ARM® CoreSight™ Architecture Specification

v2.0

ARM®

ARM IHI 0029D (ID092613)

ARM CoreSight Architecture Specification
v2.0

Release Information

The *Change history* table lists the changes made to this document.

Change history

Date	Issue	Confidentiality	Change
29 September 2004	A	Non-Confidential	First release for v1.0.
24 March 2005	B	Non-Confidential	Second release for v1.0. Editorial changes and clarifications.
27 March 2012	C	Confidential	Limited beta release for v2.0.
26 September 2013	D	Non-Confidential	First release for v2.0.

Non-Confidential Proprietary Notice

ARM Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

Product Status

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

Web Address

`http://www.arm.com`

Contents
ARM CoreSight Architecture Specification v2.0

Preface

This preface introduces the *CoreSight Architecture Specification*. It contains the following sections:

About this specification

This specification describes the CoreSight architecture that all versions of the CoreSight compliant cores, components, platforms, and systems use.

Intended audience

This specification is written for the following target audiences:

- Hardware engineers integrating CoreSight components into a CoreSight system.
- Hardware engineers designing CoreSight components.
- Software engineers writing development tools providing support for CoreSight system functionality.
- Designers of debugging hardware used to connect to a CoreSight system, such as JTAG or SWD emulators and Trace Port Analyzers.
- Advanced users of development tools providing support for CoreSight functionality.

This specification does not document the behavior of individual components unless they form a fundamental part of the architecture.

It is recommended that all users of this specification have experience of the ARM architecture.

Using this specification

The information in this specification is organized into parts, as described in this section.

Part A, CoreSight Architecture

Part A contains an introduction to the CoreSight architecture. It contains the following chapter:

Chapter A1 *Introduction*

Read this for an outline description of the components, memory maps, clock and reset requirements, system integration, and the test interface.

Part B, CoreSight Visible Component Architecture

Part B describes the CoreSight visible component architecture, that must be implemented by all CoreSight components that are visible to a debugger. It contains the following chapters:

Chapter B1 *Visible Component Architecture*

Read this for a general description of the visible component architecture.

Chapter B2 *CoreSight Programmers' Model*

Read this for a description of the CoreSight technology programmers' model.

Chapter B3 *Topology Detection Registers*

Read this for a description of the topology detection registers in CoreSight systems.

Part C, CoreSight Reusable Component Architecture

Part C describes the CoreSight reusable component architecture, that must be implemented by CoreSight components so that they can be used with other CoreSight components. It contains the following chapters:

Chapter C1 *Reusable Component Architecture*

Read this for a general description of the reusable component architecture.

Chapter C2 *AMBA APB and ATB Interfaces*

Read this for a description of the *Advanced Microcontroller Bus Architecture* (AMBA®) 3 *Advanced Peripheral Bus* (APB) interface and the *Advanced Trace Bus* (ATB) interface.

Chapter C3 *Event Interface*

Read this for a description of the event interface.

Chapter C4 *Channel Interface*

Read this for a description of the channel interface.

Chapter C5 *Authentication Interface*

Read this for a description of the authentication interface.

Chapter C6 *Timestamp Interface*

Read this for a description of the timestamp interface.

Chapter C7 *Topology Detection at the Component Level*

Read this for a description of topology detection at the component level.

Part D, CoreSight System Architecture

Part D describes the CoreSight system architecture, that must be implemented by all CoreSight systems, and provides information required by debuggers to enable them to use CoreSight systems. It contains the following chapters:

Chapter D1 *System Architecture*

Read this for a general description of the CoreSight system architecture.

Chapter D2 *System Design*

Read this for a description of system design with the CoreSight system architecture.

Chapter D3 *Physical Interface*

Read this for a description of the physical interface for CoreSight connection to a debugger.

Chapter D4 *Trace Formatter*

Read this for a description of the CoreSight trace formatter.

Chapter D5 *ROM Table*

Read this for a description of the CoreSight ROM table.

Chapter D6 *Topology Detection at the System Level*

Read this for a description of topology detection at the system level.

Chapter D7 *Compliance Criteria*

Read this for a description of the criteria that systems must comply with to satisfy CoreSight requirements.

Part E, Appendices

This specification contains the following appendices:

Appendix A *Power Requestor*

Read this for a description of the power requestor.

Appendix B *Revisions*

Read this for a description of the technical changes between released versions of this specification.

Glossary Read this for definitions of some terms used in this specification. The glossary does not contain terms that are industry standard unless the ARM meaning differs from the generally accepted meaning.

Note

ARM publishes a single glossary that relates to most ARM products, see the *ARM® Glossary* http://infocenter.arm.com/help/topic/com.arm.doc.aeg0014- . A definition in the glossary in this specification might be more detailed than the corresponding definition in the *ARM® Glossary*.

Conventions

The following sections describe conventions that this book can use:

- *Typographic conventions*
- *Signals*
- *Timing diagrams*
- *Numbers* on page xiv.

Typographic conventions

The typographical conventions are:

italic Introduces special terminology, and denotes citations.

bold Denotes signal names, and is used for terms in descriptive lists, where appropriate.

`monospace` Used for assembler syntax descriptions, pseudocode, and source code examples.

Also used in the main text for instruction mnemonics and for references to other items appearing in assembler syntax descriptions, pseudocode, and source code examples.

SMALL CAPITALS

Used for a few terms that have specific technical meanings, and are included in the *Glossary*.

Colored text Indicates a link. This can be:

- A URL, for example `http://infocenter.arm.com`.
- A cross-reference, that includes the page number of the referenced information if it is not on the current page, for example, *Numbers* on page xiv.
- A link, to a chapter or appendix, or to a glossary entry, or to the section of the document that defines the colored term, for example *Embedded Trace Buffer (ETB)* or DEVARCH.

Signals

In general this specification does not define processor signals, but it does include some signal examples and recommendations. 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:

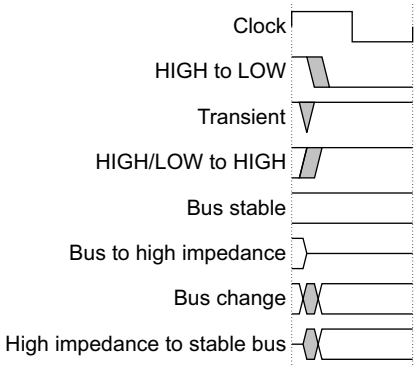
- HIGH for active-HIGH signals.
- LOW for active-LOW signals.

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

Timing diagrams

The figure named *Key to timing diagram conventions* on page xiv 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.



Key to timing diagram conventions

Numbers

Numbers are normally written in decimal. Binary numbers are preceded by `0b`, and hexadecimal numbers by `0x`. In both cases, the prefix and the associated value are written in a `monospace` font, for example `0xFFFF0000`.

Additional reading

This section lists relevant publications from ARM and third parties.

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

ARM publications

This specification contains information that is specific to CoreSight. See the following documents for other relevant information:

- *ARM® CoreSight™ Components Technical Reference Manual* (ARM DDI 0314).
- *ARM® CoreSight™ SoC Technical Reference Manual* (ARM DDI 0480).
- *ARM® CoreSight™ Technology System Design Guide* (ARM DGI 0012).
- *ARM® Embedded Trace Macrocell Architecture Specification* (ARM IHI 0014).
- *ARM® ETM™ Architecture Specification, ETMv4* (ARM IHI 0064).
- *ARM® Debug Interface Architecture Specification, ADIv5.0 to ADIv5.2* (ARM IHI 0031).
- *ARM® Architecture Reference Manual ARMv7-A and ARMv7-R edition* (ARM DDI 0406).
- *ARM® Architecture Reference Manual ARMv8, for ARMv8-A architecture profile* (ARM DDI 0487).
- *ARM® AMBA® AXI and ACE Protocol Specification AXI3, AXI4, and AXI4-Lite, ACE and ACE-Lite* (ARM IHI 0022).
- *ARM® AMBA® APB Protocol Specification* (ARM IHI 0024).
- *ARM® AMBA® ATB Protocol Specification* (ARM IHI 0032).

Other publications

This section lists relevant documents published by third parties:

- IEEE, *Standard Test Access Port and Boundary Scan Architecture*, IEEE Std 1149.1-1990.
- JEDEC, *Standard Manufacturer's Identification Code*, JEP106.

Feedback

ARM welcomes feedback on its documentation.

Feedback on this book

If you have comments on the content of this book, send e-mail to `errata@arm.com`. Give:

- The title.
- The number, ARM IHI 0029D.
- The page numbers to which your comments apply.
- A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Note

ARM tests PDFs only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the appearance or behavior of any document when viewed with any other PDF reader.

Part A
CoreSight Architecture

Chapter A1
Introduction

This chapter introduces the CoreSight architecture. It contains the following sections:

A1.1 About the CoreSight architecture

The CoreSight architecture provides a system-wide solution to real-time debug and trace. It recognizes:

- The requirement for multi-core debug and trace.
- The requirement to debug and trace the whole system beyond the core, for example buses.
- The requirement to share resources, such as pins and trace storage, between debug and trace components, to reduce silicon costs.
- The requirement for debug and trace components from multiple vendors to work together.
- The requirement to minimize pin count.
- The requirement to support increasing trace bandwidth from many sources.
- That many trace solutions already exist for a variety of purposes, and that these trace protocols cannot be rewritten to support a new trace architecture.
- The requirement for development tools to identify and configure themselves for different systems automatically.
- The requirement to control access to debug and trace functionality in the field.
- That the clock and power to parts of the system can be varied or disabled independently, and that this must not prevent the rest of the system from being debugged.
- That the time available to design in debug and trace functionality is often limited and the number of options must be minimized where possible.
- The requirement for debug monitors and other on-chip software to have access to the same debug and trace functionality as an external debugger.
- That systems are often built out of a hierarchy of reusable platforms, where each level must hide the complexities within it and designers cannot change this when they use this platform in another system.

The CoreSight architecture maintains the traditional requirements of debug and trace:

- To access debug functionality without software interaction.
- To connect to a running system without performing a reset.
- To perform certain operations, such as real-time tracing, completely non-invasively, with no effect on the behavior of the system.
- To access non-invasive functionality non-invasively.
- To minimize power consumption of debug logic when not in use.
- To capture trace over a large period of time, so that only the most recent trace is available.

The CoreSight architecture is embodied in a set of CoreSight components and compliant processors that form CoreSight systems. You can use this architecture to design additional CoreSight components.

A1.2 Structure of the CoreSight architecture

The CoreSight architecture comprises:

- A visible component architecture.
- A reusable component architecture.
- A system architecture.

For the design rationale of the CoreSight architecture see *Component access using memory-mapped interfaces instead of JTAG* on page A1-25. This explains why, for example, CoreSight uses memory-mapped access to the registers of CoreSight components.

A1.2.1 Visible component architecture

The visible component architecture specifies aspects of a component that are visible to the programming interface and to tools that access the device. All CoreSight components must comply with the visible component architecture. The visible component architecture specifies:

- The requirements of the programmers' model that all CoreSight components must conform to.
- The requirements for topology detection that enable discovery of the component layout.

For details of the visible component architecture see part B of this specification.

A1.2.2 Reusable component architecture

The reusable component architecture specifies the physical interface of a component. The reusable architecture provides rules that components must comply with so that they work correctly with other reusable components. All CoreSight components conform to the rules of the reusable architecture. If a component does not comply with the rules in the reusable component architecture you might not be able to integrate it with other CoreSight components and you might also have problems with tool compatibility during topology detection. The reusable component architecture specifies:

- The AMBA 3 APB interface for access to the registers in CoreSight components.
- The AMBA ATB interface for trace data transfer between CoreSight components.
- The channel interface for the communication of trigger events between CoreSight components.
- The authentication interface for control of access for debug.
- Topology detection infrastructure that specifies the signals that must be controlled at each interface.

You can create a homogeneous component, that performs a number of functions internally, as separate components, but you only have to present one set of the reusable component interfaces to enable integration into a larger CoreSight system.

Self-contained systems that implement only the visible component architecture are compatible with development tools, but cannot be used with other CoreSight components.

For details of the reusable component architecture, see part C of this specification.

A1.2.3 System architecture

The system architecture specifies:

- System level requirements for:
  - Clock and power domains visible to debuggers.
  - Control of the authentication interface.
  - Distinction between internal and external accesses through the AMBA 3 APB interface memory map.
- The physical interface.
- The trace formatter. See *CoreSight component types* on page A1-22 and Chapter D4 *Trace Formatter*.
- The ROM table. See *CoreSight component types* on page A1-22 and Chapter D5 *ROM Table*.
- The topology detection.
- Compliance criteria for CoreSight systems.

For details of the system architecture, see part D of this specification.

A1.3 CoreSight component types

The CoreSight architecture is embodied as a set of components from which you can implement specific SoC subsystems for debug and trace. The CoreSight components in this section are examples. You can build additional components based on the architecture.

Note

A CoreSight component is a component that implements the CoreSight visible component architecture. The *Debug Access Port* (DAP) is not a CoreSight component, but provides access to the CoreSight components.

The main elements are:

Control components

CoreSight systems can include the following control components:

- *Embedded Cross Trigger* (ECT). The ECT includes:
  - A *Cross Trigger Interface* (CTI).
  - A *Cross Trigger Matrix* (CTM).

Trace sources

CoreSight systems can include the following trace sources:

- *Embedded Trace Macrocells* (ETMs).
- *AMBA Trace Macrocells*.
- *Program Flow Trace Macrocells* (PTMs).
- *System Trace Macrocells* (STMs).

Trace links CoreSight systems can include the following trace links:

- Trace Funnels.
- Replicators.
- ATB bridges.

Trace sinks CoreSight systems can include the following trace sinks:

- *Trace Port Interface Units* (TPIUs).
- *Embedded Trace Buffers* (ETBs).
- *Trace Memory Controllers* (TMCs).

Each trace sink can include a Trace Formatter.

Debug Access Port

The DAP is not a CoreSight component but provides access to CoreSight components and other system features. The DAP can include:

- An *APB Access Port* (APB-AP).
- An *AHB Access Port* (AHB-AP).
- An *AXI Access Port* (AXI-AP).
- A *JTAG Access Port* (JTAG-AP).
- A *Serial Wire JTAG Debug Port* (SWJ-DP).
- A *JTAG Debug Port* (JTAG-DP).
- A ROM Table.

For more information on specific components see the appropriate Technical Reference Manual.

A1.4 CoreSight topology detection

CoreSight systems are defined at three levels:

- A visible component architecture.
- A reusable component architecture.
- A system architecture

The infrastructure for topology detection is reflected at each of these levels.

You can connect CoreSight components together in many different ways, depending on the requirements of the system. Debuggers must be able to detect how you have connected the components. This process is called topology detection. For details see Chapter D6 *Topology Detection at the System Level*.

CoreSight systems can have a number of interface types, as masters or slaves, and each CoreSight component specifies which interfaces are present. The debugger probes each interface to determine which other components are connected to it.

Each interface type defines which signals must be controllable by the master and slave interfaces, and how the debugger can determine the connectivity using these signals. These signals are referred to as topology detection signals. For the specification of the requirements for standard interfaces used by ARM CoreSight components see Chapter C7 *Topology Detection at the Component Level*. Interface vendors must define the requirements for other interfaces, following the rules in Chapter D6 *Topology Detection at the System Level*.

A1.4.1 Basic topology detection infrastructure

This section describes the topology detection infrastructure in a bottom up fashion, from the visible component level to the system level.

At the visible component architecture level a CoreSight system provides topology detection registers. These registers are accessible through the programmers' model and contain information about the components in the system and permit a debugger to identify the components. See Chapter B3 *Topology Detection Registers*.

At the reusable component architecture level the system defines interfaces that enable communication between the various components and enable debuggers access to the system. See Chapter C7 *Topology Detection at the Component Level*.

At the system level there is:

- A ROM table that contains the address map for the CoreSight system. See Chapter D5 *ROM Table*.
- A description of physical connections for the debugger hardware. See Chapter D3 *Physical Interface*.

There are registers to control the wires where buses exist, and enough of the system must be controllable to establish the existence of the link, for example, for ATB interface signals only **ATVALID** and **ATREADY** need to be controlled.

A1.4.2 Mechanism for topology detection

Topology detection is only required when the debugger does not already have information about the system being debugged.

Before it performs topology detection the debugger must determine which components are present in the system. It:

- Connects to the physical interface. See Chapter D3 *Physical Interface*.
- Establishes communication with the system, for example through the DAP.
- Uses the ROM table to determine which components are present.

The debugger then:

- Uses information about each component type to determine which interfaces are present on each component and how to access signals on these interfaces.
- Uses information about each interface type to determine which signals to access. Chapter C7 *Topology Detection at the Component Level* describes how to perform topology detection for each of the CoreSight interfaces.

- Uses the algorithm in Chapter D6 *Topology Detection at the System Level* to perform topology detection. This asserts and deasserts signals on each master interface in turn to check each slave interface and determine where interfaces are connected together.
- Resets the system and saves the description.

A1.5 Component access using memory-mapped interfaces instead of JTAG

CoreSight components must implement a memory-mapped interface, for example AMBA 3 APB interfaces, to provide access to their control registers.

Many existing debug components, that include ARM debug components, use JTAG to access their functionality. The reasons for using a memory-mapped interface are:

- Synchronous JTAG interfaces can cause problems. For example:
  - If a device is powered down, its JTAG controller stops and this causes all devices in the chain to be inaccessible.
  - The JTAG clock, **TCK**, must run at the lowest common speed that all devices on the chain support, that can limit the speed of access to some devices. In particular, if one of the devices is put into a power saving mode where its clock runs very slowly, access to all devices is extremely slow.
- Asynchronous JTAG interfaces on every device are expensive:
  - If there are many more debug components on the chip than just the processor cores, some might be combined in successive layers of platform hierarchy. The implementation complexity of providing a separate debug power domain that permeates all of these components is substantial.

    For power saving reasons, a separate power domain must often be implemented anyway, but this is only for very large blocks such as processor trace, and only at low process geometries.
  - This requires the hardware design to support asynchronous internal interfaces on every debug component. Because many debug components are small, a high number of asynchronous interfaces can cause the design to become large and complex.
- A memory-mapped interface is preferred over JTAG interfaces in general:
  - In a daisy-chained model, the time taken for scans to the JTAG instruction register is proportional to the number of devices in the chain. A system with a lot of devices is slowed down.
  - JTAG is not accessible to system software. Traditionally debug components have had to implement an AHB or coprocessor interface in addition to the JTAG interface to enable software access. This leads to increased logic, inconsistent programmers' models between the interfaces, and different ways to resolve access conflicts between the two interfaces.
  - JTAG auto detection is often difficult. For example, determining the length of the instruction registers is not possible if several TAP controllers are in a chain, unless they either all follow an ARM convention or they follow a non-standard trial and error technique. The IDCODE register is not always correctly implemented. The CoreSight programmers' model enables you to determine which components are in a system.
  - JTAG on each component implies a JTAG interface at the chip boundary for maximum efficiency. However JTAG is no longer the single choice for a system, because other lower-cost interfaces are available. The choice of interface depends on the choice of DAP and is not mandated by the processor.
  - JTAG does not support flow control or error response natively, and requires each component to individually implement a mechanism for this. A memory-mapped interface, such as the AMBA 3 APB interface, can provide native support, removing the need for polling and other mechanisms that are expensive in bandwidth.
- A memory-mapped interface is preferred over coprocessor access:
  - More efficient software access of the debug register file. Because it is now memory-mapped, there is no requirement for separate instructions in a monitor to access each register, as is required when using coprocessor access. This reduces code size.
  - Debug functionality can be controlled from multiple cores, rather than being restricted to the core being debugged.

Part B

CoreSight Visible Component Architecture

Chapter B1
Visible Component Architecture

This chapter describes the visible component architecture used in CoreSight systems. It contains the following section:

- *About the visible component architecture* on page B1-30.

B1.1 About the visible component architecture

The visible component architecture specifies aspects of components that are visible to the programming interface and to tools that access the device. The visible component architecture specifies the programmers' model and requirements for topology detection.

The programmers' model specifies various registers for the identification and control of the component.

The topology detection registers provide the means for the process of topology detection in the CoreSight system.

Chapter B2
CoreSight Programmers' Model

This chapter describes the CoreSight programmers' model. It contains the following sections:

B2.1 About the programmers' model

This chapter defines the standard set of registers that every CoreSight component must implement, in addition to the control registers specific to that component. Some of these registers are optional and must read as zero if not implemented.

This section also explains how software can use integration registers to determine the topology of a CoreSight system.

The basic register structure is taken from the Peripheral ID Register structure defined for ARM CoreLink components. This defines two conceptual identification registers:

- A *component identification register*, that indicates that the identification registers are present. It extends the original CoreLink specification by including a component class, that can indicate that additional registers are present.
- A *peripheral identification register* that uniquely identifies the component.

The remainder of the programmers' model depends on the component classes. The valid classes are:

`0x1` ROM table.

`0x9` CoreSight component.

`0xF` CoreLink component or system component.

Note

CoreLink components were previously called PrimeCell components.

The following sections describe the programmers' model requirements of the different component classes:

- *Component and Peripheral Identification registers* on page B2-36.
- *Class 0x1 ROM table* on page B2-39.
- *Class 0x9 CoreSight component* on page B2-40.
- *Class 0xF CoreLink, PrimeCell, or system component* on page B2-51.

Table B2-1 on page B2-33 and Figure B2-1 on page B2-34 show the memory map of a component of the CoreSight component class.

Each component occupies one or more contiguous 4KB blocks of address space. Where a component occupies more than one 4KB block, these registers must appear in the highest 4KB block. See *Spanning multiple 4KB blocks* on page B2-52.

These registers provide the CoreSight programmers' model. Every component on the Debug APB must support this programmers' model. The CoreSight programmers' model presents these registers as a set of single word (32-bit) memory-mapped registers, but for some registers:

- Only the least-significant bits of the register are valid.
- The most significant bits of the register are reserved, with access permissions that depend on the register.

Note

For any CoreSight register, a component might implement only the valid bits of the register. However software can always access the register as a 32-bit register.

When accessed as a 32-bit register, all registers are accessed in little-endian format.

Table B2-1 shows the address offsets for the CoreSight component registers, in order of their offset in the 4KB block.

Table B2-1 CoreSight component register address offsets

Offset	Type	Valid bits	Name	Description	
0xF00	RW	1	ITCTRL	Integration Mode Control register	
0xFA0	RW	0-32	CLAIMSET	Claim Tag Set register	
0xFA4	RW	0-32	CLAIMCLR	Claim Tag Clear register	
0xFA8	RO	32	DEVAFF0	Device Affinity register 0	
0xFAC	RO	32	DEVAFF1	Device Affinity register 1	
0xFB0	WO	32	LAR	Lock Access Register	
0xFB4	RO	3	LSR	Lock Status Register	
0xFB8	RO	8	AUTHSTATUS	Authentication Status register	
0xFBC	RO	32	DEVARCH	Device Architecture register	
0xFC0	RO	8-32	DEVID2	Device Configuration register 2	
0xFC4	RO	8-32	DEVID1	Device Configuration register 1	
0xFC8	RO	8-32	DEVID	Device Configuration register	
0xFCC	RO	8	DEVTYPE	Device Type Identifier register	
0xFD0	RO	8	PIDR4	Peripheral Identification Registers, PIDR0-PIDR7	
0xFD4	RO	8	PIDR5		
0xFD8	RO	8	PIDR6		
0xFDC	RO	8	PIDR7		
0xFE0	RO	8	PIDR0		
0xFE4	RO	8	PIDR1		
0xFE8	RO	8	PIDR2		
0xFEC	RO	8	PIDR3		
0xFF0	RO	8	CIDR0	Preamble	Component Identification Registers, CIDR0-CIDR3
0xFF4	RO	8	CIDR1	Component class	
0xFF8	RO	8	CIDR2	Preamble	
0xFFC	RO	8	CIDR3	Preamble	

Figure B2-1 on page B2-34 shows the registers in the 4KB window of a CoreSight component, with the valid bits of the CoreSight registers.

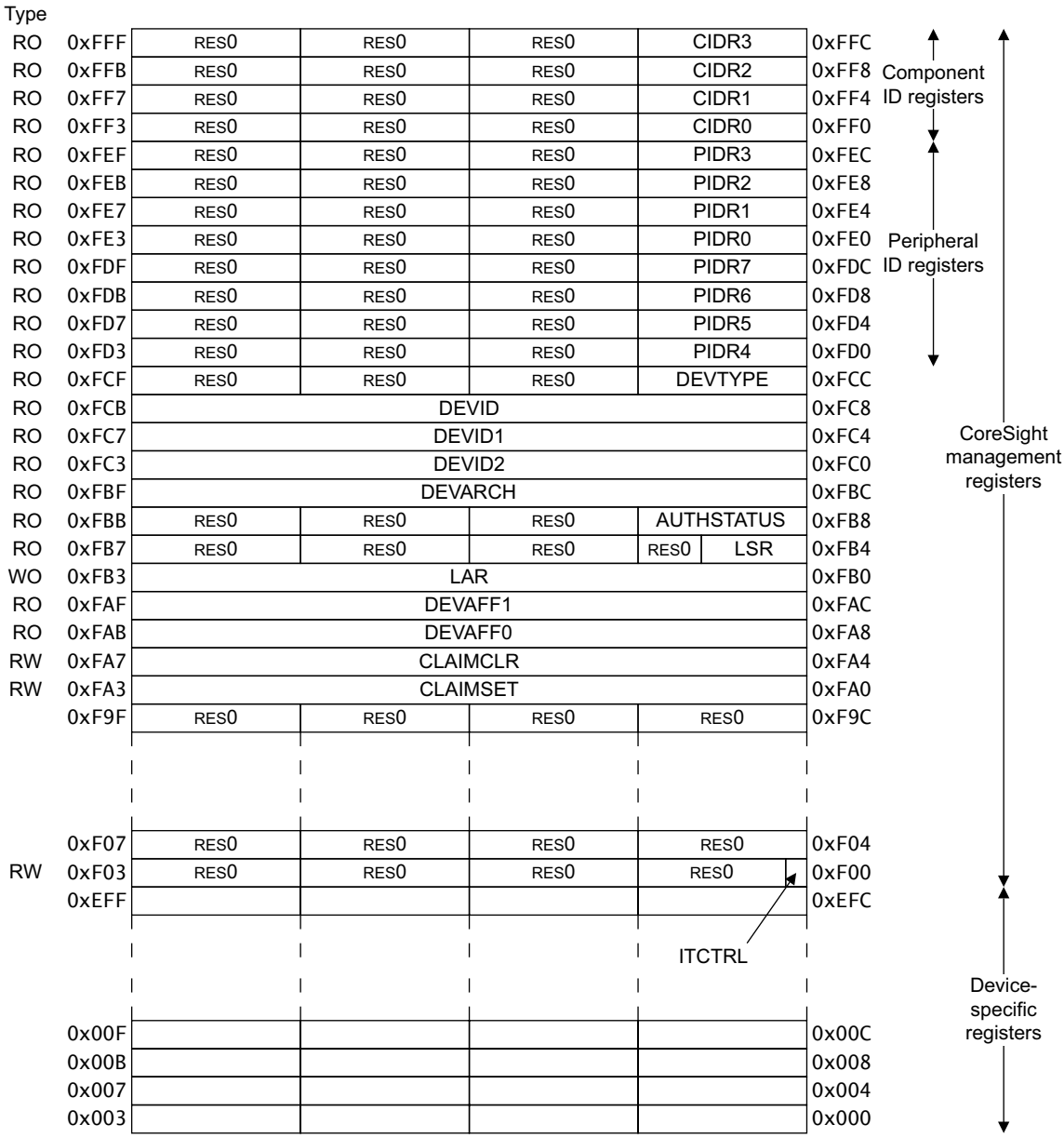


Figure B2-1 Contents of the 4KB memory window of a CoreSight component

In Figure B2-1, locations marked as RES0 behave as *Reserved locations* on page B2-35 describes.

