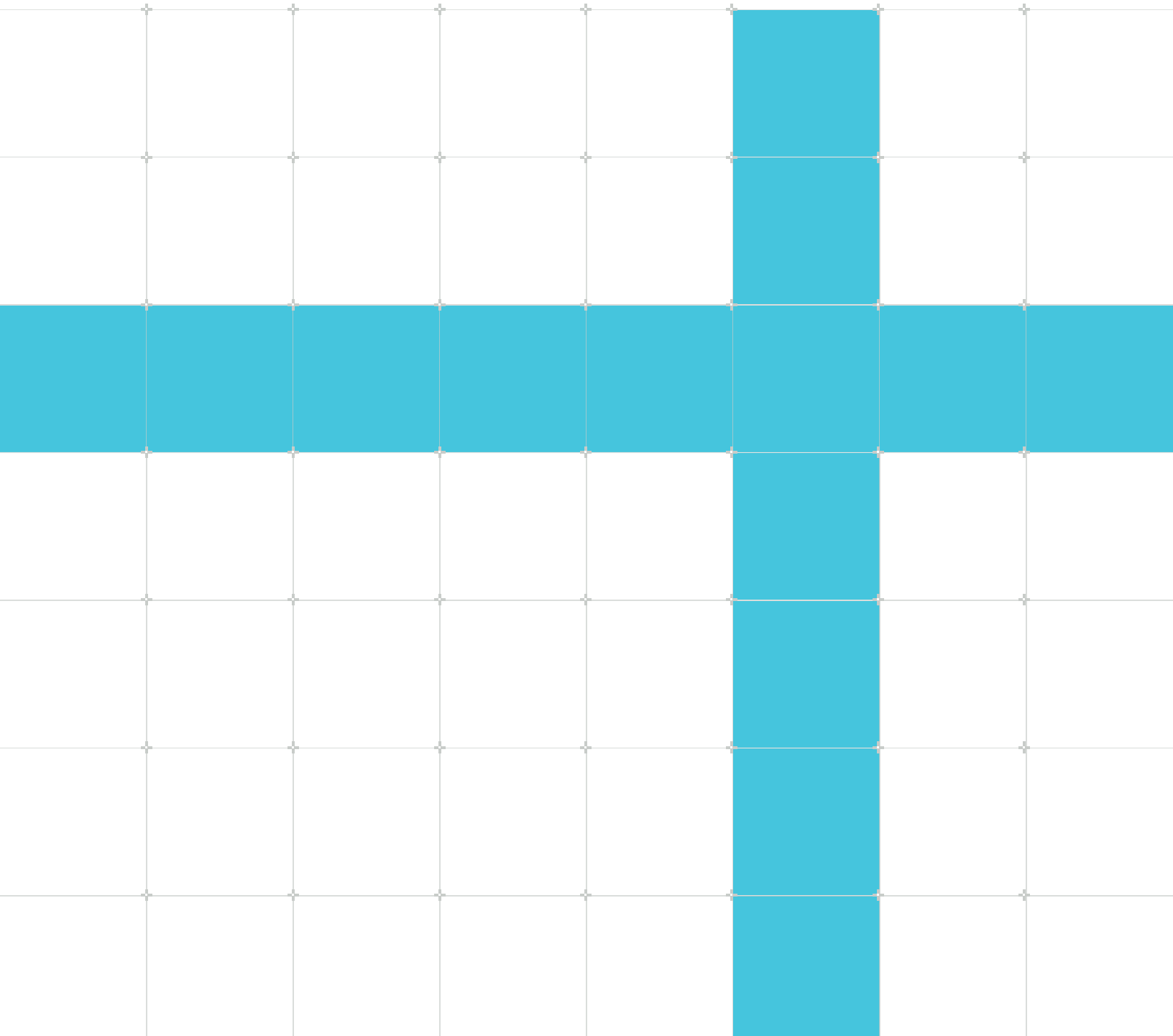
# Learn the architecture - Introducing CoreSight debug and trace

Revision r1p1

## Guide

**Non-Confidential**


**Issue 01**
102520_0101_01_en



# Learn the architecture - Introducing CoreSight debug and trace Guide



This document (102520_0101_01_en) was issued on 2025-09-15. There might be a later issue at https://developer.arm.com/documentation/102520

The product revision is r1p1.

See also: Proprietary notice | Product and document information | Useful resources

## Start reading

If you prefer, you can skip to the start of the content.

## Intended audience

SoC designers

## 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.

This document includes language that can be offensive. We will replace this language in a future issue of this document.

To report offensive language in this document, email terms@arm.com.

## 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.

# Contents

# 1. Overview

This guide introduces the debug and trace infrastructure that the Arm CoreSight architecture provides. The guide also describes the components that are suitable for use with Arm A-profile processors. It also explains the debug system context for software development on Arm processors. Using the guide, you can learn how debugging devices connect to the Arm processor cores within the chip.

If you are an SoC designer, this guide helps you design debug and trace infrastructure using Arm CoreSight IP products such as the CoreSight SoC components. It gives a high-level view of the goals when designing CoreSight debug infrastructure.

# 2. Before you begin

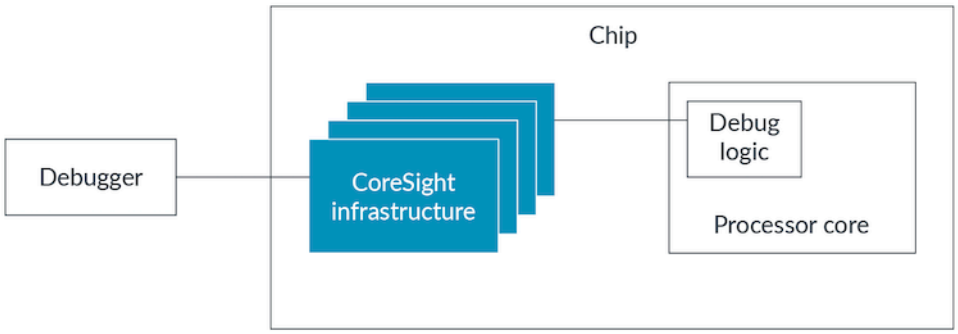
This guide assumes that you are familiar with the debug and trace capabilities in Arm A-profile processors.

