CoreSight™

v1.0

Architecture Specification

ARM®

ARM IHI 0029B

CoreSight
Architecture Specification

Release Information

Change history

Date	Issue	Change
29 September 2004.	A	First release.
24 March 2005.	B	Second draft. Editorial changes and clarifications.

Proprietary Notice

Confidentiality Status

Product Status

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

Web Address

`http://www.arm.com`

Contents

List of Tables
CoreSight Architecture Specification

List of Figures
CoreSight Architecture Specification

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 ARM CoreSight Design Kit and other 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 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

This specification is organized into the following chapters:

Part A *CoreSight Architecture*

This part contains an introduction to the CoreSight architecture.

Chapter 1 *Introduction*

Read this chapter 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*

This part contains details of the CoreSight visible component architecture, that must be implemented by all CoreSight components that are visible to a debugger.

Chapter 3 ***CoreSight Programmer's Model***

Read this chapter for a description of the CoreSight technology programmer's model.

Chapter 4 ***Topology Detection Registers***

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

Part C ***CoreSight Reusable Component Architecture***

This part contains details of the CoreSight reusable component architecture, that must be implemented by CoreSight components so that they can be used with other CoreSight components.

Chapter 5 ***Reusable Component Architecture***

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

Chapter 6 ***AMBA 3 APB Interface***

Read this chapter for a description of the *Advanced Microcontroller Bus Architecture* (AMBA™) 3 *Advanced Peripheral Bus* (APB™) interface.

Chapter 7 ***AMBA 3 ATB Interface***

Read this chapter for a description of the AMBA 3 *Advanced Trace Bus* (ATB™) interface.

Chapter 8 ***Channel Interface***

Read this chapter for a description of the channel interface.

Chapter 9 ***Authentication Interface***

Read this chapter for a description of the authentication interface.

Chapter 10 ***Topology Detection at the Component Level***

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

Part D ***CoreSight System Architecture***

This part contains details of 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.

Chapter 11 ***System Architecture***

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

Chapter 12 ***System Design***

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

Chapter 13 ***Physical Interface***

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

Chapter 14 ***Trace Formatter***

Read this chapter for a description of the CoreSight trace formatter.

Chapter 15 ***ROM table***

Read this chapter for a description of the CoreSight ROM table.

Chapter 16 ***Topology Detection at the System Level***

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

Chapter 17 ***Compliance Criteria***

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

Glossary Read the Glossary for definitions of terms used in this manual.

Conventions

This section describes the conventions that this specification uses:

- *Typographical*
- *Timing diagrams* on page xv
- *Signal naming* on page xvi
- *Numbering* on page xvi.

Typographical

This specification uses the following typographical conventions:

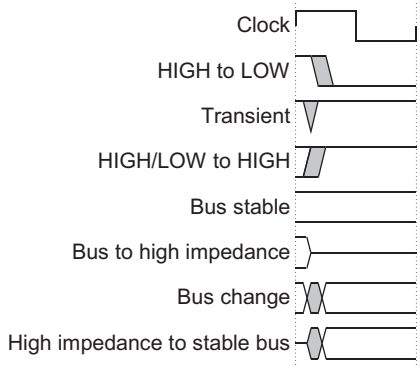
italic Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.

bold Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace`	Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`monospace italic`	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
`monospace bold`	denotes language keywords when used outside example code.
< and >	Angle brackets enclose replaceable terms for assembler syntax where they appear in code or code fragments. They appear in normal font in running text. For example: • `MRC p15, 0 <Rd>, <CRn>, <CRm>, <Opcode_2>` • The Opcode_2 value selects which register is accessed.

Timing diagrams

This specification contains one or more timing diagrams. The figure named *Key to timing diagram conventions* explains the components used in these diagrams. When variations occur they have clear labels. You must not assume any timing information that is not explicit in the diagrams.



Key to timing diagram conventions

Signal naming

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

Prefix A Denotes AMBA 3 *Advanced eXtensible Interface* (AXI™) interface global and address channel signals.

Prefix B Denotes AMBA 3 AXI interface write response channel signals.

Prefix C Denotes AMBA 3 AXI interface low-power interface signals.

Prefix H Denotes AMBA *Advanced High-performance Bus* (AHB™) signals.

Prefix n Denotes active-LOW signals except in the case of AMBA AHB, AMBA 3 APB, or AMBA 3 ATB interface reset signals. These are named **HRESETn**, **PRESETn**, and **ATRESETn** respectively.

Prefix P Denotes an AMBA 3 APB interface signal.

Prefix R Denotes AMBA 3 AXI interface read channel signals.

Prefix W Denotes AMBA 3 AXI interface write channel signals.

Numbering

<size in bits>'<base><number>

This is a Verilog method of abbreviating constant numbers. For example:

- 'h7B4 is an unsized hexadecimal value.
- 'o7654 is an unsized octal value.
- 8'd9 is an eight-bit wide decimal value of 9.
- 8'h3F is an eight-bit wide hexadecimal value of `0x3F`. This is equivalent to b00111111.
- 8'b1111 is an eight-bit wide binary value of b00001111.

Further reading

This section lists publications by ARM Limited, and by third parties.

