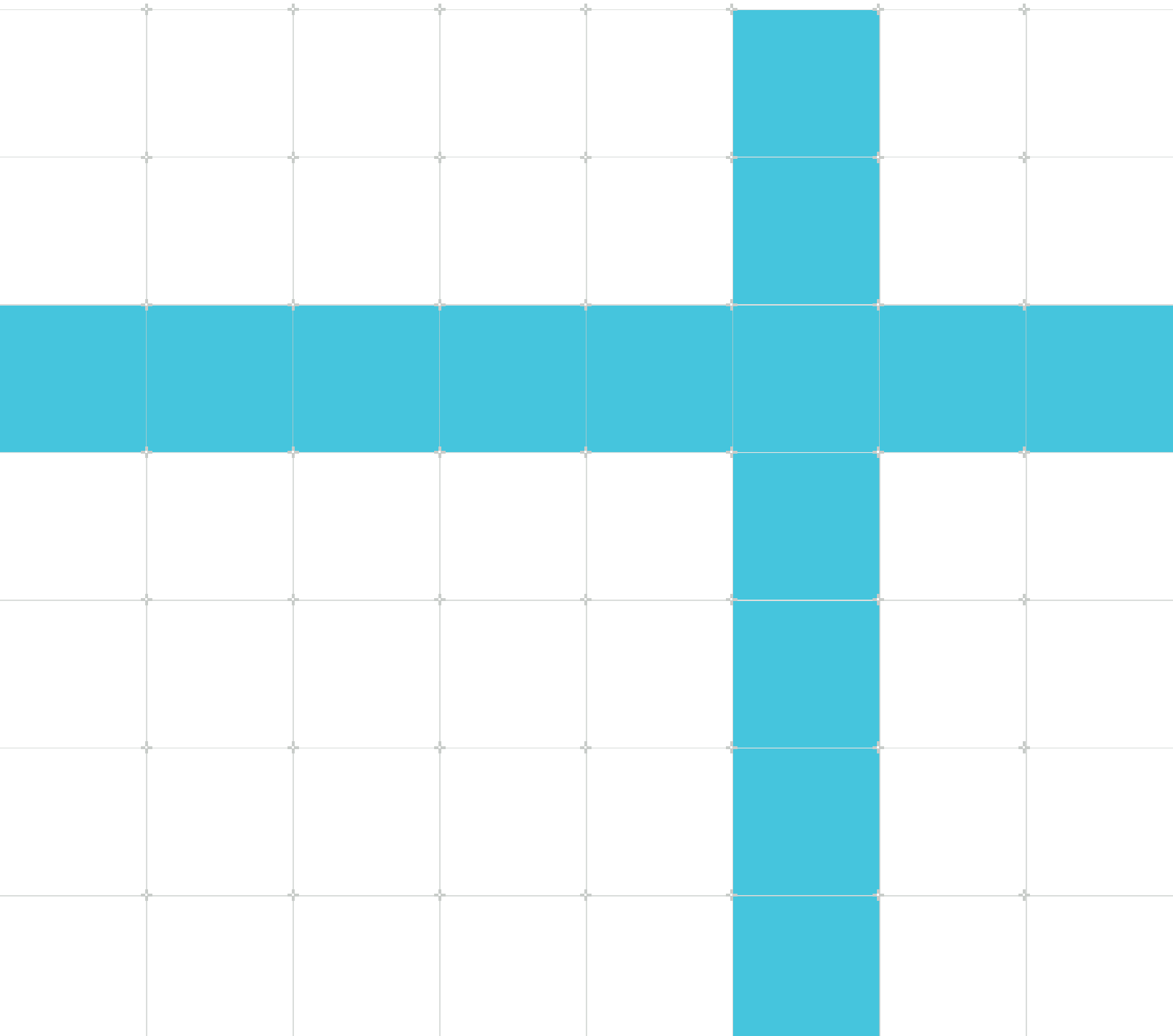
arm

Learn the architecture - AArch64 external debug

Version 1.0

Issue 01
102196_0100_01_en



Learn the architecture - AArch64 external debug

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	5 May 2022	Non-Confidential	Initial release

Proprietary Notice

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

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

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

Feedback

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

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

Inclusive language commitment

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

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

Contents

1. Overview

The Armv8-A architecture supports both self-hosted debug and external debug. This guide provides an overview of external debug, and describes the external debug features that the architecture supports. The guide explains external debug features, and steps to enable these features.

The self-hosted debug model is used when the debugger is hosted on a Processing Element (PE) that is being debugged. Debug exceptions are the basis of the self-hosted debug model. The debugger programs the debug logic to generate debug events. These debug events generate debug exceptions. You can read about this topic in our AArch64 self-hosted debug guide.

The external debug model is used when the debugger is hosted external to the PE that is being debugged. Debug state is the basis of the external debug model. The debugger programs the debug logic to generate debug events. These debug events cause the PE to enter Debug state. This guide describes external debug features in detail.