Every Arm processor core has some debug functionality that is built in. This debug functionality is used in two general modes:

- Self-hosted debug, in which code that is running on the processor core uses the debug capability to identify problems in the software,
- External debug, in which an external debugging component, the debugger, can access the debug features and use them to identify problems with the software.

For the external component to access the built-in debug capability of the processor, a debug infrastructure must be built into the chip. This infrastructure provides the connectivity between the debugger and the processor. The mechanism that Arm uses to provide this debug infrastructure is based on the CoreSight architecture. The following diagram illustrates the CoreSight infrastructure concept:

**Figure 2-1: A diagram showing the CoreSight infrastructure concept**




The CoreSight architecture defines a set of capabilities that can be designed into a processor or system level components. The system level capabilities allow a debugging component to access and use the processor debug and trace capabilities. Arm has developed a set of components that are based on this architecture. These components are used to create a customized debug infrastructure for a device, and are delivered in the CoreSight SoC products.

The CoreSight components that are essential for use with an A-profile processor can be divided into two groups:

- Debug control: Components that provide a control path from the debugger to the Debug registers that are designed into the processor cores.
- Debug co-ordination: Components that can be used to coordinate debug activities across multiple processor instances. These are referred to as the cross-trigger components.

The CoreSight trace components that are used with an A-profile processor:

- Trace Infrastructure: A set of components that can connect from the optional AMBA Trace Bus (ATB) trace interface of the processor through to the trace capture components.

The timestamp components that are delivered as part of the CoreSight SoC deliverable:

- Timestamp Infrastructure: A set of components that is used to generate and distribute a 64-bit incrementing count value to the trace generation logic.

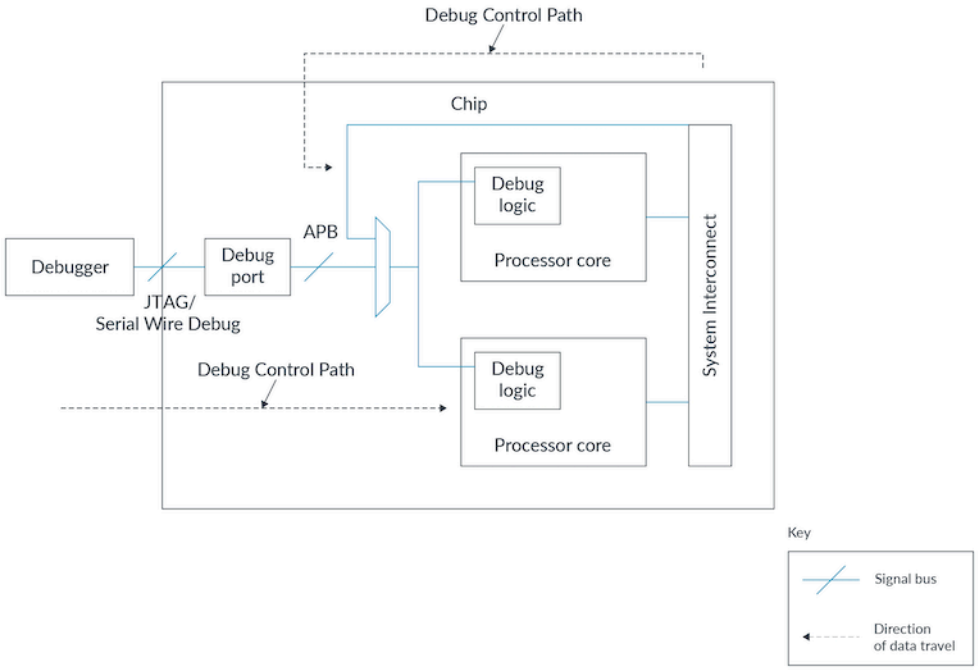
However, the timestamp infrastructure components do not apply to the Armv9-A architecture processors. This is because the Armv9-A architecture uses the software system timer incrementing counter value as the source of the trace generation count value. This means that the timestamp infrastructure components are not relevant to a design that uses Armv9-A trace sources.

# 3. Debug control

A key activity of an external debugger is to read and write the Debug registers inside each Arm processor core. To do this, an on-chip connection for the debugger needs to physically connect to the chip and then generate on-chip transactions that can access the Debug registers inside the processor core. The process is similar if debug is done using a self-hosted mode, with software that is used to control debug activity. In this case, an on-chip connection for the processor core needs to run the debug software, to access the Debug registers of the processor cores and the debug components.

Typically, external debuggers use the JTAG (IEEE 1149.1) or serial wire debug protocols to provide the connection onto the chip. These protocols are used because they limit the number of pins that are required, which helps reduce the physical cost of the debug infrastructure. Once the JTAG or serial wire transactions are on the chip, they are converted into parallel bus transactions based on the AMBA Advanced Peripheral Bus (APB) protocol. These APB transactions are transported through an APB bus network to each of the components that must be accessed by the debugger. The following diagram illustrates the basic debug control connectivity using the APB control network:

**Figure 3-1: A diagram showing the debug control connectivity using the APB control network**




If you are an SoC designer, you can create a suitable infrastructure to access all the required components. Typically, you do this using the CoreSight SoC components to design a suitable debug subsystem.

A typical debugger device controls the JTAG or serial wire transactions and target accesses at the Debug registers inside the device. This means that, when you use a debugger, you only need to decide which debug logic you want to use inside the processor cores to help debug the malfunctioning software program. You do not need to do low-level programming of CoreSight components from the debugger.

If you are an SoC designer, you might decide to provide debug access to the APB protocol bus using functional IOs, instead of using a JTAG or Serial Wire Debug Port to provide the off-chip debug access. That is, you might be able to repurpose some of the device functional connectivity and use that connectivity for debug purposes. The advantage of this approach is that dedicated IOs for debug purposes are not required. Debug could be allowed from a non-standard debugging source, for example, using software routines that are running on a PC and connecting to the device using a USB port. However, this approach requires the development of design-specific debug connectivity and software. This approach is typically followed when an SoC designer is considering a specific debug scenario.

