a CONTEXTIDR value for each Application View being debugged. For example, the operating system can provide a pseudo-file containing the CONTEXTIDR within the target file system and either:

- Set this to a unique value for each Application View
- Allow the debugger to write a value to this file for each Application View.

The operating system writes this value to CONTEXTIDR for each context switch. The debugger either reads or writes the value when it connects to a running process.

If the AArch32 Short-descriptor translation table format is being used, then CONTEXTIDR[7:0] contains an *Address Space Identifier* (ASID) which the operating system needs to make unique for each process. In order to do this, ASID values may be recycled, meaning this portion of the CONTEXTIDR is not guaranteed to be unique during the process lifetime.

If the CONTEXTIDR is used for filtering trace then the ETM or PTM can be configured to ignore the ASID portion of the CONTEXTIDR. The remaining 24 bits of CONTEXTIDR can be a unique value for the Application View.

However, if the CONTEXTIDR is being used to filter breakpoints and watchpoints, then it is IMPLEMENTATION DEFINED whether the context-matching breakpoint can be configured to ignore the ASID portion of the CONTEXTIDR.

Note: Cortex-A processors do not provide this capability.

If the processor does not provide this capability then a mechanism is required to keep the debugger's CONTEXTIDR and the operating system's ASID value aligned, such as:

- The operating system notifies the debugger that ASID is being changed for a process.
- The operating system provides a means to lock the ASID of a process that is being debugged.
- The Long-descriptor translation table format is used.

This area requires further investigation and standardization between operating systems and debuggers.

Alternatives to using CONTEXTIDR and VTTBR.VMID

An alternative to using CONTEXTIDR and VTTBR.VMID is to use support for self-hosted debug and trace (see *Standard Usage Models for Self-hosted Debug and Trace* on page 30) to context switch debug and trace registers on behalf of the debugger.

One such approach is for the external debugger to write directly to the debug and trace registers when the process being debugged is in context. However, this relies on the operating system to actively switch the values in the debug and trace registers. In practice an operating system reconstructs these values from its own data structures each time it switches to a process being debugged.

Therefore the external debugger instead cooperates with the operating system by using its normal APIs to configure debug and trace for the process being debugger, for example through a daemon process executing on the target.

See *Target software support for external debug and trace in multiprocessor systems, including big.LITTLE* on page 28.

3.2.4 Target software support for external debug and trace over power-down

The processor loses state if the processor Core power domain is powered down. To aid external debug and trace, ARM recommends that the processor implements separate Core and Debug power domains, and that software supports external debug and trace over

power-down. The Debug power domain of each processor should contain the CTI connected to that processor.

In a multiprocessor system the Debug power domains for each processor are recommended to be part of a system-wide Debug power domain. The Core power domains of each processor might be independent.

The DAP contains control bits that request the Debug power domain is powered. See *ARM Debug Interface Architecture Specification ADIv5.0 to ADIv5.2* for details.

If this is not possible, the debugger can request emulation of power-down by setting control bits in debug power control registers. These control bits require support from the system power controller for emulating power-down states. These control bits are:

- DBGPRCR.COREPURQ (ARMv7, v7.1 Debug) or EDPRCR.COREPURQ (ARMv8). Setting this bit requests that the Core power domain is powered up, and remains powered up (emulating power down). On power-on reset, DBGPRCR.CORENPDRQ is initialized to the value of COREPURQ. COREPURQ can be accessed when the Core power domain is powered-down, if the processor implements separate Core and Debug power domains.
- DBGPRCR.CORENPDRQ (ARMv7 and ARMv8). Setting this bit requests that the Core power domain remains powered up (emulating power down). In v7 Debug, setting CORENPDRQ also requests that the Core power domain is powered up, and can be accessed when the Core power domain is powered-down, if the processor implements separate Core and Debug power domains. In v7.1 Debug and ARMv8, CORENPDRQ cannot be accessed when the Core power domain is powered down or the OS Lock is locked.
- TRCPDCR.NPDRQ. Setting this bit requests that the power domain containing the trace macrocell remains powered up (emulating power down). If this bit is set to 1 when the trace macrocell is powered down, this requests that it is powered up. This allows the trace macrocell to be powered up over a power down sequence.

Full details can be found in the appropriate *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition* and trace architecture manual:

- *Embedded Trace Macrocell Architecture Specification ETMv1.0 to ETMv3.5*
- *Embedded Trace Macrocell Architecture Specification ETMv4*
- *Program Flow Trace Architecture Specification*.

Target software must also ensure that other power domains are powered-up when used by debug. For example, if the trace fabric and sink(s) require specific power domains outside the processor to be powered.

Use of CLAIM tags to negotiate power-down

ARM further recommends that debuggers and power-down software use the CLAIM tags to indicate when an external debugger is connected and so Core power domain debug logic should be saved over power-down. Table 9 shows the recommended usage for debug.

Table 9 Recommended usage model for DBGCLAIM tags

Bit	Usage
[0]	Debug in use by external debugger. Indicates that the external debugger is using the debug registers and: - Expects the operating system to save/restore them over a power-down. - Expects them to not be overwritten by self-hosted debug software.
[1]	Debug in use by self-hosted software. Indicates that the self-hosted debugger is using the debug registers and expects them not to be overwritten by an external debugger. This can also be used to communicate between software layers for power-down control.
[2]	PMU in use by external debugger. See bit [0].
[3]	PMU in use by self-hosted software. See bit [1].

Bit	Usage
[4]	Reserved to indicate CTI in use by external debugger. The CTI should be in the Debug power domain and so software does not need to support save/restore of the CTI.
[5]	Reserved to indicate CTI in use by self-hosted software.
[6]	Acknowledge from operating system for bit [0] (only required for compatibility with v7 Debug)
[7]	Acknowledge from operating system for bit [2] (only required for compatibility with v7 Debug)

For implementations of v7 Debug only, the external debugger can write to DBGCLAIM when the core is powered down, meaning it should either not clear DBGPRCR.CORENPDRQ or not program any volatile state until the power-down software acknowledges that it will save/restore the state by setting DBGCLAIM[6] to 1.

This is not required for v7.1 Debug or ARMv8 because these versions of the architecture do not allow writes to DBGCLAIM when the core is powered down.

Similarly, Table 10 shows the recommended usage for the trace macrocell.

Table 10 Recommended usage model for ETMCLAIM tags

Bit	Usage
[0]	Trace in use by external debugger. Indicates that the external debugger is using the trace registers and: - Expects the operating system to save/restore them over a power-down. - Expects them to not be overwritten by self-hosted trace software.
[1]	Trace in use by self-hosted debugger. Indicates that the self-hosted debugger is using the trace registers and expects them not to be overwritten by an external debugger. This can also be used to communicate between software layers for power-down control.
[5:2]	Reserved.
[6]	Acknowledge from operating system for bit [0] (for compatibility with ETMv3.4 or PFTv1.0)
[7]	Reserved.