After you have read this guide, you can Check your knowledge. You will understand:

- How external debug works
- What debug features are supported in external debug
- How to program debug logic to enable each external debug event

Before you begin

You need to be familiar with the basics of the Armv8-A architecture, and the Armv8-A Exception model, to understand the material that is presented in this guide. If you are not familiar with these topics, read these guides:

- Introducing the Arm architecture
- AArch64 Exception model

Debugger usage for software development is beyond the scope of this guide. To learn about debugger usage from a software development perspective, read these guides:

- Before debugging
- Debugger usage

For a general introduction to debugging on Armv8-A, read the Introduction to debug in our AArch64 self-hosted debug guide.

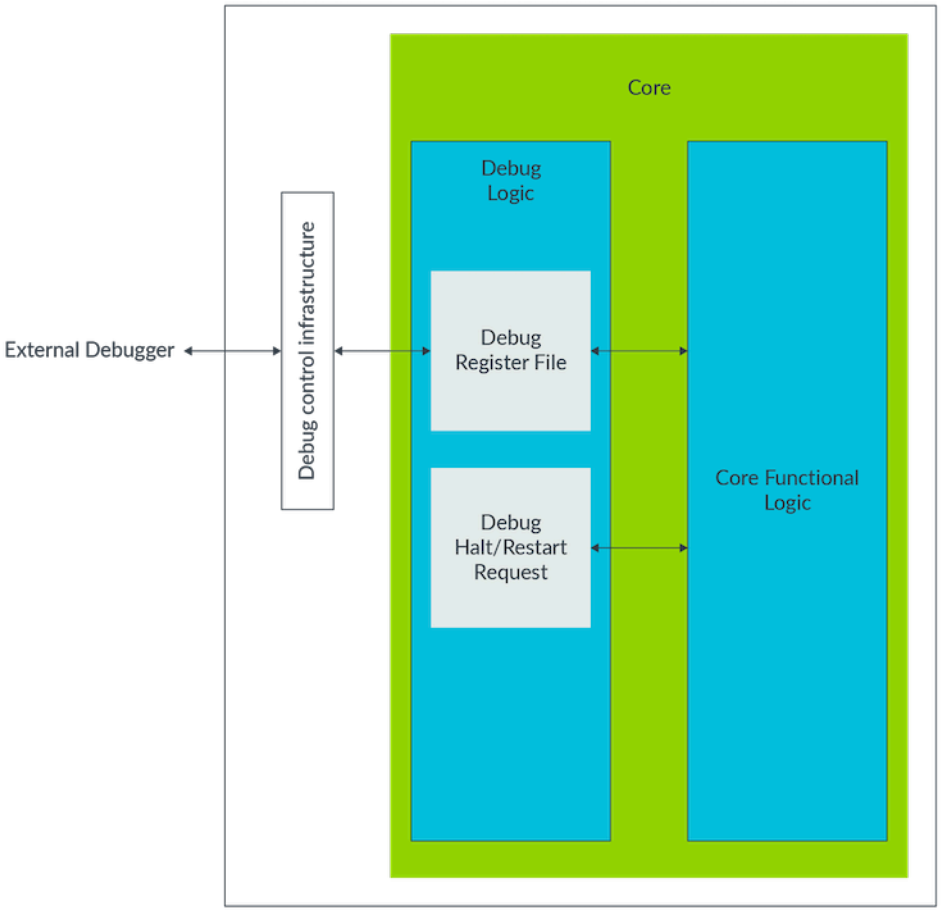
2. External debug

An external debug model is used when a debugger is hosted external to the PE that is being debugged. The external debugger can control the PE with the following operations:

- Stopping the program execution on the occurrence of debug events
- Checking the status of the architectural registers
- Modifying the status of the architectural registers
- Running instructions on the PE to access memory

The external debugger controls the entry of the core into Debug state, by configuring a debug event. A debug event is configured by either programming debug logic, or by asserting a debug halt request through a signal to the core. The following diagram illustrates the setup of an external debugger:

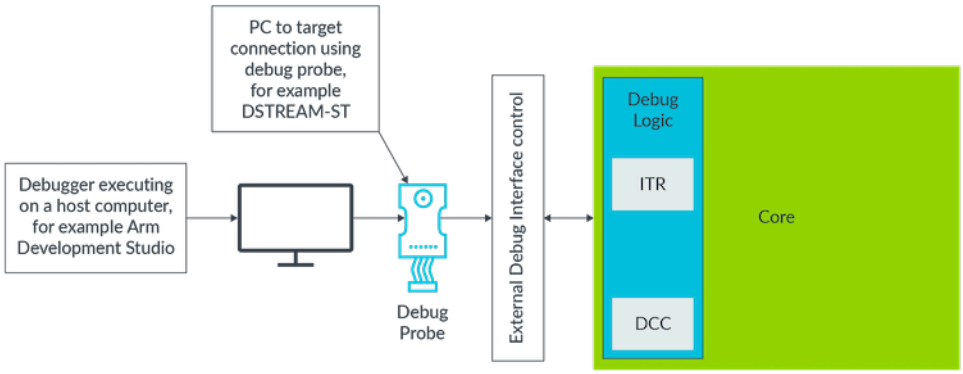
Figure 2-1: External debugger



External debug interface

In Debug state, the PE executes instructions in the Instruction Transfer Register (ITR). The external debugger inserts instructions into the ITR using the external debug interface. The Debug Communication Channel (DCC) allows communication between the PE and the external debugger. The following diagram illustrates the debug setup of an Arm core using an external debugger interface:

Figure 2-2: External debug interface



3. Debug state

Debug state is the basis of the external debug model. The external debugger programs the debug logic to cause Debug state on the occurrence of debug events. External debug is also called halting debug mode. When the PE enters Debug state, the following things happen, in this order:

- The PE stops executing instructions from the location that the Program Counter indicates. Instead, the debugger controls the PE through the external debug interface.
- The debugger uses the Instruction Transfer Register (ITR), through the external debug interface, to pass instructions to the PE to execute in Debug state.
    - Using the instructions that are executed through the ITR, the debugger can read/write architectural registers, for example general-purpose registers, System registers, and Floating Point registers.
    - Using the instructions that are executed through the ITR, the debugger can read/write to memory locations.
    - The Data Transfer Registers of the DCC are accessible by both the external debug interface and the System register interface. The DCC plays an important role in exchanging data between the PE and the external debugger.
- The PE cannot service interrupts in Debug state.

Debug state entry and exit

When the PE enters Debug state:

- The PSTATE of the PE, before entering Debug state, is stored in the Debug Saved Program Status Register (DSPSR).
- The preferred restart address is stored in the Debug Link Register (DLR).
- The PE stops execution of the code that is pointed by the Program Counter.
- Interrupts are not serviced.
- External debugger takes control of the PE.

While the PE is in Debug state, the external debugger can:

- View and modify the contents of memory locations and architectural registers, including DLR and DSPSR.
- Use the ITR to pass instructions to the PE to execute in Debug state.
- Use the Debug Communications Channel (DCC) to pass data to the PE, and to receive data from the PE.
- Change Exception levels, by using the DCPS and DRPS instructions.
- Exit Debug state by triggering a Restart request.

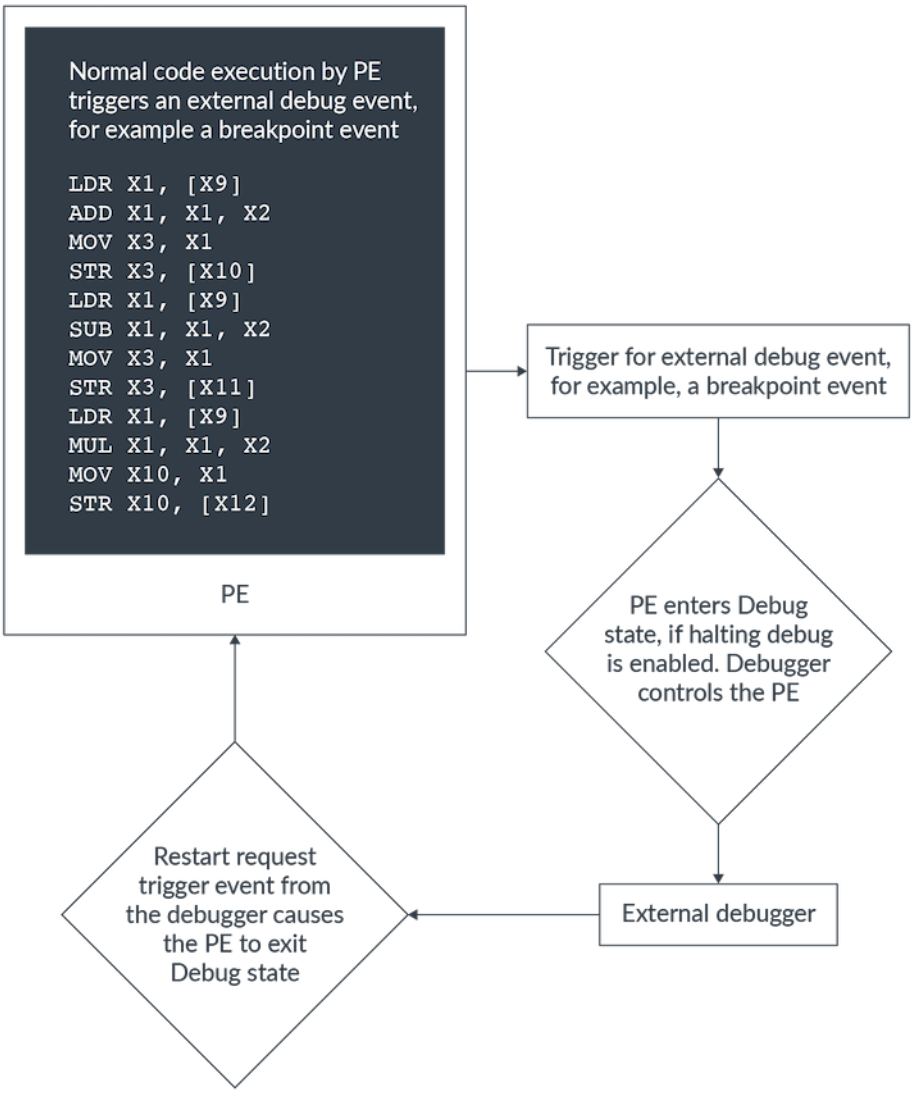
On a restart request from the external debugger, the PE exits Debug state. When the PE exits Debug state, the PE:

- Sets the Program Counter to the address in the DLR.