# 4. Debug coordination

A common debug activity is to halt a processor core, by stopping the execution of the program and allowing an external debugger to check the current status of the processor core registers.

Sometimes you might want to halt all the processor cores when one processor core halts, so that you can see the current state of all the system software.

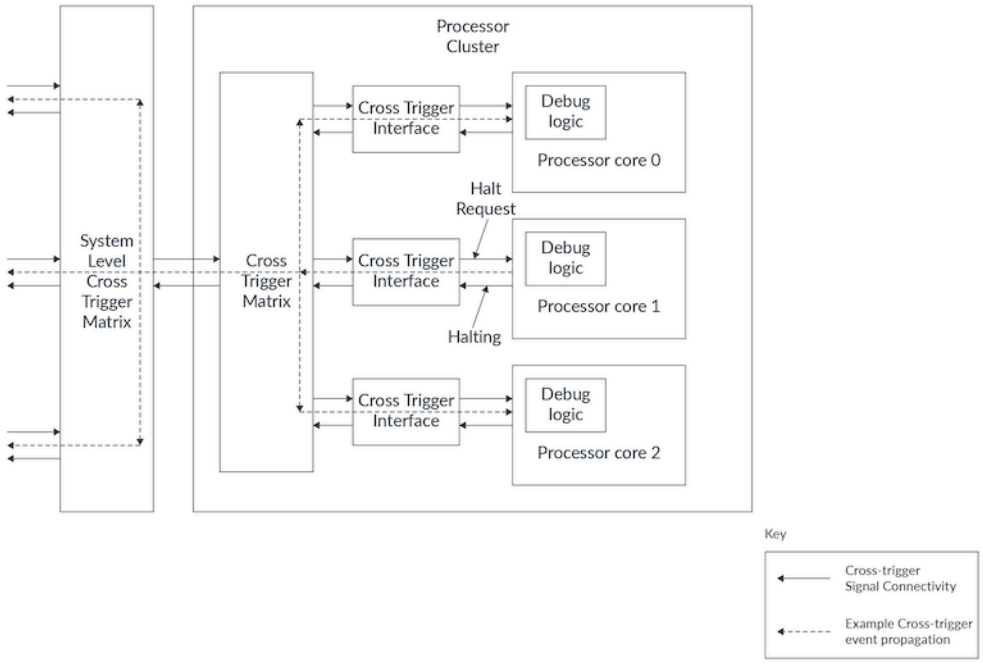
To do this, each processor cluster includes a Cross Trigger Interface (CTI) component for each processor core. The connections between the CTI and the processor core include:

- A signal to tell the processor core to halt for debug activity,
- A signal that indicates that the processor core is halting.

Each CTI also connects into a Cross Trigger Matrix (CTM) component. Activity on the signals between the CTI and the processor core can be propagated between the CTI components through this matrix. The CTI contains programmable registers that can be accessed by the debugger to decide which activity to propagate through the matrix. These registers can be programmed so that when one processor core halts, this halting activity is propagated through the CTM components to all the other CTI components in the system, as you can see in the following diagram. The activity is then forwarded to the signal that is used to tell each processor core to halt. This means that when one processor core halts, all the other processor cores should halt soon after. This behavior is referred to as the cross-halt.

If you are an SoC designer, you connect the CTM inside each cluster to the system level where it connects to a system-level CTM. The connectivity to the system-level CTM is what allows the activity at each CTI to propagate through the entire system. The following diagram illustrates how activity in one processor could be distributed to the other processor cores in the cluster and out into the rest of the system through the Cross Trigger Matrix:

**Figure 4-1: A diagram of the Cross Trigger Matrix**



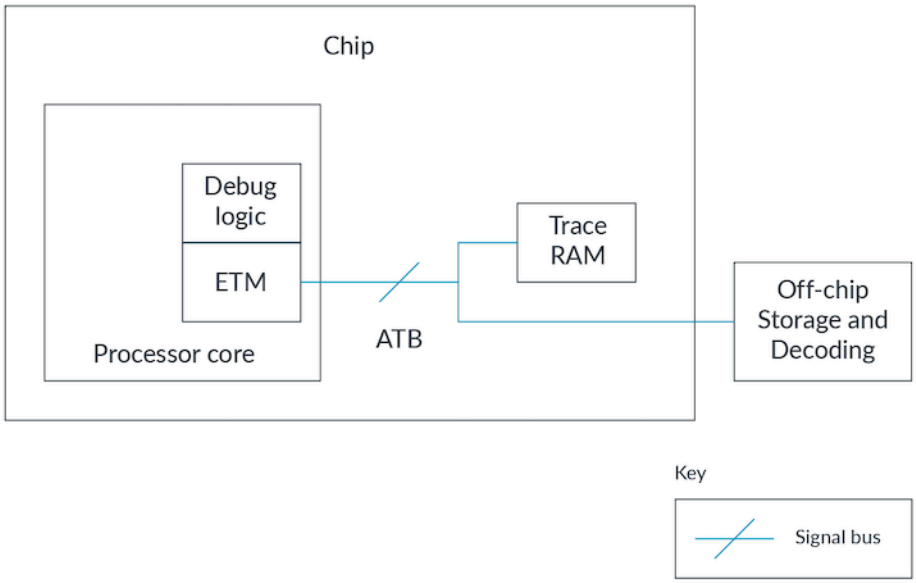

It is likely that a debugger device already knows how to program the CTI registers to achieve this cross-halt behavior. This means that, when using the debugger, you should not need to do low-level programming of the CTI components to achieve the cross-halt. However, the CTI components are not only used for managing the processor halting. The CTI components might be used to generate interrupts or to control trace capture. This means that you might need to tell the debugger how to program the CTI registers to achieve the required behavior.

# 5. CoreSight trace infrastructure

Each processor can be paired with an Embedded Trace Macrocell (ETM), which can generate trace data. This trace data can be captured in an on-chip buffer or output via a dedicated trace bus for off-chip capture.