If neither self-hosted nor the external debugger is using the shared resources then the operating system should not save/restore them. This allows for the case where the debugger has requested emulation of power-down. Otherwise, the restore code could overwrite state that has been updated whilst in emulated power-down.

Using OS Unlock Catch to delay programming

If the debugger attempts to access a debug register when the processor Core power domain is completely off, or in a low-power state where the Core power domain registers cannot be accessed, and that access returns an error, it must retry the access. However, if the Core power domain is regularly put into such a state, this can lead to unreliable debugger behavior.

In a multiprocessor system utilizing dynamic power control it might be rare for all the processors to be powered-up simultaneously, which is necessary for duplicating programming across all processors.

A debugger can request emulation of power down in this case, as described in *Target software support for external debug and trace over power-down* above.

Alternatively, the debugger can program the registers when the processor is halted in Debug state, as this means the processor is powered-up, and:

- To program the registers, the debugger halts the processor:
  - On an ARMv7 processor by writing 1 to DBGDRCR.HRQ.
  - On an ARMv8 processor or an ARMv7 processor with CTI, using the External Debug Request from the CTI.
- If the processor implements separate Core and Debug power domains, this can be done even when the processor is powered down: if the processor is powered down, it will halt when the processor powers up.

Note: The processor will reset into Secure state. If halting is not allowed in Secure state then the processor will not halt until the Secure monitor returns to the hypervisor or guest operating system in Non-secure state.

- When the processor halts, if the OS Lock is locked and software on the processor implements save/restore of the debug registers then this indicates the processor was either about to power down or has just powered up.

  Note: If the save/restore sequence executes in Secure state and halting is not allowed in Secure state then the processor will not halt during the save/restore sequences.

  Any values written at this point by the debugger would be overwritten by the restore sequence. Hence the debugger should enable the OS Unlock Catch debug event to halt the processor again after the restore sequence has completed. The registers can then be programmed and the OS Unlock Catch debug event disabled.

This method allows the debugger to avoid using emulation of power down.

Once all processors have been programmed in this way, the target software can be left to save/restore the debug registers over power down.

3.2.5 Target software support for external debug and trace in multiprocessor systems, including big.LITTLE

For a description of multiprocessor systems, including big.LITTLE, see *Power management and multiprocessor systems*, *including big.LITTLE* on page 18.

Power control in MP systems is an orthogonal issue. See *Target software support for external debug and trace over power-down* above.

For external debug of an MP system (big.LITTLE or otherwise), there are a number of possible strategies, including:

- *Pinning a process or guest*
- *Migrating programming*
- *Duplicating programming*.

These are described in more detail below. Other aspects of supporting MP systems are the same as for self-hosted debug. See *Software support for self-hosted debug and trace in multiprocessor systems, including big.LITTLE* on page 46.

Pinning a process or guest

The process or guest is pinned a particular processor whilst debugging. That is, it executes only that processor, meaning the debugger can treat the system as a uniprocessor.

The operating system or hypervisor controlling the process being debugged offers an IMPLEMENTATION DEFINED interface to the debugger to allow it to pin the process.

However, such an interface may be subject to abuse by applications, for example in a heterogeneous multiprocessor to pin a process to a higher performance processor.

Migrating programming

The system software includes migration of the debug and trace programming between processors.

If an operating system or hypervisor supports self-hosted debug, then this is an extension of the context switching for self-hosted debug. To fully support external debug, the context switch must use the architecture's support for debug over power-down, and, in particular, the OS Lock function to ensure that the registers are not changed by an external debugger during a context switch. The external debugger can request the context is switched using the same mechanisms as for supporting power-down described above.

For a big.LITTLE system, this is an extension of the OSPM support for powering-down one processor and migrating the state to another.

Duplicating programming

The external debugger has to program the same debug and trace settings across all processors in the system where the process being debugged might execute. To be able to do so, either all processors must be powered on when the settings are programmed, or the programming of powered off processors delayed until such time as they are powered on.

To fully support external debug, the OSPM must save and restore the debug and trace settings for processors as it powers them off and on, using the architecture's support for debug over power-down, and, in particular, the OS Lock function to ensure that the registers are not changed by an external debugger during a save/restore. The external debugger can request the context is saved/restored using the mechanisms described above.

When using this technique, the OSPM does not migrate the state between processors.

This means that the *Duplicating programming* and *Migrating programming* strategies are fundamentally incompatible. The external debugger must understand the strategy used by the OSPM. ARM recommends use of the *Duplicating programming* strategy for external debugging.

4 Standard Usage Models for Self-hosted Debug and Trace

Self-hosted debug and self-hosted trace support a fuller range of debug views, through software controls, as shown in Figure 9.