- Restores PSTATE from DSPSR.
- Begins execution of instructions pointed by the Program Counter.

The following diagram illustrates Debug state entry and exit:

Figure 3-1: Debug state entry and exit



External debug is useful for:

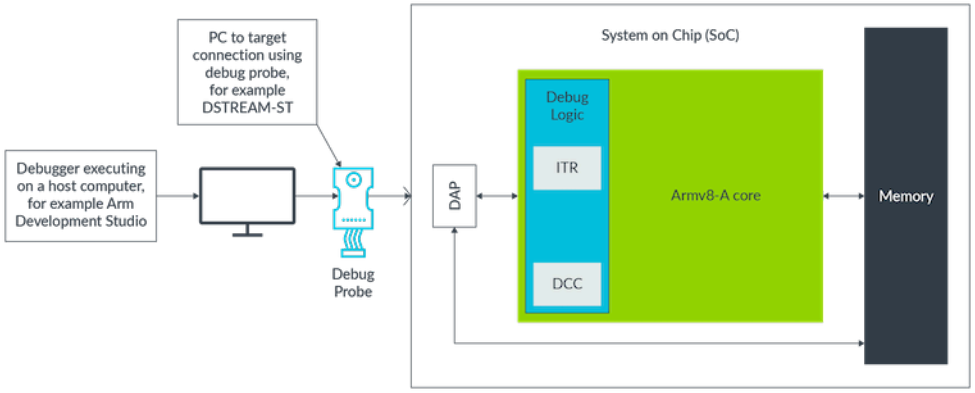
- Hardware bring-up: Debugging during development when a system is first powered up and some software functionality is not available.
- Debugging PEs that are deeply embedded inside systems

4. Debug Access Port

In the external debug model, Debug registers are accessed using the external debug interface. The means to access external debug interface is **IMPLEMENTATION DEFINED**. However, most Armv8-A systems include a Debug Access Port (DAP) to access the external debug interface for off-chip external debuggers. On-chip external debuggers, for example, using a second PE to debug a PE, use the memory mapped interface to access the external debug interface.

The following diagram illustrates access to the external debug interface using a DAP:

Figure 4-1: External debug interface using DAP



External Debug registers, also called halt mode debug registers, are usually prefixed with ED, for example, EDSCR. Two important external Debug registers are:

- EDSCR: External Debug Status and Control Register
- EDECR: External Debug Execution Control Register

5. Enabling external debug

External debug mode does not include any global enable bits. For each external debug event, corresponding control bits can be programmed to enable halting of the processor for that external debug event.

Breakpoints and watchpoints are resources that are shared between self-hosted debuggers and external debuggers. Setting EDSCR.HDE to 1 causes breakpoints and watchpoints to halt the PE.

External debuggers may need to authenticate themselves. If an external debugger does not authenticate itself, halting of an Arm implementation might be prohibited. Details of the authentication are **IMPLEMENTATION DEFINED**. Authentication may exist in hierarchical fashion. This means that a level of authentication might be required during which halting might be enabled at Non-secure execution of the PE. Another level of authentication might be required to allow halting in a Secure execution of the PE.