The words at offsets `0xF00-0xFFC` are the CoreSight management registers. These are common to all CoreSight components. This area is reserved for CoreSight registers and must not be used by device-specific control registers. The CoreSight management registers include:

- The Peripheral ID registers, at offsets `0xFD0-0xFEC`.
- The Component ID registers, at offsets `0xFF0-0xFFC`.

The remaining words can be used for component specific registers. ARM recommends that:

- Control registers start at address `0x000` and continue upwards.
- Any registers used purely for integration purposes start at address `0xEFC` and continue downwards.

B2.2 Reserved locations

Figure B2-1 on page B2-34 shows the memory map of the registers that make up a CoreSight component, or the final 4KB block of a CoreSight component that comprises more than one 4KB block. For the device-specific registers, the register type, RW, RO, or WO, is IMPLEMENTATION DEFINED.

Software can access a CoreSight component using aligned word accesses. This means that, in the programmers' model, a CoreSight component comprises a set of word-aligned 32-bit registers.

Table B2-2 defines the behavior on accesses to reserved registers and fields.

Table B2-2 Behavior on accesses to reserved registers and fields

Access to	Behavior
Reserved registers	RES0
Unimplemented registers	RAZ/WI
Reserved fields in registers	RES0 or RES1
Unimplemented fields in registers	RES0 or RES1
Unimplemented bits in implemented fields	RAZ/WI

Reads of write-only registers are considered accesses to Reserved registers. Writes to read-only registers are considered accesses to Reserved registers. The following tables show the specific meaning of each of the behaviors.

Table B2-3 shows the required behavior of a CoreSight component, for registers defined as RW, RO, or WO.

Table B2-3 CoreSight component behavior

Behavior	Component behavior on reads			Component behavior on writes		
	RW	RO	WO	RW	RO	WO
RES0	RAZ	RAZ	RAZ	WI	WI	WI
RES1	RAO	RAO	RAO	WI	WI	WI
RAZ/WI	RAZ	RAZ	RAZ	WI	WI	WI

Table B2-4 shows the required behavior of software when accessing a CoreSight component, for registers defined as RW, RO, or WO.

Table B2-4 Software behavior

Behavior	Software behavior on reads			Software behavior on writes		
	RW	RO	WO	RW	RO	WO
RES0	Treat as UNKNOWN	Treat as UNKNOWN	Do not read	Preserve	Do not write	Preserve
RES1	Treat as UNKNOWN	Treat as UNKNOWN	Do not read	Preserve	Do not write	Preserve
RAZ/WI	Expect zero	Expect zero	Do not read	Are ignored	Do not write	Are ignored

Programming a reserved value into a register, or field within a register, might result in CONSTRAINED UNPREDICTABLE behavior of the component. Usually this involves mapping the behavior to one or more permitted behaviors.

B2.3 Component and Peripheral Identification registers

All component classes require the implementation of the Component and Peripheral Identification registers, as described in:

- *Component Identification Registers, CIDR0-CIDR3*.
- *Peripheral Identification Registers, PIDR0-PIDR7* on page B2-37.

B2.3.1 Component Identification Registers, CIDR0-CIDR3

Table B2-5 shows the Component Identification Registers.

Table B2-5 Component Identification Registers

Name	Offset	Bits	Field	Value	Description
CIDR3	0xFFC	[7:0]	PRMBL_3	0xB1	Preamble
CIDR2	0xFF8	[7:0]	PRMBL_2	0x05	Preamble
CIDR1	0xFF4	[7:4]	CLASS	See Table B2-6	Component class
		[3:0]	PRMBL_1	0x0	Preamble
CIDR0	0xFF0	[7:0]	PRMBL_0	0x0D	Preamble

Table B2-6 shows the component class values that you can use in the Component ID Register 1.

Table B2-6 CLASS field encodings

Value	Description
0x0	Generic verification component.
0x1	ROM table. See *Class 0x1 ROM table* on page B2-39.
0x2-0x8	Reserved.
0x9	CoreSight component. See*Class 0x9 CoreSight component* on page B2-40.
0xA	Reserved.
0xB	Peripheral Test Block.
0xC-0xD	Reserved.
0xE	Generic IP component.
0xF	CoreLink, PrimeCell, or system component with no standardized register layout, for backwards compatibility. See *Class 0xF CoreLink, PrimeCell, or system component* on page B2-51.

B2.3.2 Peripheral Identification Registers, PIDR0-PIDR7

Table B2-7 shows the Peripheral Identification Registers.

Table B2-7 Peripheral Identification Registers

Name	Offset	Bits	Field	Value	Description
PIDR7	0xFDC	[7:0]	-	RES0	Reserved.
PIDR6	0xFD8	[7:0]	-	RES0	Reserved.
PIDR5	0xFD4	[7:0]	-	RES0	Reserved.
PIDR4	0xFD0	[7:4]	SIZE	-	4KB count.
		[3:0]	DES_2	-	JEP106 continuation code.
PIDR3	0xFEC	[7:4]	REVAND	-	RevAnd.
		[3:0]	CMOD	-	Customer Modified.
PIDR2	0xFE8	[7:4]	REVISION	-	Revision.
		[3]	JEDEC	1	Always set. Indicates that a JEDEC assigned value is used.
		[2:0]	DES_1	-	JEP106 identification code, bits[6:4].
PIDR1	0xFE4	[7:4]	DES_0	-	JEP106 identification code, bits[3:0].
		[3:0]	PART_1	-	Part number, bits[11:8].
PIDR0	0xFE0	[7:0]	PART_0	-	Part number, bits[7:0].

A component is uniquely identified by the following fields:

- JEP106 continuation code.
- JEP106 identification code.
- Part Number.
- Revision.
- Customer Modified.
- RevAnd.

The meaning of the fields is as follows:

JEP106 continuation code, JEP106 identification code (DES_2, DES_1, DES_0)

These indicate the designer of the component and not the implementer, except where the two are the same. To obtain a number, or to see the assignment of these codes, contact JEDEC http://www.jedec.org .

A JEDEC code takes the following form:

- A string of zero or more numbers, all of the value 0x7F.
- A following 8-bit number, that is not 0x7F, and where bit[7] is an odd parity bit.

For example, ARM Limited is assigned the code 0x7F 0x7F 0x7F 0x7F 0x3B.

The encoding used in the Peripheral Identification Registers is as follows:

- The continuation code is the number of times 0x7F appears before the final number. For example, for ARM Limited this is 0x4.
- The identification code is bits[6:0] of the final number. For example, for ARM Limited this is 0x3B.

Part number (PART_1, PART_0)

This is selected by the designer of the component.

Revision (REVISION)

The REVISION field is an incremental value starting at `0x0` for the first design of a component. This only increases by 1 for both major and minor revisions and is simply used as a look-up to establish the exact major and minor revision.

Customer Modified (CMOD)

Where the component is reusable IP, this value indicates if the customer has modified the behavior of the component. In most cases this field is zero.

RevAnd (REVAND)

This field indicates minor errata fixes specific to this design, for example metal fixes after implementation. In most cases this field is zero. ARM recommends that component designers ensure this field can be changed by a metal fix if required, for example by driving it from registers that reset to zero.

4KB Count (SIZE)

This is a 4-bit value that indicates the total contiguous size of the memory window used by this component in powers of 2 from the standard 4KB. If a component only requires the standard 4KB then this should read as `0x0`, 4KB only, for 8KB set to `0x1`, 16KB == `0x2`, 32KB == `0x3`, and so on. For more explanation on the value of this field, see *Spanning multiple 4KB blocks* on page B2-52 and Table B2-12 on page B2-52. For ROM tables, this value must be zero. See *Expansion above 960 entries* on page D5-141.

B2.4 Class 0x1 ROM table

For the definition of the ROM table format see Chapter D5 *ROM Table*.

B2.5 Class 0x9 CoreSight component

CoreSight components must implement an additional set of registers, referred to as the CoreSight management registers. Addresses `0xF00` to `0xFCC` are reserved for use by CoreSight management registers.

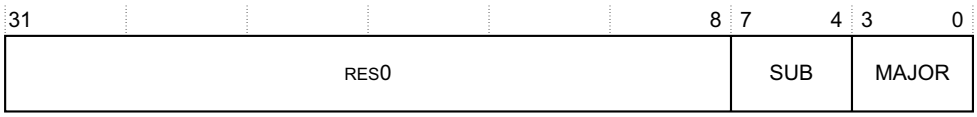
Any reads from unimplemented or reserved registers in `0xF00` to `0xFFF` must return zero, and writes must be ignored. See *Reserved locations* on page B2-35.

For components with a component class `0x9`, two or more functionally different components are permitted to share the same Part number, so long as they have different values of the DEVTYPE Register. However, ARM recommends that each functionally different component has a unique Part number.

B2.5.1 Device Type Identifier Register, DEVTYPE

The Device Type Identifier Register is read-only. It provides a debugger with information about the component when the Part number field is not recognized. The debugger can then report this information.

The DEVTYPE bit assignments are:



Description Indicates the type of functionality the component supports.

Bits [31:8] RES0

[7:4] SUB type

[3:0] MAJOR type.

Short Name DEVTYPE.

Location `0xFCC`.

Table B2-8 shows the device type encoding for the Device Type Identifier Register.

Table B2-8 Device type encoding

MAJOR type [3:0]		SUB type [7:4]	
Value	**Description**	**Value**	**Description**
`0x0`	Miscellaneous	`0x0`	Other, undefined.
		`0x1-0x3`	Reserved.
		`0x4`	Validation component.
		`0x5-0xF`	Reserved.
`0x1`	Trace Sink	`0x0`	Other.
		`0x1`	Trace port, for example TPIU.
		`0x2`	Buffer, for example ETB.
		`0x3`	Basic trace router.
		`0x4-0xF`	Reserved.

Table B2-8 Device type encoding (continued)

MAJOR type [3:0]		SUB type [7:4]	
Value	**Description**	**Value**	**Description**
0x2	Trace Link	0x0	Other.
		0x1	Trace funnel, Router.
		0x2	Filter.
		0x3	FIFO, Large Buffer.
		0x4-0xF	Reserved.
0x3	Trace Source	0x0	Other.
		0x1	Associated with a processor core.
		0x2	Associated with a DSP.
		0x3	Associated with a Data Engine or Coprocessor.
		0x4	Associated with a Bus, stimulus derived from bus activity.
		0x5	Reserved.
		0x6	Associated with software, stimulus derived from software activity.
		0x7-0xF	Reserved.
0x4	Debug Control	0x0	Other.
		0x1	Trigger Matrix, for example ECT.
		0x2	Debug Authentication Module. See *Control of authentication interfaces* on page D2-108
		0x3	Power requestor.
		0x4-0xF	Reserved.
0x5	Debug Logic	0x0	Other.
		0x1	Processor core.
		0x2	DSP.
		0x3	Data Engine or Coprocessor.
		0x4	Bus, stimulus derived from bus activity.
		0x5	Memory, tightly coupled device such as *Built In Self Test* (BIST).
		0x6-0xF	Reserved.

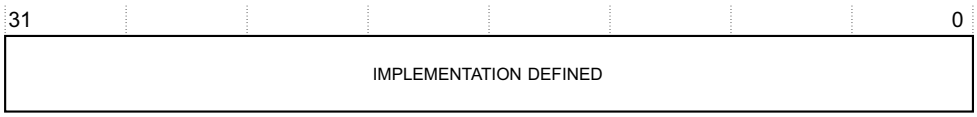
Table B2-8 Device type encoding (continued)

MAJOR type [3:0]		SUB type [7:4]	
Value	**Description**	**Value**	**Description**
0x6	Performance Monitor	0x0	Other.
		0x1	Associated with a processor.
		0x2	Associated with a DSP.
		0x3	Associated with a Data Engine or Coprocessor.
		0x4	Associated with a bus, stimulus derived from bus activity.
		0x5	Associated with a memory management unit that conforms to the ARM System MMU Architecture.
		0x6-0xF	Reserved.
0x7-0xF	Reserved	-	-

B2.5.2 Device Configuration Register, DEVID

The Device Configuration Register is read-only.

The DEVID bit assignments are:



Description This register is IMPLEMENTATION DEFINED for each Part Number and Designer. This indicates the capabilities of the component. The entire 32-bit field can be used because the data width is determined by the particular component. Unused bits must be RAZ. If the component is configurable then it is recommended that this register reflects any changes to a standard configuration.

Bits [31:0] IMPLEMENTATION DEFINED.

Short Name DEVID.

Location 0xFC8.

B2.5.3 Device Configuration Register 1, DEVID1

The DEVID1 characteristics are:

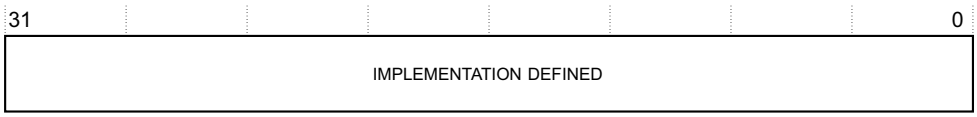
Purpose Contains an IMPLEMENTATION DEFINED value.

Usage constraints There are no usage constraints.

Configurations Available in all implementations.

Attributes A 32-bit RO management register that contains an IMPLEMENTATION DEFINED value. See also *CoreSight component register address offsets* on page B2-33.

The DEVID1 bit assignments are:



Bits[31:0] The contents are IMPLEMENTATION DEFINED.

B2.5.4 Device Configuration Register 2, DEVID2

The DEVID2 characteristics are:

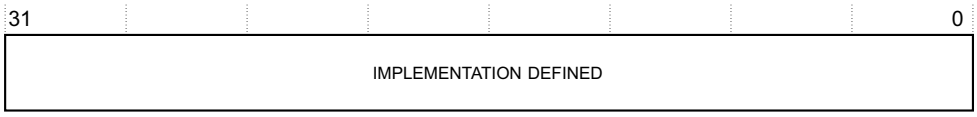
Purpose Contains an IMPLEMENTATION DEFINED value.

Usage constraints There are no usage constraints.

Configurations Available in all implementations.

Attributes A 32-bit RO management register that contains an IMPLEMENTATION DEFINED value. See also *CoreSight component register address offsets* on page B2-33.

The DEVID2 bit assignments are:

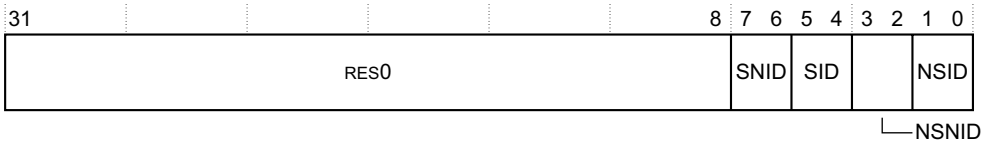


Bits[31:0] The contents are IMPLEMENTATION DEFINED.

B2.5.5 Authentication Status Register, AUTHSTATUS

The Authentication Status Register is read-only. For details of authentication, see *Control of authentication interfaces* on page D2-108.

The AUTHSTATUS bit assignments are:



Description Reports the required security level and current status of those enables. Where functionality changes on a given security level then this change in status must be reported in this register.

Bits [31:8] RES0.

[7:6] SNID, Secure non-invasive debug.

[5:4] SID, Secure invasive debug.

[3:2] NSNID, Non-secure non-invasive debug.

[1:0] NSID, Non-secure invasive debug.

Short Name AUTHSTATUS.

Location `0xFB8`.

Table B2-9 shows the description for each pair of bits in each debug set.

Table B2-9 Authentication status bits values and meanings

Value	Description
0b00	Functionality not implemented or controlled elsewhere
0b01	Reserved
0b10	Functionality disabled
0b11	Functionality enabled

Table B2-10 shows how you must set the fields in Table B2-9.

Table B2-10 Recommended authentication status bit field settings

Field	Debug level not supported	Debug level supported
Secure invasive debug	0b00	If (**SPIDEN** and **DBGEN**) 0b11 else 0b10
Non-secure invasive debug	0b00	If (**DBGEN**) 0b11 else 0b10
Secure non-invasive debug	0b00	If (**SPNIDEN** or (**SPIDEN** and **DBGEN**)) and (**NIDEN** or **DBGEN**) 0b11 else 0b10
Non-secure non-invasive debug	0b00	If (**NIDEN** or **DBGEN**) 0b11 else 0b10

Components not designed for Secure systems

Some components might not distinguish between Secure and Non-secure debug. For example, a trace component for a simple bus might connect to a Secure or a Non-secure bus, and its enable signals connect differently depending on which bus the component connects to. This could result in:

- A component that indicates only Non-secure debug capabilities but that is performing only Secure debug functions.
- A component that indicates only Secure debug capabilities but that is performing only Non-secure debug functions.

Debuggers must be aware of this possibility.

B2.5.6 Device Architecture Register, DEVARCH

The DEVARCH characteristics are:

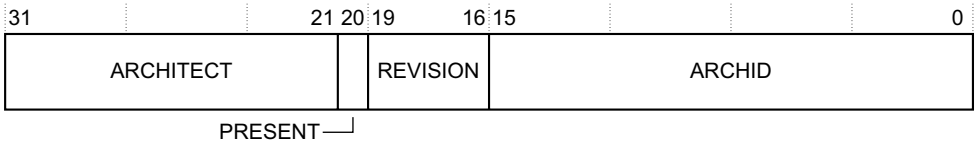
Purpose Identifies the architect and architecture of a CoreSight component. The architect might differ from the designer of a component, for example ARM defines the architecture but another company designs and implements the component.

Usage constraints There are no usage constraints.

Configurations Available in all implementations.

Attributes A 32-bit RO management register that returns an IMPLEMENTATION DEFINED value. See also *CoreSight component register address offsets* on page B2-33.

The DEVARCH bit assignments are:



ARCHITECT, bits[31:21]

Defines the architect of the component:

Bits[31:28] Indicates the JEP106 continuation code.

Bits[27:21] Indicates the JEP106 identification code.

See the *Standard Manufacturers Identification Code* for information about JEP106. For components where ARM is the architect, this 11-bit field returns `0x23B`.

PRESENT, bit[20] Indicates the presence of this register:

0 = DEVARCH register is not present so bits[31:0] must be RAZ.

1 = DEVARCH register is present.

REVISION, bits[19:16]

Architecture revision. Returns the revision of the architecture that the ARCHID field specifies.

ARCHID, bits[15:0] Architecture ID. Returns a value that identifies the architecture of the component.

Table B2-11 lists the ARCHID values for some example components where ARM is the architect.

Table B2-11 Example ARCHID values

ARCHID	Description
`0x4A13`	*Embedded Trace Macrocell* (ETMv4) architecture
`0x1A14`	*Cross Trigger Interface* (CTI) architecture
`0x6A15`	Processor debug architecture
`0x2A16`	Processor *Performance Monitor* (PMU) architecture
`0x0A31`	Basic trace router
`0x0A34`	Power requestor
`0x0A63`	*System Trace Macrocell* (STM) architecture

B2.5.7 Device Affinity Register 0, DEVAFF0

The DEVAFF0 characteristics are:

Purpose Enables a debugger to determine if two components have an affinity with each other.

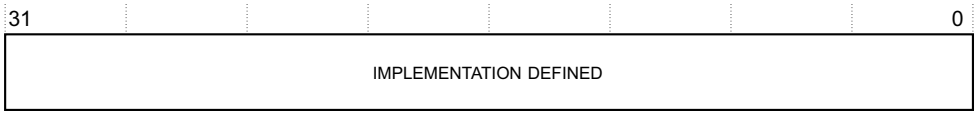
For example, when a trace macrocell connects to a processor, the DEVAFF0 and DEVAFF1 registers in both components must contain identical values that are unique. This enables the debugger to identify how the components relate to each other, without performing topology detection.

Usage constraints There are no usage constraints.

Configurations Available in all implementations.

Attributes A 32-bit RO management register that returns an IMPLEMENTATION DEFINED value. See also *CoreSight component register address offsets* on page B2-33.

The DEVAFF0 bit assignments are:



Bits[31:0] The contents are IMPLEMENTATION DEFINED. If a component has no unique association with another component then this field is RAZ.

Examples of the content that DEVAFF0 contains are:

- For ARM architecture processors, it returns the lower 32 bits of the processor MPIDR, that is, MPIDR[31:0].
- For a CTI that connects to an ARM architecture processor, it returns the lower 32 bits of the processor MPIDR.

B2.5.8 Device Affinity Register 1, DEVAFF1

The DEVAFF1 characteristics are:

Purpose Enables a debugger to determine if two components have an affinity with each other.

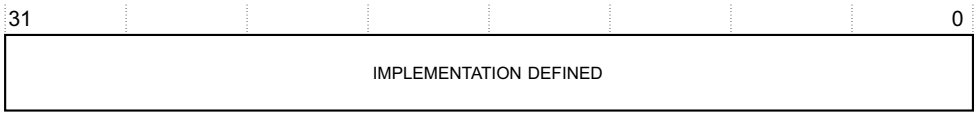
For example, when a trace macrocell connects to a processor, the DEVAFF0 and DEVAFF1 registers in both components must contain identical values that are unique. This enables the debugger to identify how the components relate to each other, without performing topology detection.

Usage constraints There are no usage constraints.

Configurations Available in all implementations.

Attributes A 32-bit RO management register that returns an IMPLEMENTATION DEFINED value. See also *CoreSight component register address offsets* on page B2-33.

The DEVAFF1 bit assignments are:



Bits[31:0] The contents are IMPLEMENTATION DEFINED. If a component has no unique association with another component then this field is RAZ.

Examples of the content that DEVAFF1 contains are:

- For ARM architecture processors, it returns the upper 32 bits of the processor MPIDR, that is, MPIDR[63:32].
- For a CTI that connects to an ARM architecture processor, it returns the upper 32 bits of the processor MPIDR.

B2.5.9 Lock registers

The Lock registers prevent accidental access to the registers of CoreSight components. Software that is being debugged might accidentally write to memory used by CoreSight components. This might disable those components, making the software impossible to debug. The CoreSight programmers' model includes a Lock Status Register, LSR, and a Lock Access Register, LAR, to control software access to CoreSight components to ensure that the likelihood of accidental access to CoreSight components is extremely small.

A software monitor that accesses debug registers must unlock the component before accessing any registers, and lock the component again before exiting the monitor. In this way the software being debugged can never access an unlocked CoreSight component.

It is recommended that external accesses from a debugger are not subject to the Lock registers, and therefore that external reads of the LSR return zero. For information on how CoreSight components can distinguish between internal and external accesses, see *Debug APB interface memory map* on page D2-109.

If the system includes several bus masters capable of accessing the CoreSight components, for example several processors, then it is possible for software running on one processor, processor A, to accidentally access the component while it is being programmed by a debug monitor running on the other processor, processor B. It is not possible for the component to distinguish between accesses from the two processors. The probability of this occurring, and of processor A disabling the component, is low, but you must consider this possibility when designing your system in case there are special circumstances that make this more likely.

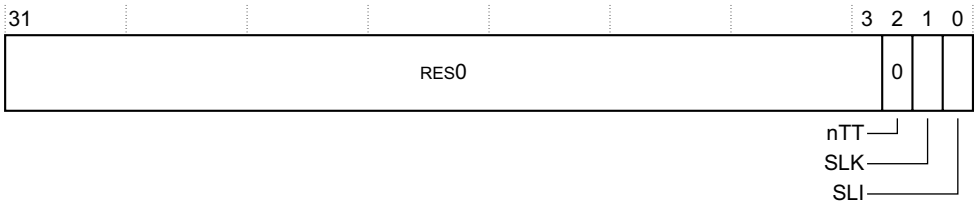
Note

The claim tag cannot be used to manage accesses to the Lock registers, because access to the claim tag is subject to the Lock registers.

Lock Status Register, LSR

This register indicates the status of the lock control mechanism.

The LSR bit assignments are:



Description This indicates the status of the Lock control mechanism. This lock prevents accidental writes by code under debug. This register must always be present although there might not be any lock-access control mechanism. The lock mechanism, where present and locked, must block write accesses to any control register, except the LAR. For most components this covers all registers except for the LAR.

Bits [31:3] RES0.

[2] nTT. This bit is always zero, which indicates that the component implements a 32-bit LAR.

[1] SLK. Returns the Software Lock status:

- 0 = Access permitted.
- 1 = Write access to the component is blocked. All writes to control registers are ignored. Reads are permitted.

  Note

  When present, the default reset value of this bit is 1.

[0] SLI, indicates that a Software Lock control mechanism exists for this device.

Short name LSR.

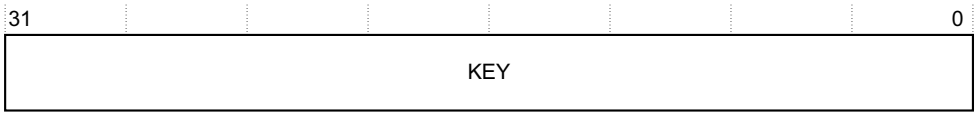
Location `0xFB4`.

Note

- If no lock control mechanism exists then this register must be RAZ.
- Some components have two programmable views, one only visible from external tools and the other visible from software running on-chip. In this case, the externally visible memory map must have this register be RAZ, because no lock access mechanism is required for that view.

Lock Access Register, LAR

The LAR bit assignments are:



If LSR[0] == `0b0` then this register is not present.

Description This is used to enable write access to device registers.

Bits [31:0] KEY. A write of `0xC5ACCE55` enables further write access to this device. An invalid write has the affect of removing write access.

Short name LAR.

Location `0xFB0`.

B2.5.10 Claim tag registers

Often there are a number of debug agents that must cooperate to control the resources that the CoreSight components make available. For example, an external debugger and a debug monitor running on the target might both require control of the breakpoint resources of a processor. It is important that a debug agent does not reprogram debug resources that another debug agent is using.

The claim tag registers provide a number of bits that can be separately set and cleared to indicate if functionality is in use by a debug agent. All debug agents must implement a common protocol in order to use these bits. This protocol is not defined in this specification, but a number of protocols are suggested in *Claim tag protocols* on page B2-49 to illustrate how these bits can be used.

Claim Tag Clear Register, CLAIMCLR

This register is used in conjunction with *Claim Tag Set Register, CLAIMSET* on page B2-49.

The CLAIMCLR bit assignments are:

Description This register forms one half of the claim tag value. This location enables individual bits to be cleared, write, and returns the current claim tag value, read.

The width (n) of this register can be determined from reading CLAIMSET:

Read Current claim tag value.

Write Each bit is considered separately:
0 = No effect.
1 = Clear this bit in the claim tag.

Bits [31:n] RAZ/WI.

[n–1:0] A bit programmable register bank that is zero at reset.

Short name CLAIMCLR.

Location `0xFA4`.

Claim Tag Set Register, CLAIMSET

This is used in conjunction with *Claim Tag Clear Register, CLAIMCLR* on page B2-48.

The CLAIMSET bit assignments are:

Description This register forms one half of the claim tag value. This location allows individual bits to be set, write, and returns the number of bits that can be set, read.

Read Each bit is considered separately:

0 = This claim tag bit is not implemented.

1 = This claim tag bit is implemented.

Write Each bit is considered separately:

0 = No effect.

1 = Set this bit in the claim tag.

You can determine how many claim bits are implemented by reading this register. For example, if four bits are implemented, a read of this register returns `0x0000000F`. If no claim tag is implemented, then a read of this register returns `0x00000000`.

ARM recommends that a minimum of four claim bits are implemented.

Bits [31:n] RAZ/WI.