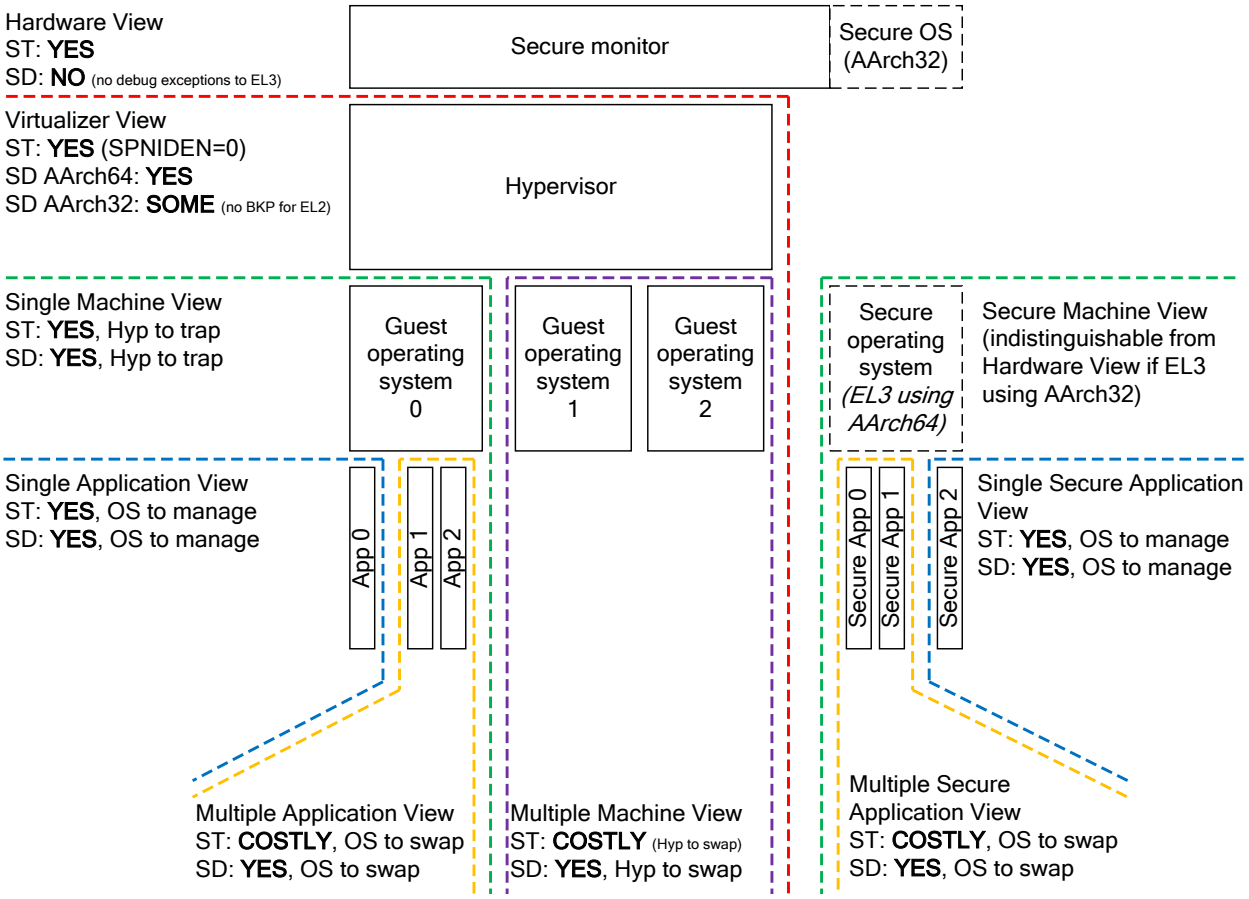


Figure 9: Debugging the software views: self-hosted debug and trace

Hardware View

Self-hosted debug and trace of the *Hardware View* means a Monitor supporting a debug monitor that allows debug of itself and both Secure and Non-secure states, and the Monitor ensuring that Non-secure state has no visibility of Secure state using debug.

Virtualizer View

Self-hosted debug and trace of the *Virtualizer View* means a Hypervisor supporting a debug monitor that allows debug of itself and all guests underneath it, and the Monitor ensuring that Non-secure state has no visibility of Secure state using debug.

Single Machine View

Self-hosted debug and trace of the *Single Machine View* means a (Non-secure) operating system supporting a debug monitor that allows debug of itself and all applications below it, and the Hypervisor ensuring that the single operating system does not have visibility of other guests. Only one operating system at a time has this capability.

Multiple Machine View

Self-hosted debug and trace of the *Multiple Machine View* means a (Non-secure) Hypervisor supporting multiple guest operating systems each of which enables debug and trace of *Single Machine View*, or lower.

Secure Machine View

Self-hosted debug and trace of the *Secure Machine View* means a monitor supporting a Secure operating system which enables debug and trace of *Single Machine View*, or lower, possibly at the same time as supporting an independent *Virtualizer View* or any other independent Non-secure debug or trace view.

Note: An *independent* view means that the system is actively supporting concurrent usage of both the Secure Machine View and the other view. For example, by allowing both a Secure operating system to trace the Secure Machine View and a Hypervisor to trace the Virtualizer View, with independent operation.

Single Application View

Self-hosted debug and trace of the *Single Application View* means a (Non-secure) operating system supporting a debug monitor that allows debug of one application running below it, typically by a separate debug application, whilst ensuring that applications do not have visibility of other applications.

Multiple Application View

Self-hosted debug and trace of the *Multiple Application View* means a (Non-secure) operating system supporting a debug monitor that allows debug of multiple application running below it, typically by one or more debug applications, whilst ensuring that applications do not have visibility of other applications.

Single Secure Application and Multiple Secure Application Views

These views are the same as the *Single Application View* and *Multiple Application View*, only in Secure state.

Disabled

If a layer of software does not want to restrict debug to its own view, it should be configured to enable (some) lower layer views. ARM recommends that:

- If the *Hardware View* is not being used, the Secure monitor should enable the *Virtualizer View*, and optionally the *Secure Machine View*. Supporting the *Secure Machine View* implies a cost on each security state switch.
- If the *Virtualizer View* is not being used, the hypervisor should enable the *Single Machine View* or *Multiple Machine View*
- If the *Single Machine View* is not being used, the operating system should enable the *Multiple Application View*.

In a closed environment, such as a consumer tablet or smart-phone, with restricted access for loading and developing applications, an operating system can choose to disable all forms of debug.

4.1 Platform support for self-hosted debug and trace

In order to support different views of self-hosted debug and trace, the platform must support different configurations of the authentication interface as described in the following sections.

4.1.1 Hardware View

The authentication signal settings and Secure monitor responsibilities for supporting the self-hosted debug and trace Hardware View are shown in:

- Table 11 for ARMv7
- Table 12 for ARMv8.

Table 11 Enabling self-hosted debug and trace, Hardware View, ARMv7

	Self-hosted trace	Self-hosted debug
NIDEN	HIGH	-
SPNIDEN	HIGH	-
DBGEN	-	HIGH
SPIDEN	-	HIGH

Table 12 Enabling self-hosted debug and trace, Hardware View, ARMv8

	Self-hosted trace	Self-hosted debug
NIDEN	HIGH	-
SPNIDEN	HIGH	-
DBGEN	-	-
SPIDEN	-	-

4.1.2 Virtualizer View

The authentication signal settings, and Hypervisor and Secure monitor responsibilities for supporting the self-hosted debug and trace Virtualizer View are shown in:

- Table 13 for ARMv7
- Table 14 for ARMv8.

Table 13 Enabling self-hosted debug and trace, Virtualizer View, ARMv7

	Self-hosted trace	Self-hosted debug
NIDEN	HIGH	HIGH
SPNIDEN	LOW	LOW
DBGEN	-	HIGH
SPIDEN	LOW	LOW

Table 14 Enabling self-hosted debug and trace, Virtualizer View, ARMv8

	Self-hosted trace	Self-hosted debug
NIDEN	HIGH	-
SPNIDEN	LOW	-
DBGEN	-	-
SPIDEN	LOW	-

4.1.3 Single Machine View

The authentication signal settings, and operating system and Hypervisor responsibilities for supporting the self-hosted debug and trace Single Machine View are shown in:

- Table 15 for ARMv7
- Table 16 for ARMv8.