Handshake between the external debugger and the PE

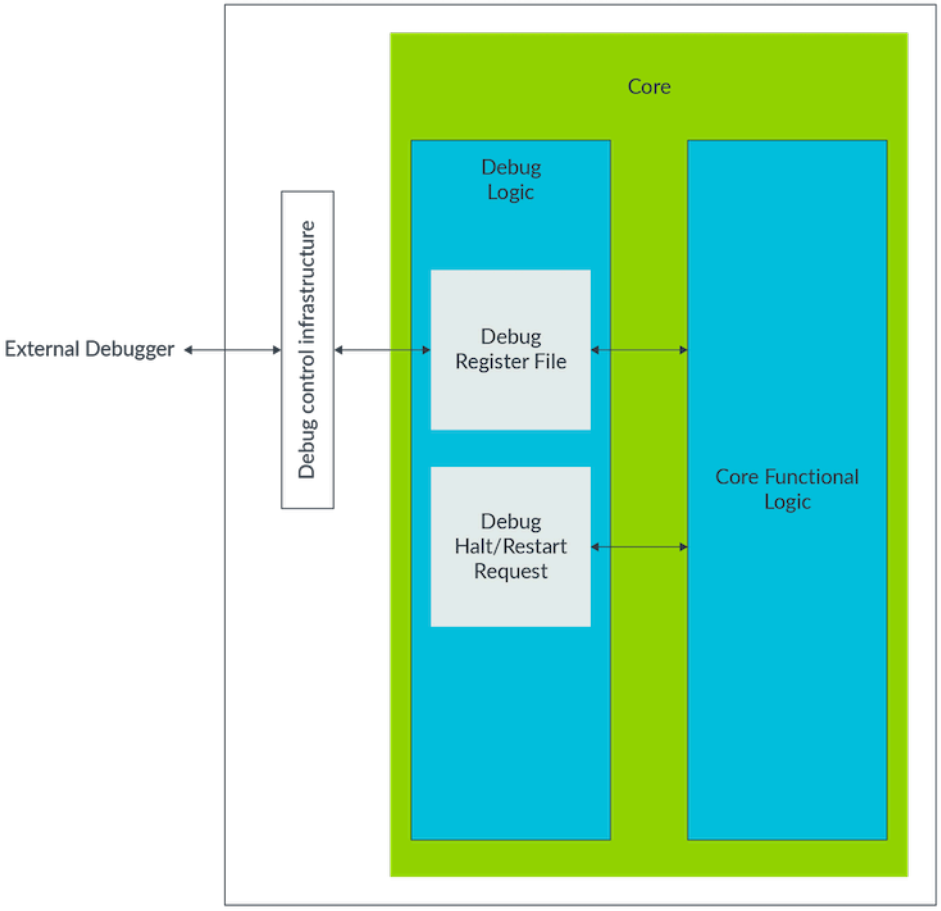
The external debugger takes control of the PE when the PE enters Debug state. The external debugger also initiates the PE to exit Debug state, to continue normal execution of the program. The mechanism that is used to transfer control between the external debugger and the PE is called a handshake. Let's look at this in more detail.

A typical handshake between the external debugger and the PE includes the following actions, in this order:

- The external debugger configures the debug logic of the PE to generate debug events at the point of interest in the program execution. For example, the external debugger configures a breakpoint on the start address of a subroutine.
- When the program reaches the point of interest a debug event is generated, causing the PE to enter Debug state. The external debugger registers of the PE indicate the status of the PE in Debug state.
- The external debugger polls for the external debug registers of the PE to indicate Debug state.
- The external debugger queries the program state and issues a restart request to the PE.

The following diagram and table illustrate the handshake sequence between the external debugger and the core:

Figure 5-1: External debugger and core handshake sequence



Step	External debugger action	Core action
Step 1	The external debugger programs Debug logic for the debug event, either by programming the Debug logic or by asserting a debug halt request.	
Step 2	The external debugger polls its status register until the core enters debug state.	
Step 3		During normal code execution, the core enters debug state on an debug event.
Step 4	The external debugger takes control of the core.	
Step 5	The external debugger engages in its activities, for example, it reads status, modifies status, and runs instructions.	
Step 6	Post external debugger activities, the external debugger requests debug exit.	
Step 7		The core exits debug state when possible.
Step 8		The core resumes normal execution based on the Program Counter and PSTATE.

6. External debug events

In this section of the guide, we describe the debug events that can cause the PE to enter Debug state. Here is a summary of each event:

- External debug request debug event: This event is an input trigger event to the processor. When this event is asserted, the PE enters Debug state.
- Halt instruction debug event: This event is generated whenever the PE executes an HLT instruction.
- Halting step debug event: This event is generated immediately after the PE executes an instruction or an exception, while the PE is not in Debug state.
- Exception catch debug event: This event is generated immediately after the PE executes an exception or an exception return.
- Reset catch debug event: This event is generated immediately after reset, before the PE begins execution following the reset.
- Software access debug event: This event is generated when the PE attempts access to Debug system registers, for example, the breakpoint and watchpoint control registers.
- OS unlock catch debug event: This event is generated when the OS lock state changes from locked to unlocked.
- Breakpoint event: This event is generated whenever the PE attempts to execute an instruction from a particular address.
- Watchpoint event: This event is generated whenever the PE accesses data from a particular address.

Breakpoints and watchpoints are resources that are shared between self-hosted debuggers and external debuggers. If halt mode debug is enabled, a breakpoint event or watchpoint event causes the PE to enter Debug state. Otherwise, a debug exception is generated if self-hosted debug is enabled.

Let's look at each of these events in more detail.

External debug request event

An external debug request event is an input trigger event to the processor. When this event is asserted, the PE enters Debug state. The debugger asserts an external debug request to force the PE to enter Debug state, and then the external debugger controls the PE. Cross Trigger Interface Registers are used to generate external debug request events.

If an external debug request is asserted when the PE is already in Debug state, the request is ignored. To exit Debug state, the external debugger asserts a restart request to the processor. A restart request is an input signal to the processor. When a restart request is asserted, the processor exits Debug state.

Halt instruction debug event

A halt instruction debug event allows the external debugger to take control of the PE when the PE attempts to execute the HLT instruction. The HLT instruction is the halting software breakpoint instruction. When the HLT instruction is committed for execution, the halt instruction debug event is generated if `EDSCR.HDE` is 1. On execution of HLT instruction, the PE enters Debug state and gives control to the external debugger.