If you are an SoC designer, you can use this bus for trace capture, so that the trace capture is independent of the functional interconnect. You might decide to capture the trace data on-chip in a dedicated trace RAM instance. Alternatively, you might decide to send the trace data off-chip for decoding and analysis. The following diagram illustrates the concepts of on-chip trace capture and storage in a trace RAM and off-chip trace capture and storage:

**Figure 5-1: A diagram showing the trace capture and storage**




If you are an SoC designer, you can provide the debugger device with the on-chip connectivity information for the CoreSight ATB infrastructure. If the debugger device has access to the on-chip connectivity, it can determine how to program the processor ETM and the CoreSight ATB components to capture and analyze the trace data. This means that, when using the debugger, you should not need to program these components directly using the debugger.

# 6. External debug control infrastructure components

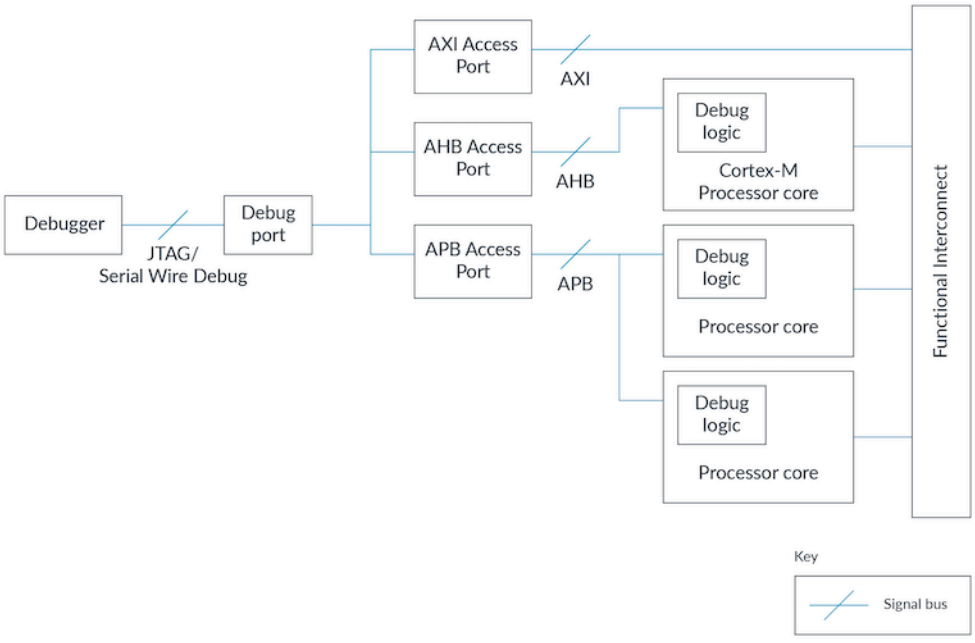
The external debug control infrastructure includes two key pieces:

- Debug port
- Access ports

The following diagram illustrates the concept of connecting multiple access ports to a single debug port to access different parts of an SoC. There are three types of access ports included:

- An Advanced Peripheral Bus (APB) access port for access to an APB network
- An Advanced eXtensible Interface (AXI) access port for accessing a functional subsystem
- An Advanced High-performance Bus (AHB) access port for accessing the debug logic of a Cortex-M processor

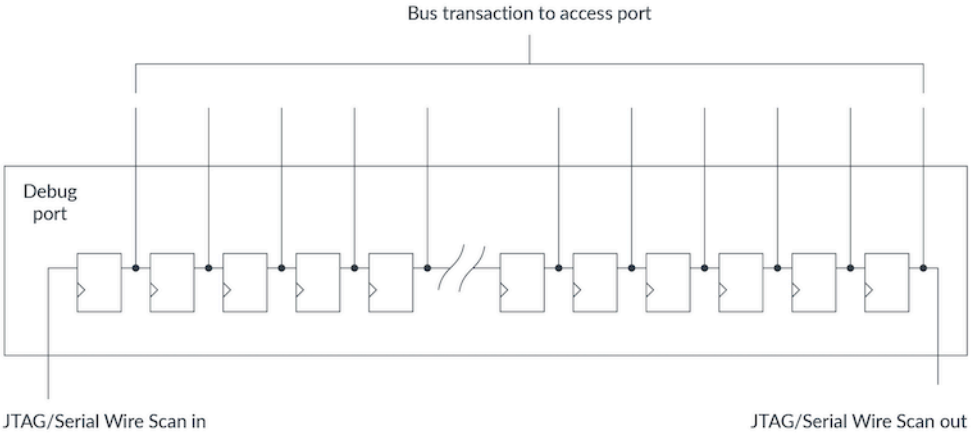
**Figure 6-1: A diagram showing the access ports connected to a single debug port**




A typical debug port has a JTAG or serial wire connection with the debugger. The debugger drives serial scan accesses on the port, which scan values into scan chain registers inside the debug port. The scan chain registers generate a bus transaction that is directed to an access port register.

The following diagram illustrates the concept of converting the serial scan chain accesses of a JTAG or Serial Wire Debug Port into parallel bus accesses, by scanning in a value for each bus signal one bit at a time, and then generating the bus transaction from each of the register bits:

**Figure 6-2: A diagram showing the conversion of serial scan input into parallel bus transactions**




Each access port can drive a different memory space. The type of access port that is used depends on the bus protocol that is required for accesses into each memory space.

The Debug registers inside the processor cores, and the Debug registers for CoreSight components, like the Cross Trigger Interface (CTI) components, are accessed through buses that are based on the APB protocol. This means that an APB access port is used to access the memory space for these debug control registers. If you are an SoC designer, you can set the precise address locations for the registers for each component when you design the CoreSight debug subsystem.

The Debug registers for the Cortex-M processors are accessed through an AMBA Advanced High-performance Bus (AHB) interface. This means that an AHB access port is used for debug accesses into a Cortex-M processor. The connection from an AHB access port into a Cortex-M processor allows the AHB access port to direct accesses to all locations in the Cortex-M functional memory space, not just the Debug registers of the Cortex-M processor. The functional address space of a Cortex-M processor is 4 GB. Each AHB access port can only access a 4 GB address space. This means that each Cortex-M processor core must be accessed by a dedicated AHB access port component.