Table 15 Enabling self-hosted debug and trace, Single Machine View, ARMv7

	Self-hosted trace	Self-hosted debug
NIDEN	HIGH	HIGH

	Self-hosted trace	Self-hosted debug
SPNIDEN	LOW	LOW
DBGEN	-	HIGH
SPIDEN	LOW	LOW

Table 16 Enabling self-hosted debug and trace, Single Machine View, ARMv8

	Self-hosted trace	Self-hosted debug
NIDEN	HIGH	-
SPNIDEN	LOW	-
DBGEN	-	-
SPIDEN	LOW	-

4.1.4 Multiple Machine View

The authentication signal settings and Hypervisor responsibilities for supporting the self-hosted debug and trace Multiple Machine View are shown in:

- Table 17 for ARMv7
- Table 18 for ARMv8.

Table 17 Enabling self-hosted debug and trace, Multiple Machine View, ARMv7

	Self-hosted trace	Self-hosted debug
NIDEN	HIGH	HIGH
SPNIDEN	LOW	LOW
DBGEN	-	HIGH
SPIDEN	LOW	LOW

Table 18 Enabling self-hosted debug and trace, Multiple Machine View, ARMv8

	Self-hosted Trace	Self-hosted Debug
NIDEN	HIGH	-
SPNIDEN	LOW	-
DBGEN	-	-
SPIDEN	LOW	-

4.1.5 Secure Machine View

The authentication signal settings and Secure monitor responsibilities for supporting the self-hosted debug and trace Secure Machine View are shown in:

- Table 19 for ARMv7
- Table 20 for ARMv8.

Table 19 Enabling self-hosted debug and trace, Secure Machine View, ARMv7

	Self-hosted trace	Self-hosted debug
NIDEN	HIGH	HIGH
SPNIDEN	HIGH (see text)	HIGH (see text)
DBGEN	-	HIGH

	Self-hosted trace	Self-hosted debug
SPIDEN	-	HIGH (see text)

Table 20 Enabling self-hosted debug and trace, Secure Machine View, ARMv8

	Self-hosted trace	Self-hosted debug
NIDEN	HIGH	-
SPNIDEN	HIGH (see text)	-
DBGEN	-	-
SPIDEN	-	-

4.1.6 Single Application View

The authentication signal settings and operating system responsibilities for supporting the self-hosted debug and trace *Single Application View* are shown in:

- Table 21 for ARMv7
- Table 22 for ARMv8.

Table 21 Enabling self-hosted debug and trace, Single Application View, ARMv7

	Self-hosted trace	Self-hosted debug
NIDEN	HIGH	HIGH
SPNIDEN	LOW	LOW
DBGEN	-	HIGH
SPIDEN	LOW	LOW

Table 22 Enabling self-hosted debug and trace, Single Application View, ARMv8

	Self-hosted trace	Self-hosted debug
NIDEN	HIGH	-
SPNIDEN	LOW	-
DBGEN	-	-
SPIDEN	LOW	-

4.1.7 Multiple Application View

The authentication signal settings and operating system responsibilities for supporting the self-hosted debug and trace Multiple Application View are shown in:

- Table 23 for ARMv7
- Table 24 for ARMv8.

Table 23 Enabling self-hosted debug and trace, Multiple Application View, ARMv7

	Self-hosted trace	Self-hosted debug
NIDEN	HIGH	HIGH
SPNIDEN	LOW	LOW
DBGEN	-	HIGH
SPIDEN	LOW	LOW

Table 24 Enabling self-hosted debug and trace, Multiple Application View, ARMv8

	Self-hosted trace	Self-hosted debug
NIDEN	HIGH	-
SPNIDEN	LOW	-
DBGEN	-	-
SPIDEN	LOW	-

4.1.8 Disabled

For devices in very closed environments, such as dedicated secure processors, self-hosted debug can be disabled at higher levels.

For ARMv7, tying **DBGEN** and **NIDEN** LOW disables all external and self-hosted debug and trace.

For ARMv8, software must actively not enable self-hosted debug.

4.2 Software support for self-hosted debug and trace

4.2.1 Hardware View

AArch32 Secure monitor support for self-hosted debug and trace, Hardware View

It is not possible use self-hosted features to debug the Hardware View because there is no way to route debug exceptions from Non-secure state to Secure state.

This could be achieved using proxies at the lower levels to combine Virtualizer View and Secure Machine View debug. AArch32 does not support self-hosted debug of Hyp mode (see above).

To support self-hosted trace of the Hardware View, an AArch32 Secure monitor must:

- Configure the ETM and trace fabric, including any funnels and trace sinks.
- If CP14 access to the ETM or PTM is implemented, set NSACR.TTA to 1 to prevent Non-secure access to the ETM or PTM system registers.
- Enable the ETM or PTM.

AArch64 Secure monitor support for self-hosted debug and trace, Hardware View

It is not possible use self-hosted features to fully debug the *Hardware View* because there is no way to route debug exceptions to EL3 from lower exception levels.

This could be achieved using proxies at the lower levels to combine *Virtualizer View* and *Secure Machine View* debug. EL3 using AArch64 and EL2 using AArch32 do not support self-hosted debug (see above).

If Secure software is making use of the debug features, an AArch64 Secure monitor must:

- Set MDCR_EL3.EDAD to 1 to disable external debug access to the debug registers.
- Set MDCR_EL3.TDA to 1 to disable Non-secure access to the debug registers.

To support self-hosted trace of the Hardware View, an AArch64 Secure monitor must:

- Configure the ETM and trace fabric, including any funnels and trace sinks.
- If system register access to the ETM is implemented, set CPTR_EL3.TTA to 1 to prevent Non-secure access to the ETM system registers.
- Enable the ETM.

4.2.2 Virtualizer View

AArch32 Hypervisor support for self-hosted debug and trace, Virtualizer View

To enable self-hosted debug of the Virtualizer View, an AArch32 hypervisor must:

- Set DBGDSCRext.MDBGen to 1, to enable Monitor debug-mode.
- Set HDCR.TDE to 1, to route debug events to the hypervisor.
- Set HDCR.{TDA,TDRA,TDOSA} to 1, to trap any access to the debug registers from the guest to the hypervisor.
- Configure the stage 2 MMU translation to prevent any guest access to the debug registers through the debug APB.

In AArch32 state, debug exceptions not generated by debug hardware in Hyp mode, so only `BKPT` instructions can be used to debug the hypervisor itself.

To enable self-hosted trace of the Virtualizer View, an AArch32 hypervisor must:

- Configure the ETM and trace fabric, including any funnels and trace sinks.
- If CP14 access to the ETM or PTM is implemented, set HCPTR.TTA to 1 to prevent guest access to the ETM or PTM system registers.
- Configure the stage 2 MMU translation to prevent any guest access to the trace registers through the debug APB.
- Enable the ETM or PTM.