The debugger usually replaces the program instruction with an HLT instruction to trigger a halt instruction debug event. After this event, the external debugger controls the PE.

To exit Debug state, the external debugger asserts a restart request to the processor.

Halting step debug event

A halting step debug event allows the external debugger to take control of the program that is being debugged. This happens after the PE executes every individual instruction or exception, while the PE is not in Debug state. When halting step debug event is enabled, the PE enters Debug state whenever an instruction or exception is retired in the PE.

Programming a halting step debug event:

- When the PE is in Debug state, the external debugger enables halting step by writing the `EDECR.SS` to 1 .
- The external debugger signals the PE to exit Debug state.
- After exiting Debug state, the PE executes the instruction that is pointed by the Program Counter (PC). The PE enters Debug state before executing the next instruction, which gives control to the external debugger.

To exit Debug state, the external debugger asserts a restart request to the processor.

Exception catch debug event

An exception catch debug event allows the external debugger to take control whenever the PE executes an exception or exception return. For each Exception level, ED Exception Catch Control Register (EDECCR) includes independent control bits for exception and exception return. Setting the bit in EDECCR forces the PE to enter Debug state when the PE executes an exception, or exception return, with a target Exception level that is indicated in EDECCR.

When an exception catch debug event is generated on exception entry, the PE enters Debug state as part of the exception entry, before the first instruction of the exception handler is executed.

When an exception catch debug event is generated on exception return, the PE enters Debug state, before the execution of the first instruction at the exception return address.

To exit Debug state, the external debugger asserts a restart request to the processor.

Reset catch debug event

A reset catch debug event allows the external debugger to take control of the PE immediately after reset.

To enable a reset catch debug event, set the Reset Catch debug Event (RCE) bit in the External Debug Execution Control Register (EDECR) or Cross Trigger Interface Device Control register (CTIDEVCTL).

When the external debugger sets the RCE bit, the reset catch debug event is generated on a reset of the PE. The PE enters Debug state and gives control to the external debugger, before the execution of the first instruction after reset. The reset can be either a Warm reset or a Cold reset of the PE.

To exit Debug state, the external debugger asserts a restart request to the processor.

Software access debug event

A software access debug event allows the external debugger to take control of the PE when the PE software tries to access the following registers:

- Debug Breakpoint Value Registers (`DBGBVRn_EL1`), including DBGBXVRn from AArch32 state
- Debug Breakpoint Control Registers (`DBGBCRn_EL1`)
- Debug Watchpoint Value Registers (`DBGWVRn_EL1`)
- Debug Watchpoint Control Registers (`DBGWCRn_EL1`)

The external debugger can enable the software access debug event by setting the TDA-bit in the External Debug Status and Control Register (EDSCR).

When the external debugger sets the TDA bit, the PE enters Debug state, and provides control to the external debugger when the PE software tries to access any of the following Debug system registers:

- AArch64: `DBGBCR<n>_EL1`, `DBGBVR<n>_EL1`, `DBGWCR<n>_EL1`, `DBGWVR<n>_EL1`.
- AArch32: `DBGBCR<n>`, `DBGBVR<n>`, `DBGBXVR<n>`, `DBGWCR<n>`, `DBGWVR<n>`.

Memory mapped accesses to the above registers do not generate a software access debug event.

To exit Debug state, the external debugger asserts a restart request to the processor.

OS unlock catch debug event

Some Debug register contents are lost during powerdown of the PE. To enable the debugging ability of the processor after reset, Debug register contents must be saved into a non-volatile memory before the powerdown, and restored when the system comes out of reset. The Armv8-A architecture enables saving of Debug register contents with the OS save mechanism and restoring the Debug register contents with the OS restore mechanism.

The PE software performs the save restore. While saving the Debug register contents, the PE software locks the OS lock so that external debug accesses to Debug registers are forbidden. After restoring the Debug register contents, the PE software unlocks the OS lock.

The OS unlock catch debug event allows the external debugger to take control of the PE, immediately after the OS restore sequence. This event forces the PE to enter Debug state, immediately after restoring the Debug register contents after reset.

The control bit EDECR.OSUCE enables an OS unlock catch debug event.

Breakpoint event

A breakpoint event is generated whenever the PE tries to execute an instruction from a particular address. Hardware breakpoint registers can be programmed with the address of a program. When the PE tries to execute an instruction from the programmed address, a breakpoint event is generated.

When a breakpoint event is generated, it generates an entry to Debug state if:

- `EDSCR.HDE` == one. Halting debug is enabled for these events.
- Authentication signals indicate that debugger has sufficiently authenticated itself for the current Security state of the PE.