[n–1:0] A bit programmable register bank which sets the claim tag value. A read returns a logic 1 for all implemented locations.

Short name CLAIMSET.

Location `0xFA0`.

Claim tag protocols

This section describes some example claim tag protocols:

Protocol 1: Set common bit to claim

In this scenario, debug functionality is only claimed on a few rare well-defined points, for example when the target is powered up or when a debugger is connected.

Each bit in the claim tag corresponds to an area of debug functionality, shared between all debug agents. For example, four bits can control four areas of functionality. The following shows a pseudocode implementation of this protocol:

```
read claim tag bit
if (bit is set)
    functionality is not available
else
    set bit
    use functionality
```

Protocol 2: Set private bit to claim

In this scenario, debug functionality is also only claimed on a few rare well-defined points, but it is necessary to be able to determine which other agent has claimed functionality.

Each bit in the claim tag corresponds to an area of debug functionality for a debug agent. For example, four bits can control two areas of functionality each for two debug agents. The following shows a pseudocode implementation of this protocol:

```
read all claim tag bits for this functionality
if (any bits are set)
    functionality is not available
```

```
else
    set bit for this agent
    use functionality
```

Protocol 3: Set private bit and check for race

In this scenario, debug functionality is claimed regularly and it is possible for two debug agents to attempt to claim it at the same time. Each bit in the claim tag corresponds to an area of debug functionality for a debug agent, as in protocol 2. The following shows a pseudocode implementation of this protocol:

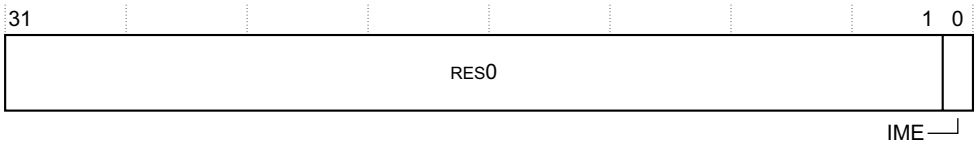
```
read all claim tag bits for this functionality
if (any bits are set)
    functionality is not available
else
    set bit for this agent
    read all claim tag bits for this functionality
    if (any bits are set by other agents)
        clear bit for this agent
        wait a random amount of time
        go back to start
    else
        use functionality
```

If using this protocol, it is important to implement the random wait, otherwise it is possible for two debug monitors operating at the same speed to repeatedly claim and release the functionality indefinitely.

B2.5.11 Integration Mode Control Register, ITCTRL

The Integration Mode Control Register is Read/Write. This register is used to enable topology detection. For more information see Chapter B3 *Topology Detection Registers*.

The ITCTRL bit assignments are:



Description This register enables the component to switch between functional mode and integration mode. The default behavior is functional mode. In integration mode the inputs and outputs of the component can be directly controlled for the purpose of integration testing and topology solving.

Bits [31:1] RES0.

[0] IME. When set, the component enters integration mode, enabling topology detection or integration testing to be performed. At reset the component must enter functional mode. If no integration functionality is implemented, this register might be RAZ.

Short name ITCTRL.

Location `0xF00`.

Note

When a device has been in integration mode, it might not function with the original behavior. After performing integration or topology detection, you must reset the system to ensure correct behavior of CoreSight and other connected system components that are affected by the integration or topology detection.

B2.6 Class 0xF CoreLink, PrimeCell, or system component

This class can be used for components that are unrelated to the CoreSight system. No registers other than the Peripheral ID Registers and Component ID Registers are specified for this class of component.

B2.7 Spanning multiple 4KB blocks

If the registers of a component, including the 256 bytes reserved for CoreSight management registers, do not fit within 4KB then one or more additional 4KB blocks are required.

You must implement the CoreSight programmers' model in the last 4KB block. You do not have to implement the programmers' model in the other 4KB blocks. Each 4KB window must be allocated consecutively without gaps. Table B2-12 shows the full list of memory block requirements and the impacts to the component in terms of available registers and required address lines.

Table B2-12 Spanning multiple 4KB windows

Number of 4KB blocks	Value of PIDR4.SIZE	Total memory window used	Component specific registers available	Expected PADDRDBG input[a]
1	0x0	4KB, 1K words	960 words	**PADDRDBG[11:2]**
2	0x1	8KB	1984 words	**PADDRDBG[12:2]**
4	0x2	16KB, 4K words	4032 words	**PADDRDBG[13:2]**
8	0x3	32KB, 8K words	8128 words	**PADDRDBG[14:2]**
16	0x4	64KB	16320 words	**PADDRDBG[15:2]**
32	0x5	128KB	32704 words	**PADDRDBG[16:2]**
64	0x6	256KB, 64K words	65472 words, 63.94K words	**PADDRDBG[17:2]**
128	0x7	512KB	127.94K words	**PADDRDBG[18:2]**
256	0x8	1MB, 256K words	255.9K words	**PADDRDBG[19:2]**
512	0x9	2MB	~512K words	**PADDRDBG[20:2]**
1024	0xA	4MB	~1M words	**PADDRDBG[21:2]**
2048	0xB	8MB	~2M words	**PADDRDBG[22:2]**
4096	0xC	16MB	~4M words	**PADDRDBG[23:2]**
8192	0xD	32MB	~8M words	**PADDRDBG[24:2]**
16384	0xE	64MB, 16M words	~16M words	**PADDRDBG[25:2]**
Reserved	0xF	-	-	-

a. **PADDRDBG[1:0]** are not required on blocks as all transfers under the AMBA 3 APB protocol are 32-bit, word, aligned. **PADDRDBG[31]** might also be required. For more information see *Use of PADDRDBG[31]* on page C2-69.

Figure B2-2 on page B2-53 shows how a component that requires four 4KB windows appears in the memory map.

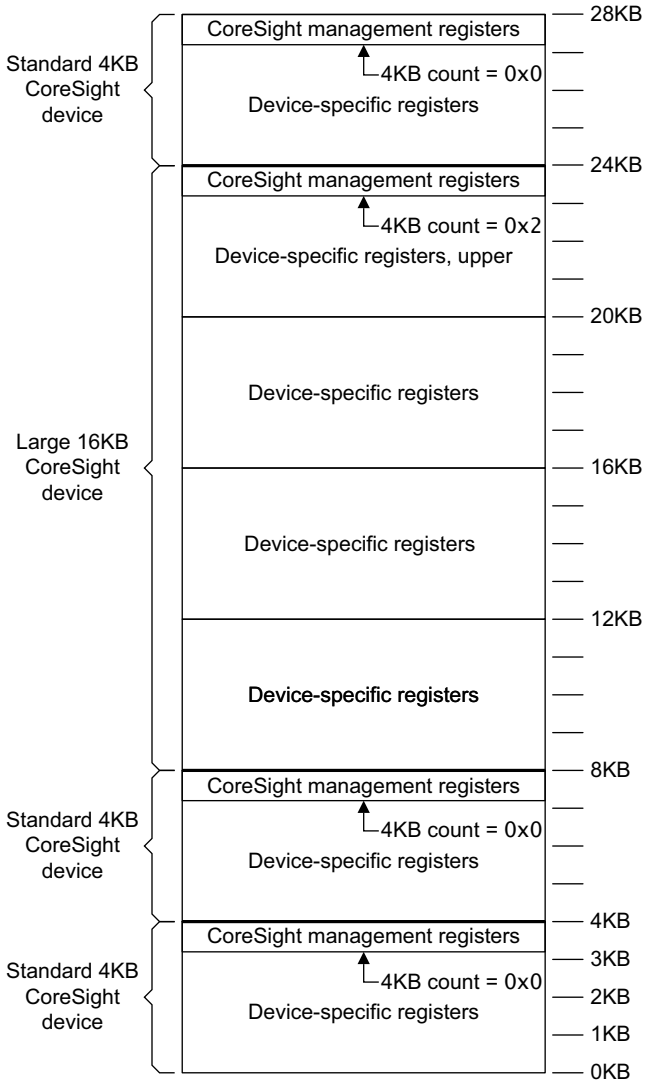


Figure B2-2 How multiple 4KB windows are spanned

B2.7.1 Alternative addressing methods

Alternative methods for extending the address space available to a component are possible, but are not recommended.

Second CoreSight component

You can implement additional address space as an additional CoreSight component. However, if you do this you must provide a method for linking this component back to the original component through topology detection. This is not recommended.

Additional linked address space

You can implement an additional area of address space for the component, provided that this address space is not within the space used by CoreSight components conforming to this programmers' model. If you do this, you must provide a means for determining the address of the additional space from the programmers' model of the component. This limits the system design options and is not recommended.

Chapter B3
Topology Detection Registers

This chapter describes the CoreSight topology detection registers. It contains the following sections:

B3.1 About topology detection registers

CoreSight systems can have a number of interface types, as masters or slaves, and each CoreSight component specifies which interfaces are present. The debugger probes each interface to determine which other components are connected to it.

Each interface type defines that signals must be controllable by the master and slave interfaces, and how the debugger can determine the connectivity using these signals. These signals are referred to as topology detection signals. For the specification of the requirements for standard interfaces used by ARM CoreSight components see Chapter C7 *Topology Detection at the Component Level*. Interface vendors must define the requirements for other interfaces, following the rules in Chapter D6 *Topology Detection at the System Level*.

B3.2 Requirements for topology detection signals

For each topology detection input, it must be possible to read the state of that input. For each topology detection output, it must be possible to drive the state of that output without affecting other topology detection signals. It is not necessary to implement topology detection registers on the interface through which the component is programmed, the AMBA 3 APB interface, because this connectivity is described by the ROM table. Topology detection can be invasive. See Chapter D6 *Topology Detection at the System Level*.

B3.2.1 Recommended method

ARM recommends that topology detection registers are implemented as follows:

- Implement a topology detection mode that isolates the topology detection signals.
- For each topology detection output, provide a register that sets the value of that output when in topology detection mode.
- For each topology detection input, provide a register that returns the value of that input when in topology detection mode.

B3.3 Combination with integration registers

Many components implement integration registers, providing the same level of control for the majority of inputs and outputs, instead of just those required for topology detection. This enables rapid integration testing when validating a SoC built from these components, because a testbench can thoroughly prove the connectivity between two components without knowledge of the underlying functionality of those components.

If you are designing a component that implements integration registers, it is recommended that you reuse these registers for topology detection. You can use the ITCTRL to select both integration mode and topology detection mode. See *Integration Mode Control Register, ITCTRL* on page B2-50.

B3.4 Interfaces that are not connected or implemented

Some components do not implement a fixed number of interfaces. Usually this is because some interfaces might not be connected. To the debugger, there is no difference between an interface that is not present and one that is unconnected.

If the component requires that the interface is still usable when connected to a non-CoreSight component that is not capable of topology detection, the programmers' model must indicate if the interface is connected or not.

If the component can only be connected to other CoreSight components, the tools can assume that the interface does not exist if they fail to find any connected interfaces during topology detection. In this case, the programmers' model does not need to indicate whether the interface is connected but, if it does, some time can be saved during topology detection.

B3.5 Variant interfaces

Usually the connections between interfaces, when detected, do not change. However, sometimes it is necessary for a number of components to share one component. For example, a component tracing the operation of a processor might switch to trace the operation of a different processor. It is important that the conditions under which this can occur are well understood.

The connections between interfaces can only change if all the following apply:

- An interface is defined as being variant between multiple connections.
- The programmers' model of the affected component controls the configuration by selecting between a number of alternative connections for that interface.
- The programmers' model indicates how many alternative connections are valid, to reduce the autodetection time. It is recommended that this number is small, no more than 32. This number must not change when any switching occurs.

If this is too inflexible, you must build a separate CoreSight component to perform the multiplexing operation. Topology detection can then be performed between this new component and the components it is connected to.

When a switch has occurred, topology detection must be repeated to determine the new connections. Because this might be invasive, it is recommended that topology detection is performed for all switches that are likely to occur in advance.

B3.5.1 External multiplexing

Figure B3-1 shows an example of how variable connections can be implemented using an external multiplexer. This example shows:

- A variant connection that can be connected to one of *n* inputs.
- A selection register that selects the input to use.
- A register that indicates that there are *n* connections to select between. This register can be read by a debugger to determine which values of the selection register are valid. It is tied to the value *n* outside the component.

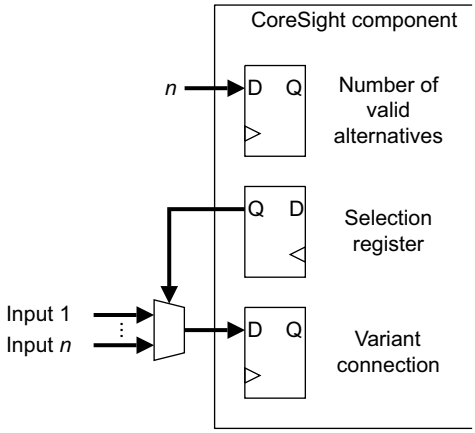


Figure B3-1 External multiplexing of connections

B3.6 Documentation requirements for topology detection registers

The component must have documentation that defines the interfaces present on that component. The definition of each interface must include:

- Its name, using the format listed in Chapter D6 *Topology Detection at the System Level*.
- If the interface supports variable connections:
  - How many connections are valid.
  - How to switch between connections.
- How to control the topology detection signals listed for that interface in Chapter D6 *Topology Detection at the System Level*.

B3.6.1 Interfaces where topology detection is not possible

If an interface can be connected to a non-CoreSight component, topology detection might not be possible. In this case it must still be possible to determine how it has been connected. The documentation must define how this can be determined from the programmers' model.

Part C

CoreSight Reusable Component Architecture

Chapter C1
Reusable Component Architecture

This chapter describes the reusable component architecture. It contains the following section:

- *About the reusable component architecture* on page C1-66.

C1.1 About the reusable component architecture

This reusable component architecture specifies the rules that must be followed for a component to be used with other CoreSight components using the CoreSight architecture.

The reusable component architecture defines the physical interfaces of the component. It enables you to connect components together easily.

Self-contained systems can implement just the visible component architecture. This will not affect compatibility with debuggers, but will prevent the IP being used with other CoreSight components.

You do not have to implement an interface if the functionality it provides is not required by a component. For example, a component with no programmable registers does not need to implement the AMBA 3 APB interface.

You can create a component that performs a number of functions internally as separate components, but presents only one set of the re-usable component interfaces externally. This is encouraged as a means to implement pre-built platforms with the CoreSight infrastructure already integrated. You can integrate the platform into a larger system as if it is a single CoreSight component.

Chapter C2
AMBA APB and ATB Interfaces

This chapter describes the AMBA APB interface that is used to program CoreSight components and the AMBA ATB interface that transfers the trace data. It contains the following sections:

C2.1 AMBA APB interface

The following sections describe:

- *About the AMBA APB interface*.
- *AMBA APB interface signals*.
- *AMBA 3 APB interface width* on page C2-69.
- *Use of PADDRDBG[31]* on page C2-69.
- *Alternative views of the register file* on page C2-69.

C2.1.1 About the AMBA APB interface

The AMBA APB interface is used to program CoreSight components.

The interface supports:

- Simple, non-pipelined operation.
- Implementation of 8-, 16-, or 32-bit slaves.
- Slave stalling.
- Slave error response.

See the *ARM® AMBA® APB Protocol Specification* for more information.

Some legacy debug components implement a JTAG TAP controller to access their functionality. The rationale for using memory-mapped access on-chip is discussed in *Component access using memory-mapped interfaces instead of JTAG* on page A1-25.

The bus to which all CoreSight components are connected is referred to as the Debug APB interface.

C2.1.2 AMBA APB interface signals

All signals are suffixed with **DBG** to indicate that this is the Debug APB interface used for accessing CoreSight components. Table C2-1 shows the signals in the interface. The clamp value is the value that an output must be clamped to when the component is powered down or disabled. See the *ARM® AMBA® APB Protocol Specification* for more information.

Table C2-1 Signals on the Debug APB interface

Name	Direction			Description
	Master	Slave	Clamp value	
PCLKDBG	Input	Input	-	The rising edge of **PCLKDBG** times all transfers on the AMBA 3 APB interface.
PRESETDBGn	Input	Input	-	This signal resets the interface and is active LOW.
PADDRDBG[31:2]	Output	Input	0	This bus indicates the address of the transfer. You do not have to implement unused bits. For information on the special use of bit[31] see *Use of PADDRDBG[31]* on page C2-69.
PSELDBG	Output	Input	0	This signal indicates that the slave device is selected and a data transfer is required. There is a **PSELDBG** signal for each slave.
PENABLEDBG	Output	Input	0	This signal indicates the second and subsequent cycles of an AMBA 3 APB interface transfer.
PWRITEDBG	Output	Input	0	When HIGH this signal indicates a write access and when LOW a read access.

Table C2-1 Signals on the Debug APB interface (continued)

Name	Direction			Description
	Master	Slave	Clamp value	
PWDATADBG[31:0]	Output	Input	0	The write data bus is driven by the master during write cycles, when **PWRITEDBG** is HIGH. The write data bus can be up to 32-bits wide.
PREADYDBG	Input	Output	1	The ready signal is used by the slave to extend an AMBA 3 APB interface transfer.
PRDATADBG[31:0]	Input	Output	0	The read data bus is driven by the selected slave during read cycles, when **PWRITEDBG** is LOW. The read data bus can be up to 32-bits wide.
PSLVERRDBG	Input	Output	1	This signal is returned in the second cycle of the transfer, and indicates an error response. CoreSight components should only use this signal to indicate that the component is unavailable, for example because of power down.

C2.1.3 AMBA 3 APB interface width

The AMBA 3 APB interface is 32-bits wide.

Chapter B2 *CoreSight Programmers' Model* describes the model compatible with a 32-bit AMBA 3 APB interface.

C2.1.4 Use of PADDRDBG[31]

The memory map of the Debug APB interface is split to allow the source of an access to be determined, as described in section *Debug APB interface memory map* on page D2-109. A component can use **PADDRDBG[31]** to distinguish between internal and external accesses. Most components use this to control the Lock access mechanism defined in *Lock registers* on page B2-46.

C2.1.5 Alternative views of the register file

There can be several ways of accessing the registers of a component. In particular, it is often useful for some or all of the debug registers in a processor to be visible through dedicated instructions, for example as registers in a linked coprocessor. You can do this provided that it is also possible to access the debug functionality of the component using the AMBA 3 APB interface.

C2.2 AMBA ATB interface

The AMBA ATB interface carries trace around an SoC.

Any CoreSight component, or platform, that has trace capabilities has an AMBA ATB interface. An interface that generates trace data is a master on the AMBA ATB interface and an interface that receives trace data is a slave on the AMBA ATB interface.

The AMBA ATB interface supports:

- Stalling of data, using valid and ready responses.
- Byte-sized packets, control signals to indicate the number of bytes valid in a cycle.
- Originating component marker, each data packet has an associated ID.
- Any trace protocol or data agnostic requirements about the format of the data.
- Check-pointing of data from all originating components.

See the *ARM® AMBA® ATB Protocol Specification* for more information.

Chapter C3
Event Interface

This chapter describes the event interface. It contains the following section:

- *About the event interface* on page C3-72.

C3.1 About the event interface

A CoreSight system uses the event interface to transfer events between components. It is most commonly used for communicating cross-trigger events between debug components and a *Cross Trigger Interface* (CTI).

The event interface signals are:

EVENTCLK Clock. This signal is usually mapped onto an existing clock signal.

EVENTRESETn Reset. This signal is usually mapped onto an existing reset signal.

EVENT This signal indicates the event, and is usually mapped onto a signal of a different name that describes its purpose.

An event is indicated by a rising edge on **EVENT**. Therefore an event can be signaled at most once every two **EVENTCLK** cycles.

If the event has a duration then the falling edge on **EVENT** indicates its completion.

There is no back-pressure mechanism defined in this interface, therefore events that are close together might merge. For example, if the event interface crosses an asynchronous boundary to a slower clock domain, events in close succession might merge into a single event.

Chapter C4
Channel Interface

This chapter describes the channel interface. It contains the following sections:

C4.1 About the channel interface

CoreSight components need to pass events between one another. For example:

- For two processors to stop at the same time, they need to signal to each another when the they have stopped.
- To perform advanced profiling functions, profiling events from many different sources in the system need to be shared.

CoreSight technology from ARM provides the *Cross Trigger Interface* (CTI), that enables events to be passed between components. However:

- Some systems require more event signals than are supported by a CTI.
- In a platform-oriented system it is necessary to connect these event signals together within the platform, exporting only a set of standard interfaces for extension at higher levels.

The channel interface passes events between components. It connects multiple CTIs together, and can be supported by a CoreSight component directly if required. The channel interface is a specialist type of event interface, as Chapter C3 *Event Interface* describes. Figure C4-1 shows how you can use the channel interface.

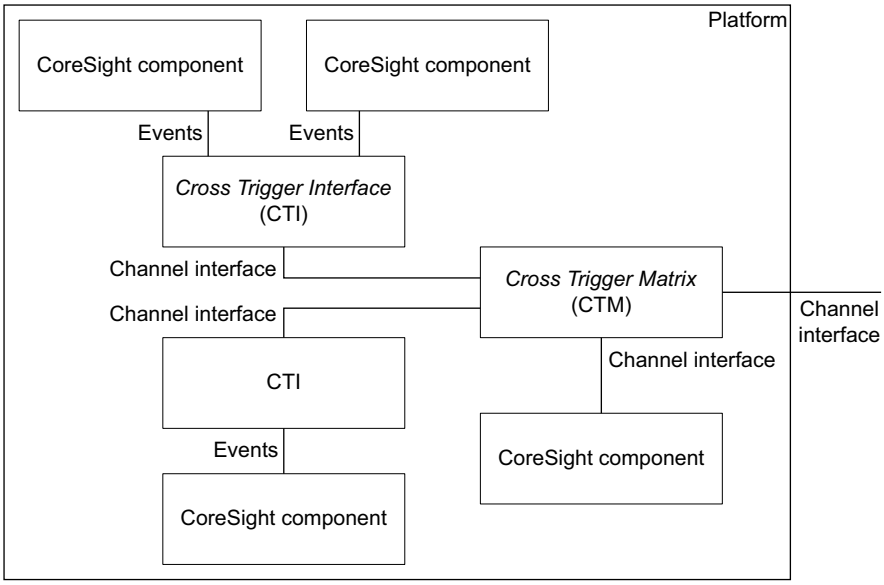


Figure C4-1 Use of the channel interface

The channel interface supports:

- A variable number of event channels.
- Bidirectional communication.
- Synchronous or asynchronous variants.

C4.1.1 Channel interface limitations

The channel interface is designed to pass events between components with minimum overhead. If multiple events are presented to the channel interface in close succession, these might be interpreted as a single event.

Figure C4-2 on page C4-75 shows events generated in a fast clock domain, Clock A, being passed to a slower clock domain, Clock B. In Clock A two separate events can be seen. These events are too close together for Clock B, that shows them as a single event.

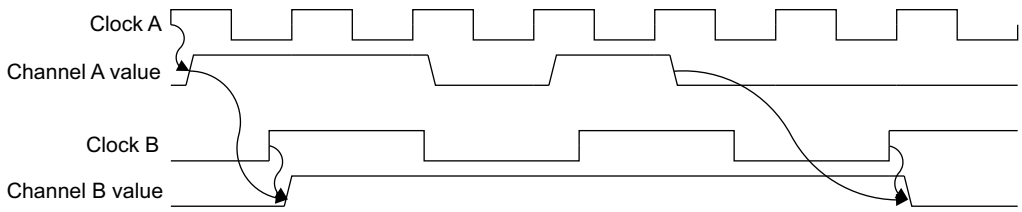


Figure C4-2 Event merging in the channel interface

The channel interface is suitable for the following:

- Transmission of an event that happens only once, for example a trigger signal to an ETB or TPIU to end trace capture.
- Transmission of a low-speed signal level where precision is not important.
- Transmission of a signal subject to handshaking using another channel in the channel interface.
- Transmission of a signal subject to software handshaking, for example an interrupt request.
- Transmission of events to be counted that do not occur close together, for example the number of times a peripheral causes an interrupt.

The channel interface is not suitable for the following:

- Transmission of events to be counted that occur close together, for example the number of instructions executed by a processor over a period of time.

C4.2 Channels

The channel interface consists of two components:

- Channel outputs. These indicate events generated by the component.
- Channel inputs. These indicate events generated by other components.

Events indicated on the channel outputs are indicated on the channel inputs of other components, but are not indicated on the channel inputs of the component that generated them.

Components must treat all channels identically. It must be possible for the debugger to control which channels are used for which purposes.

The interface supports an IMPLEMENTATION DEFINED number of channels. ARM recommends that at least four channels are implemented.

If a system consists of subsystems with different numbers of channels, and there is a requirement to pass events between these subsystems, then:

- A subset of the channels from the subsystem with the greater number is connected to all of the channels in the other subsystem.
- The set of channels connected is always a contiguous set from channel 0.

For example, in a system where subsystem A has eight channels and subsystem B has four channels, channels 0-3 from subsystem A are connected to channels 0-3 in subsystem B. Channels 4-7 are not connected to subsystem B.

C4.3 Channel interface signals

Table C4-1 shows the set of signals required by an asynchronous channel interface. The clamp value is the value that an output must be clamped to when the component is powered down or disabled.

Table C4-1 Asynchronous channel interface signals

Name	Direction	Clamp value	Description
CHIN[n-1:0]	Input	-	Channel input
CHINACK[n-1:0]	Output	1	Channel input acknowledge
CHOUT[n-1:0]	Output	0	Channel output
CHOUTACK[n-1:0]	Input	-	Channel output acknowledge

Figure C4-3 shows how the asynchronous interface uses a basic 4-phase handshaking protocol. The same protocol is used by **CHOUT** and **CHOUTACK**.

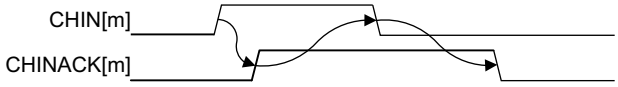


Figure C4-3 Channel interface handshaking

Table C4-2 shows the set of signals required by a synchronous channel interface.

Table C4-2 Synchronous channel interface signals

Name	Direction	Clamp value	Description
CHCLK	Input	-	Clock
CHIN[n-1:0]	Input	-	Channel input
CHOUT[n-1:0]	Output	0	Channel output

C4.4 Channel connections

The channel interface is bidirectional, and therefore you must take care to connect the correct signals together. Figure C4-4 shows how to connect two asynchronous channel interfaces together.

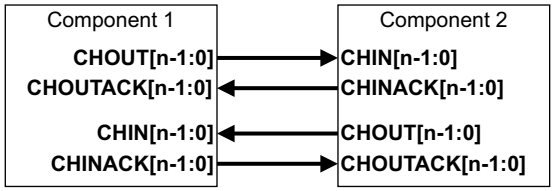


Figure C4-4 Asynchronous channel interface connection

Figure C4-5 shows how to connect two synchronous channel interfaces together.

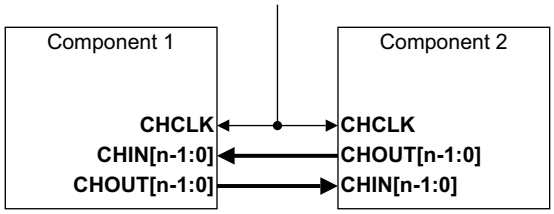


Figure C4-5 Synchronous channel interface connection

A component can support:

- Only the input channels. This is appropriate for components that do not generate events, but have to react to events from other components.
- Only the output channels. This is appropriate for components that generate events, but do not have to react to events from other components.
- Both the input and output channels.

If a component does not support both sets of channels, the unsupported outputs must be clamped as shown in Table C4-1 on page C4-77 and Table C4-2 on page C4-77.