AArch64 Hypervisor support for self-hosted debug and trace, Virtualizer View

To enable self-hosted debug of the Virtualizer View, an AArch64 hypervisor must:

- Set MDSCR_EL1.MDE to 1, to enable Monitor debug-mode for hardware breakpoints and watchpoints.
- Set MDCR_EL2.TDE to 1, to route debug events to the hypervisor.
- Set MDCR_EL2.KDE to 1, if required, to enable debug exceptions from within the hypervisor.
- Set MDCR_EL2.{TDA,TDRA,TDOSA} to 1, to trap any access to the debug registers from the guest to the hypervisor.
- Configure the stage 2 MMU translation to prevent any guest access to the debug registers through the debug APB.

To enable self-hosted trace of the Virtualizer View, an AArch64 hypervisor must:

- Configure the ETM and trace fabric, including any funnels and trace sinks.
- If system register access to the ETM is implemented, set CPTR_EL2.TTA to 1 to prevent guest access to the ETM system registers.
- Configure the stage 2 MMU translation to prevent any guest access to the trace registers through the debug APB.
- Enable the ETM.

AArch32 Secure monitor support for self-hosted debug, Virtualizer View

To prevent self-hosted debug of the Virtualizer View affecting Secure state operation, an AArch32 Secure monitor must:

- If the processor is ARMv8, set SDCR.SPD to 0b10 to disable self-hosted debug in Secure privileged modes.
- Set SDER.SUIDEN to 0 to disable self-hosted debug in Secure User mode.

AArch64 Secure monitor support for self-hosted debug, Virtualizer View

To prevent self-hosted debug of the Virtualizer View affecting Secure state operation, an AArch64 Secure monitor must:

- Set MDCR_EL3.SDD to 1 to disable self-hosted debug in AArch64 Secure state.
- If Secure state uses AArch32:

- Set MDCR_EL3.SPD32 to 0b10 to disable self-hosted debug in AArch64 Secure privileged modes.
- Set SDER32_EL3.SUIDEN to 0 to disable self-hosted debug in Secure User mode.

4.2.3 Single Machine View

AArch32 operating system support for self-hosted debug and trace, Single Machine View

To enable self-hosted debug of the Single Machine View, an AArch64 operating system must:

- Set DBGDSCRext.MDBGen to 1, to enable Monitor debug-mode.
- Configure the MMU translation to prevent any application access to the debug registers through the debug APB.

To enable self-hosted trace of the Virtualizer View, an AArch64 operating system must:

- Configure the ETM and trace fabric, including any funnels and trace sinks.
- Configure the MMU translation to prevent any application access to the trace registers through the debug APB.
- Enable the ETM.

AArch64 operating system support for self-hosted debug and trace, Single Machine View

To enable self-hosted debug of the Single Machine View, an AArch64 operating system must:

- Set MDSCR_EL1.MDE to 1, to enable Monitor debug-mode for hardware breakpoints and watchpoints.
- Set MDSCR_EL1.KDE to 1, if required, to enable debug exceptions from within the operating system.
- Manage PSTATE.D at exception entry and return.
- Configure the MMU translation to prevent any application access to the debug registers through the debug APB.

To enable self-hosted trace of the Single Machine View, an AArch64 operating system must:

- Configure the ETM and trace fabric, including any funnels and trace sinks.
- Configure the MMU translation to prevent any application access to the trace registers through the debug APB.
- Enable the ETM.

AArch32 hypervisor support for self-hosted debug and trace, Single Machine View

To support self-hosted debug of the Single Machine View by an operating system, an AArch32 hypervisor must:

- Discover which operating system is using self-hosted debug. It can do this by setting HDCR.TDA to 1 in order to trap the first access to the debug system registers.
- On switching from the operating system using self-hosted debug:
  - Swap out the value of DBGDSCRext, and set DBGDSCRext.MDBGen to 0 for the new operating system.
  - Set HDCR.TDA to 1 to prevent the new operating system accessing debug system registers.
- On switching in the operating system using self-hosted debug:
  - Restore the value of DBGDSCRext.

    - Set HDCR.TDA to 0 to allow access to debug system registers.
- If required, set HDCR.TDRA to 1 and configure the stage 2 MMU translation to prevent any guest access to the debug registers through the debug APB.
- If required, set HDCR,TDOSA to 1, to prevent any guest from using the OS Save and Restore mechanism over power down and to prevent any guest from enabling Halting debug-mode.

To support self-hosted trace of the Single Machine View by an operating system, an AArch32 hypervisor must:

- Discover which operating system is using self-hosted trace. It can do this by:
    - If CP14 access to the trace registers is implemented, setting HCPTR.TTA to 1 in order to trap the first access to the trace system registers.
    - Configuring the stage 2 MMU translation to trap any guest access to the trace registers through the debug APB.
- On switching from the operating system using self-hosted trace:
    - Swap out the value of trace enable register and disable the ETM or PTM for the new operating system.
    - If CP14 access to the trace registers is implemented, set HCPTR.TTA to 1 to prevent the new operating system accessing trace system registers.
    - Configuring the stage 2 MMU translation to trap the new operating system accessing the trace registers through the debug APB.
- On switching in the operating system using self-hosted trace:
    - If CP14 access to the trace registers is implemented, set HCPTR.TTA to 0 to allow access to the trace system registers.
    - Configuring the stage 2 MMU translation to allow access to the trace registers through the debug APB.

If an ETR is used and this accesses memory through a System MMU, then the hypervisor must configure the System MMU context for the ETR for stage 2 translations to match the operating system using self-hosted trace.

In order to modify this context the hypervisor must first:

- Disable the ETM or PTM.
- Issue a trace flush at the ETR.
- Poll the ETR to ensure the flush is complete.

AArch64 hypervisor support for self-hosted debug and trace, Single Machine View

To support self-hosted debug of the Single Machine View by an operating system, an AArch64 hypervisor must:

- Discover which operating system is using self-hosted debug. It can do this by setting MDCR_EL2.TDA to 1 in order to trap the first access to the debug system registers.
- On switching from the operating system using self-hosted debug:
    - Swap out the value of MDSCR_EL1, and set MDSCR_EL1.MDE to 0 for the new operating system.
    - Set MDCR_EL2.TDA to 1 to prevent the new operating system accessing debug system registers.
- On switching in the operating system using self-hosted debug:
    - Restore the value of MDSCR_EL1.
    - Set MDCR_EL2.TDA to 0 to allow access to debug system registers.
- If required, set MDCR_EL2.TDRA to 1 and configure the stage 2 MMU translation to prevent any guest access to the debug registers through the debug APB.

- If required, set MDCR_EL2,TDOSA to 1, to prevent any guest from using the OS Save and Restore mechanism over power down and to prevent any guest from enabling Halting debug-mode.