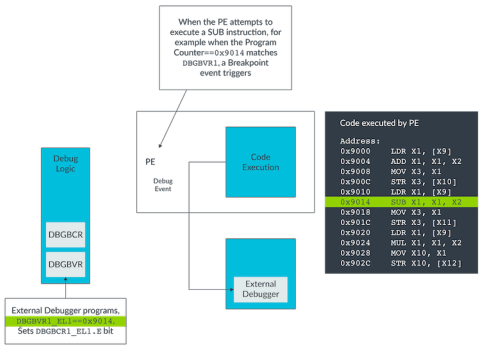
Breakpoint setup uses hardware registers, so is commonly referred to as hardware breakpoint.

Steps to program a breakpoint event:

- The external debugger programs the address of the instruction into the Breakpoint Value Register DBGBVR.
- The external debugger sets the enable bit `DBGBCR.E` to enable the breakpoint event.

The following diagram illustrates a breakpoint event in which the PE tries to execute a SUB instruction for which a breakpoint is set up:

Figure 6-1: Breakpoint event



Hardware breakpoints can be programmed to generate a breakpoint event, based on the following breakpoint matches when the PE executes:

- Unlinked instruction address match: The PE executes from a virtual address with the same value as the DBGBVR register, and the current state of the PE matches the settings in DBGBCR.
- Unlinked Context ID match: The PE executes an instruction while `CONTEXTIDR_EL1/CONTEXTIDR_EL2` is the same as the DBGBVR register, and the current state of the PE matches with the settings in DBGBCR. `DBGBCR.BT` defines whether `CONTEXTIDR_EL1` or `CONTEXTIDR_EL2` is used. These breakpoints are useful to route the control to the external debugger when an application or operating system is scheduled for execution.

- Unlinked VMID match: The PE executes an instruction while VTTBR_EL2.VMID matches the DBGBVR register contents, and the current state of the PE matches the settings in DBGBCR. These breakpoints are useful to route the control to the external debugger when an operating system is scheduled for execution.
- Unlinked Context ID and VMID match: The PE executes an instruction while `CONTEXTIDR_EL1` and `VTTBR_EL2.VMID` matches the contents DBGBVR register, and the current state of the PE matches the settings in DBGBCR. These breakpoints are useful to route the control to the external debugger when an application in an operating system is scheduled for execution.
- Linked breakpoints: Address matching breakpoints can be linked to context or VMID breakpoints. These breakpoints are useful to route the control to the external debugger when the PE executes from an instruction address in an application or operating system, or an application within an operating system.

The Breakpoint Control Register, `DBGBCR<n>_EL1`, contains controls for the breakpoint, for example an enable control.

The Breakpoint Value Register, `DBGBVR<n>_EL1`, holds the value that is used for breakpoint matching. This value is one of the following:

- An instruction virtual address
- One or two Context IDs
- A VMID value
- A concatenation of both a Context ID value and a VMID value

The Armv8-A architecture provides for 2-16 hardware breakpoints to be implemented. How many hardware breakpoints an implementation supports is an **IMPLEMENTATION DEFINED** choice. Depending on the availability of the hardware breakpoint units, that number of hardware breakpoints can be simultaneously set up on an implementation. The register ID_AA64DFR0_EL1.BRPs indicates how many breakpoint units are implemented.

Individual breakpoint unit can be enabled or disabled by programming the E-bit of the individual debug breakpoint control register, `DBGBCR<n>_El1.E`.

Each breakpoint unit has a corresponding control register. Depending on how many breakpoints are implemented, the registers are numbered in line with this, so that:

- `DBGBCR0_EL1` and `DBGBVR0_EL1` are for breakpoint number zero.
- `DBGBCR1_EL1` and `DBGBVR1_EL1` are for breakpoint number one.
- `DBGBCR2_EL1` and `DBGBVR2_EL1` are for breakpoint number two.
- `DBGBCR<n>_EL1` and `DBGBVR<n>_EL1` are for breakpoint number n.

Watchpoint event

A watchpoint event is generated when a PE accesses data memory from a particular address or address range. Hardware watchpoint registers can be programmed with the address of a data memory location. When the PE attempts to access data from the programmed address, a watchpoint event is generated.

When a watchpoint event is generated, the PE enters Debug state if:

- `EDSCR.HDE` == 1, Halting debug is enabled for these events.
- Authentication signals indicate that the debugger has sufficiently authenticated itself for the current Security state of the PE.

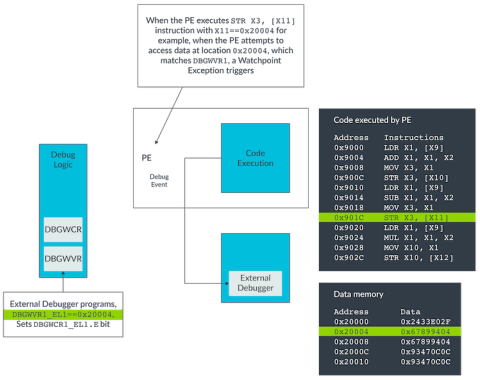
A watchpoint never generates a watchpoint debug event on an instruction fetch.