If a component supports both input and output channels, the component must not reflect events on an input channel to the corresponding output channel.

C4.5 Synchronous and asynchronous conversions

Figure C4-6 shows a circuit that makes it possible to convert between synchronous and asynchronous versions of this interface.

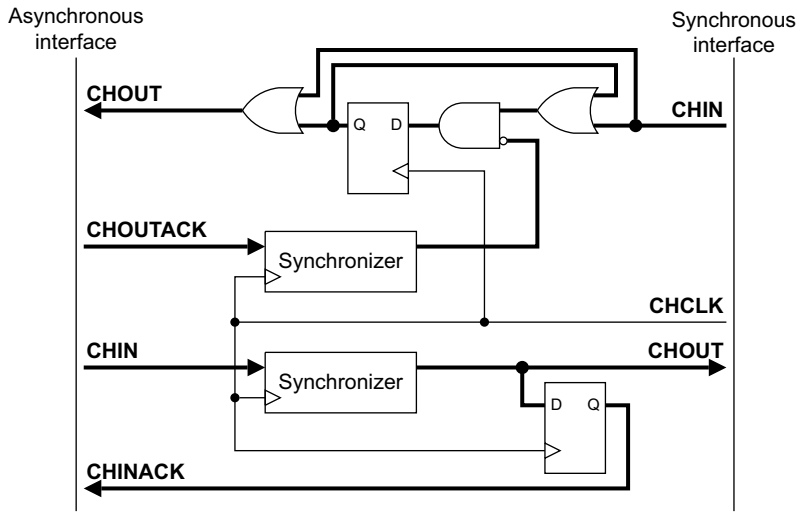


Figure C4-6 Asynchronous to synchronous converter

If you implement a synchronous to asynchronous converter, you increase the likelihood of events being merged as described in *Channel interface limitations* on page C4-74.

Chapter C5
Authentication Interface

This chapter defines the system requirements that control access to debug and trace peripherals, and how those requirements are met by CoreSight-compliant devices. It contains the following sections:

C5.1 About the authentication interface

The authentication interface aims to restrict access to debug and trace functionality for the following reasons:

- To prevent unauthorized people from modifying the behavior of the system, for example to prevent a mobile phone from reporting a fake identification number to the network. This requires authenticated access to invasive debug functions such as traditional core debug, but permits non-invasive tracing and profiling functions.
- To prevent unauthorized people from reverse engineering a product or discovering secrets stored within it, for example to read encryption keys. This requires authenticated access to all debug and trace functions.

It does not prevent against accidental access of debug functionality by rogue code, making a system impossible to debug. This is managed by the LAR that is optional in all CoreSight components, see *Lock Access Register, LAR* on page B2-48, and by the software lockout feature of the DAP.

C5.2 Definitions of Secure and invasive debug

This section defines Secure debug and invasive debug.

C5.2.1 Definition of Secure debug

A Non-secure debug operation is any operation where instructions executing on-chip with Non-secure privileges or operations external to the system, can cause the same effect. Any other operation is a Secure debug operation.

Debug operations that monitor the time taken by a Secure routine are not therefore considered Secure debug operations, because this can be measured by a combination of off-chip timing and Non-secure on-chip event generation. Operations that affect the time taken by a Secure routine are considered Secure debug operations.

C5.2.2 Definition of invasive debug

Any operation that changes the defined behavior of the system is invasive.

This includes any changes to the contents of memory, and insertion of instructions into a processor pipeline. It does not necessarily include effects that change the number of cycles taken to perform an operation, unless the number of cycles is defined architecturally.

An implementation can treat an IMPLEMENTATION DEFINED set of effects as invasive which change the observable behavior of the system but do not change the defined behavior of the system. This includes most effects that change the number of cycles taken to perform an operation.

C5.3 Authentication interface signals

Table C5-1 and Table C5-2 show the authentication interface signals that a component might support. If a component uses non-invasive enables then it must import the invasive equivalent, for example **SPIDEN** with **SPNIDEN**.

Table C5-1 Asynchronous authentication interface signals

Name	Direction	Description
DBGEN	Input	Invasive debug enable
NIDEN	Input	Non-invasive debug enable
SPIDEN	Input	Secure invasive debug enable
SPNIDEN	Input	Secure non-invasive debug enable

Table C5-2 Synchronous authentication interface signals

Name	Direction	Description
AUTHCLK	Input	Clock
AUTHRESETn	Input	Reset
DBGEN	Input	Invasive debug enable
NIDEN	Input	Non-invasive debug enable
SPIDEN	Input	Secure invasive debug enable
SPNIDEN	Input	Secure non-invasive debug enable

Use of the asynchronous authentication interface is deprecated.

C5.4 Authentication rules

The authentication rules are as follows:

1. For asynchronous interfaces, all signals must be sampled asynchronously. For synchronous interfaces, all signals are sampled synchronously on the rising edge of **AUTHCLK**. Typically, **AUTHCLK** is mapped onto another clock signal. It is IMPLEMENTATION DEFINED when a change of any of the authentication signals takes effect.

   For example, a processor core might ignore changes to the authentication signals while in Debug state. By extension, it is possible that a component only observes the signals on reset, but it is recommended that more frequent changes are permitted.

   ARM recommends that processors implementing the authentication interface specify a sequence of instructions that, when executed, wait until changes to the authentication signals have taken effect before continuing.

2. If **DBGEN** is LOW then no invasive debug must be permitted.

   Invasive debug is any debug operation that might cause the behavior of the system to be modified. Non-invasive debug, such as trace, is unaffected.

3. If **NIDEN** is LOW and **DBGEN** is LOW then no debug is permitted.

   This includes non-invasive debug.

4. If **NIDEN** is LOW and **DBGEN** is HIGH then this indicates that invasive debug and non-invasive debug are permitted. ARM recommends that these signals are not driven in this way.

   To ensure that a component that is non-invasive is correctly enabled, it must also import **DBGEN** in addition to **NIDEN** and internally OR the result.

5. If **SPIDEN** is LOW then no Secure invasive debug must be permitted.

6. If **SPNIDEN** is LOW and **SPIDEN** is LOW then no Secure debug is permitted.

7. If **SPNIDEN** is LOW and **SPIDEN** is HIGH then invasive and non-invasive Secure debug is permitted. ARM recommends that these signals are not driven in this way. To ensure that a component that is non-invasive is correctly enabled, it must also import **SPIDEN** in addition to **SPNIDEN**, and internally OR the result.

   Rules 5 to 7 are similar to rules 2 to 4, for Secure debugging. This is for systems that separate secure and non-secure data, for example systems implementing ARM Security Extensions. Secure non-invasive debug is any debug operation that might cause secure data to become known to a debugger. Secure invasive debug is any debug operation that might cause secure data to be changed by a debugger. If a debug component supports Secure non-invasive debug functions, **SPNIDEN**, then it must also observe the Secure invasive signal, **SPIDEN**.

8. If **SPIDEN** is HIGH and **DBGEN** is LOW then no invasive debug is permitted. ARM recommends that these signals are not driven in this way. To ensure that a component that supports Secure invasive debug is correctly controlled, it must import **DBGEN** in addition to **SPIDEN**, and internally AND the result.

9. If **SPNIDEN** is HIGH and **NIDEN** is LOW then no debug is permitted. ARM recommends that these signals are not driven in this way. To ensure that a component that supports Secure non-invasive debug is correctly controlled, it must import **NIDEN** in addition to **SPNIDEN**, and internally AND the result.

10. If the value of any of the authentication signals change, it is IMPLEMENTATION DEFINED when this will take effect.

    Pipeline effects mean that it is not generally possible for these signals to be precise. It is not recommended that they are used to enable and disable debug around specific regions of code without a full understanding of the pipeline behavior of the system.

The authentication rules can be summarized as follows:

- **SPIDEN**, **DBGEN**, **SPNIDEN**, and **NIDEN** enable Secure invasive debug, Non-secure invasive debug, Secure non-invasive debug, and Non-secure non-invasive debug respectively.

- Invasive functionality might require non-invasive functionality to be enabled to function correctly. Therefore where invasive debugging is enabled, non-invasive debugging must also be enabled.
- Secure functionality must be disabled if the corresponding Non-secure functionality is disabled.

For a debug component which supports the authentication interface, Table C5-3 and Table C5-4 show the equations that define whether a particular level of debug functionality is permitted:

Table C5-3 Component without Secure debug capabilities

Debug functionality	Equation
Invasive debug	**DBGEN**
Non-invasive debug	**DBGEN** | **NIDEN**

Table C5-4 Component with Secure debug capabilities

Debug functionality	Equation
Non-secure Invasive debug	**DBGEN**
Non-secure Non-invasive debug	**DBGEN** | **NIDEN**
Secure Invasive debug	**DBGEN** & **SPIDEN**
Secure Non-invasive debug	((**SPIDEN** & **DBGEN**) | **SPNIDEN**) & (**DBGEN** | **NIDEN**)

Table C5-5 shows the restrictions and their effects. Numbers in brackets indicate the rules that apply in each case, S indicates Secure, and NS indicates Non-secure.

Table C5-5 Authentication signal restrictions

SPIDEN	DBGEN	SPNIDEN	NIDEN	Legal signal combination	Invasive debug permitted		Non-invasive debug permitted	
					S	NS	S	NS
0	0	0	0	Yes	No (2,5)	No (2)	No (3,6)	No (3)
0	0	0	1	Yes	No (2,5)	No (2)	No (6)	Yes
0	0	1	0	No[a] (9)	No (2,5)	No (2)	No (3,6)	No (3)
0	0	1	1	Yes	No (2,5)	No (2)	Yes	Yes
0	1	0	0	No (4)	No (5)	Yes (4)	No (6)	Yes (4)
0	1	0	1	Yes	No (5)	Yes	No (6)	Yes
0	1	1	0	No (4)	No (5)	Yes (4)	Yes (4)	Yes (4)
0	1	1	1	Yes	No (5)	Yes	Yes	Yes
1	0	0	0	No (7)	No (2)	No (2)	No (3)	No (3)
1	0	0	1	No (7)	No (2)	No (2)	Yes (7)	Yes
1	0	1	0	No[a] (8,9)	No (2)	No (2)	No (3)	No (3)
1	0	1	1	No[a] (8)	No (2)	No (2)	Yes	Yes
1	1	0	0	No (4,7)	Yes (7)	Yes (4)	Yes (4,7)	Yes (4)

Table C5-5 Authentication signal restrictions (continued)

SPIDEN	DBGEN	SPNIDEN	NIDEN	Legal signal combination	Invasive debug permitted		Non-invasive debug permitted	
					S	NS	S	NS
1	1	0	1	No (7)	Yes (7)	Yes	Yes (7)	Yes
1	1	1	0	No (4)	Yes (4)	Yes (4)	Yes (4)	Yes (4)
1	1	1	1	Yes	Yes	Yes	Yes	Yes

a. These signal combinations were permitted in previous versions of the CoreSight architecture but are deprecated from v2.0.

C5.5 User mode debugging

Individual components might offer greater control over the level of debug permitted. For example, some processors implementing ARM Security Extensions are capable of granting permission to debug specific secure processes by permitting debugging of Secure User mode without permitting debugging of Secure privileged modes. This level of control is extended to the *Embedded Trace Macrocell* (ETM). For more information, see the *ARM® Embedded Trace Macrocell Architecture Specification.*

Figure C5-1 shows how the four signals of the CoreSight authentication interface interact with the two registers controlled by the Secure *Operating System* (OS), **SUIDEN** and **SUNIDEN**:

- If **DBGEN** is asserted, **NIDEN** is ignored and assumed asserted.
- If **SPIDEN** is asserted, **SPNIDEN** is ignored and assumed asserted.
- In all other cases, the permissions represented by all the boxes bounding each level of debug functionality must be granted before that level of debug functionality is enabled.

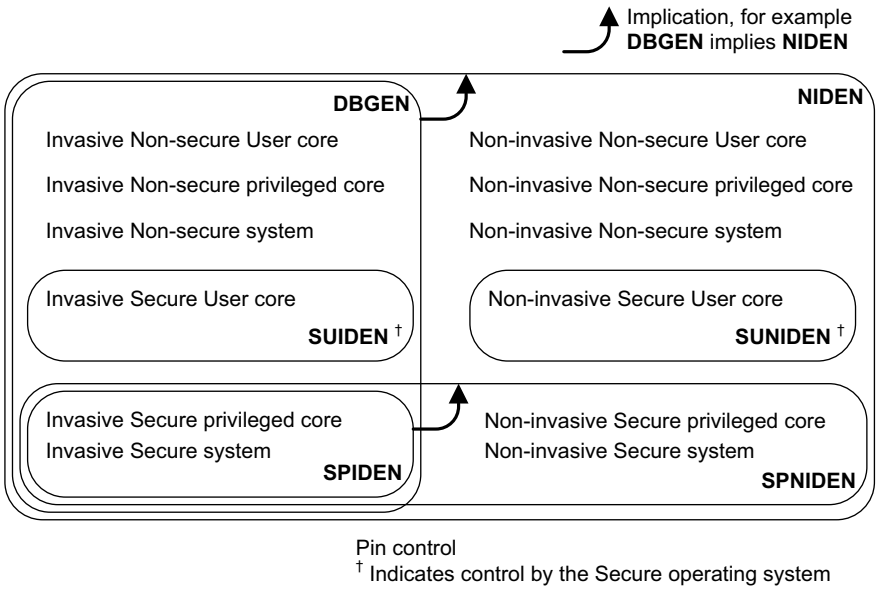


Figure C5-1 Interaction between CoreSight and ARM Security Extensions

C5.6 Control of the authentication interface

The authentication interface is controlled at the system level. For more information, see *Control of authentication interfaces* on page D2-108.

C5.7 Exemptions in the authentication interface

Debug functions that are only software accessible do not have to be controlled by the authentication interface. Instead, standard mechanisms for controlling software access to privileged and secure resources can be used.

Chapter C6
Timestamp Interface

This chapter describes the timestamp interface. It contains the following section:

C6.1 About the timestamp interface

A CoreSight system uses the wide timestamp interface to distribute a time value to debug components. Typically this time value is included in a trace stream to permit correlation of events in multiple trace streams. It contains the following signals:

TSCLK Interface clock.

TSRESETn Interface reset.

TSVALUEB[63:0] Timestamp value. The value is encoded as a natural binary number.

A value of 0 indicates that the timestamp is UNKNOWN. This might occur when the timestamp value source is disabled or when the timestamp value is being reset.

In a system with multiple components which implement the timestamp interface, the same timestamp is used by all components. Some systems might use different clocks or different timestamp distribution mechanisms and therefore there might be skew between the timestamp values observed by the components.

Some components might not implement a full 64-bit timestamp. These components use an IMPLEMENTATION DEFINED subset of the 64-bit timestamp value. ARM recommends the subset provides the largest set of unique timestamp values that the component can observe. This subset might depend on the clock speed of the component.

ARM recommends that the timestamp resolution is at least 10% of the fastest processor in the system.

Chapter C7
Topology Detection at the Component Level

This chapter describes how to detect components connected to the AMBA ATB interface and where they are logically located in any corresponding hierarchical connection. It contains the following sections:

C7.1 About topology detection at the component level

This chapter describes how to perform topology detection on each interface type. Chapter B3 *Topology Detection Registers* describes the topology detection requirements of CoreSight components. Chapter D6 *Topology Detection at the System Level* describes how debuggers can use this information to detect the topology of a target system.

C7.2 Interface types for topology detection

Each component has a number of interfaces, each containing one or more signals. Each interface is defined in terms of:

- A name, for example channel interface.
- A direction:
  - Master, always connected to one or more slaves of the same type.
  - Slave, always connected to one or more masters of the same type.
  - Bidirectional, always connected to one or more identical bidirectional interfaces.
  - Probe, read-only interface, used to trace the activity of a bus without affecting the behavior of that bus.

The list of interfaces must be defined for each block for the purposes of topology detection. Not all signals have to be part of an interface, these signals are not visible to topology detection. The signals that make up the interface must be strictly defined.

C7.2.1 Interfaces on standard components

Table C7-1 shows the interfaces present on common CoreSight components. See the appropriate Technical Reference Manual for specific interface details.

Table C7-1 Interfaces on some example components

Programmable component	Interfaces
CoreSight ETM[a] CoreSight PTM[b]	• AMBA ATB interface, master. • 2x event, master. **EXTOUT[1:0]**. • 4x event, slave. **EXTIN[3:0]**. • Event, master. **TRIGOUT**. • Variant: CoreETM, slave. The number of core interfaces can be read from the programmers' model of the ETM or PTM. The state of **DBGACK** can be driven directly in all ARM cores, including those that are not CoreSight compliant.
CoreSight ETB	• AMBA ATB interface, slave: • Event, master. **ACQCOMP**. • Event, master. **FULL**. • Event, slave. **TRIGIN**. • Event, slave. **FLUSHIN**.
TPIU	• AMBA ATB interface, slave: • Event, slave. **TRIGIN**. • Event, slave. **FLUSHIN**.
DAP	No topology detection interfaces.
HTM	• AMBA ATB interface, master: • 2x event, master. **HTMEXTOUT[1:0]**. • 2x event, master. **HTMEXTIN[1:0]**. • Event, master. **HTMTRIGGER**. • Variant: AHB, probe. The number of AHB interfaces can be read from the programmers' model of the HTM. The method to perform topology detection of this interface is not defined.

Table C7-1 Interfaces on some example components (continued)

Programmable component	Interfaces
CoreSight Funnel	• 8x AMBA ATB interface, slave. • AMBA ATB interface, master.
CTI	• 8x event, slave. **TRIGIN[7:0]**. • 8x event, master. **TRIGOUT[7:0]**. • Channel, bidirectional.
VIC (PL190/192)	32x event, master: **VICINTSOURCE[n]**.

a. An ETM that implements the ETMv3 or ETMv4 architecture.

b. A PTM that implements the PFTv1 architecture.

C7.3 Interface requirements for topology detection

For all controllable signals each interface type specifies:

- The signals on the master interface that must be controllable or observable.
- The signals on the slave interface that must be controllable or observable.
- The transitions on the interface that must be performed:
  - Before topology detection can begin.
  - To assert the master interface.
  - To check the slave interface is asserted.
  - To deassert the master interface.
  - To check the slave interface is deasserted.

If the interface is bidirectional, each interface to be tested must in turn be treated as a master while the other interfaces of that type are treated as slaves. See Chapter D6 *Topology Detection at the System Level*.

For signals that must be controllable, it must be possible to independently control the value of outputs, and read the value of inputs. See Chapter B3 *Topology Detection Registers*.

Usually each master interface specifies one output, and the slave interface specifies the corresponding input. The choice of signal must take the following into account:

- *Intermediate non-programmable components*.
- *Multi-way connections*.

C7.3.1 Intermediate non-programmable components

Sufficient control signals must be available to enable the interface to be driven to an active state so that it passes through any intermediate non-programmable components. For example, in AMBA ATB interfaces, **ATVALID** must be controllable, because if LOW an intermediate bridge will not pass any control signals through it.

C7.3.2 Multi-way connections

In a multi-way connection:

- Asserting and deasserting a master signal might cause an effect to be seen on multiple slaves.
- Asserting and deasserting a slave signal might cause an effect to be seen on multiple masters.

Sufficient signals must be controllable to cause the arbitration logic to route between the master and slave.

C7.4 Signals for topology detection

Table C7-2 shows the controllable signals for each interface type. See Table C7-1 on page C7-95.

Table C7-2 Controllable signals for each interface type

Interface	Master wire(s)	Slave wire(s)
AMBA ATB interface	**ATVALID**	**ATVALID**, **ATREADY**
CoreETM	**DBGACK**	**DBGACK**
Event	**EVENT**	**EVENT**, **EVENTACK**, if present
Channel, bidirectional	**CHOUT[0]**	**CHIN[0]**, **CHINACK[0]**, if asynchronous

In some ARM processors the value of **DBGACK** can only be controlled from a JTAG debugger.

In some ARM processors, the value of **DBGACK** cannot be controlled and cannot be used for topology detection. To perform topology detection on these processors, one of the following mechanisms must be used:

- A different controllable signal, which is IMPLEMENTATION DEFINED.
- The Device Affinity registers, DEVAFF0 and DEVAFF1, indicate the association of a processor with the ETM.

The event interface is defined for miscellaneous point-to-point connections carrying a one-bit signal. An event interface might implement an acknowledge signal, that if implemented must be controllable. Substitute **EVENT** and **EVENTACK** for the signal names with the names of the equivalent signals in the appropriate interface.

Methods for performing topology detection between masters and slaves of key interface types are given in the following sections:

- *AMBA ATB interface signals for topology detection*.
- *Core ETM signals for topology detection* on page C7-99.
- *Event signals for topology detection* on page C7-99.
- *Channel interface signals for topology detection* on page C7-100.

Refer to these sections in conjunction with the algorithm given in *Detection algorithm* on page D6-151. Specification of the topology detection requirements for each interface form part of the specification for that interface. Since it is impractical to do this at this time, the topology detection requirements are instead listed here for convenience.

C7.4.1 AMBA ATB interface signals for topology detection

Table C7-3 shows the topology detection sequence for an AMBA ATB interface.

Table C7-3 Topology detection for the AMBA ATB interface

Signal	Condition
Master preamble	**ATVALID** ← 0
Slave preamble	**ATREADY** ← 0
Master assert	**ATVALID** ← 1
Slave check asserted	**ATVALID** == 1
Slave post-assert	**ATREADY** ← 1

Table C7-3 Topology detection for the AMBA ATB interface (continued)

Signal	Condition
Master deassert	**ATVALID** ← 0
Slave check deasserted	**ATVALID** == 0
Slave post-deassert	**ATREADY** ← 0

C7.4.2 Core ETM signals for topology detection

Table C7-4 shows the topology detection sequence for a core ETM interface.

Table C7-4 Topology detection for the core ETM signals

Signal	Condition
Master preamble	**DBGACK** ← 0
Slave preamble	None
Master assert	**DBGACK** ← 1
Slave check asserted	**DBGACK** == 1
Slave post-assert	None
Master deassert	**DBGACK** ← 0
Slave check deasserted	**DBGACK** == 0
Slave post-deassert	None

C7.4.3 Event signals for topology detection

Table C7-5 shows the topology detection sequence for an event interface.

Table C7-5 Topology detection for the event interface

Signal	Condition
Master preamble	**EVENT** ← 0
Slave preamble	**EVENTACK** ← 0, if present
Master assert	**EVENT** ← 1
Slave check asserted	**EVENT** == 1
Slave post-assert	**EVENTACK** ← 1, if present
Master deassert	**EVENT** ← 0
Slave check deasserted	**EVENT** == 0
Slave post-deassert	**EVENTACK** ← 0, if present

C7.4.4 Channel interface signals for topology detection

Table C7-6 shows the topology detection sequence for a channel interface.

Table C7-6 Topology detection for a channel interface

Signal	Condition
Master preamble	**CHOUT[0]** ← 0
Slave preamble	**CHINACK[0]** ← 0, if present
Master assert	**CHOUT[0]** ← 1
Slave check asserted	**CHIN[0]** == 1
Slave post-assert	**CHINACK[0]** ← 1, if present
Master deassert	**CHOUT[0]** ← 0
Slave check deasserted	**CHIN[0]** == 0
Slave post-deassert	**CHINACK[0]** ← 0, if present

Part D

CoreSight System Architecture

Chapter D1
System Architecture

This chapter describes the system architecture. It contains the following section:

- *About the system architecture* on page D1-104.

D1.1 About the system architecture

This system architecture specifies:

- Rules that must be followed by all systems implementing CoreSight components.
- Additional information required by debuggers to use a CoreSight system.

The system architecture describes aspects of the system that might have an impact on various CoreSight components, for example the clock and power domains. The system architecture also deals with aspects of the complete SoC that are important for external tools that use CoreSight technology in the SoC.

Chapter D2
System Design

This chapter describes CoreSight system design. It contains the following sections:

D2.1 About system design

This chapter describes how to consider the following when integrating CoreSight components into a system:

- The clock and power domain structure visible to debuggers.
- Control of the signals in the authentication interface.
- How debug of the memory map for the AMBA 3 APB interface distinguishes between internal and external accesses.

D2.2 Clock and power domains

CoreSight is suitable for use in systems with many clock and power domains. However, all CoreSight systems can be considered to consist of the following clock and power domains:

System domain This is the domain in which most non-debug functionality resides. The clock frequencies in this domain can be asynchronous to the other domains and can vary over time to respond to varying performance requirements. The clocks can be stopped, and the power can be removed leading to the loss of all state.

Debug domain This is the domain in which most debug functionality resides. The power can be removed or the clocks can be stopped when debug functionality is not required, to reduce power consumption of the device.

Always on domain This is the domain in which the power controller and interface to the debugger resides. The power is never removed, even when the device is dormant. This enables the debugger to connect to the device even when powered down.

The debugger interface can make the following requests to the system, controlled by the debugger:

- Power up everything in the system domain. While this request is made, all logic in the system domain must be kept permanently powered up, and the clocks must be kept running.
- Power up everything in the debug domain. While this request is made, all logic in the debug domain must be kept permanently powered up, and the clocks must be kept running.
- Reset everything in the debug domain. When this request is made, all logic in the debug domain must be reset to its initial state.

The debugger interface is managed by the *Debug Access Port* (DAP). See the appropriate CoreSight Technical Reference Manual. You can implement more or fewer clock and power domains than this:

- You can implement more clock and power domains by subdividing one of the above clock and power domains, provided that all the clock and power domains respond to the appropriate debugger requests shown above. For example, you can implement two system clock domains, provided that both clocks are kept permanently running during a System Power Up request.
- You can implement fewer clock and power domains by combining two or more of the above clock and power domains, provided that the above debugger requests are still operational. For example, you can combine the system and debug power domains, provided that the combined domain is always powered up whenever a System Power Up request or a Debug Power Up request is made.

D2.3 Control of authentication interfaces

A CoreSight system prevents unauthorized debugging by disabling debug functionality, rather than by preventing access to the debug registers. This is controlled by the authentication interface. For more information about the authentication interface, see Chapter C5 *Authentication Interface*.

Each signal can be driven in one of the following ways:

- Tied LOW. This is most appropriate for production systems where the specified debug functionality is not required. This prevents in-the-field debugging. There is usually an alternative development chip with the same functionality enabled.
- Tied HIGH. This is most appropriate for prototype or development systems where authentication is not required.
- Connected to a fuse that is blown in production parts to disable debug functionality. This prevents in-the-field debugging.
- Driven by a custom authentication module, that unlocks debug functionality after a successful authentication sequence. This is the most flexible option. In systems where high security is required, it is recommended that a challenge-response mechanism is used, based on an on-chip random number generator or a hardware key unique to that device.

When secure debugging is enabled, secure operations are visible to the outside world, and in some cases to software running in the Non-secure world.

ARM recommends that devices are split into development and production devices:

- Development devices can have secure debugging enabled by authorized developers. All secure data must be replaced by test data suitable for development purposes, where financial loss is minimal if the test data is disclosed.
- Production devices can never have secure debugging enabled. These devices are loaded with the real secure data.

D2.4 Memory system design

This section describes issues that affect how CoreSight registers are made available to system software.

D2.4.1 Debug APB interface memory map

The Debug APB interface splits into two views. This enables CoreSight components to distinguish between internal accesses from system software, for example a debug monitor, and external accesses from a debugger. Figure D2-1 shows how the lower 2GB represents internal accesses, and the upper 2GB represents external accesses.