An AMBA Advanced eXtensible Interface (AXI) access port is typically used for accesses into the main functional memory space of the SoC, by connecting the access port to the main functional interconnect of the device. This provides a path for the debugger to direct accesses to DDR memory and SRAM locations as part of a debugging session.

Another way for the debugger to access memory would be through the processor cores, using either:

- Direct transactions to a Cortex-M processor memory space through the AHB access port, or
- The Debug registers of an A-profile processor core, to instruct the processor core to access the memory space on behalf of the debugger.

Using the processor core to access the memory space means that the read or write could hit on the processor core caches and be controlled by the processor core memory map controls, for

example the Memory Management Unit (MMU). This memory access approach might be useful when debugging software if you want to determine the view of memory that the software is seeing from the processor core. On the other hand, it might be useful for the debugger to access the memory location directly through an AXI access port, bypassing the processor core caches and the processor core memory map controls. This would allow the debugger to confirm the current value of the physical memory location.

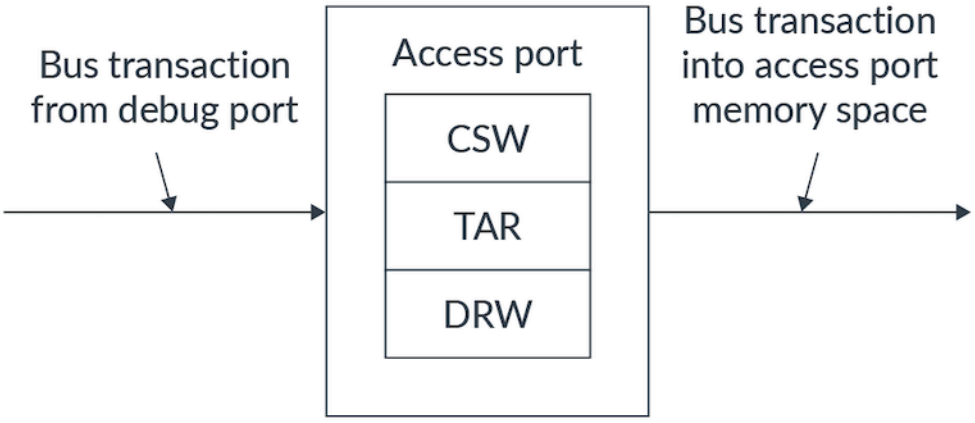
# 7. Access port control

There are three main registers that the external debugger can use to generate accesses into the memory space of an access port:

- Control/Status Word (CSW)
- Transfer Address Register (TAR)
- Data Read/Write (DRW)

The following diagram illustrates the concept of an access port. With an access port, transactions from a debug port are used to write to registers inside the access port. The values that are written to these registers generate bus transactions on the output of the access port:

**Figure 7-1: A conceptual diagram of an access port**




First, the debugger accesses CSW to set up the properties for the bus transaction into the access port address space. These properties include the size of the data transaction, the security of the data transaction, and caching options.

Next, the debugger accesses TAR to set the address for the transaction into the access port memory space.

Finally, the debugger generates a read or write transaction to DRW.

When the debugger does a write transaction, the data that the debugger has written to DRW forms the write data of the write transaction that is sent to the address set in TAR. The properties of the transaction are set by the values in CSW.

When the debugger does a read transaction to DRW, the address in TAR and the properties that are set in CSW generate a read transaction into the access port address space. The data that is returned from the TAR address is sent back to the debug port.

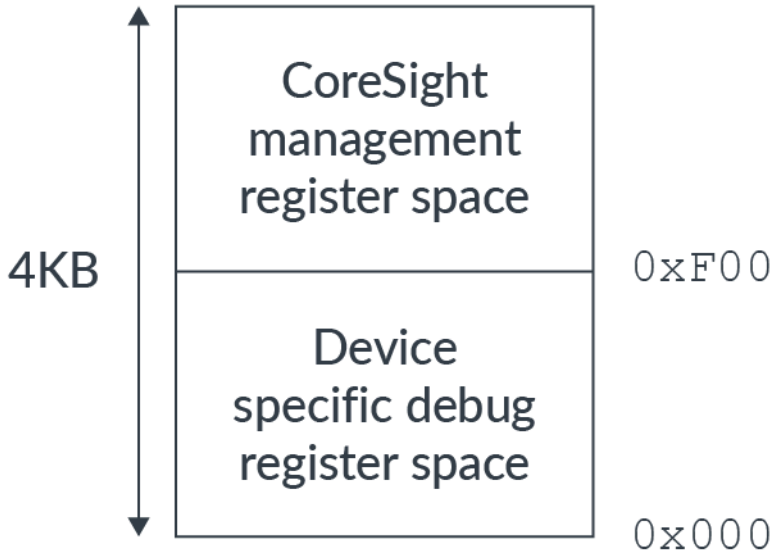
By using the debug port and access ports, the debugger can access the full debug space of the device. If you are a debug tool vendor, you can reference the Arm Debug Interface Architecture Specification for the details of the JTAG/SW debug port and access port component behavior and registers.

# 8. Debug Access Port address space

The debug address space that is accessed by an access port consists of debug control registers for CoreSight-compliant components. A CoreSight-compliant component has a 4 KB set of debug control registers, or a set of debug control registers that is a multiple of 4 KB. Typically, these registers are accessed using an APB interface, though any protocol that can perform device memory type accesses is permissible.

The following diagram illustrates how the 4 KB register space for a CoreSight component is divided into a set of predefined CoreSight Management registers and a set of device specific registers:

**Figure 8-1: A diagram of the register space for a CoreSight component**




The register space address range is divided between CoreSight component management registers and device-specific Debug registers. The Technical Reference Manual (TRM) for a CoreSight SoC product defines the device-specific debug control registers for each of the CoreSight SoC components. The Arm A-profile architecture defines the debug control registers for the processor cores. Any processor core Debug register that is **IMPLEMENTATION DEFINED** is defined in the TRM for that processor core.

The 4 KB set of Debug registers are placed in the address map of the APB access port. The debugger directs accesses to Debug registers for each component to control the debugging behavior.