To support self-hosted trace of the Single Machine View by an operating system, an AArch64 hypervisor must:

- Discover which operating system is using self-hosted trace. It can do this by:
  - If CP14 access to the trace registers is implemented, setting CPTR_EL2.TTA to 1 in order to trap the first access to the trace system registers.
  - Configuring the stage 2 MMU translation to trap any guest access to the trace registers through the debug APB.
- On switching from the operating system using self-hosted trace:
  - Swap out the value of trace enable register and disable the ETM for the new operating system.
  - If system register access to the trace registers is implemented, set CPTR_EL2.TTA to 1 to prevent the new operating system accessing trace system registers.
  - Configuring the stage 2 MMU translation to trap the new operating system accessing the trace registers through the debug APB.
- On switching in the operating system using self-hosted trace:
  - If system register access to the trace registers is implemented, set CPTR_EL2.TTA to 0 to allow access to the trace system registers.
  - Configuring the stage 2 MMU translation to allow access to the trace registers through the debug APB.

If an ETR is used and this accesses memory through a System MMU, then the hypervisor must configure the System MMU context for the ETR for stage 2 translations to match the operating system using self-hosted trace.

In order to modify this context the hypervisor must first:

- Disable the ETM.
- Issue a trace flush at the ETR.
- Poll the ETR to ensure the flush is complete.

Secure monitor support for self-hosted debug and trace, Single Machine View

The Secure monitor should be programmed as for *Virtualizer View*. See:

- *AArch32 Secure monitor support for self-hosted debug, Virtualizer View* above
- *AArch64 Secure monitor support for self-hosted debug, Virtualizer View* above.

Note: The external debugger can reprogram the debug registers to force exceptions in guest operating systems other than that being debugged.

4.2.4 Multiple Machine View

Operating system support for self-hosted debug and trace, Multiple Machine View

The operating system should be programmed as for the *Single Machine View*. See:

- *AArch32 operating system support for self-hosted debug and trace, Single Machine View* above
- *AArch64 operating system support for self-hosted debug and trace, Single Machine View* above.

Hypervisor support for self-hosted debug and trace, Multiple Machine View

To support self-hosted debug and/or trace of the Multiple Machine View by an operating system, a hypervisor must implement similar support as for the Single Machine View. See:

- *AArch32 hypervisor support for self-hosted debug and trace, Single Machine View* above
- *AArch64 hypervisor support for self-hosted debug and trace, Single Machine View* above.

To support self-hosted debug of multiple operating system contexts, the hypervisor must:

- Additionally switch the entire debug context with the context of operating systems using debug. This can be done lazily.

To support self-hosted trace of multiple operating system contexts, the hypervisor must:

- Switch the entire ETM or PTM context with the context of the operating system using trace. This can be done lazily.
- Virtualize the trace fabric and sink(s) peripherals and switch their contexts with the context of the operating system using trace, if this differs between operating systems. This can be done lazily.
- If an ETR is used with a System MMU, then before configuring a new System MMU context for the ETR:
  - Disable the ETM or PTM.
  - Issue a trace flush at the ETR.
  - Poll the ETR to ensure the flush is complete.
- If an ETB is used, then before switching to a new trace context:
  - Disable the ETM or PTM.
  - Issue a trace flush at the ETB.
  - Poll the ETB to ensure the flush is complete.
  - Preserve the ETB contents with the old trace context.

However, in a multiprocessor system, the trace fabric and sink(s), including ETFs and ETBs, are potentially used simultaneously by the multiple applications being debugged. See *Software support for self-hosted debug and trace in multiprocessor systems, including big.LITTLE* on page 46.

Secure monitor support for self-hosted debug and trace, Multiple Machine View

See *Secure monitor support for self-hosted debug and trace, Single Machine View* above.

4.2.5 Secure Machine View

AArch32 Secure monitor support for self-hosted debug and trace, Secure Machine View

To support self-hosted debug of the Secure Machine View, an AArch32 secure monitor must:

- On switching to Secure state:
  - If supporting any Non-secure debug view save the Non-secure debug context.
  - If the processor is ARMv7 then assert **SPIDEN** HIGH to enable self-hosted debug in Secure state.
  - If the processor is ARMv8 then set:
    - SDCR.EDAD to 1 to disable external debug access in Secure state.
    - SDCR.SPD to 0b11 to enable self-hosted debug in Secure state.

    - Load the Secure debug context.
- On switching to Non-secure state:
    - Save the Secure debug context
    - If the processor is ARMv7 then de-assert **SPIDEN** LOW to disable self-hosted debug in Secure state.
    - If the processor is ARMv8 then set:
        - SDCR.EDAD to 0 to enable external debug access in Non-secure state.
        - SDCR.SPD to 0b10 to disable self-hosted debug in Secure state.
    - If supporting any Non-secure debug view load the Non-secure debug context.

If the processor is ARMv7 and does not support dynamic control over SPIDEN, then **SPIDEN** must be asserted HIGH. However, this means that Non-secure software can program a breakpoint or watchpoint to generate a debug exception inside Secure monitor.

To support self-hosted trace of the Secure Machine View, an AArch32 secure monitor must:

- On switching to Secure state:
    - If supporting any independent Non-secure trace view then:
        - If there is a live Non-secure trace context, save the ETM or PTM, trace fabric, and trace sink(s) contexts.
        - Otherwise set NSACR.TTA to 0.
    - Assert **SPNIDEN** HIGH to enable self-hosted trace in Secure state.
    - Load the Secure contexts.
- On switching to Non-secure state:
    - Save the Secure ETM or PTM, trace fabric, and trace sink(s) contexts.
    - De-assert **SPIDEN** LOW to disable self-hosted debug in Secure state.
    - If supporting any independent Non-secure trace view then:
        - If there are live Non-secure trace contexts, load them.
        - Otherwise set NSACR.TTA to 1 to detect use of the ETM or PTM in Non-secure state.

If the processor does not support dynamic control over **SPNIDEN**, then **SPNIDEN** must be asserted HIGH. However, this means that Non-secure software can program the ETM or PTM to collect trace of the Secure monitor.

If an ETR is used with a System MMU and supporting any independent Non-secure trace, then before configuring a new System MMU context for the ETR:

- Disable the ETM or PTM.
- Issue a trace flush at the ETR.
- Poll the ETR to ensure the flush is complete.

If an ETB is used and supporting any independent Non-secure trace, then before switching to a new trace context:

- Disable the ETM or PTM.
- Issue a trace flush at the ETB.
- Poll the ETB to ensure the flush is complete.
- Preserve the ETB contents with the old trace context.