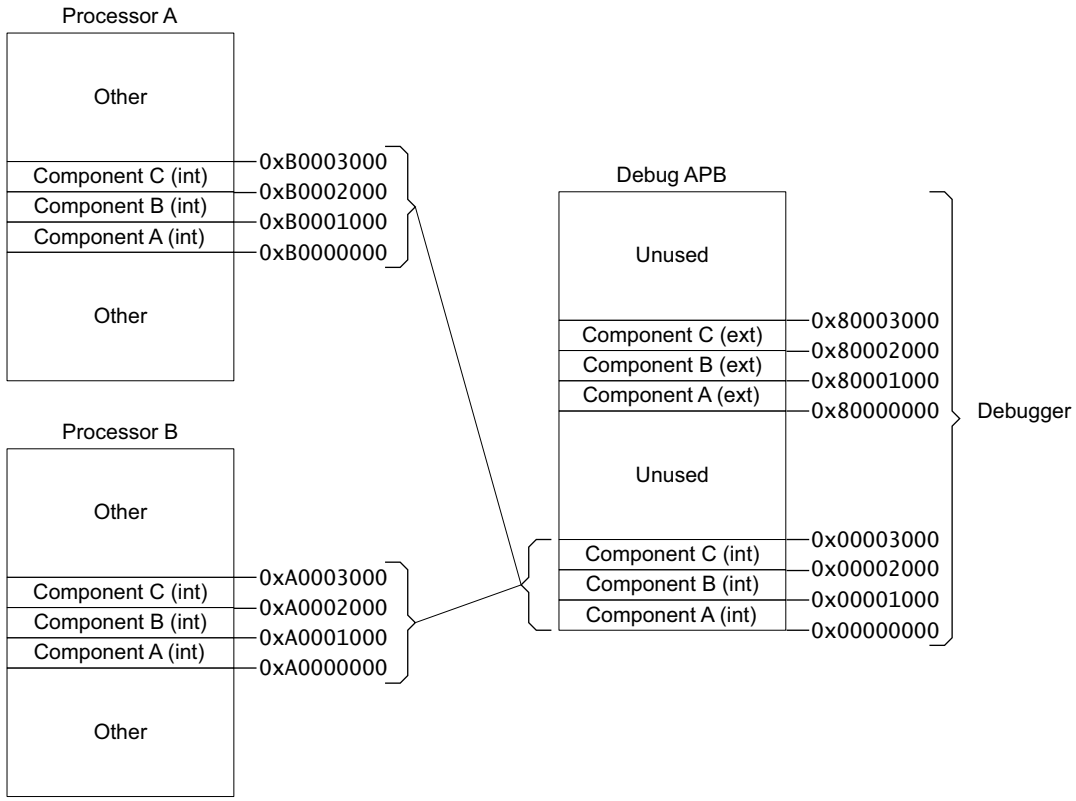


Figure D2-1 AMBA 3 APB interface memory map

It is not possible for system software to access the upper 2GB of the Debug APB interface address space. The DAP ensures this by setting **PADDRDBG[31]** LOW for all internal accesses.

While it is intended that external accesses use the upper 2GB of the address space for the Debug APB interface, it is possible for a debugger to directly access the lower 2GB if required. This enables the debugger to simulate an internal access, for example when debugging a debug monitor. A debugger can also access the lower 2GB by stopping a processor and issuing accesses through that processor.

D2.4.2 Access to the Debug APB interface

When designing a CoreSight system, you must ensure that the registers of CoreSight components are visible to privileged software.

Note

You must not prevent non-secure software from accessing the registers of CoreSight components, even if those components can debug secure software, because this seriously restricts debugging of non-secure software.

In an uncached system, the memory system ensures that unprivileged software cannot access CoreSight components by using information in the memory bus to determine if it is privileged. In a cached system, this information cannot be relied on, and therefore access control requires that the cache is correctly configured. In these systems, the region of memory corresponding to the Debug APB interface must be marked as Non-cacheable, privileged access only, accessible from Secure and Non-secure states.

If a custom authentication module is used, care must be taken to ensure that the authentication module can be accessed while debug is disabled. It is recommended that the authentication module is a CoreSight component, connected to the Debug APB interface.

Chapter D3
Physical Interface

This chapter describes the external pin interface, timing, and connector type required for the trace port on a target system. It contains the following sections:

D3.1 About the physical interface

This chapter defines the connectors used for debug communication and trace output from a CoreSight system. It describes the following connectors:

- *ARM JTAG 20* on page D3-113.
- *CoreSight 10 and CoreSight 20 connectors* on page D3-115.
- *ARM MICTOR* on page D3-119.

D3.2 ARM JTAG 20

The ARM JTAG 20 connector is a 20-way 2.54mm pitch connector. It supports the following interfaces:

- JTAG interface. This is based on IEEE 1149.1-1990 and includes the ARM **RTCK** signal.
- *Serial Wire Debug* (SWD) interface.
- *Serial Wire Output* (SWO) interface.

Figure D3-1 shows the ARM JTAG 20 connector pinout.

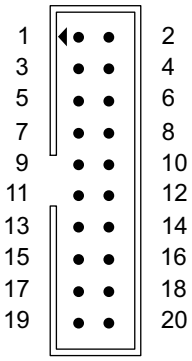


Figure D3-1 ARM JTAG 20 connector pinout

Table D3-1 shows the ARM JTAG 20 pinout as used on the target board.

Table D3-1 ARM JTAG 20 interface pinout table

Pin	Signal name
1	**VTREF**
2	**NC**
3	**nTRST**
4	**GND**
5	**TDI**
6	**GND**
7	**TMS/SWDIO**
8	**GND**
9	**TCK/SWCLK**
10	**GND**
11	**RTCK**
12	**GND**
13	**TDO/SWO**
14	**GND**
15	**nSRST**
16	**GND**
17	**DBGRQ/TRIGIN**

Table D3-1 ARM JTAG 20 interface pinout table (continued)

Pin	Signal name
18	**GND**
19	**DBGACK/TRIGOUT**
20	**GND**

See *Signal details* on page D3-124 for a description of the signals in Table D3-1 on page D3-113.

D3.3 CoreSight 10 and CoreSight 20 connectors

The CoreSight 10 and CoreSight 20 connectors are used for debug targets requiring JTAG, SWD, SWO, and low-bandwidth trace connectivity.

This section describes 10-way and 20-way connectors that are mounted on debug target boards. These are specified as 0.050 inch pitch two-row pin headers, Samtec FTSH or equivalent. See Samtec www.samtec.com.

The connectors are grouped into compatible sets according to the functions supported. Some targets support communication by both SWD and JTAG using the SWJ-DP block to switch between protocols.

The connector pin layouts are described in:

- *Combined pin names*.
- *CoreSight 10 pinouts* on page D3-116.
- *CoreSight 20 pinouts including trace* on page D3-117.

Note

The equivalent pin numbers on a CoreSight-compatible *Matched Impedance Connector* (MICTOR) connector pinout are shown in the tables. This enables you to build a target where the debug communication signals are brought in parallel to both connectors so that the target can be debugged using either physical connector.

D3.3.1 Combined pin names

Table D3-2 shows a summary of the pin names.

Table D3-2 Summary of pin names

Pin	Combined pin name	Pin	Combined pin name
1	VTref	11	Gnd/TgtPwr+Cap
2	**TMS/SWDIO**	12	**TraceClk/RTCK**
3	GND	13	Gnd/TgtPwr+Cap
4	**TCK/SWCLK**	14	**TraceD0/SWO**
5	GND	15	GND
6	**TDO/SWO/EXTa/TraceCtl**	16	**TraceD1/nTRST**
7	Key	17	GND
8	**TDI/EXTb/TraceCtl**	18	**TraceD2/TrigIn**
9	GNDDetect	19	GND
10	**TraceCtl/nSRST**	20	**TraceD3/TrigOut**

See *Signal details* on page D3-124 for a description of the signals in Table D3-2.

The following sections describe the use of pins 6, 8, 11, and 13:

- *EXTa and EXTb pins*.
- *GND/TgtPwr+Cap pins* on page D3-116.

EXTa and EXTb pins

Some pins on the connector have functions that are only used for certain connection layouts. Where a pin is not required for a particular debug communication protocol, it can be reused for a user-defined target function. These pins are labeled **EXTa** and **EXTb** in the tables in this chapter. You must select any alternate functions carefully, and also consider the effects of connecting debug equipment capable of communicating using multiple protocols.

GND/TgtPwr+Cap pins

There are two usage options for these pins:

Target boards

Standard target boards can connect these two pins directly to signal ground, GND.

Some special target boards, typically those for evaluation or demonstration purposes, can use these pins to supply power to the target board. In this case, the target board must include capacitors between each of the pins and signal ground. The capacitors must be situated extremely close to the connector so that they maintain an effective AC ground, that is, high frequency signal return path. Typical values for the capacitors are 10nF.

Debug equipment

Debug communication equipment designed to work with special valuation or demonstration target boards provides operating current, typically up to 100mA, at a target-specific supply voltage, for example, 3.3V, 5V, or 9V. ARM recommends that the debug equipment includes some protection if you connect it to standard target boards that have connected these pins directly to GND, for example, a current limiting circuit. This debug equipment must include capacitors between each of these power pins and signal ground. The capacitors must be situated extremely close to the connector so that they maintain an effective AC ground, that is, high frequency signal return path. Typical values for the capacitors are 10nF.

Standard debug communication equipment can connect these pins directly to GND. It is also possible for these pins to have only a high frequency signal return path to ground, using 10nF capacitors. This option is also compatible in the unlikely case where a target board has a connection between the debug connector **TgtPwr** pins, but is powered from another source.

D3.3.2 CoreSight 10 pinouts

There are two types of the pinout for a 10-pin connector, one supporting communication using SWD, and one using JTAG, and these are arranged to facilitate dynamic switching between the protocols.

SWD is the preferred protocol for debugging because it provides more data bandwidth over fewer pins, therefore freeing some for use by application functions. JTAG can be used where the target is communicating with a tool chain that does not support SWD, or with test tools performing board-level boundary scan testing, where it might be acceptable to sacrifice the functional pins multiplexed with JTAG.

Table D3-3 shows the CoreSight 10 for targets using SWD or JTAG for debug communication, and includes an optional *Serial Wire Output* (SWO) signal for conveying application and instrumentation trace.

Table D3-3 CoreSight 10 for SWD or JTAG systems

Pin name for SWD	Pin number		Pin name for JTAG	Pin number	
	10-way	MICTOR		10-way	MICTOR
VTref	1	12	VTref	1	12
SWDIO	2	17	**TMS**	2	17
GND	3	-	GND	3	-
SWCLK	4	15	**TCK**	4	15
GND	5	-	GND	5	-
SWO	6	11	**TDO**	6	11
Key	7	-	Key	7	-

Table D3-3 CoreSight 10 for SWD or JTAG systems (continued)

Pin name for SWD	Pin number		Pin name for JTAG	Pin number	
	10-way	MICTOR		10-way	MICTOR
NC/EXTb	8	19	**TDI**	8	19
GNDDetect	9	-	GNDDetect	9	-
nSRST	10	9	**nSRST**	10	9

The SWD layout is typically used in a CoreSight system that uses a SWJ-DP operating in SWD mode.

The JTAG layout is typically used in a CoreSight system including a JTAG-DP, or one with a SWJ-DP operating in JTAG mode, possibly because it is cascaded with other JTAG TAPs.

Note

- A target board can use this connector for performing board-level boundary scan but then switch its SWJ-DP into SWD mode for debugging according to the layout shown in Table D3-3 on page D3-116. This frees up pins 6 and 8 for either application functions or SWO.
- You do not have to choose the switching mode at the time of chip or board development. The connector can be switched and the target board operated in either SWD or JTAG mode.

D3.3.3 CoreSight 20 pinouts including trace

20-way connectors include support for a narrow trace port, up to four data bits, operating at moderate speed, up to 100 MSamples/sec.

Table D3-4 shows the CoreSight 20 for targets using SWD or JTAG for debug communication, and includes an optional **SWO** signal for conveying application/instrumentation trace. Alternatively, a target trace port operating in CoreSight normal or bypass modes might convey the **TraceCtl** signal on pin 6.

Both pin 6 and pin 8 in the SWD layout are shown with alternative extra signals, **EXTa** and **EXTb**. This enables flexibility to communicate other signals on these pins. For example, future target systems and trace equipment might convey two more trace data signals on these pins.

Table D3-4 CoreSight 20 for future SWD or JTAG systems

Pin name for SWD	Pin number		Pin name for JTAG	Pin number	
	20-way	MICTOR		20-way	MICTOR
VTref	1	12	VTref	1	12
SWDIO	2	17	**TMS**	2	17
GND	3	-	GND	3	-
SWCLK	4	15	**TCK**	4	15
GND	5	-	GND	5	-
SWO/EXTa/TraceCtl	6	11	**TDO**	6	11
Key	7	-	Key	7	-
NC/EXTb	8	(19)	**TDI**	8	19
GNDDetect	9	-	GNDDetect	9	-

Table D3-4 CoreSight 20 for future SWD or JTAG systems (continued)

Pin name for SWD	Pin number		Pin name for JTAG	Pin number	
	20-way	MICTOR		20-way	MICTOR
nSRST	10	9	**nSRST**	10	9
Gnd/TgtPwr+Cap	11	-	Gnd/TgtPwr+Cap	11	-
TraceClk	12	6	**TraceClk**	12	6
Gnd/TgtPwr+Cap	13	-	Gnd/TgtPwr+Cap	13	-
TraceD0	14	38	**TraceD0**	14	38
GND	15	-	GND	15	-
TraceD1	16	28	**TraceD1**	16	28
GND	17	-	GND	17	-
TraceD2	18	26	**TraceD2**	18	26
GND	19	-	GND	19	-
TraceD3	20	24	**TraceD3**	20	24

The SWD layout is typically used in a CoreSight system that uses a SWJ-DP operating in SWD mode.

The JTAG layout is typically used in a CoreSight system including a JTAG-DP, or one with a SWJ-DP operating JTAG mode, possibly because it is cascaded with other JTAG TAPs. This layout s the recommended debug connection for a processor built with support for instruction trace, that is, including an ETM.

Note

- A target board can use this layout for performing board-level boundary scan but then switch its SWJ-DP into SWD mode for debugging according to the layout shown in Table D3-4 on page D3-117. This frees up pins 6 and 8 for either application functions or SWO.
- You do not have to choose the switching mode at the time of chip or board development. The connector can be switched and the target board operated in either SWD or JTAG mode.

D3.4 ARM MICTOR

The following sections describe:

- *Target system connector*.
- *Target connector description*.
- *Decoding requirements for trace capture devices* on page D3-122.
- *Electrical characteristics* on page D3-123.

D3.4.1 Target system connector

The specified target system connector is the AMP MICTOR. This connector supports:

- JTAG interface. This is based on IEEE 1149.1-1990 and includes the ARM **RTCK** signal.
- Trace port interface, with up to 16 data pins.
- *Serial Wire Debug* (SWD) interface.
- *Serial Wire Output* (SWO) interface.
- Optional power supply pin.
- Reference voltage pin to enable support of a range of target voltages.
- Optional system reset request pin.
- Optional additional trigger pins for communicating with the target.

For tracing with large port widths, greater than 16 data pins, two connectors are required. See *Single target connector pinout* on page D3-120 and *Dual target connector pinout* on page D3-121.

The AMP MICTOR connector is a high-density matched-impedance connector. This connector has several important attributes:

- Direct connection to a logic analyzer probe using a high-density adapter cable with termination, for example HPE5346A from Agilent.
- Matching impedance characteristics, enabling the connector to be used at high speeds.
- A large number of ground fingers to ensure good signal integrity.
- Inclusion of one or both of the JTAG or SWD run-time control signals on the connector, enabling a single debug connection to the target.

Table D3-5 lists the AMP part numbers for the four possible connectors.

Table D3-5 Connector part numbers

AMP part number	Description
2-767004-2	Vertical, surface mount, board to board or cable connectors
767054-1	
767061-1	
767044-1	Right angle, straddle mount, board to board or cable connector

D3.4.2 Target connector description

This section contains details of the physical layout of the connector and recommended board orientation as follows:

- *Single target connector pinout* on page D3-120.
- *Dual target connector pinout* on page D3-121.

Single target connector pinout

Figure D3-2 shows how the connector and PCB can be oriented on the target system. This shows the view from above the PCB. The trace connector is mounted near to the edge of the board to minimize the intrusiveness of the TPA when the interconnect is a direct PCB to PCB link.

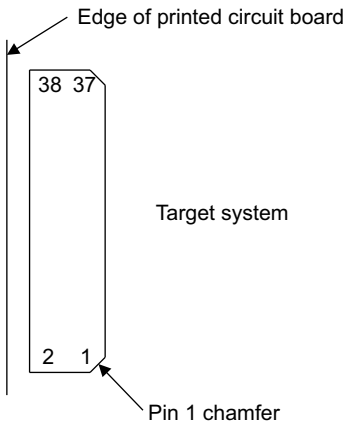


Figure D3-2 Recommended orientation for one connector

Table D3-6 shows all the connections on a single MICTOR connector, showing where differences exist between this and the connections specified in the *ARM® Embedded Trace Macrocell Architecture Specification*. If a signal is not implemented on the target system then it must be replaced with a logic 0 connection.

Table D3-6 Single target connector pinout

Pin	Signal	Pin	Signal
38	**TRACEDATA[0]**	37	**TRACEDATA[8]**
36	**TRACECTL**	35	**TRACEDATA[9]**
34	Logic 1	33	**TRACEDATA[10]**
32	Logic 0	31	**TRACEDATA[11]**
30	Logic 0	29	**TRACEDATA[12]**
28	**TRACEDATA[1]**	27	**TRACEDATA[13]**
26	**TRACEDATA[2]**	25	**TRACEDATA[14]**
24	**TRACEDATA[3]**	23	**TRACEDATA[15]**
22	**TRACEDATA[4]**	21	**nTRST**
20	**TRACEDATA[5]**	19	**TDI**
18	**TRACEDATA[6]**	17	**TMS/SWDIO**
16	**TRACEDATA[7]**	15	**TCK/SWCLK**
14	**VSupply**	13	**RTCK**
12	**VTRef**	11	**TDO/SWO**
10	No connection, was **EXTTRIG**	9	**nSRST**
8	**TRIGOUT/DBGACK**	7	**TRIGIN/DBGRQ**

Table D3-6 Single target connector pinout (continued)

Pin	Signal	Pin	Signal
6	**TRACECLK**	5	**GND**
4	No connection	3	No connection
2	No connection	1	No connection

Dual target connector pinout

Figure D3-3 shows the arrangement for two connectors. ARM recommends that they are placed in line, with pins 1 separated by 1.35 inches.

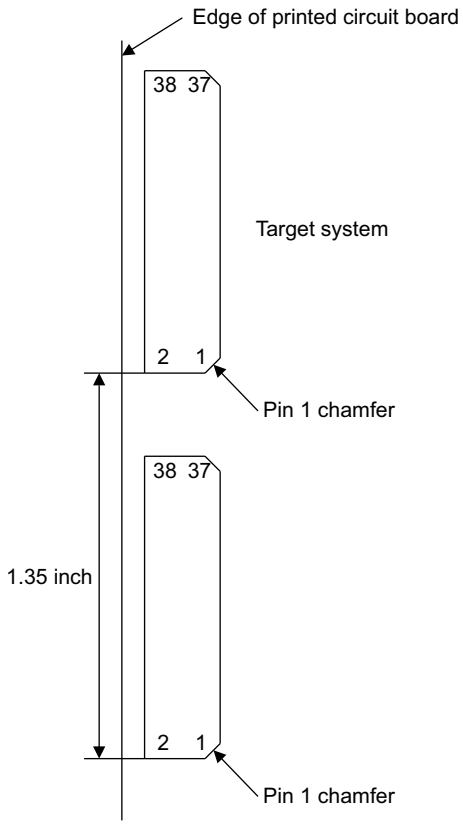


Figure D3-3 Recommended orientation for two connectors

Table D3-7 shows the connections for the secondary MICTOR connector. For the primary connector see Table D3-6 on page D3-120. If a signal is not implemented on the target system then it must be replaced with a logic 0 connection.

Table D3-7 Dual target connector pinout

Pin	Signal	Pin	Signal
38	**TRACEDATA[16]**	37	**TRACEDATA[24]**
36	Logic 0	35	**TRACEDATA[25]**
34	Logic 1	33	**TRACEDATA[26]**
32	Logic 0	31	**TRACEDATA[27]**

Table D3-7 Dual target connector pinout (continued)

Pin	Signal	Pin	Signal
30	Logic 0	29	**TRACEDATA[28]**
28	**TRACEDATA[17]**	27	**TRACEDATA[29]**
26	**TRACEDATA[18]**	25	**TRACEDATA[30]**
24	**TRACEDATA[19]**	23	**TRACEDATA[31]**
22	**TRACEDATA[20]**	21	No connection
20	**TRACEDATA[21]**	19	No connection
18	**TRACEDATA[22]**	17	No connection
16	**TRACEDATA[23]**	15	No connection
14	No connection	13	No connection
12	**VTRef**	11	No connection
10	No connection	9	No connection
8	No connection	7	No connection
6	**TRACECLK**	5	**GND**
4	No connection	3	No connection
2	No connection	1	No connection

D3.4.3 Decoding requirements for trace capture devices

Table D3-8 shows the three conditions that must be decoded by *Trace Capture Devices* (TCDs), for example, a TPA or a logic analyzer.

Table D3-8 Trace capture device decoding

TRACECTL	TRACEDATA[0]	TRACEDATA[1]	Trigger	Capture	Description
0	x	x	No	Yes	Normal trace data
1	0	0	Yes	Yes	Trigger packet
1	0	1	Yes	No	Trigger
1	1	x	No	No	Trace disable

Normal trace data

When trace data is indicated, only the full field of **TRACEDATA[n:0]** has to be stored. **TRACECTL** can be discarded to permit more efficient packing of data within the TCD.

Trigger packet

This is an unused encoding within CoreSight but must be implemented to maintain cross compatibility with ETMv3.x Trace Ports. All the **TRACEDATA** signals must be stored because there is further information emitted on this cycle, **TRACECTL** can be discarded. For more information see the *ARM® Embedded Trace Macrocell Architecture Specification*.

Trigger

A trigger is used as a marker to enable the TCD to stop capture after a predetermined number of cycles. No data is output on this cycle, **TRACEDATA[n:0]** and **TRACECTL** must not be captured.

Trace disable

This indicates that the current cycle must not be captured because it contains no useful information.

D3.4.4 Electrical characteristics

Debug equipment must be able to deal with a wide range of signal voltage levels. Typical ASIC operating voltages can range from 1V to 5V, although 1.8V to 3.3V is common.

It is important that you keep the track length differences as small as possible to minimize skew between signals. Crosstalk on the trace port must be kept to a minimum because it can cause erroneous trace results. Stubs on these traces can cause unpredictable responses, especially at high frequencies, and it is recommended that no stubs exist on the trace lines. If stubs are necessary you must make them as small as possible.

The trace port clock line, **TRACECLK** must be series-terminated as close as possible to the pins of the driving ASIC.

The maximum capacitance that is presented by the trace connector, cabling and interfacing logic must be less than 15pF.

There are no inherent restrictions on operating frequency, other than ASIC pad technology and TPA limitations. You must consider the following to maximize the speed at which trace capture is possible.

Figure D3-4 shows the timing for **TRACECLK**.

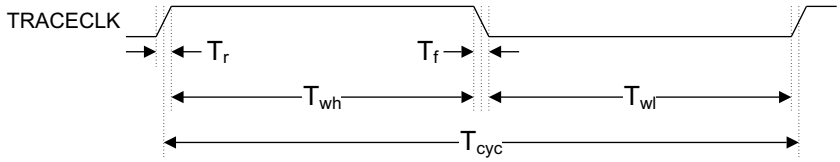


Figure D3-4 TRACECLK specification

It is recommended that trace ports provide a **TRACECLK** as symmetrical as possible, because both edges are used to capture trace. Figure D3-5 shows the setup and hold requirements of the trace data pins, **TRACEDATA[n:0]** and **TRACECTL**, with respect to **TRACECLK**.

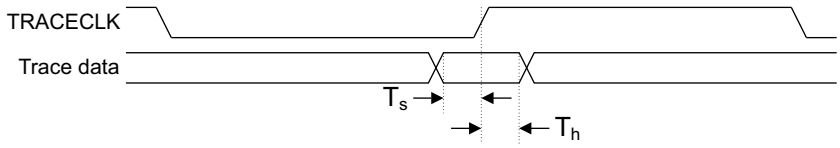


Figure D3-5 Trace data specification

ARM recommends that:

- The T_s, setup time, and T_h, hold time, are as large as possible, and that both are positive, because this is a requirement of some TPAs.
- The TPAs are able to delay each trace data signal individually by up to a whole clock period, to compensate for trace ports where T_s and T_h are not balanced or vary between data signals.

D3.5 Signal details

The signals on the target connector pins are:

- *VTRef output*.
- *TRACECLK output*.
- *TRACECTL output*.
- *TRACEDATA[n:1] output*.
- *Logic 1 input* on page D3-125.
- *Logic 0 input* on page D3-125.
- *TRIGIN/DBGRQ input* on page D3-125.
- *TRIGOUT/DBGACK output* on page D3-125.
- *nSRST input* on page D3-125.
- *nTRST input* on page D3-125.
- *TDI input* on page D3-125.
- *TMS input* on page D3-125.
- *SWDIO input/output* on page D3-125.
- *TCK/SWCLK input* on page D3-125.
- *RTCK output* on page D3-125.
- *TDO output* on page D3-126.
- *SWO output* on page D3-126.
- *VSupply output* on page D3-126.
- *GND* on page D3-126.
- *No connection* on page D3-126.

D3.5.1 VTRef output

The **VTRef** signal is intended to supply a logic-level reference voltage to enable debug equipment to adapt to the signaling levels of the target board.

Note

VTRef does not supply operating current to the debug equipment.

Target boards must supply a voltage that is nominally between 1V and 5V. With ±10% tolerance, this is minimum 0.9V, maximum 5.5V. The target board must provide a sufficiently low DC output impedance so that the output voltage does not change by more than 1% when supplying a nominal signal current, 0.4mA. Debug equipment that connects to this signal must interpret it as a signal rather than a power supply pin and not load it more heavily than a signal pin. The recommended maximum source or sink current is 0.4mA.

D3.5.2 TRACECLK output

The trace port must be sampled on both edges of this clock. There is no requirement for this clock to be linked to a core clock.

D3.5.3 TRACECTL output

This signal indicates if trace can be stored this cycle, in conjunction with **TRACEDATA[1:0]**. This signal does not have to be stored.

D3.5.4 TRACEDATA[n:1] output

This signal can be any size and represents the data generated from the trace system. To decompress the data an understanding of the data stream is required, the data can be wrapped up within the Formatter protocol, see *AMBA ATB interface* on page C2-70, or direct data from a single trace source.

D3.5.5 Logic 1 input

This is a signal pin that is at a voltage level and interpreted as logic 1, typically a resistor pull-up to **VTRef**.

D3.5.6 Logic 0 input

This is a signal pin that is at a voltage level and interpreted as logic 0, typically a resistor pull-down to **GND**.

D3.5.7 TRIGIN/DBGRQ input

This signal is used to change the behavior of on-chip logic, for example by connecting it to a Cross Trigger Interface. ARM recommends that this pin is pulled to a defined state, LOW, to avoid unintentional requests being made to any connected logic on-chip.

D3.5.8 TRIGOUT/DBGACK output

This signal can be connected to on-chip trigger generation logic such as a Cross Trigger Interface to enable events to be propagated to external devices.

D3.5.9 nSRST input

This is an open-collector output from the run control unit to the target system reset. This might also be an input to the run control unit so that a reset initiated on the target can be reported to the debugger.