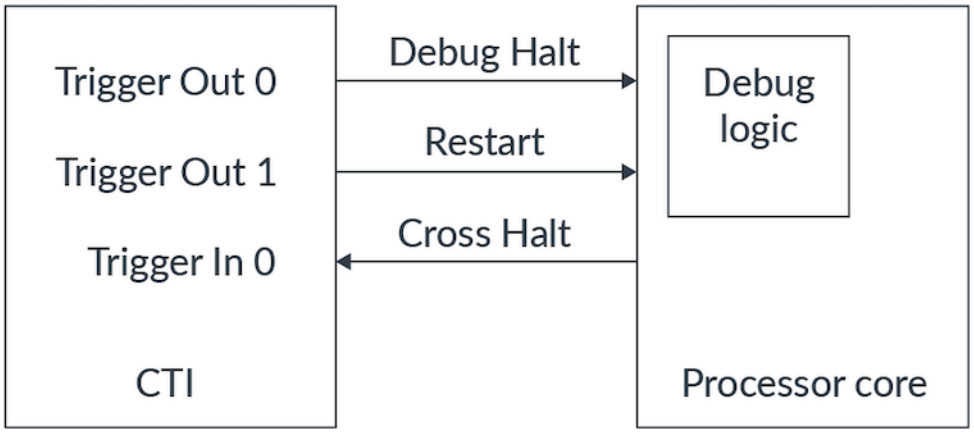
# 9. Cross-trigger components

The cross-triggering infrastructure consists of two components:

- Cross Trigger Interface (CTI)
- Cross Trigger Matrix (CTM)

It is likely that most of the CTI components in an SoC are associated with processor cores and are built into processor clusters. You can see some typical connections between a CTI and a processor core in the following diagram:

**Figure 9-1: A diagram showing connections between a CTI and a processor core**

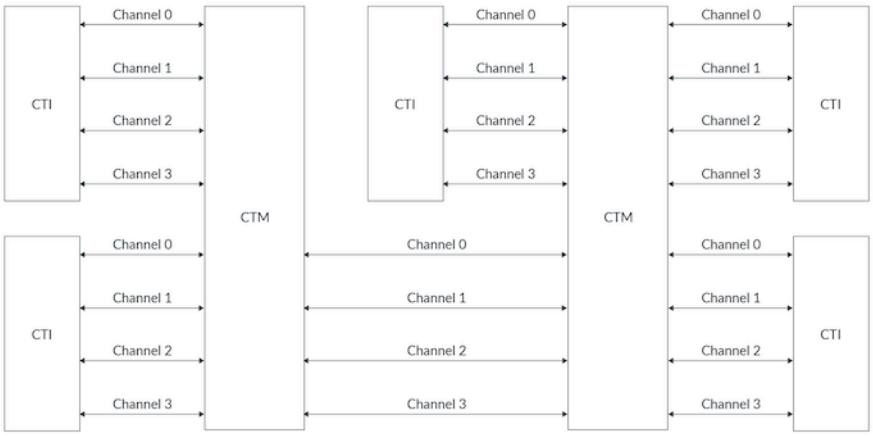



These connections between the CTI and the component are called trigger events. Trigger events are pulses or level-sensitive signals. The Technical Reference Manual for the processor describes the precise trigger event connections between the processor core and the CTI component.

The connections between CTIs through the Cross Trigger Matrix are formed of channels. The matrix consists of four channels: Channel 0, Channel 1, Channel 2, and Channel 3. Each CTI is connected to all four channels.

The following diagram gives an example of how multiple CTI components could be connected using CTM components:

**Figure 9-2: A diagram showing multiple CTI components**




Debug control registers inside the CTI components map trigger event signals to one or more channels. When an input trigger event signal activates, pulses high, or is held high, this activity is propagated along any cross-trigger channel to which the event is mapped. Because the channels connect to all other CTI components, this activity propagates to all other CTI components in the design.

When a channel input to a CTI activates, this activity is propagated to any trigger event output to which the channel is mapped.

This connectivity through the channels makes it possible for activity on a trigger event input in one CTI component to propagate to the trigger event output of another CTI component.

The main CTI registers that are used to program this behavior are shown in the following table:

**Table 9-1: The main CTI registers used to program trigger events**

| CTI register name | Register bit behavior |
|---|---|
| CTICONTROL[31:0] | Bit[0] = 1 : CTI Enable |
| CTIINEN#[31:0]<br><br>(# = Input trigger Number) | Bit[0] = 1 : Map Input trigger # to Channel 0<br><br>Bit[1] = 1 : Map Input trigger # to Channel 1<br><br>Bit[2] = 1 : Map Input trigger # to Channel 2<br><br>Bit[3] = 1 : Map Input trigger # to Channel 3 |

| CTI register name | Register bit behavior |
|---|---|
| CTIOUTEN#[31:0]<br><br>(# = Output trigger Number) | Bit[0] = 1 : Map Output trigger # to Channel 0<br><br>Bit[1] = 1 : Map Output trigger # to Channel 1<br><br>Bit[2] = 1 : Map Output trigger # to Channel 2<br><br>Bit[3] = 1 : Map Output trigger # to Channel 3 |
| CTIGATE[31:0] | Bit[0] = 1 : Enable event propagation on Channel 0<br><br>Bit[1] = 1 : Enable event propagation on Channel 1<br><br>Bit[2] = 1 : Enable event propagation on Channel 2<br><br>Bit[3] = 1 : Enable event propagation on Channel 3 |

Refer to the Technical Reference Manual for your processor or CoreSight SoC product for a more detailed description of these registers.