These are the steps to program a watchpoint event, in this order:

- The external debugger programs the address of the data into the watchpoint value register, DBGWVR.
- The external debugger enables the watchpoint in the Watchpoint Control Register, DBGWCR by setting the enable bit `DBGWCR.E`.

The following diagram illustrates a watchpoint event when the PE tries to execute an instruction for which a watchpoint is set up:

Figure 6-2: Watchpoint event



Lower bits of DBGWVR can be masked in the address comparison. This allows the ED to set a watchpoint event over a range of addresses.

A watchpoint can be:

- Programmed to generate watchpoint debug events on read accesses only, on Write-Accesses only, or on both types of access
- Linked to a Linked Context breakpoint, so that a watchpoint debug event is only generated if the PE is in a particular context when the address match occurs

The Armv8-A architecture provides for 2-16 hardware watchpoints to be implemented. How many hardware watchpoints an implementation supports is an **IMPLEMENTATION DEFINED** choice . Depending on the availability of the hardware watchpoint units, that number of watchpoints can be simultaneously set up on an implementation. The ID_AA64DFR0_EL1.WRPs register indicates how many watchpoint units are implemented.

An individual watchpoint unit can be enabled or disabled by programming the E-bit of the individual debug watchpoint control register, `DBGWCR<n>_EL1.E`.

Each watchpoint unit has a corresponding control register. Depending on how many watchpoints are implemented, the registers are numbered in line with this, so that:

- `DBGWCR0_EL1` and `DBGWVR0_EL1` are for watchpoint number zero.
- `DBGWCR1_EL1` and `DBGWVR1_EL1` are for watchpoint number one.
- `DBGWCR2_EL2` and `DBGWVR2_EL1` are for watchpoint number two.
- `DBGWCR<n>_EL1` and `DBGWVR<n>_EL1` are for watchpoint number n.

7. Embedded Cross Trigger

Embedded Cross Trigger (ECT) is the mechanism that is used by an external debugger in an Armv8-A system to generate debug events. ECT also routes events from one agent to another agent within the system.

The external debugger uses ECT to generate a debug request trigger or a restart request trigger. These triggers route events between PEs. For example, these triggers can route a cross halt event from one PE to other PEs, so that all PEs in a system halt whenever one PE in the system halts.

The ECT in an Armv8-A implementation supports:

- Cross Trigger Interface (CTI)
- Input and output triggers
- Cross Trigger Matrix (CTM)

The CTI provides the ability to route trigger events across various components of the Armv8-A system. Examples include routing trigger events to the interrupt controller across, for example, trace units, or across PEs in the system.

If there are multiple CTI blocks in the system, we can use the CTM to pass events between the CTI blocks. The CTM uses the following IO channels to pass events between CTM blocks:

- Input triggers – Trigger event inputs from the PE to the CTI.
- Output triggers – Trigger event outputs from the CTI to the PE.
- Input channels – Channel event inputs from the CTM to the CTI. These are directly accessible by the registers CTIAPPPULSE, CTIAPPSET, and CTIAPPCLEAR, which are present in the ECT programmers model.
- Output channels – Channel event outputs from the CTI to the CTM.

A few of the CTI output triggers that are present in the ECT of an Armv8-A PE have fixed meanings:

- Output trigger event 0 – Debug request trigger event. This is an output trigger event from the CTI and an input trigger event to the processor, asserted by the CTI to force the processor into Debug state. The trigger event is asserted until acknowledged by the debugger. If the processor is already in Debug state, the processor ignores the trigger event, but the CTI continues to assert it until it is removed by the debugger.
- Output trigger event 1 – Restart request trigger event. This is an output trigger event from the CTI to force the PE to exit Debug state.
- Input trigger event 0 – Cross-halt Request trigger event. This is an input trigger from the PE to the CTI. This event can be routed to other agents in the system. For example, it can be routed as a debug request event to another PE.

8. Check your knowledge

Q: What is the debug model called when the debugger is external to the Processor Element (PE)?

External debug

Q: What is the basis for the external debug model?

Debug state

Q: List four debug events that are supported by external debug.

The possible answers for this question are: External debug request, Halt instruction, Halting step, Exception catch, Reset catch, Software access debug event, OS unlock catch debug event, Breakpoint event, and Watchpoint event.

Q: What is the basis for the self-hosted debug model?

Debug exceptions

Q: How do you disable external debug?

There are no global bits to disable external debug.

Q: How many hardware breakpoints and watchpoints are supported in an Arm core?

The number of hardware breakpoints and watchpoints that are supported in an Arm core is **IMPLEMENTATION DEFINED**.

9. Related information

Here are some resources related to material in this guide:

- Arm Architecture Reference Manual: Information on AArch64
- Arm Community: Ask development questions, and find articles and blogs on specific topics from Arm experts
- AArch64 self-hosted debug guide

10. Next steps

This guide has introduced the concept of the debug model for the Armv8-A AArch64 with emphasis on external debugging. Understanding this information helps you to program the debug logic for external debug.

This guide also mentioned the concept of self-hosted debug. If you want to learn more about self-hosted debugging, read our AArch64 self-hosted debug guide.