You must pull this pin HIGH on the target to avoid unintentional resets when there is no connection.

For more details on the use of **nSRST**, see the *ARM® Debug Interface Architecture Specification*.

D3.5.10 nTRST input

The **nTRST** signal is an open-collector input from the run control unit to the **Reset** signal on the target JTAG port. This pin must be pulled HIGH on the target to avoid unintentional resets when there is no connection.

D3.5.11 TDI input

TDI is the Test Data In signal from the run control unit to the target JTAG port. ARM recommends that this pin is set to a defined state.

D3.5.12 TMS input

TMS is the Test Mode Select signal from the run control unit to the target JTAG port. This pin must be pulled up on the target so that the effect of any spurious **TCK**s when there is no connection is benign. For connectors that share JTAG and SWD signals, this is shared with the **SWDIO** signal.

D3.5.13 SWDIO input/output

The Serial Wire Data I/O pin sends and receives serial data to and from the target during debugging. For connectors that share JTAG and SWD signals, this is shared with the **TMS** signal.

D3.5.14 TCK/SWCLK input

TCK/**SWCLK** is the Test Clock signal from the run control unit to the target JTAG or SWD port. ARM recommends that this pin is pulled to a defined state.

D3.5.15 RTCK output

RTCK is the Return Test Clock signal from the target JTAG port to the run control unit. Some targets, such as ARM7TDMI-S™, must synchronize the JTAG port to internal clocks. To assist in meeting this requirement, you can use a returned, and re-timed, **TCK** to dynamically control the **TCK** rate.

D3.5.16 TDO output

TDO is the Test Data Out from the target JTAG port to the run control unit. This signal must be set to its inactive drive state, tristate, when the JTAG state machine is not in the Shift-IR or Shift-DR states. For connectors that share JTAG and SWD signals, this is shared with the **SWO** signal.

D3.5.17 SWO output

The Serial Wire Output pin can be used to provide trace data. For connectors that share JTAG and SWD signals, this is shared with the **TDO** signal.

D3.5.18 VSupply output

The **VSupply** signal enables the target board to supply operating current to debug equipment so that an additional power supply is not required. This might not be used by all debug equipment. The V_{DD} power rail typically drives the pin on the target board. Target board documentation indicates the **VSupply** pin voltage and the current available. Target boards must supply a voltage that is nominally between 2V and 5V. With ±10% tolerance, this is minimum 1.8V, and maximum 5.5V. A target board that drives this pin must provide a minimum of 250mA, and 400mA is recommended. Debug equipment must indicate the required supply voltage range and the current consumption over that range. This enables the user to determine if an external power supply is required to power the debug equipment. Target boards might have a limited amount of current available for external debug equipment, so a backup mechanism to power the debug equipment must be provided where **VSupply** is not connected, or is insufficient. For some hardware, this signal is unused.

D3.5.19 GND

This must be connected to 0V on the target board to provide a signal return and logic reference.

D3.5.20 No connection

No connection must be made to this pin.

Chapter D4
Trace Formatter

This chapter describes trace formatter requirements for CoreSight-compliant devices. It contains the following sections:

D4.1 About trace formatters

Formatters are methods for wrapping Trace Source IDs into the output Trace Data stream. This chapter specifies the format used by Trace Sinks to embed AMBA ATB interface source IDs into a single trace stream. For more information about the AMBA ATB protocol, see *AMBA ATB interface* on page C2-70. The protocol:

- Permits trace from several sources to be merged into a single stream and later separated.
- Does not place any requirements or constraints on the data that is emitted from trace sources.
- Is suitable for high-speed real-time decode.
- Can be transmitted and stored as a bitstream without the need for separate alignment information.
- Can be decoded even if the start of the trace is lost.
- Indicates to the TPA the position of the trigger signal around which trace capture is usually centered, eliminating the need for a separate pin.
- Indicates to the TPA when the trace port is inactive, eliminating the need for a separate flow control pin.

When only a single trace source is used, it is possible to disable the formatting to achieve better data throughput, provided that the embedded trigger and flow control information is not required by the TPA.

D4.2 Frame descriptions

The formatter protocol outputs data in 16-byte frames. Each frame consists of:

- Seven bytes of data.
- Eight mixed-use bytes, each of which contains:
  - One bit to indicate the use of the remaining seven bits.
  - Seven bits that can be data or a change of trace source ID.
- One byte of auxiliary bits, where each bit corresponds to one of the eight mixed-use bytes:
  - If the corresponding byte was data, this bit indicates the remaining bit of that data.
  - If the corresponding byte was an ID change, this bit indicates when that ID change takes effect.

Figure D4-1 shows the structure of a formatter frame.

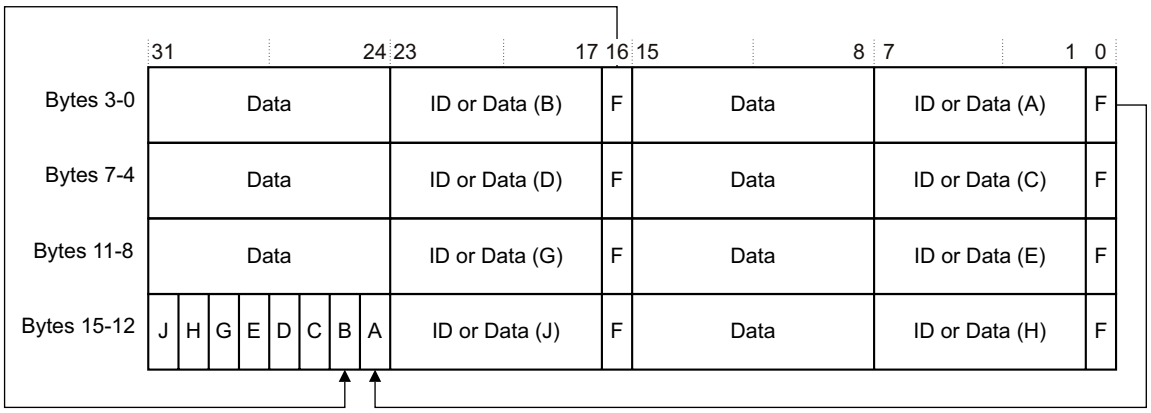


Figure D4-1 Formatter frame structure

Each time the ID changes, at least one byte of data must be output for that ID. Table D4-1 shows the meaning of each bit in a formatter frame. It is output least significant bit first, starting with bit 0.

Table D4-1 Meaning of bits in a formatter frame

Byte number	Bits	Description
0	0	ID or Data control for bits[7:1], see bits in Figure D4-1 marked F.
	7:1	Depends on bit 0: 0 = Data[7:1]. 1 = New ID.
1	7:0	Data[7:0].
2	7:0	ID or Data, see byte 0.
3	7:0	Data[7:0].
4	7:0	ID or Data, see byte 0.
5	7:0	Data[7:0].
6	7:0	ID or Data, see byte 0.
7	7:0	Data[7:0].
8	7:0	ID or Data, see byte 0.
9	7:0	Data[7:0].

Table D4-1 Meaning of bits in a formatter frame (continued)

Byte number	Bits	Description
10	7:0	ID or Data, see byte 0.
11	7:0	Data[7:0].
12	7:0	ID or Data, see byte 0.
13	7:0	Data[7:0].
14	7:0	ID or Data, see byte 0.
15	0	Auxiliary bit for byte 0, see bit in Figure D4-1 on page D4-129 marked A. The meaning of this bit depends on whether byte 0 contains data or a new ID: Data = Data[0]. New ID: 0 = Byte 1 corresponds to the new ID. 1 = Byte 1 corresponds to the old ID. The new ID takes effect from the next data byte after byte 1.
	1	Auxiliary bit for byte 2, see bit in Figure D4-1 on page D4-129 marked B. See bit 0. If byte 2 contains a new ID, this bit indicates whether the new ID takes effect from byte 3 or the following data byte.
	2	Auxiliary bit for byte 4, see bit in Figure D4-1 on page D4-129 marked C. See bit 0.
	3	Auxiliary bit for byte 6, see bit in Figure D4-1 on page D4-129 marked D. See bit 0.
	4	Auxiliary bit for byte 8, see bit in Figure D4-1 on page D4-129 marked E. See bit 0.
	5	Auxiliary bit for byte 10, see bit in Figure D4-1 on page D4-129 marked G. See bit 0.
	6	Auxiliary bit for byte 12, see bit in Figure D4-1 on page D4-129 marked H. See bit 0.
	7	Auxiliary bit for byte 14, see bit in Figure D4-1 on page D4-129 marked J. See bit 0. If byte 14 is a new ID, this bit is reserved. It must be zero, and must be ignored when decompressing the frame. The new ID takes effect from the first data byte of the next frame.

D4.2.1 Frame example

Two trace sources with IDs of `0x03` and `0x15` generate trace data and are interleaved on the trace bus one word of data at a time.

The following stream of bytes is output by the formatter:

`0x07`, `0xAA`, `0xA6`, `0xA7`, `0x2B`, `0xA8`, `0x54`, `0x52`, `0x52`, `0x54`, `0x07`, `0xCA`, `0xC6`, `0xC7`, `0xC8`, `0x1C`.

Figure D4-2 on page D4-131 shows the frame this represents.

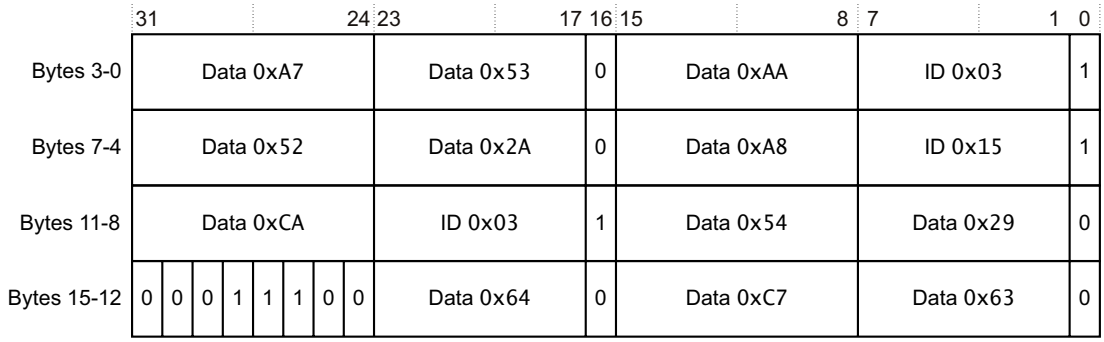


Figure D4-2 Example formatter frame

Table D4-2 shows how this frame is decoded.

Table D4-2 Decoding the example formatter frame

Byte	Comments	Data
0	Bit[0] is set, so this indicates a new ID. The new ID is `0x03`. Bit[0] of byte 15 is clear, so the new ID takes effect immediately.	-
1	Data byte corresponding to the new ID.	`0xAA`, ID `0x03`
2	Bit[0] is clear, so this is a data byte. Bit[0] of the data is taken from bit[1] of byte 15.	`0xA6`, ID `0x03`
3	Data byte.	`0xA7`, ID `0x03`
4	Bit[0] is set, so this indicates the new ID. The new ID is `0x15`. Bit[2] of byte 15 is set, so the next data byte continues to use the old ID.	-
5	Data byte.	`0xA8`, ID `0x03`
6	Bit[0] is clear, so this is a data byte. Bit[0] of the data is taken from bit[3] of byte 15.	`0x55`, ID `0x15`
7	Data byte.	`0x52`, ID `0x15`
8	Bit[0] is clear, so this is a data byte. Bit[0] of the data is taken from bit[4] of byte 15.	`0x53`, ID `0x15`
9	Data byte.	`0x54`, ID `0x15`
10	Bit[0] is set, so this indicates the new ID. The new ID is `0x03`. Bit[5] of byte 15 is clear, so the new ID takes effect immediately.	-
11	Data byte.	`0xCA`, ID `0x03`
12	Bit[0] is clear, so this is a data byte. Bit[0] of the data is taken from bit[6] of byte 15.	`0xC6`, ID `0x03`
13	Data byte.	`0xC7`, ID `0x03`
14	Bit[0] is clear, so this is a data byte. Bit[0] of the data is taken from bit[7] of byte 15.	`0xC8`, ID `0x03`
15	Auxiliary bits.	-

D4.2.2 Frame synchronization packet

The frame synchronization packet enables a TPA or trace decompresser to find the start of a frame. It is output periodically between frames. It is output least significant bit first, starting with bit[0]. In continuous mode the TPA might discard all frame synchronization packets after the start of a frame is found. See *Modes of operation* on page D4-134 for more information about continuous mode.

Figure D4-3 on page D4-132 shows a frame synchronization packet.

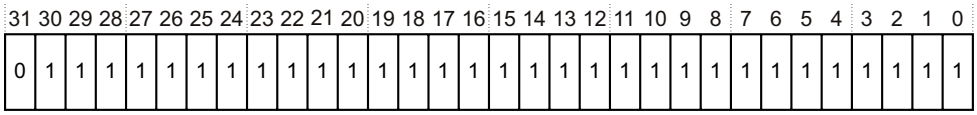


Figure D4-3 Full frame synchronization packet

This sequence cannot occur at any other time, provided that ID `0x7F` has not been used. See *Special trace source IDs* for more information on reserved source IDs.

Note

Frame synchronization packets and frame data are always multiples of 32-bits. However, because halfword synchronization packets can occur within frames and between frames, this does not mean that frame synchronization packets or frame data always start on a 32-bit boundary, and might occur on 16-bit boundaries.

D4.2.3 Halfword synchronization packet

This packet enables a TPA to detect when the trace port is idle and there is no trace to be captured. It is output between frames or within a frame. If it appears within a frame, it is always aligned to a 16-bit boundary. It is output least significant bit first, starting with bit[0].

This packet is only generated in continuous mode. When this packet is detected by a TPA, it must be discarded. It does not form part of a formatter frame. See *Modes of operation* on page D4-134 for more information about continuous mode. Figure D4-4 shows a halfword synchronization packet.

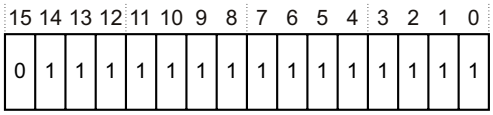


Figure D4-4 Halfword synchronization packet

Note

Frame synchronization packets and frame data are always multiples of 32-bits. However, because halfword synchronization packets can occur within frames and between frames, this does not mean that frame synchronization packets or frame data always start on a 32-bit boundary, and might occur on 16-bit boundaries.

D4.2.4 Special trace source IDs

The following IDs are for special purposes and must not be used under normal operation:

`0x00` This indicates a NULL trace source. Any data following this ID change must be ignored by the debugger. This is sometimes required where there is insufficient trace data available to complete a formatter frame.

`0x7F` This must never be used because it conflicts with the synchronization packet encodings.

`0x7E` Reserved for future use.

`0x7D` Indicates a trigger within the trace stream and is accompanied by one byte of data for each trigger. The value of each data byte indicates the ID of the trigger. A data byte value of zero indicates that the trigger ID is UNKNOWN.

`0x70-0x7C` Reserved.

D4.2.5 Data overheads

The formatter protocol adds an overhead of 6% to the bandwidth requirement of a trace port. It also requires one byte of additional trace every time the source ID changes. Components that arbitrate between trace sources should aim to minimize the frequency with which the source ID is changed. The CoreSight Trace Funnel can be configured with a minimum switching frequency for this reason.

The formatter can be bypassed under certain circumstances to eliminate this overhead. See *Bypass* on page D4-134.

D4.2.6 Repeating the trace source ID

If a large amount of trace is generated consecutively by a single source ID, the ID must be repeated periodically. This ensures that the debugger can determine the source of the trace even when the beginning of the trace has been lost.

ARM recommends that the source ID is repeated approximately every 10 frames.

D4.2.7 Indication of alignment points

In most trace protocols it is necessary to periodically indicate the beginning of a packet. This is called alignment synchronization. Most protocols achieve this by periodically outputting a packet similar to the Frame Synchronization Packet.

For protocols where this is not possible, you can use the formatter protocol to indicate the position of synchronization points. To do this, a trace source uses two source IDs. The trace source outputs trace using the first ID, then switches to the second ID at the first alignment point. It switches back to the first ID at the next alignment point, and continues switching on each subsequent alignment point.

A trace source that uses ID switching in this manner cannot use bypass mode. See *Bypass* on page D4-134.

D4.3 Modes of operation

The formatter can operate in one of three modes. Not all modes are supported by all components implementing a formatter. For example, an ETB does not need to support continuous mode.

D4.3.1 Bypass

In this mode, the trace is output without modification. No formatting information is inserted into the trace stream. Bypass mode can be used if all the following apply:

- Only one trace source ID is in use.
- If the trace is to be output over a trace port, the **TRACECTL** pin is implemented, and the TPA supports this pin. This enables the TPA to detect the trigger and to detect when no trace is available for capture. See *Decoding requirements for trace capture devices* on page D3-122.
- The debugger does not need to report the position of the trigger as seen by the trace sink. In bypass mode, the trigger ID `0x7D` is not generated.

To ensure that all trace is output from a trace sink when stopping trace, extra data might be added to the end of the trace stream. See *Bypass mode* on page D4-135.

D4.3.2 Normal

The formatter is enabled, and the **TRACECTL** pin is used to indicate when no trace is available for capture and to indicate the trigger. Half-synchronization packets are not generated. The TPA does not have to decode any part of the trace stream.

This mode is the easiest to support by TPAs designed for ETMs.

D4.3.3 Continuous

The formatter is enabled, but the **TRACECTL** pin is not used. The TPA must decode part of the formatter protocol to determine the position of the trigger and to detect when no trace is available for capture.

This mode is not implemented by trace sinks that do not output their data for a trace port.

D4.4 Flush of trace data at the end of operation

The formatter must ensure that all trace is output at the end of the trace. This is requested by an AMBA ATB protocol flush. When this occurs, there might be insufficient trace remaining to complete a frame, or to use all the pins in the trace port. This section describes how the remaining trace is output.

Note

When tracing is resumed, there might be some leftover trace generated by this flush sequence that is output before any new trace is output. You must look for the first synchronization packet in the protocol before starting to decompress the trace.

D4.4.1 Bypass mode

When running in bypass mode, the formatter must output an extra sequence at the end of the trace. This ensures that all trace stored in the formatter is output, even if, for example, there is insufficient trace to use all the pins of a trace port.

The sequence consists of a single bit set, followed by a number of zeros. This does not relate to the real trace data and must always be removed before decompression when the trace sink has been requested to stop trace output.

The following two examples show sequences observed on a 32-bit trace port. Figure D4-5 shows an example of how the last AMBA ATB protocol transaction left three bytes within the formatter.

31 24	23 16	15 8	7 0
0xAA [Real Data]	0x55 [Real Data]	0xAA [Real Data]	0x55 [Real Data]
0x01	0x55 [Real Data]	0xAA [Real Data]	0x55 [Real Data]
0x00	0x00	0x00	0x00
0x00	0x00	0x00	0x00

Figure D4-5 End of session example 1

Figure D4-6 shows an example of how the trace finishes on a 32-bit trace port boundary.

31 24	23 16	15 8	7 0
0x55 [Real Data]	0xAA [Real Data]	0x55 [Real Data]	0xAA [Real Data]
0x55 [Real Data]	0xAA [Real Data]	0x55 [Real Data]	0xAA [Real Data]
0x00	0x00	0x00	0x01
0x00	0x00	0x00	0x00

Figure D4-6 End of session example 2

D4.4.2 Normal and continuous mode

When running in normal or continuous mode, the formatter must complete the frame currently being output, using the null ID encoding, ID `0x00`. Any data associated with this ID can be ignored. Additional frames of data corresponding to the null ID can be generated to ensure that all trace has been output.

Chapter D5
ROM Table

This chapter describes the CoreSight ROM table. It contains the following sections:

D5.1 About the ROM table

This chapter describes the CoreSight ROM table. This table contains a list of components in the system, and must be present in all systems. Debuggers can use the ROM table to determine which components are implemented in a system.

D5.2 ROM table format

The following sections describe the format of the ROM table:

- *Component Identification Registers, 0xFF0-0xFFC*.
- *Peripheral Identification Registers, 0xFD0-0xFEC*.
- *MEMTYPE Register, 0xFCC*.
- *Reserved, 0xF00-0xFC8* on page D5-140.
- *ROM entries, 0x000-0xEFC* on page D5-140.

D5.2.1 Component Identification Registers, 0xFF0-0xFFC

The ROM table has a specific class, specified in *Component Identification Registers, CIDR0-CIDR3* on page B2-36, `0xB105100D`, spread over the lower byte of each word, as specified in *Component Identification Registers, CIDR0-CIDR3* on page B2-36.

D5.2.2 Peripheral Identification Registers, 0xFD0-0xFEC

The Peripheral Identification Registers uniquely identify the SoC or platform. If any of the components pointed to by the ROM are changed or any of their connections are changed, then these registers must be updated. Where a ROM table is implemented as part of a standard component, the following fields must be available for customer modification:

- JEP106 continuation code, four bits.
- JEP106 identification code, seven bits.
- Part number, 12 bits.
- Revision, four bits.
- Customer modified, four bits.

Note

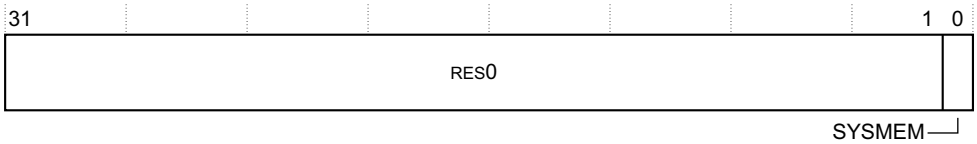
- The Peripheral ID is used by the debugger to name the description of the chip. When topology detection is performed, the description of the chip is saved along with the Peripheral ID. If that chip is connected to the debugger again, the debugger reads the Peripheral ID and retrieves the saved description. If two different chips have the same Peripheral ID then the debugger might retrieve the wrong description. In these circumstances, you must request that the debugger perform topology detection again.
- The 4KB Count must be RAZ.

D5.2.3 MEMTYPE Register, 0xFCC

Every ROM table must implement a MEMTYPE Register.

The MEMTYPE Register identifies the type of memory present on the bus that connects the DAP to the ROM table. In particular, it identifies whether system memory is connected to the bus. The MEMTYPE Register is a read-only register, at offset `0xFCC` from the base address of the ROM table.

The MEMTYPE bit assignments are:



Bits[31:1] RES0.

SYSMEM, bit[0]

System memory present. Indicates whether system memory is present on the bus that connects to the ROM table. The possible values are:

- 0 = System memory not present on bus. This is a dedicated debug bus.
- 1 = System memory is also present on this bus.

The value of bit[0] of the MEMTYPE Register tells you about the addresses that can be accessed:

- When MEMTYPE[0]==0 the ROM table indicates all the valid addresses in the memory system and the result of accessing any other address is UNPREDICTABLE. See the *ARM® Debug Interface Architecture Specification, ADIv5.0 to ADIv5.2* for more information.
- When MEMTYPE[0]==1 there might be other valid addresses in the memory system. The result of accessing these addresses is IMPLEMENTATION DEFINED:
  - This specification does not include any mechanism that a debugger can use to discover what addresses it can access, other than those indicated by the ROM table.
  - If a debugger accesses addresses other than those indicated by the ROM table this might have side effects on the system.

D5.2.4 Reserved, 0xF00-0xFC8

Registers in this region are RES0. They are reserved.

D5.2.5 ROM entries, 0x000-0xEFC

Each ROM table entry is 32-bits wide and occupies 4 bytes, aligned to 4-byte boundaries.

The first entry is at address 0x000. Each subsequent entry is at the next 4-byte boundary, until a value of 0x00000000 is read which is the final entry.

Table D5-1 shows the format of each entry in a ROM table.

Table D5-1 ROM entries

Offset in entry	Bits	Description
0	[31:0]	Entry[31:0]

Table D5-2 shows the format that the entries take.

Table D5-2 ROM entry format

Bits	Name	Description
[31:12]	Address offset	Base address of the highest 4KB block for that component, relative to the ROM address. Negative values are permitted using twos complement. ComponentAddress = ROMAddress + (AddressOffset SHL 12).
[11:9]	-	RES0.
[8:4]	Power domain ID	Indicates the power domain ID of the component. This field: • Is only valid if bit[2]==1, otherwise this field must be RAZ. • Supports up to 32 power domains using values 0x00 to 0x1F.
[3]	-	RES0.

Table D5-2 ROM entry format (continued)

Bits	Name	Description
[2]	Power domain ID valid	Indicates if the Power domain ID field contains a power domain ID: 0 = Power domain ID not provided. 1 = Power domain ID provided in bits[8:4].
[1]	Format	0 = 8-bit format. 1 = 32-bit format. For valid ROM table entries, as indicated by bit[0]==1, this bit is always 1.
[0]	Entry present	This bit indicates whether an entry is present at this location in the ROM table. Its possible values are: 0 = Entry not present. 1 = Entry present. See *Empty entries and the end of the ROM table* for more information.

The last entry has the value `0x00000000`, that is reserved.

If the component occupies several consecutive 4KB blocks, the base address of the highest block in memory is given. See *Class 0xF CoreLink, PrimeCell, or system component* on page B2-51. The base address of the block that contains the Management registers must be specified and from this, it is possible to establish the actual memory footprint of that component.

A ROM table might not use power domain IDs. All the components pointed to by the ROM table are in the same power domain as the ROM table. See *Use of power domain IDs* on page D5-143 for more information.

ARM recommends the power requestor that Appendix A *Power Requestor* describes.

Expansion above 960 entries

If more than 960 entries are required then expansion can be achieved by using one entry in the table to identify a second ROM table, giving 1919 possible entries (959+960). This method of linking ROM tables together can be repeated to create more entries if required.

Empty entries and the end of the ROM table

The descriptions of the debug components are stored in sequential locations in the ROM table, starting at the ROM table base address. However, a ROM table entry can be marked as *not present* by setting bit[0] of the entry to 0.

When scanning the ROM table, an entry marked as not present must be skipped. However you must not assume that an entry that is marked as not present represents the end of the ROM table. For example, a ROM table might be generated using static configuration tie-offs that indicate the presence or absence of particular devices, giving *not present* entries in the ROM table.

Note

- If the top-level ROM table is generated using static configuration tie-offs, then the Peripheral ID Register values must also depend on these tie-offs. This is because each possible topology must have a unique Peripheral ID.
- *ROM table hierarchy* on page D5-142 describes how there can be a hierarchy of ROM tables, and that such a hierarchy must have a top-level ROM table.

The end of the ROM table

Immediately after the last component entry in the ROM table there must be a blank ROM table entry. This means a 32-bit entry with the value `0x00000000`.

D5.3 ROM table hierarchy

Each entry can point to a CoreSight component or another ROM table. If it points to another ROM table, then that ROM must be read for CoreSight components, and might itself have entries for further ROM tables.

Each component, including other ROM tables, must appear only once in all the ROM tables visible by a debug agent. Figure D5-1 shows two ROM tables pointing to the same component, in this case ROM Table D.

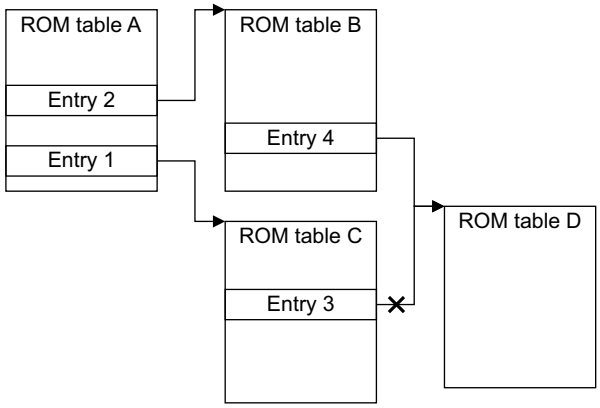


Figure D5-1 Prohibited duplicate ROM table references

A ROM table entry must not point to a ROM table that directly or indirectly points to it. Figure D5-2 shows an example of this prohibited circular reference.