# 10. Programming the cross-halt

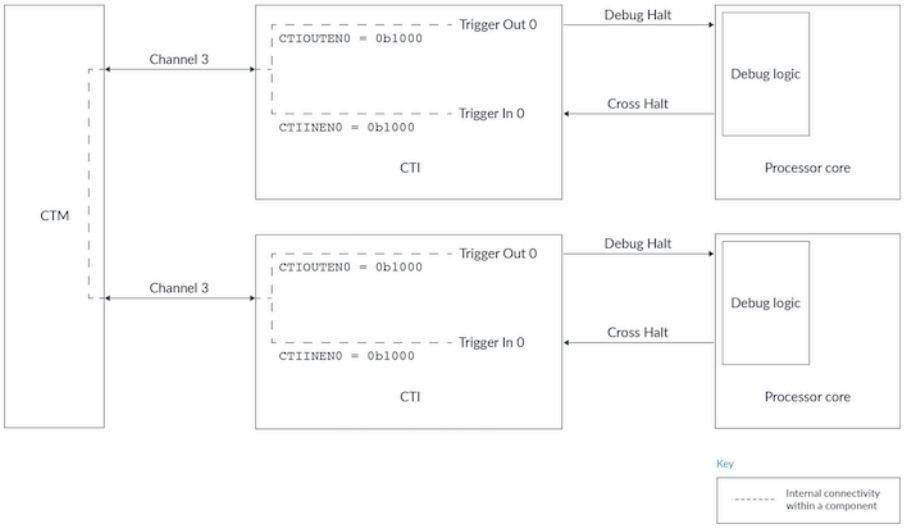
Cross-halt behavior happens when one processor core enters the Halting debug state, which triggers all the processor cores to enter the Halting debug state. This is a common situation that is encountered when debugging a system.

To generate the cross-halt behavior, the debugger must make use of the Debug Halt and Cross Halt trigger events. The Debug Halt event is the signal to the processor core to enter the debug state. The Cross Halt event is the signal from the processor core that it is entering the debug state. To achieve the cross-halt behavior, the debugger maps the Debug Halt event and Cross Halt event signals for every processor core in the system to the same cross-trigger channel. For example, if the debugger mapped the events to Channel 3 it would program the registers as follows:

- Enable the CTI: CTICONTROL = `0b1`
- Map input trigger 0 to Channel 3: CTIINEN0 = `0b1000`
- Map output trigger 0 to Channel 3: CTIOUTEN0 = `0b1000`
- Enable event propagation on Channel 3: CTIGATE = 0b1XXX

The following diagram illustrates how Trigger Inputs 0 and Trigger Outputs 0 for multiple CTI components can be mapped to channel 3 of the Cross Trigger Matrix:

**Figure 10-1: A diagram showing mapping of the Cross Trigger Matrix**




The details of all the CTI Debug registers are found in the Technical Reference Manual for the CoreSight SoC product.

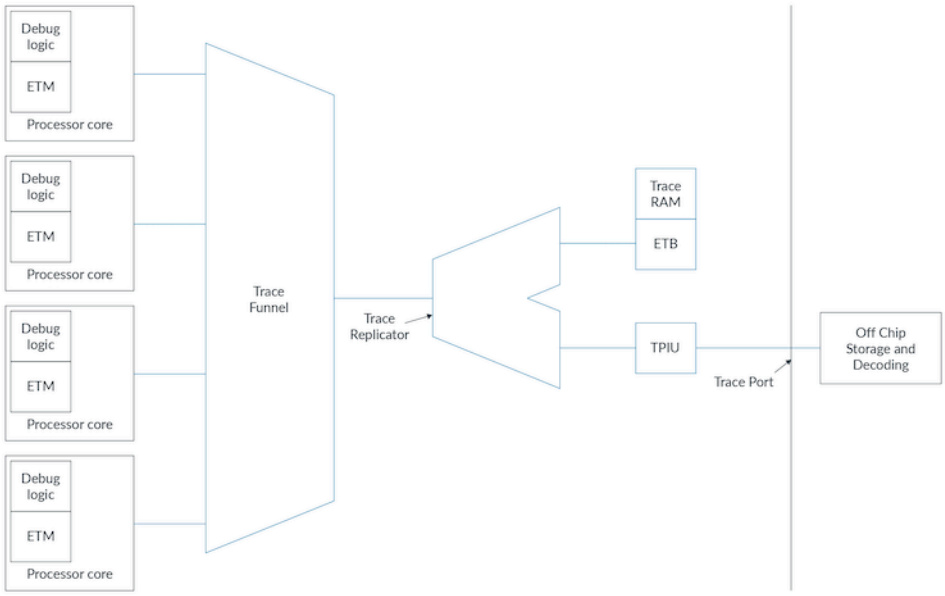
# 11. ATB trace capture components

The ATB trace capture is typically achieved using multiple components, including:

- Trace funnel
- Trace replicator
- Trace Port Interface Unit (TPIU)
- Embedded Trace Buffer (ETB)

The following diagram illustrates some of the typical components that are used in a trace distribution and capture infrastructure:

**Figure 11-1: A diagram showing typical components used in trace/capture**




Each trace bus for a processor core typically has a data width of 32 bits. However, during normal operation, the ETM only generates an ATB transaction when it has enough trace data to fill the full width of the trace bus. This means that there might be many cycles while the trace bus of a processor core is inactive, while the ETM gathers enough trace data to generate a new transaction.

The presence of these idle cycles means that the trace buses from each processor core can be merged into single trace bus using a funnel component. The funnel component arbitrates between the different trace sources. When multiple trace sources attempt to send a transaction in the same cycle, it accepts the transaction from one source and stalls the other transactions. Merging the trace streams into a single bus reduces the area cost of the trace infrastructure.

If you are an SoC designer, you can decide to store the trace data on the chip itself inside a dedicated trace RAM. Alternatively, you can send the trace data directly off-chip through a trace port, where it can be captured and analyzed directly by a trace port analyzer.

There are advantages and disadvantages to both approaches. This means that some designs might include multiple methods of storing and capturing the trace data. When multiple trace targets are included, the trace data packets need a path to all the trace capture targets that are included in the design. This means that a trace replicator component is included in the trace capture path to drive the single ATB bus to two destinations. The trace replicator can be programmed to send all the trace data to both destinations. Alternatively, the trace replicator can be programmed to send trace data from different trace sources to different destinations.

The standard trace component for sending trace data off-chip is the Trace Port Interface Unit (TPIU). This converts the ATB transactions into a format that can be sent off-chip through a trace port. Trace capture hardware can be connected to the trace port to capture and decode the trace for analysis. A trace port typically has low bandwidth compared to the on-chip trace bus. If a typical trace port is 16 bits wide running at 200 MHz DDR, the on-chip trace data bus might be up to 128 bits wide and potentially run at GHz frequencies. This means that the TPIU component is suitable for debug scenarios that only generate a low bandwidth of trace, but where the debug scenario might require many hours of trace capture.