AArch64 Secure monitor support for self-hosted debug and trace, Secure Machine View

To support self-hosted debug of the Secure Machine View, an AArch32 secure monitor must:

- To enable self-hosted debug in Secure state:
  - Set MDCR_EL3.SDD to 0, if the Secure operating system is using AArch64.
  - Set MDCR_EL3.SPD32 to 0b11, otherwise.
  - Or both.
- On switching to Secure state:
  - If supporting any Non-secure debug view then:
    - If there is a live Non-secure debug context, save it.
    - Otherwise set MDCR_EL3.TDA to 0 .
  - Set MDCR_EL3.EDAD to 1 to disable external debug access in Secure state.
  - Load the Secure debug context.
- On switching to Non-secure state:
  - Save the Secure debug context.
  - Set MDCR_EL3.EDAD to 0 to enable external debug access in Secure state.
  - If supporting any Non-secure debug view then.
    - If there is a live Non-secure debug context, load it.
    - Otherwise set MDCR_EL3.TDA to 1 to detect use of debug system registers in Non-secure state.

To support self-hosted trace of the Secure Machine View, an AArch32 secure monitor must:

- On switching to Secure state:
  - If supporting any independent Non-secure trace view then:
    - If there is a live Non-secure trace context, save the ETM, trace fabric, and trace sink(s) contexts.
    - Otherwise set CPTR_EL3.TTA to 0.
  - Assert **SPNIDEN** HIGH to enable self-hosted trace in Secure state.
  - Load the Secure contexts.
- On switching to Non-secure state:
  - Save the Secure ETM or PTM, trace fabric, and trace sink(s) contexts.
  - De-assert **SPIDEN** LOW to disable self-hosted debug in Secure state.
  - If supporting any independent Non-secure trace view then:
    - If there are live Non-secure trace contexts, load them.
    - Otherwise set CPTR_EL3.TTA to 1 to detect use of the ETM in Non-secure state.

If the processor does not support dynamic control over **SPNIDEN**, then **SPNIDEN** must be asserted HIGH. However, this means that Non-secure software can program the ETM or PTM to collect trace of the Secure monitor.

If an ETR is used with a System MMU and supporting any independent Non-secure trace, then before configuring a new System MMU context for the ETR:

- Disable the ETM.
- Issue a trace flush at the ETR.

- Poll the ETR to ensure the flush is complete.

If an ETB is used and supporting any independent Non-secure trace, then before switching to a new trace context:

- Disable the ETM.
- Issue a trace flush at the ETB.
- Poll the ETB to ensure the flush is complete.
- Preserve the ETB contents with the old trace context.

4.2.6 Single Application View

AArch32 operating system support for self-hosted debug and trace, Single Application View

To support self-hosted debug of the Single Application View, an AArch32 operating system must:

- Initialize the debug system registers.
- On switching to an application being debugged:
  - Set DBGDSCRext.MDBGen to 1, to enable Monitor debug-mode.
- On switching to any other application:
  - Set DBGDSCRext.MDBGen to 0, to disable Monitor debug-mode.

To support self-hosted trace of the Single Application View, an AArch32 operating system must:

- Initialize the ETM or PTM and trace fabric, including any funnels and trace sinks.
- On switching to an application being traced:
  - Enable the ETM or PTM.
- On switching to any other application:
  - Disable the ETM or PTM.

Typically the debugger also runs as an application. To support this, an AArch32 operating system must provide interfaces to allow that application to:

- Select an application to debug or trace.
- Configure the debug, ETM or PTM, and trace fabric, including any funnels and trace sinks, on behalf of the debugger.
- If an ETR or ETB is being used, allow the debugger application access to the collected trace buffers.

AArch64 operating system support for self-hosted debug and trace, Single Application View

To support self-hosted debug of the Single Application View, an AArch64 operating system must:

- Initialize the debug system registers.
- On switching to an application being debugged:
  - Set MDSCR_EL1.MDBGen to 1, to enable Monitor debug-mode.
- On switching to any other application:
  - Set MDSCR_EL1.MDBGen to 0, to disable Monitor debug-mode.

To support self-hosted trace of the Single Application View, an AArch64 operating system must:

- Initialize the ETM and trace fabric, including any funnels and trace sink.
- On switching to an application being traced:

    - Enable the ETM.
- On switching to any other application:
    - Disable the ETM.

Typically the debugger also runs as an application. To support this, an AArch64 operating system must provide interfaces to allow that application to:

- Select an application to debug or trace.
- Configure the debug, ETM and trace fabric, including any funnels and trace sinks, on behalf of the debugger.
- If an ETR or ETB is being used, allow the debugger application access to the collected trace buffers.

Hypervisor and secure monitor support for self-hosted debug and trace, Single Application View

The hypervisor and secure monitor should be programmed as for Single Machine View. See:

- *AArch32 hypervisor support for self-hosted debug and trace, Single Machine View* above
- *AArch64 hypervisor support for self-hosted debug and trace, Single Machine View* above
- *Secure monitor support for self-hosted debug and trace, Single Machine View* above.

4.2.7 Multiple Application View

Operating system support for self-hosted debug and trace, Multiple Application View

The operating system should be programmed as for Single Application View. See:

- *AArch32 operating system support for self-hosted debug and trace, Single Application View* above
- *AArch64 operating system support for self-hosted debug and trace, Single Application View* above.

To support self-hosted debug of multiple applications, the operating system must:

- Additionally switch the entire debug context with the context of applications using debug. This can be done lazily.

To support self-hosted trace of multiple applications, the operating system must:

- Switch the entire ETM or PTM context with the context of the application using trace. This can be done lazily.
- Virtualize the trace fabric and sink(s) peripherals and switch their contexts with the context of the application using trace.
- If an ETR is used then before configuring the ETR for a new application:
    - Disable the ETM or PTM.
    - Issue a trace flush at the ETR.
    - Poll the ETR to ensure the flush is complete.
- If an ETB is used, then before switching to a new trace context:
    - Disable the ETM or PTM.
    - Issue a trace flush at the ETB.
    - Poll the ETB to ensure the flush is complete.
    - Preserve the ETB contents with the old trace context.

However, in a multiprocessor system, the trace fabric and sink(s), including ETFs and ETBs, are potentially used simultaneously by the multiple applications being debugged.

See *Software support for self-hosted debug and trace in multiprocessor systems, including big.LITTLE* on page 46.

Hypervisor and secure monitor support for self-hosted debug and trace, Single Application View

See *Hypervisor and secure monitor support for self-hosted debug and trace, Single Application View* above.

4.2.8 Single Secure Application and Multiple Secure Application Views

These views are the same as the *Single Application View* and *Multiple Application View*, except that the Secure operating system must behave as for the *Secure Machine View*, and:

- For self-hosted debug:
  - In AArch32, SDER.SUIDEN must be set to 1 for each application being debugged.
  - In AArch64, MDCR_EL3.SDD must be 0.
- For self-hosted trace, in AArch32, SDER.SUNIDEN must be set to 1 for each application being traced.