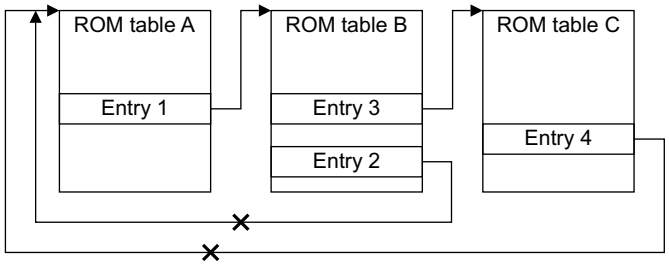


Figure D5-2 Prohibited circular ROM table references

The ID of a ROM table accessed from another ROM table is not used, and does not have to be unique:

- It can be set to a value representing the subsystem it describes, permitting that subsystem to be implemented on its own in the future.
- It can be set to the ID of the parent ROM.
- It can be set to a value reserved by an implementer for ROM tables only accessed from other ROM tables.

D5.4 Use of power domain IDs

If a ROM table includes any valid entries with a valid power domain ID then it must include a valid entry that points to a power requestor. The power requestor must not have a valid power domain ID and must be in same power domain as the ROM table. The power requestor permits a debugger to request power to the power domain IDs specified in the ROM table.

For any component with a valid power domain ID, the debugger must access the power requestor to request that power is applied to the component, otherwise the component might not be powered or might be powered down at any time.

The power domain ID is specific to components identified by the ROM table. This enables hierarchies of power domains to be constructed with each level enabling access to a level below.

Figure D5-3 shows an example ROM table that indicates the locations of three components.

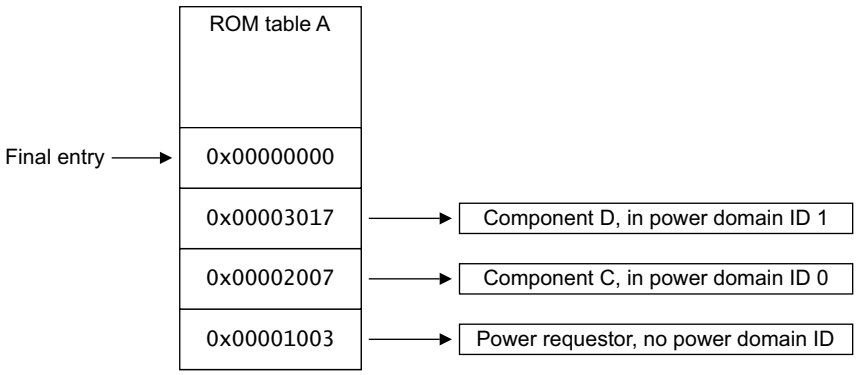


Figure D5-3 Single ROM table with power domain IDs

In Figure D5-3, the first component has no valid power domain ID and is the power requestor. The other two components, C and D, have power domain IDs of 0 and 1 respectively. The debugger requests power for these components, by using the power requestor.

Figure D5-4 shows an example system with nested power domains.

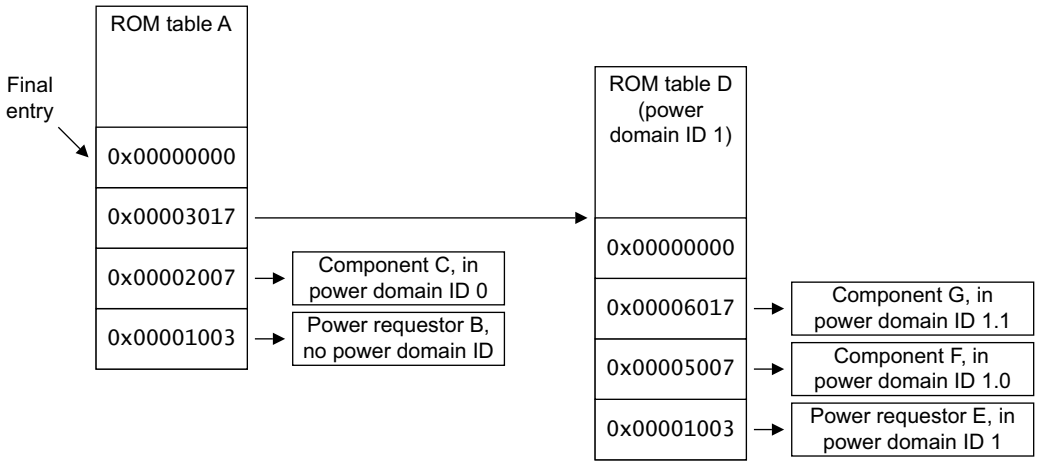


Figure D5-4 Multiple ROM tables with nested power domain IDs

In Figure D5-4, ROM table A is the top-level ROM table and indicates the presence of:

- Power requestor B, which must be in the same power domain as the ROM.
- Component C, which is in power domain ID 0.
- ROM table D, which is in power domain ID 1.

ROM table D indicates the presence of:

— Power requestor E, which is in the same power domain as ROM table D.

— Components F and G, with power domain IDs 0 and 1 respectively.

The power domain IDs indicated by ROM table D are different from the power domain IDs indicated by ROM table A and are nested within the power domain ID 1 indicated in ROM table A.

D5.4.1 Algorithm to discover power domain IDs

For each ROM table in the system, starting at the top-level ROM table:

1. For each valid entry in the ROM table, if the entry indicates no valid power domain ID, inspect the component to determine if it is a power requestor.

   Note

   If no power requestor is indicated in this ROM table then all entries in this ROM table must not have valid power domain IDs.

2. For every ROM table entry:
   a. If bit[2]==1 then power domain ID information is present, To request power for this entry, use the power domain ID in bits[8:4] to program the power requestor.
   b. If bit[2]==0 then no power domain ID information is present, so the component is powered.

D5.5 Location of the ROM table

It is necessary to provide a pointer to the top-level ROM table, and this section describes how to do this. While entries in a ROM table are always relative addresses, the top-level pointer to a ROM table always takes the form of an absolute address.

D5.5.1 From DAP

Each memory *Access Port* (AP) contains a register that either:

- Indicates the base address of a ROM table.
- Indicates the base address of a CoreSight component, that must be the only CoreSight component accessible from that AP.
- Indicates that no CoreSight components are accessible from this AP.

For implementation detail see the appropriate CoreSight Technical Reference Manual.

D5.5.2 From processor cores

The operating system or debug monitor must be aware of the memory map of the system in order to find the ROM table and CoreSight components.

Chapter D6
Topology Detection at the System Level

This chapter describes topology detection at the system level. It contains the following sections:

D6.1 About topology detection at the system level

Chapter B3 *Topology Detection Registers* describes the topology detection requirements of CoreSight components. Chapter C7 *Topology Detection at the Component Level* describes how to perform topology detection on each interface type. This chapter describes how debuggers can use this information to detect the topology of a target system.

D6.2 Detection

When connecting to a CoreSight system, a debugger:

1. Finds the DAP.
2. Ensures that the system is fully powered up, and that its clocks are running. The DAP provides facilities to assist this.
3. Looks for a ROM table giving the location of all components.
4. Compares the Peripheral ID of the ROM table against a list of saved system descriptions. For information on this ID, see *Peripheral Identification Registers, 0xFD0-0xFEC* on page D5-139.
5. If the description of the system with this ID is saved, that description is used. Otherwise the debugger:
   a. Identifies each component.
   b. Looks up information known about that component to determine what interfaces are supported and how to control them for topology detection.
   c. Performs topology detection. See *Detection algorithm* on page D6-151.
   d. Saves the description for later use.

D6.2.1 Saved descriptions

Topology detection might be invasive. It is therefore important that the description of the system is saved when discovered, so that the debugger can be connected non-invasively in the future. It must be possible for the user of the debugger to force topology detection to be redone, in case two different targets are accidentally produced with the same ROM table ID.

Note

Software running on the system must be able to determine the topology of the CoreSight system. Such software must not cease to function when topology detection registers are enabled.

D6.3 Components that are not recognized

When an unrecognized component is encountered, the JEDEC code and CoreSight component class of the component is used to indicate to the user what sort of component has been encountered and who to ask for further information. Alternatively, the DEVARCH register, if present, can be used to determine the generic architecture of a component. The component must be otherwise ignored.

D6.4 Detection algorithm

ARM recommends that a debugger connecting to a system executes the following algorithm, to determine the topology of the system:

```
for each component, c
        execute (component preamble) for c
for each interface type, t
        for each master interface and bidirectional interface of type t, m
                execute (master preamble) for interface m
        for each slave interface and bidirectional interface of type t, s
                execute (slave preamble) for interface s
        for each master interface and bidirectional interface of type t, m
                execute (master assert) for interface m
                for each slave interface and bidirectional interface of type t, s
                        if (slave check asserted) for interface s
                                record connection between m and s
                for each slave interface and bidirectional interface of type t, s
                       execute slave post-assert for interface s
                execute (master deassert) for interface m
                for each slave interface and bidirectional interface of type t, s
                        if not (slave check deasserted) for interface s
                                raise error
                for each slave interface and bidirectional interface of type t, s
                        execute (slave post-deassert) for interface s
for each component, c
        execute (component postamble) for c
```

Signals for topology detection on page C7-98 describes preambles, and assert and deassert sequences for common interfaces. If a component does not specify a preamble or postamble then they are as follows:

Component preamble

Set bit 0 of the ITCTRL.

Component postamble

Clear bit 0 of the ITCTRL.

Note

When a device has been in integration mode, it might not function with the original behavior. After performing integration or topology detection, you must reset the system to ensure correct behavior of CoreSight and other connected system components that are affected by the integration or topology detection.

Chapter D7
Compliance Criteria

This chapter describes the requirements for CoreSight compliance. It contains the following sections:

D7.1 About compliance classes

This chapter defines the requirements that a system must meet to claim CoreSight compliance. It refers to specific revisions of components available from ARM.

These requirements are aimed at interoperability between debuggers, and only cover behavior that is visible to such tools. The following behavior is specified:

- Minimum functionality. This functionality must be available in all compliant systems.
- Optional functionality. It is recommended that debuggers aiming to support compliant systems support this functionality.

Note

All systems can implement additional functionality, where it does not affect use of the minimum functionality. Debuggers might not be able to support this additional functionality.

Two levels of compliance are defined:

- CoreSight debug. This is the basic level of compliance. Note that a processor supporting CoreSight debug does not need to comply with the CoreSight visible component architecture, although this makes it easier to build a CoreSight system.
- CoreSight trace. This includes all the requirements for CoreSight debug, and adds trace functionality.

The level of compliance is claimed for debugging each processor in the system. For example, a system incorporating three processors might claim CoreSight trace for the first processor, CoreSight debug for the second, and no CoreSight compliance for the third.

D7.2 CoreSight debug

This section defines the CoreSight debug compliance class.

Note

A CoreSight component is a component that implements the CoreSight visible component architecture. The DAP is not a CoreSight component, but provides access to the CoreSight components.

D7.2.1 Minimum debug functionality

Systems claiming CoreSight debug compliance must conform to the following:

- Each CoreSight system must contain exactly one DAP, implementing a *JTAG Debug Port* (JTAG-DP) or a *Serial Wire Debug Port* (SW-DP) component, the JTAG or Serial Wire interface of which is accessible to the tools. For more information about implementing multiple systems containing DAPs, see *Multiple DAPs* on page D7-160.
- All CoreSight components must:
  - Be accessible from the DAP through a MEM-AP.
  - Be discoverable through a valid ROM table, that must itself conform to the above requirements for CoreSight components.
- All processors claiming CoreSight debug compliance must either:
  - Conform to the CoreSight visible component architecture, conforming to the above requirements for CoreSight components.
  - Be accessible using a JTAG TAP controller, which is connected either:

    In series with the JTAG TAP controller of the DAP JTAG-DP, connected to the **TDI** side of the DAP as Figure D7-8 on page D7-160 shows.

    In a chain of TAP controllers controlled by the DAP *JTAG Access Port* (JTAG-AP).
- All debug functionality must be visible and detectable, with its clocks running, when Debug Power Up is requested in the DAP JTAG-DP programmers' model, except where hidden because of security restrictions.
- All debug functionality must be operational when System Power Up is requested in the DAP JTAG-DP programmers' model, except where hidden because of security restrictions.
- All debug functionality must be reset to its initial state when Debug Reset is requested in the DAP JTAG-DP programmers' model.
- For each CoreSight component and JTAG controlled processor, all inputs and outputs defined as type event are connected to a *Cross Trigger Interface* (CTI) component, unless there is only one component in the system with event inputs or outputs, in which case no CTI is required. For ARM JTAG controlled processors, the required connections are documented in the *CoreSight Technology System Design Guide*.
- All channel interfaces of CoreSight components, for example those present on CTIs, are connected together, so that the channels are shared between all components. CoreSight technology from ARM provides a *Cross Trigger Matrix* (CTM) for connecting three or more channel interfaces together where required.
- No additional logic is permitted between components where this is visible to the tools, except where stated otherwise in this specification, for example external multiplexing as discussed in *Variant interfaces* on page B3-60.
- ARM recommends that the system is CoreSight compliant at all times, but it is recognized that this might not be achievable in some systems. If a system requires certain operations to be performed before it complies with the CoreSight compliance criteria, you must clearly state what these operations are, and clearly state that it is not CoreSight compliant until they have been performed.

D7.2.2 Optional debug functionality

CoreSight debug systems can also implement DAP visibility of system components through an AHB-AP or APB-AP.

Single-core debug

Figure D7-1 shows CoreSight debug in a single-core system in its simplest form. In this configuration no trace capabilities are provided. The processor is accessed using JTAG, through the DAP to ensure that it can be powered down without affecting other components on the master JTAG TAP chain.

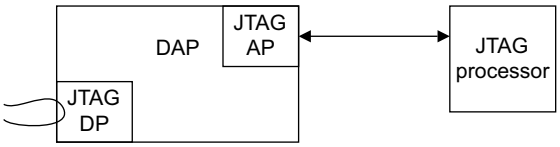


Figure D7-1 Single core with JTAG debug access

Multi-core debug

Figure D7-2 shows a multi-core CoreSight debug system. Here:

- One of the processors is a full CoreSight component.
- Cross triggering is supported between processors.
- Both processors provide access to program the CTI and processor with interfaces that comply with the CoreSight architecture.

This system could also use AHB-AP to provide access to system memory, although this has not been done in this case.

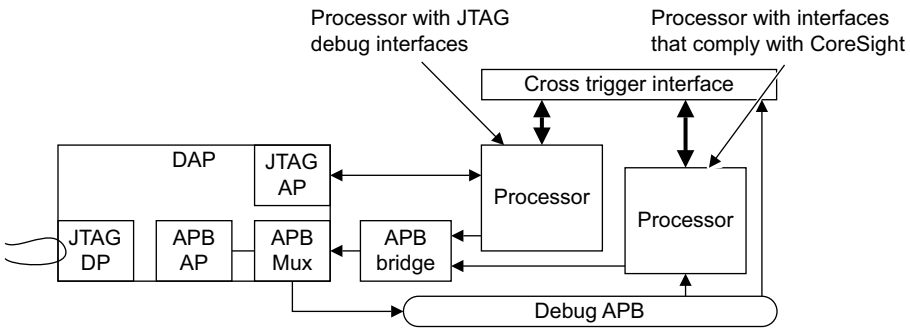


Figure D7-2 Multi-core system

D7.3 CoreSight trace

The section defines the CoreSight trace compliance class.

D7.3.1 Minimum trace functionality

Systems claiming CoreSight trace compliance must conform to the minimum requirements for CoreSight debug, plus the following:

- All processors claiming CoreSight trace compliance must either:
  - If it is an ARM-compatible processor, implement an ARM CoreSight ETM.
  - If it is any other type of processor, implement a trace solution that:
    Complies with the CoreSight visible component architecture.
    Provides at least instruction trace for the processor in question as a CoreSight trace source.
- The system implements one or more trace sinks:
  - If a TPIU is implemented, its output is connected to a compliant connector as defined in Chapter D3 *Physical Interface*.
- All CoreSight trace sources drain into one or more of the trace sinks:
  - Where two or more trace sources drain into the same trace sink, they are connected through one or more CoreSight trace funnels.
  - The trace cannot travel through multiple paths to reach the same endpoint. See example in Figure D7-3.

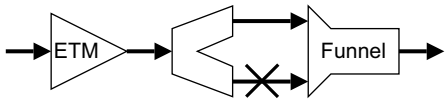


Figure D7-3 Non-compliant replicator and CoreSight trace funnel connection

A particular example that must be avoided is feedback. See example in Figure D7-4.

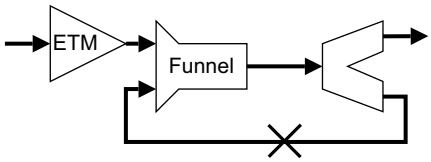


Figure D7-4 Non-compliant feedback loop

D7.3.2 Optional trace functionality

CoreSight debug systems can also implement CoreSight debug optional functionality and tracing of AHB buses using the ARM *AHB Trace Macrocell* (HTM).

Basic single-core trace

Figure D7-5 on page D7-158 shows an example system with single-core trace using the CoreSight infrastructure. The CoreSight-compliant ETM outputs directly to a TPIU for direct output of core trace off-chip. The tracing of only a single trace source enables the TPIU to be configured in bypass mode because source IDs do not need to be embedded in the trace data.

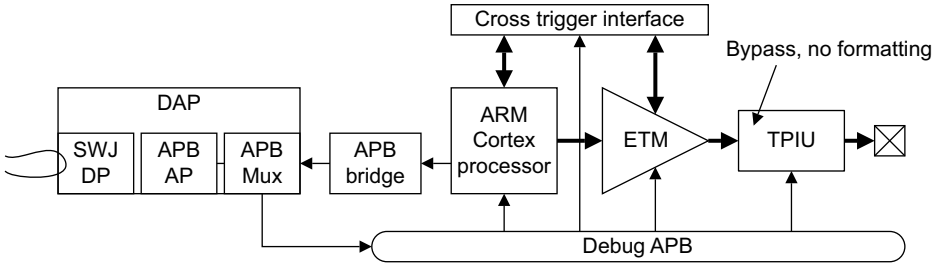


Figure D7-5 Single-core trace with formatting bypass

Advanced single-core trace

Figure D7-6 shows an example system with full trace capabilities in a single-core system. The ETM provides ARM processor tracing, and the HTM provides bus tracing. The CoreSight trace funnel combines trace from both sources into a single trace stream, that is then replicated to provide on-chip storage using the CoreSight ETB and output off-chip using the TPIU. Components can be configured using the DAP and cross triggered using the CTIs, through the CTM.

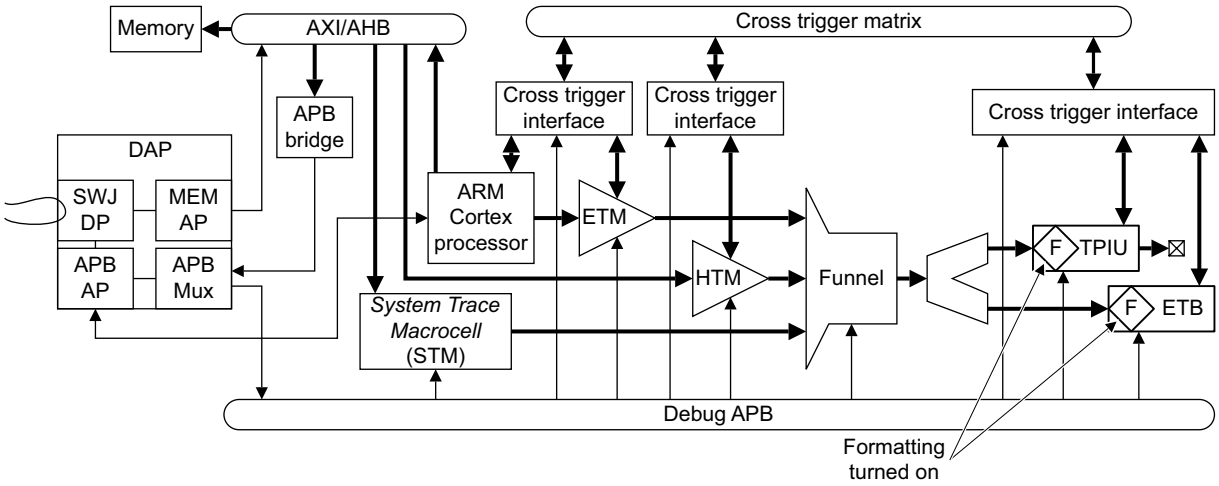


Figure D7-6 Full CoreSight trace with single core

Multi-core trace

Figure D7-7 on page D7-159 shows a system with an ARM processor and a DSP. A third smaller subsystem is added to support merging of multiple CoreSight AMBA ATB interfaces into a single trace stream.

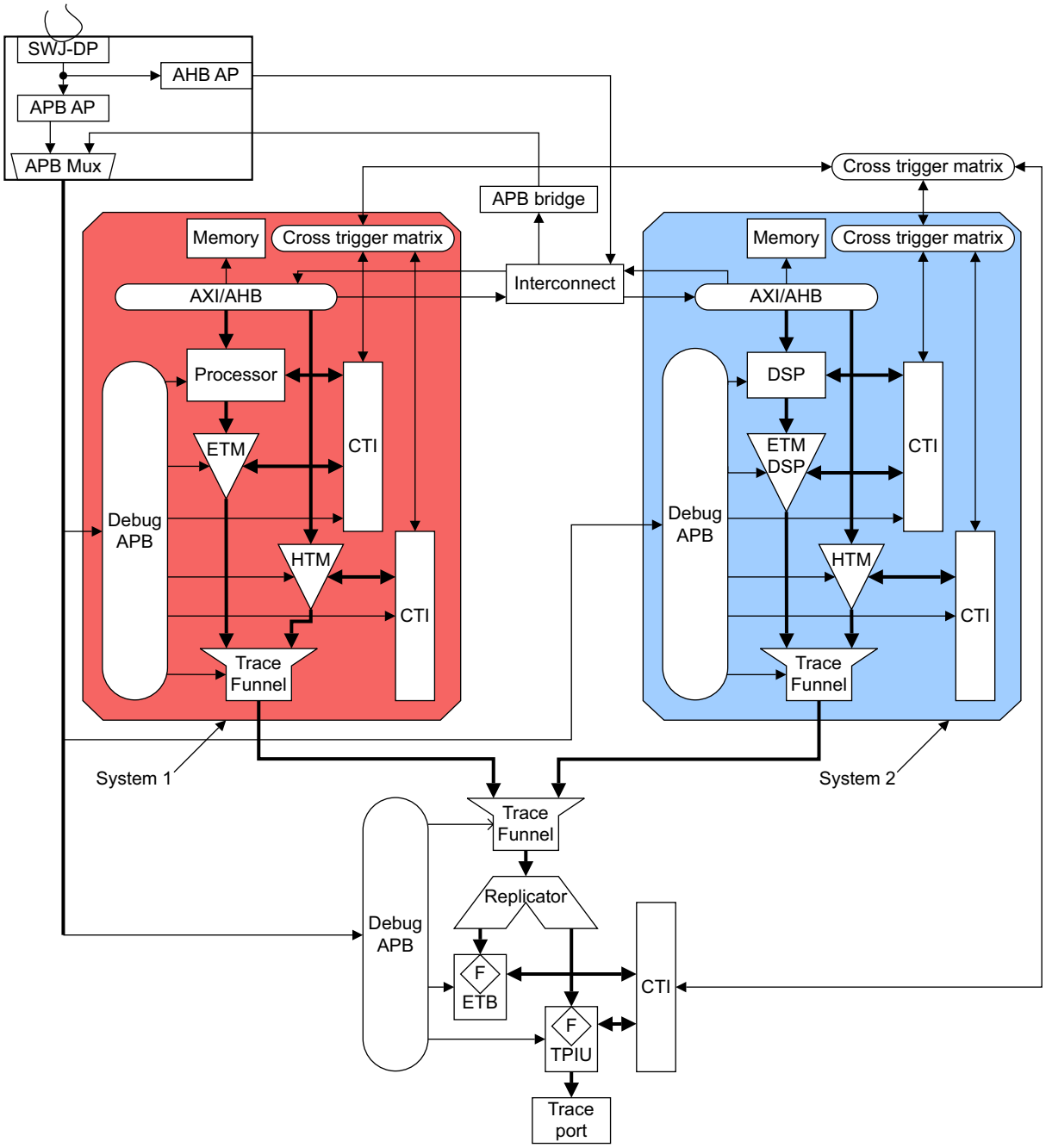


Figure D7-7 Full system trace with ARM processor and CoreSight compliant DSP

D7.4 Multiple DAPs

A system is defined as follows:

- All components are accessible through a single DAP, a MEM-AP, or a JTAG-AP.
- All components before a DAP in a serial JTAG TAP chain.

The following rules apply to the arrangement of multiple DAPs:

- The order of JTAG TAP controllers cannot be interleaved between systems. For example, if there are two systems sharing a JTAG TAP chain, each with a DAP and two JTAG processors connected in series with the DAP, the connections might be as Figure D7-8 shows and must not be as Figure D7-9 shows.

  In Figure D7-8:

  — System 1 is defined as the two processors before the first DAP in the TAP chain.

  — System 2 is defined as the two processors before the second DAP in the TAP chain.

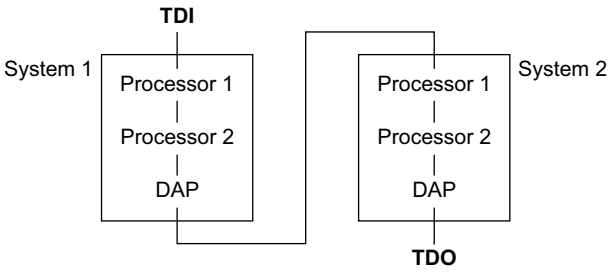


Figure D7-8 JTAG connections across systems

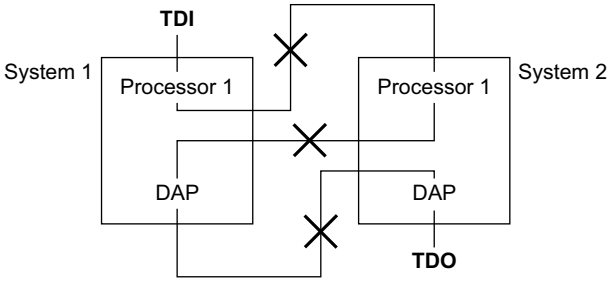


Figure D7-9 Non-compliant interleaved JTAG connections across systems

- Additional JTAG TAP controllers can be implemented in series with JTAG TAP controllers of the CoreSight systems. For example, in Figure D7-10, processor A is not part of either CoreSight system 1 or 2. Processor A is considered by the debugger to be part of system 2, because the DAP closest to the **TDO** side of processor A is in system 2. If the debugger does not recognize processor A then it is ignored, otherwise the debugger attempts topology detection on system 2 with processor A, and fails to find any connections between processor A and system 2.