There are multiple trace components that can be used for capturing trace on-chip, including:

- Embedded Trace Buffer (ETB) for capture to a dedicated SRAM
- Embedded Trace Router (ETR) for capture to a shared or dedicated AXI slave, including system DDR

Both the ETB and the ETR convert the ATB transaction data into a format that can be stored in a RAM for later analysis. The data in the RAM is read out by a debugger through the APB debug interface of the trace component, and is then decoded and analyzed.

An ETB can support high-bandwidth trace, and typically runs at the same frequency as the other ATB trace components. However, the amount of trace data that can be captured into the ETB RAM is limited by the RAM size. This means that the ETB component is suitable for debug scenarios that can generate high bandwidth trace data but only over a short period of time.

An ETR also supports high-bandwidth trace capture, but can potentially capture far more trace data into the system SRAM or DDR when the memory region is not in use by the system.

Details of all the trace infrastructure component registers can be found in the Technical Reference Manual for the CoreSight SoC product.

# 12. Check your knowledge

Q: What two general methods can be used for debug control using a CoreSight subsystem?

*A: On-chip self-hosted debug and off-chip external debug*

Q: What is the general purpose of the CoreSight components?

*A: To create a CoreSight infrastructure to access the Arm debug and trace capabilities*

Q: What CoreSight infrastructure subsystems would you expect to see for an A-profile processor subsystem?

*A: Debug APB control, cross-trigger network, and, optionally, an ATB trace capture network*

Q: What is the primary function of a JTAG/Serial Wire Debug Port?

*A: To convert serial scan transactions into bus transactions*

Q: When configuring the cross-triggers, does the user program the CTI components, the CTM components, or both?

*A: CTI components only. CTM components are not programmable.*

Q: What component would you use to capture ATB trace if you were expecting low bandwidth trace generation over a long period?

*A: A Trace Port Interface Unit (TPIU)*

# 13. Related information

Here are some resources related to material in this guide:

- Arm community - Ask development questions, and find articles and blogs on specific topics from Arm experts
- Arm Debug Interface Architecture Specification
- CoreSight Architecture Specification
- CoreSight SoC-400 Technical Reference Manual
- CoreSight SoC-600 Technical Reference Manual

# 14. Next steps

This guide introduced the concept of the CoreSight infrastructure as a mechanism to control and access the Arm processor debug and trace functionality. We discussed the CoreSight control infrastructure that is used to program the CoreSight components, the CoreSight cross trigger infrastructure that is used to coordinate debug activity, and the CoreSight ATB infrastructure that is used to distribute trace data across the chip.

The guide has examined details of each of these infrastructures, to identify use cases for each infrastructure, and to explain why you might include components in the CoreSight subsystem for your design.

To learn more about the details of the CoreSight components that are used in your device, consult the relevant technical reference manual for your device. These manuals are listed in Related information.

# Proprietary Notice

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

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

The validity, construction and performance of this notice shall be governed by English Law.



Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

# Product and document information

Read the information in these sections to understand the release status of the product and documentation, and the conventions used in Arm documents.

## Product status

All products and services provided by Arm require deliverables to be prepared and made available at different levels of completeness. The information in this document indicates the appropriate level of completeness for the associated deliverables.

### Product completeness status

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

### Product revision status

The r1p1 identifier indicates the revision status of the product described in this manual, where:

***rx*** Identifies the major revision of the product.
***py*** Identifies the minor revision or modification status of the product.

## Conventions

The following subsections describe conventions used in Arm documents.

### Glossary

The Arm Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the Arm Glossary for more information: developer.arm.com/glossary.

### Typographic conventions

Arm documentation uses typographical conventions to convey specific meaning.

| Convention | Use |
|---|---|
| italic | Citations. |
| **bold** | Terms in descriptive lists, where appropriate. |
| `monospace` | Text that you can enter at the keyboard, such as commands, file and program names, and source code. |
| `monospace underline` | A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name. |

| Convention | Use |
|---|---|
| <and> | Encloses replaceable terms for assembler syntax where they appear in code or code fragments.<br><br>For example:<br><br>`MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>` |
| SMALL CAPITALS | Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, **IMPLEMENTATION DEFINED**, **IMPLEMENTATION SPECIFIC**, **UNKNOWN**, and **UNPREDICTABLE**. |


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


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


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


This information is important and needs your attention.


This information might help you perform a task in an easier, better, or faster way.

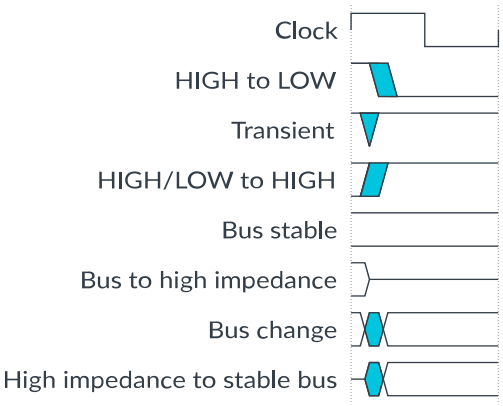

This information reminds you of something important relating to the current content.

## Timing diagrams

The following figure explains the components used in timing diagrams. Variations, when they occur, have clear labels. You must not assume any timing information that is not explicit in the diagrams.

Shaded bus and signal areas are undefined, so the bus or signal can assume any value within the shaded area at that time. The actual level is unimportant and does not affect normal operation.

**Figure 1: Key to timing diagram conventions**




## Signals

The signal conventions are:

**Signal level**

The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means:

- HIGH for active-HIGH signals.
- LOW for active-LOW signals.

**Lowercase n**

At the start or end of a signal name, n denotes an active-LOW signal.

# Useful resources

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

Arm documents are available on developer.arm.com/documentation.

Confidential documents are only available to licensees, when logged in. Each document link in the tables below provides direct access to the online version of the document.