4.2.9 Disabled

For devices in very closed environments, such as dedicated secure processors, software must actively not enable self-hosted debug.

4.3 Additional software considerations for self-hosted debug and trace

4.3.1 Co-operation with external debug and trace

See *Software considerations for external debug and trace* on page 22.

4.3.2 Discovery

Because debug and trace are mostly unused when a system is in service, it is advantageous to detect first use of debug or trace and disable any support prior to first use, in particular, any context switching of state.

Software can use configurable traps on register access to do this. See also *Software considerations for external debug and trace* on page 22.

4.3.3 Configuring debug and trace by a self-hosted debugger

A self-hosted debugger must use features of the debug and trace architectures to restrict debug operations to the specific view being debugged.

A debugger also use the CONTEXTIDR and VMID comparators in debug and trace to restrict debug operations to specific instances of each view. These might be used, for example, to reduce overhead as software might only need to update the CONTEXTIDR and/or VMID values when switching between two contexts.

See also *Configuring debug and trace by an external debugger* on page 23

4.3.4 The debug logic state to preserve for context switching

For details of the state required to be saved for an application, guest operating system or security state, see the relevant architecture reference manual:

- *Embedded Trace Macrocell Architecture Specification ETMv1.0 to ETMv3.5*
- *Embedded Trace Macrocell Architecture Specification ETMv4*
- *Program Flow Trace Architecture Specification*
- *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition*

- *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile*.

4.3.5 Software support for self-hosted debug and trace in multiprocessor systems, including big.LITTLE

For a description of multiprocessor systems, including big.LITTLE, see *Power management and multiprocessor systems*, *including big.LITTLE* on page 18.

The tactics for supporting migration depends on the view of the system that the software being migrated has, and whether the system is homogeneous or heterogeneous.

Operating system software that supports a *Multiple Application View* of debug will naturally support homogeneous MP systems, as the state of debug and trace components unique to each processor is switched with the context of the application.

However, components such as the trace fabric and sink(s), including ETFs and ETBs, and trace sources such as a *System Trace Macrocell* (STM) are shared between processors. If multiple debug and trace contexts can be live simultaneously, software must manage and/or virtualize the components. For example, by demultiplexing the multiple trace streams generated by simultaneously tracing multiple program flows.

However, big.LITTLE systems are typically heterogeneous, which creates further complications, depending on the scheduling model:

- *big.LITTLE Cluster Migration*
- *big.LITTLE CPU Migration*
- *big.LITTLE MP*.

big.LITTLE Cluster Migration

If each processor is switched as an individual element, the task of migrating debug and trace state is the same as for *big.LITTLE CPU Migration* below.

big.LITTLE CPU Migration

When switching debug context of a complete processor (that is, a *Single Machine View*; for example, the OSPM is part of a Hypervisor), the OSPM acts as it would when managing a *Multiple Machine View* on a uniprocessor.

If the OSPM is further responsible for switching a Hypervisor (for example, the OSPM is part of a Secure monitor), it must treat it as if managing debug of the *Virtualizer View*.

big.LITTLE MP

Notionally, the big.LITTLE MP sharing model is no different to a standard SMP system, and therefore the OSPM has a *Multiple Application View* of the processes and manages debug and trace context accordingly.

Complications for heterogeneous systems

However, in each case the heterogeneous issues must be addressed.

Note: See also *Target software support for external debug and trace in multiprocessor systems, including big.LITTLE* on page 28.

With respect to the PMU:

- For ARMv7, the *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition* requires:
  - For architecture-defined events, that is, those with event numbers in the range 0x00 to 0x3F, if the event is not implemented then an event counter programmed with that event number does nothing.
  - For IMPLEMENTATION DEFINED events, that is, those with event numbers in the range 0x40 to 0xFF, the same rule applies.

  However, ARM recognizes that ARMv7 implementations may wish to stray from the precise definition of *IMPLEMENTATION DEFINED* and have undocumented events.

- For ARMv8, the *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile* requires:
    - The same rule for architecture-defined events.
    - For IMPLEMENTATION DEFINED events, that is, those with event numbers in the range 0x040 to 0x3FF, it is UNKNOWN what event, if any, is counted.

ARM recommends that:

- Implementations reserve event numbers in the range 0x40 to 0xBF for events defined by the *ARM recommendations for IMPLEMENTATION DEFINED event numbers* in the *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition*. Any unimplemented events in this range do not count any event.
- Any further IMPLEMENTATION DEFINED or undocumented events are in the range 0xC0 to 0xFF (ARMv7) or 0x3FF (ARMv8).

The OSPM can provide two different views of the monitors in a big.LITTLE system:

- A logical view, where the user sees PMU events relating to a logical processor, which can, at any time, be physically instantiated by a big or a LITTLE processor.
- A physical view, where the user sees PMU events relating to an actual physical processor.

For simple tools, the OSPM software can duplicate the PMU state. The user of the tool is presented with a unified performance report for the software. This provides a logical view.

Note: A tool might be a profiling tool or could be a service layer within an operating system that supports such tools. The OSPM software might be an operating system, or a hypervisor or similar.

Following the recommendations above simplifies the role for the OSPM software, as it guarantees that if an event in the range 0x00 to 0xBF is only defined on one of the processors that it will not have unpredictable behavior on the other processor(s).

Tools must use only events in the range 0xC0 to 0xFF with caution, and interpret the results of any microarchitectural event according to the heterogeneous nature of the system.

Note: For simple tools, the tool itself may not be aware that the system is heterogeneous. Therefore, this caution applies the user, who presumably does.

Software executing at EL2 can also make use of HDCR.HPMN to restrict the number of counters available to a guest operating system, for example if in a heterogeneous cluster the number of counters is not the same on all processors.

Software at EL1 that uses PMCR to detect the number of counters will get the virtualized number of counters set by software at EL2. Alternatively, software can use other detection mechanisms such as *flattened device trees* (FDTs) that are outside the scope of this document.

Similar approaches can be used for simple debug and trace tools.

More advanced tools can be aware of the heterogeneous nature of the system and use a physical view of the monitors to provide a performance report for each (type of) processor individually.

To support this use case, ARM recommends that the OSPM provide an API for tools to independently control debug, PMU and trace hardware when executing on the different types of processor.

These services provide a virtual programmers' model, with debug resources for each type of processor. The OSPM is responsible for ensuring the correct context is loaded each time the software being debugged is switched.

For example, in a big.LITTLE system, the OSPM can provide a system call to:

- Configure a first virtual counter to count an event when running on the big processor.
- Configure a second virtual counter to count an event when running on the LITTLE processor.

An OSPM can simultaneously support both logical and physical views, or may support only one view. For more information see *Power State Coordination Interface*.