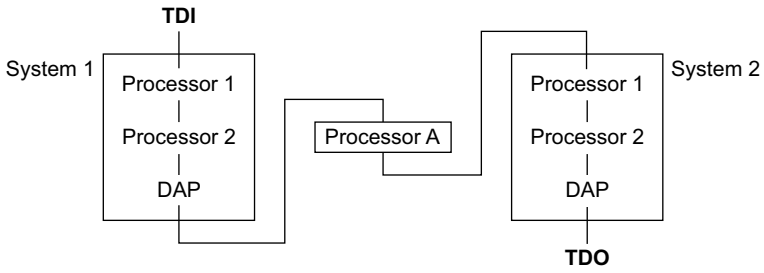


Figure D7-10 Systems with additional JTAG TAP controllers

- The DAP for one system cannot be accessed through the JTAG-AP of the DAP for another system, as shown in Figure D7-11 on page D7-161.

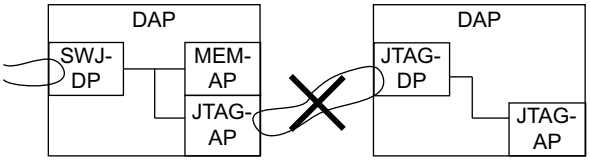


Figure D7-11 Non-compliant DAP connection

Part E
Appendices

Appendix A
Power Requestor

This appendix describes the power requestor which ARM recommends that some CoreSight components implement. It contains the following sections:

A.1 About the power requestor

The power requestor belongs to the component class `0x9`, CoreSight component.

The power requestor can control the application or removal of power with up to 32 power domains.

A.2 Register descriptions

Table A-1 shows the power requestor registers, in order of their address offset in the 4KB block.

Table A-1 Power requestor register summary

Offset	Type	Valid bits	Name	Description
0x000	RW	0-32	CDBGPWRUPREQ	Debug Power Request Register
0x004	RO	0-32	CDBGPWRUPACK	Debug Power Request Acknowledge Register
0x008-0xEFC	-	-	-	Reserved
0xF00	RW	1	ITCTRL	Integration Mode Control Register
0xF04-0xF9C	-	-	-	Reserved
0xFA0	RW	0-32	CLAIMSET	Claim Tag Set Register
0xFA4	RW	0-32	CLAIMCLR	Claim Tag Clear Register
0xFA8-0xFAC	-	-	-	Reserved
0xFB0	WO	32	LAR	Lock Access Register
0xFB4	RO	3	LSR	Lock Status Register
0xFB8	RO	8	AUTHSTATUS	Authentication Status Register
0xFBC	RO	32	DEVARCH	Device Architecture Register
0xFC0-0xFC4	-	-	-	Reserved
0xFC8	RO	6	DEVID	Device Configuration Register
0xFCC	RO	8	DEVTYPE	Device Type Identifier Register
0xFD0-0xFDC	RO	8	PIDR4-PIDR7	Peripheral Identification Registers, PIDR0-PIDR7
0xFE0-0xFEC	RO	8	PIDR0-PIDR3	
0xFF0-0xFFC	RO	8	CIDR0-CIDR3	Component Identification Registers, CIDR0-CIDR3

A.2.1 Debug Power Request Register, CDBGPWRUPREQ

The Debug Power Request Register is read/write. It controls whether a power request is active for a power domain. The register can issue power requests for up to 32 power domains.

The CDBGPWRUPREQ bit assignments are:

Description Controls the power requests to a maximum of 32 power domains.

Bits[31:0] Each bit controls whether a power request is active the corresponding power domain:

Bit[*n*]==0, power request is not active for power domain *n*.

Bit[*n*]==1, power request is active for power domain *n*.

The size of this field is IMPLEMENTATION DEFINED and is specified by DEVID.NUMREQ. Unimplemented bits are RAZ/WI.

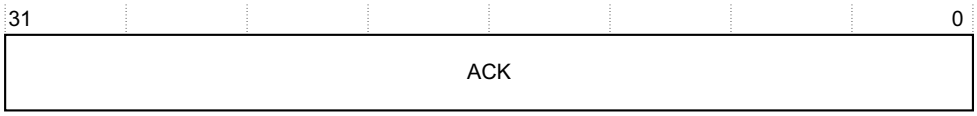
Short Name CDBGPWRUPREQ.

Location `0x000`.

A.2.2 Debug Power Request Acknowledge Register, CDBGPWRUPACK

The Debug Power Request Acknowledge Register is read-only. It returns the status of the power requests that CDBGPWRUPREQ issues. The register can monitor the power requests for up to 32 power domains.

The CDBGPWRUPACK bit assignments are:



Description Monitors the power requests to a maximum of 32 power domains.

Bits[31:0] Each bit monitors whether the corresponding power domain is powered:

Bit[*n*]==0, power domain *n* is not powered. Accesses to a component in power domain *n* might not be successful.

Bit[*n*]==1, power domain *n* is powered.

The size of this field is IMPLEMENTATION DEFINED and is specified by DEVID.NUMREQ. Unimplemented bits are RAZ.

Short Name CDBGPWRUPACK.

Location `0x004`.

A.2.3 Integration Mode Control Register, ITCTRL

The function of this register is as *Integration Mode Control Register, ITCTRL* on page B2-50 describes.

A.2.4 Claim Tag Set Register, CLAIMSET

The function of this register is as *Claim Tag Set Register, CLAIMSET* on page B2-49 describes.

A.2.5 Claim Tag Clear Register, CLAIMCLR

The function of this register is as *Claim Tag Clear Register, CLAIMCLR* on page B2-48 describes.

A.2.6 Lock Access Register, LAR

The function of this register is as *Lock Access Register, LAR* on page B2-48 describes.

A.2.7 Lock Status Register, LSR

The function of this register is as *Lock Status Register, LSR* on page B2-47 describes.

A.2.8 Authentication Status Register, AUTHSTATUS

The function of this register is as *Authentication Status Register, AUTHSTATUS* on page B2-43 describes.

A.2.9 Device Architecture Register, DEVARCH

This register implements the DEVARCH register as *Device Architecture Register, DEVARCH* on page B2-44 describes. The register returns `0x47700A34` so the values of the register fields are:

ARCHITECT `0x23B`. This is an 11-bit field.

PRESENT 1.

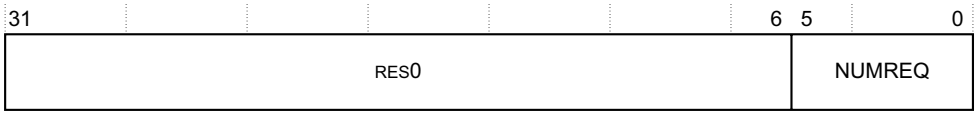
REVISION 0x0. This indicates revision 0 of the power requestor architecture.

ARCHID 0x0A34. Indicates that the architecture is a power requestor.

A.2.10 Device Configuration Register, DEVID

The Device Configuration Register is read-only.

The DEVID bit assignments are:



Description Indicates how many power domains the power requestor supports.

Bits [31:6] RES0.

[5:0] NUMREQ. Indicates the number of power requests that:

- CDBGPWRUPREQ.REQ can issue.
- CDBGPWRUPACK.ACK can monitor.

Short Name DEVID.

Location 0xFC8.

A.2.11 Device Type Identifier Register, DEVTYPE

The function of this register is as *Device Type Identifier Register, DEVTYPE* on page B2-40 describes, where:

- The MAJOR field returns 0x4. This indicates a debug control component.
- The SUB field returns 0x3. Indicates the component is a power requestor.

A.2.12 Peripheral Identification Registers, PIDR[7:0]

The function of these registers are as *Peripheral Identification Registers, PIDR0-PIDR7* on page B2-37 describes. The values of the following PIDR fields are IMPLEMENTATION DEFINED:

- DES_2, DES_1, and DES_0.
- PART_1, PART_0.
- REVISION.
- CMOD.
- REVAND.

A.2.13 Component Identification Registers, CIDR0-CIDR3

The function of these registers are as *Component Identification Registers, CIDR0-CIDR3* on page B2-36 describes, except CIDR1[7:4] returns 0x9. This indicates that the power requestor is a Class 0x9 CoreSight component.

A.3 Powering non-visible components

Some components do not have a visible programmers' model, for example a *Cross Trigger Matrix* (CTM) which is used in cross-triggering components in a CoreSight system. When requesting power for a visible component, power must be supplied to any non-visible components that provide functionality related to the visible component.

For example, if there are two *Cross Trigger Interfaces* (CTIs) connected through a CTM, if power is requested for the CTIs then the CTM must also be powered.

Appendix B
Revisions

This appendix describes the main technical changes between released versions of this specification.

Table B-1 Differences between v1.0 and v2.0

Change	Location
Renamed register fields for consistency across ARM documentation	Entire document
Clarified that all registers are accessed in little-endian format	*About the programmers' model* on page B2-32
Added new registers to Class `0x9` CoreSight component	*Device Configuration Register 1, DEVID1* on page B2-42 *Device Configuration Register 2, DEVID2* on page B2-43 *Device Architecture Register, DEVARCH* on page B2-44 *Device Affinity Register 0, DEVAFF0* on page B2-45 *Device Affinity Register 1, DEVAFF1* on page B2-46
Added new interfaces	Chapter C3 *Event Interface* Chapter C6 *Timestamp Interface*
Updated channel interface	Chapter C4 *Channel Interface*
Updated the definition of the authentication interface to deprecate some previously permitted encodings.	Chapter C5 *Authentication Interface*
Updated the connector information	Chapter D3 *Physical Interface*
Updated the permitted trace ID values to include `0x7D`	*Special trace source IDs* on page D4-132
Added the power requestor and ROM table values	*ROM entries, 0x000-0xEFC* on page D5-140 Appendix A *Power Requestor*

Glossary

This glossary describes some of the technical terms used in ARM documentation.

Advanced eXtensible Interface (AXI)
An AMBA bus protocol that supports:

- Separate phases for address or control and data.
- Unaligned data transfers using byte strobes.
- Burst-based transactions with only start address issued.
- Separate read and write data channels.
- Issuing multiple outstanding addresses.
- Out-of-order transaction completion.
- Addition of register stages to provide timing closure.

The AXI protocol includes optional signaling extensions for low-power operation.

See also AXI Coherency Extensions (ACE).

Advanced High-performance Bus (AHB)
An AMBA bus protocol supporting pipelined operation, with the address and data phases occurring during different clock periods. This means the address phase of a transfer can occur during the data phase of the previous transfer. AHB provides a subset of the functionality of the AMBA AXI protocol.

See also Advanced Microcontroller Bus Architecture (AMBA) and AHB-Lite.

Advanced Microcontroller Bus Architecture (AMBA)
The AMBA family of protocol specifications is the ARM open standard for on-chip buses. AMBA provides solutions for the interconnection and management of the functional blocks that make up a *System-on-Chip* (SoC). Applications include the development of embedded systems with one or more processors or signal processors and multiple peripherals.

Advanced Peripheral Bus (APB)
An AMBA bus protocol for ancillary or general-purpose peripherals such as timers, interrupt controllers, UARTs, and I/O ports. It connects to the main system bus through a system-to-peripheral bus bridge that helps reduce system power consumption.

Advanced Trace Bus (ATB)
A bus used by trace devices to share CoreSight capture resources.

AHB
See Advanced High-performance Bus (AHB).

AHB Access Port (AHB-AP)
An optional component of the DAP that provides an AHB interface to a SoC.

CoreSight supports access to a system bus infrastructure using the *AHB Access Port* (AHB-AP) in the *Debug Access Port* (DAP). The AHB-AP provides an AHB master port for direct access to system memory. Other bus protocols can use AHB bridges to map transactions. For example, you can use AHB to AXI bridges to provide AHB access to an AXI bus matrix.

See also Debug Access Port (DAP).

AHB Trace Macrocell (HTM)
A trace source that makes bus information visible. This information cannot be inferred from the processor using just a trace macrocell. HTM trace can provide:

- An understanding of multi-layer bus utilization.
- Software debug. For example, visibility of access to memory areas and data accesses.
- Bus event detection for trace trigger or filters, and for bus profiling.

See also Advanced High-performance Bus (AHB).

AHB-AP
See AHB Access Port (AHB-AP).

AHB-Lite
A subset of the full AMBA AHB protocol specification. It provides all of the basic functions required by the majority of AMBA AHB slave and master designs, particularly when used with a multi-layer AMBA interconnect.

Aligned
A data item stored at an address that is divisible by the number of bytes that defines its data size is said to be aligned. Aligned doublewords, words, and halfwords have addresses that are divisible by eight, four, and two respectively. An aligned access is one where the address of the access is aligned to the size of each element of the access.

AMBA
See Advanced Microcontroller Bus Architecture (AMBA).

APB
See Advanced Peripheral Bus (APB).

APB Access Port (APB-AP)
An optional component of the *Debug Access Port* (DAP) that provides an APB interface to a SoC, usually to its main functional buses or dedicated debug buses.

APB-AP
See APB Access Port (APB-AP).

ATB
See Advanced Trace Bus (ATB).

ATB bridge
A synchronous ATB bridge provides a register slice that helps timing closure by adding a pipeline stage. It also provides a unidirectional link between two synchronous ATB domains.

An asynchronous ATB bridge provides a unidirectional link between two ATB domains with asynchronous clocks. It supports connection of components with ATB ports in different clock domains.

See also Advanced Trace Bus (ATB).

AXI
See Advanced eXtensible Interface (AXI).

AXI Coherency Extensions (ACE)
The *AXI Coherency Extensions* (ACE) provide additional channels and signaling to an AXI interface to support system level cache coherency.

Cold reset
Also known as power-on reset. Starting the processor by turning power on. Turning power off and then back on again clears main memory and many internal settings. Some program failures can lock up the processor and require a cold reset to restart the system. In other cases, only a warm reset is required.

See also Warm reset.

Core reset
See Warm reset.

CoreSight
ARM on-chip debug and trace components, that provide the infrastructure for monitoring, tracing, and debugging a complete system on chip.

See also CoreSight ECT and CoreSight ETM.

CoreSight ECT
See Embedded Cross Trigger (ECT).

CoreSight ETB
See Embedded Trace Buffer (ETB).

CoreSight ETM
See Embedded Trace Macrocell (ETM).

Cross Trigger Interface (CTI)
Part of an *Embedded Cross Trigger* (ECT) device. In an ECT, the CTI provides the interface between a processor or ETM and the CTM.

Cross Trigger Matrix (CTM)
In an ECT device, the CTM combines the trigger requests generated by CTIs and broadcasts them to all CTIs as channel triggers.

CTI
See Cross Trigger Interface (CTI).

CTM
See Cross Trigger Matrix (CTM).

DAP
See Debug Access Port (DAP).

DBGTAP
See Debug Test Access Port (DBGTAP).

Debug Access Port (DAP)
A block that acts as a master on a system bus and provides access to the bus from an external debugger.

Debug Test Access Port (DBGTAP)
A debug control and data interface based on IEEE 1149.1 JTAG *Test Access Port* (TAP). The interface has four or five signals.

Debugger
A debugging system that includes a program, used to detect, locate, and correct software faults, together with custom hardware that supports software debugging.

DNM
See Do-Not-Modify (DNM).

Do-Not-Modify (DNM)
A value that must not be altered by software. DNM fields read as UNKNOWN values, and must only be written with the value read from the same field on the same processor.

Doubleword
A 64-bit data item. Doublewords are normally at least word-aligned in ARM systems.

Doubleword-aligned
A data item having a memory address that is divisible by eight.

ECT
See Embedded Cross Trigger (ECT).

Embedded Cross Trigger (ECT)
A modular system that supports the interaction and synchronization of multiple triggering events with an SoC. It comprises:
- *Cross Trigger Interface* (CTI).
- *Cross Trigger Matrix* (CTM).

Embedded Trace Buffer (ETB)
A Logic block that extends the information capture functionality of a trace macrocell.

Embedded Trace Macrocell (ETM)
A hardware macrocell that, when connected to a processor, outputs trace information on a trace port. The ETM provides processor driven trace through a trace port compliant to the ATB protocol. An ETM always supports instruction trace, and might support data trace.

ETB *See* Embedded Trace Buffer (ETB).

ETM *See* Embedded Trace Macrocell (ETM).

Event In an ARM trace macrocell:

Simple An observable condition that a trace macrocell can use to control aspects of a trace.

Complex A boolean combination of simple events that a trace macrocell can use to control aspects of a trace.

Formatter In an ETB or TPIU, an internal input block that embeds the trace source ID in the data to create a single trace stream.

Halfword A 16-bit data item. Halfwords are normally halfword-aligned in ARM systems.

Host A computer that provides data and other services to another computer. Especially, a computer providing debugging services to a target being debugged.

HTM *See* AHB Trace Macrocell (HTM).

IEEE 1149.1 The IEEE Standard that defines TAP. Commonly referred to as JTAG.

See *IEEE Std 1149.1-1990 IEEE Standard Test Access Port and Boundary-Scan Architecture* specification available from the IEEE Standards Association `http://standards.ieee.org`.

IGN An abbreviation for Ignore, when describing the behavior of a register or memory access.

IMPLEMENTATION DEFINED
Behavior that is not defined by the architecture, but is defined and documented by individual implementations.

IMPLEMENTATION SPECIFIC
Behavior that is not architecturally defined, and might not be documented by an individual implementation. Used when there are a number of implementation options available and the option chosen does not affect software compatibility.

See also IMPLEMENTATION DEFINED.

In-Circuit Emulator A device enabling access to and modification of the signals of a circuit while that circuit is operating.

Instruction Synchronization Barrier (ISB)
An operation to ensure that any instruction that comes after the ISB operation is fetched only after the ISB has completed.

Instrumentation trace A component for debugging real-time systems through a simple memory-mapped trace interface. It providing `printf` style debugging.

Intelligent Energy Manager
An energy manager solution consisting of both software and hardware components that function together to prolong battery life in an ARM processor based device.

ISB *See* Instruction Synchronization Barrier (ISB).

JTAG *See* Joint Test Action Group (JTAG).

Joint Test Action Group (JTAG)
An IEEE group focussed on silicon chip testing methods. Many debug and programming tools use a *Joint Test Action Group* (JTAG) interface port to communicate with processors.

See *IEEE Std 1149.1-1990 IEEE Standard Test Access Port and Boundary-Scan Architecture* specification available from the IEEE Standards Association `http://standards.ieee.org`.

JTAG Access Port (JTAG-AP)
An optional component of the DAP that provides debugger access to on-chip scan chains.

JTAG-AP *See* JTAG Access Port (JTAG-AP).

JTAG-DP *See* Debug Access Port (DAP).

nSRST Abbreviation of *System Reset*. The electronic signal that causes the target system other than the TAP controller to be reset.

See also nTRST and Joint Test Action Group (JTAG).

nTRST Abbreviation of *TAP Reset*. The electronic signal that causes the target system TAP controller to be reset.

See also nSRST and Joint Test Action Group (JTAG).

Power-on reset *See* Cold reset.

Program Flow Trace (PFT)
The *Program Flow Trace* (PFT) architecture assumes that any trace decompressor has a copy of the program being traced, and generally outputs only enough trace for the decompressor to reconstruct the program flow. However, its trace output also enables a decompressor to reconstruct the program flow when it does not have a copy of parts of the program, for example because the program uses self-modifying code.

A *Program Flow Trace Macrocell* (PTM) implements the Program Flow Trace architecture.

RAO *See* Read-As-One (RAO).

RAO/WI Read-As-One, Writes Ignored.

In any implementation, the bit must read as 1, or all 1s for a bit field, and writes to the field must be ignored.

Software can rely on the bit reading as 1, or all 1s for a bit field, and on writes being ignored.

RAZ *See* Read-As-Zero (RAZ).

RAZ/WI Read-As-One, Writes Ignored.

In any implementation, the bit must read as 0, or all 0s for a bit field, and writes to the field must be ignored.

Software can rely on the bit reading as 0, or all 0s for a bit field, and on writes being ignored.

Read-As-One (RAO) In any implementation, the bit must read as 1, or all 1s for a bit field.

Read-As-Zero (RAZ) In any implementation, the bit must read as 0, or all 0s for a bit field.

RealView ICE
An ARM JTAG interface unit for debugging embedded processor cores that uses a DBGTAP or Serial Wire interface.

Replicator
In an ARM trace macrocell, enables two trace sinks to be wired together and to operate independently on the same incoming trace stream. The input trace stream is output onto two independent ATB ports.

RES0
Hardware must implement the bit as Read-As-Zero, and must ignore writes to the field.

Software must not rely on the field reading as all 0s, and except for writing back to the register must treat the value as if it is UNKNOWN. Software must use an SBZP policy to write to the field.

This description can apply to a single bit that should be written as its preserved value or as 0, or to a field that should be written as its preserved value or as all 0s.

In body text, the term RES0 is shown in SMALL CAPITALS.

See also Read-As-Zero (RAZ), Should-Be-Zero-or-Preserved (SBZP), UNKNOWN.

RES1
Hardware must implement the field as Read-As-One, and must ignore writes to the field.

Software must not rely on the field reading as all 1s, and except for writing back to the register it must treat the value as if it is UNKNOWN. Software must use an SBOP policy to write to the field.

This description can apply to a single bit that should be written as its preserved value or as 1, or to a field that should be written as its preserved value or as all 1s.

In body text, the term RES1 is shown in SMALL CAPITALS.

See also Read-As-One (RAO), Should-Be-One-or-Preserved (SBOP), UNKNOWN.

Reserved
Unless otherwise stated in the architecture or product documentation, reserved:
- Instruction and 32-bit system control register encodings are UNPREDICTABLE.
- 64-bit system control register encodings are Undefined.
- Register bit fields are RES0.

SBOP
See Should-Be-One-or-Preserved (SBOP).

SBZP
See Should-Be-Zero-or-Preserved (SBZP).

Serial Wire Debug (SWD)
A debug implementation that uses a serial connection between the SoC and a debugger. This connection normally requires a bidirectional data signal and a separate clock signal, rather than the four to six signals required for a JTAG connection.

Serial Wire Debug Port (SWDP)
The interface for Serial Wire Debug.

Serial Wire JTAG Debug Port (SWJ-DP)
A Debug Port (DP) that combines functionality for both JTAG and Serial Wire Debug for communicating between a debugger and a target system.

Should-Be-One-or-Preserved (SBOP)
The Large Physical Address Extension modifies the definition of SBOP for register bits that are reallocated by the extension, and as a result are SBOP in some but not all contexts. For more information see the *ARM® Architecture Reference Manual ARMv7-A and ARMv7-R edition* and *ARM® Architecture Reference Manual ARMv8, for ARMv8-A architecture profile*. The generic definition of SBOP given here applies only to bits that are not affected by this modification.

Must be written as 1, or all 1s for a bit field, by software if the value is being written without having been previously read, or if the register has not been initialized. If the processor that is writing to the register has read the register since the processor was last reset, it must preserve the value of the field by writing the value that it previously read from the field.

Hardware must ignore writes to these fields.

If a value is written to the field that is neither 1 (or all 1s for a bit field), nor a value previously read for the same field on the same processor, software must expect an UNPREDICTABLE result.

This description can apply to a single bit that should be written as its preserved value or as 1, or to a field that should be written as its preserved value or as all 1s.

See also Should-Be-Zero-or-Preserved (SBZP).

Should-Be-Zero-or-Preserved (SBZP)

The Large Physical Address Extension modifies the definition of SBZP for register bits that are reallocated by the extension, and as a result are SBZP in some but not all contexts. For more information see the *ARM® Architecture Reference Manual ARMv7-A and ARMv7-R edition* and *ARM® Architecture Reference Manual ARMv8, for ARMv8-A architecture profile*. The generic definition of SBZP given here applies only to bits that are not affected by this modification.

Must be written as 0, or all 0s for a bit field, by software if the value is being written without having been previously read, or if the register has not been initialized. If the processor that is writing to the register has read the register since the processor was last reset, it must preserve the value of the field by writing the value that it previously read from the field.

Hardware must ignore writes to these fields.

If a value is written to the field that is neither 0 (or all 0s for a bit field), nor a value previously read for the same field on the same processor, software must expect an UNPREDICTABLE result.

This description can apply to a single bit that should be written as its preserved value or as 0, or to a field that should be written as its preserved value or as all 0s.

See also Should-Be-One-or-Preserved (SBOP).

SWJ-DP

See Serial Wire JTAG Debug Port (SWJ-DP)

SWD

See Serial Wire Debug (SWD).

SWDP

See Serial Wire Debug Port (SWDP).

TAP Controller

Logic on a device that enables access to some or all of that device for test purposes. The circuit functionality is defined in IEEE1149.1.

See also Joint Test Action Group (JTAG).

TCD

See Trace Capture Device (TCD).

TCK

The electronic clock signal that times data on the TAP data lines **TMS**, **TDI**, and **TDO**.

See also Test Data Input (TDI) and Test Data Output (TDO).

Test Access Port (TAP)

The collection of four mandatory and one optional terminals that form the input/output and control interface to a JTAG boundary-scan architecture. The mandatory terminals are **TDI**, **TDO**, **TMS**, and **TCK**. The optional terminal is **nTRST**. This signal is mandatory in ARM processors because it is used to reset the debug logic.

See also Joint Test Action Group (JTAG), TAP Controller, TCK, Test Data Input (TDI), Test Data Output (TDO), and TMS.

Test Data Input (TDI)
Test Data Input (TDI) is the electronic signal input to a TAP controller from the data source (upstream). Usually this is seen connecting the RealView ICE run control unit to the first TAP controller.

See also Joint Test Action Group (JTAG), RealView ICE, *and* TAP Controller.

Test Data Output (TDO)
Test Data Output (TDO) is the electronic signal output from a TAP controller to the downstream data sink. Usually this connects the last TAP controller to the RealView ICE run control unit.

See also Joint Test Action Group (JTAG), RealView ICE, *and* TAP Controller.

TMS
Test Mode Select.

TPA
See Trace Port Analyzer (TPA).

TPIU
See Trace Port Interface Unit (TPIU).

Trace Capture Device (TCD)
A generic term to describe Trace Port Analyzers, logic analyzers, and on-chip trace buffers.

Trace funnel
In an ARM trace macrocell, a device that combines multiple trace sources onto a single bus.

See also AHB Trace Macrocell (HTM) and CoreSight.

Trace hardware
A term for a device that contains an ARM trace macrocell.

Trace port
A port on a device, such as a processor or ASIC, used to output trace information.

Trace Port Analyzer (TPA)
A hardware device that captures trace information output on a trace port. This can be a low-cost product designed specifically for trace acquisition, or a logic analyzer.

Trace Port Interface Unit (TPIU)
Drains trace data and acts as a bridge between the on-chip trace data and the data stream captured by a TPA.

Trigger
In the context of tracing, a trigger is an event that instructs the debugger to stop collecting trace and display the trace information around the trigger position, without halting the processor. The exact information that is displayed depends on the position of the trigger within the buffer.

UNK
An abbreviation indicating that software must treat a field as containing an UNKNOWN value.

Software must not rely on the field reading as zero.

See also UNKNOWN.

UNKNOWN
An UNKNOWN value does not contain valid data, and can vary from moment to moment, instruction to instruction, and implementation to implementation. An UNKNOWN value must not be a security hole or documented or promoted as having a defined value or effect.

UNP
See UNPREDICTABLE.

UNPREDICTABLE
For an ARM trace macrocell, means that the behavior of the macrocell cannot be relied on. When applied to the programming of an event resource, only the output of that event resource is UNPREDICTABLE. UNPREDICTABLE behavior can affect the behavior of the entire system, because the trace macrocell can cause the processor to enter debug state, and external outputs can be used for other purposes.

For a processor means the behavior cannot be relied on. UNPREDICTABLE behavior must not represent a security hole. UNPREDICTABLE behavior must not hang the processor, or any parts of the system.

Warm reset Also known as a core reset. Initializes most of the processor functionality, excluding the debug controller and debug logic. This type of reset is useful if you are using the debugging features of a processor.

See also Cold reset.

Word A 32-bit data item. Words are normally word-aligned in ARM systems.