ARM Limited periodically provides updates and corrections to its documentation. See `http://www.arm.com` for current errata sheets, addenda, and the ARM Limited Frequently Asked Questions list.

ARM publications

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

- *CoreSight Design Kit Technical Reference Manual* (ARM DDI 0314)
- *CoreSight Technology System Design Guide* (ARM DGI 0012)
- *CoreSight Design Kit 11 Implementation and Integration Manual* (ARM DII 0092)
- *Embedded Trace Macrocell Architecture Specification* (ARM IHI 0014)
- *CoreSight ETM9 Technical Reference Manual* (ARM DDI 0315)
- *CoreSight ETM9 Implementation Guide* (ARM DDI 0093)
- *CoreSight ETM9 Integration Manual* (ARM DII 0094)
- *CoreSight ETM11 Technical Reference Manual* (ARM DDI 0318)
- *CoreSight ETM11 Implementation Guide* (ARM DII 0097)
- *CoreSight ETM11 Integration Manual* (ARM DII 0098)
- *ARM Architecture Reference Manual* (ARM DDI 0100)
- *AMBA 3 APB protocol specification* (ARM IHI 0024)
- *AMBA AXI protocol specification* (ARM IHI 0022).

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 Limited welcomes feedback on the CoreSight architecture and its documentation.

Feedback on the product

If you have any comments or suggestions about this product, contact your supplier giving:

- the product name
- a concise explanation of your comments.

Feedback on this specification

If you have any comments on this specification, send email to `errata@arm.com` giving:

- the title
- the number
- the relevant page number(s) to which your comments apply
- a concise explanation of your comments.

ARM Limited also welcomes general suggestions for additions and improvements.

Part A
CoreSight Architecture

Chapter 1
Introduction

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

1.1 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 keep silicon costs down.
- 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.

1.2 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 1-10. This explains why, for example, CoreSight uses memory mapped access to the registers of CoreSight components.

1.2.1 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 programmer's 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 *CoreSight Visible Component Architecture*.

1.2.2 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 components in the CoreSight Design Kit 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 3 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 *CoreSight Reusable Component Architecture*.

1.2.3 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 1-6 and Chapter 14 *Trace Formatter*
- the ROM table, see *CoreSight component types* on page 1-6 and Chapter 15 *ROM table*
- the topology detection
- compliance criteria for CoreSight systems.

For details of the system architecture see Part D *CoreSight System Architecture*.

1.3 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*

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).

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)
- a *JTAG Access Port* (JTAG-AP)
- a *JTAG Debug Port* (JTAG-DP)
- an *APB Multiplexer* (APB-Mux)
- a ROM Table.

For more information on specific components see the *CoreSight Design Kit Technical Reference Manual* and the *CoreSight Technology System Design Guide*.

1.4 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 16 *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 10 *Topology Detection at the Component Level*. Interface vendors must define the requirements for other interfaces, following the rules in Chapter 16 *Topology Detection at the System Level*.

1.4.1 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 programmer's model and contain information about the components in the system and permit a debugger to identify the components, see Chapter 4 *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 10 *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 15 *ROM table*,

- a description of physical connections for the debugger hardware, see Chapter 13 *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 AMBA 3 ATB interface signals only **ATVALID** and **ATREADY** need to be controlled.

1.4.2 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 13 *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 10 *Topology Detection at the Component Level* describes how to perform topology detection for each of the CoreSight interfaces.
- Uses the algorithm in Chapter 16 *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.

1.5 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 register.

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

- Synchronous JTAG interfaces, that are implemented in all synthesizable ARM designs, can cause problems:
  - If a device is powered down its JTAG controller stops, causing 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. These issues become significantly more important with *Intelligent Energy Management* (IEM).
- 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 core 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 programmer's 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 programmer's 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.

Most CoreSight components do not provide JTAG access to debug registers. The disadvantages of this are:

- Increased complexity for implementing a system with basic debug functionality. The CoreSight Design Kit provides an integration kit that ensures that the DAP has been correctly integrated.

Most CoreSight processors do not provide direct access to their debug registers in the programmer's model, for example using ARM coprocessor instructions. However, processors can continue to offer direct access in addition to or instead of access using the CoreSight memory-mapped architecture. The disadvantages of not providing access in the programmer's model are:

- Increased complexity for implementing a system with software access to debug functionality. In the CoreSight Design Kit, the DAP includes an AMBA 3 APB slave interface so that all the CoreSight devices can be made available to software in the same way as adding an extra peripheral.

- Software accessing the debug register file must know its base address. Software can discover this by using the ROM table.
- Slower access to debug register file. Because the debug registers are effectively peripherals, accesses from a monitor running on the processor are over one or more bridges between busses. A processor implementing access in the programmer's model can provide more direct access to these registers.

Part B

CoreSight Visible Component Architecture

Chapter 2
Visible Component Architecture

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

2.1 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 programmer's model and requirements for topology detection.

The programmer's 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 3
CoreSight Programmer's Model

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

3.1 About the programmer's model

This chapter defines the standard set of registers that you must implement on all CoreSight components, in addition to the control registers specific to components. Some registers are optional and must read as zero when they are not implemented. This section explains how you can use integration registers to understand a CoreSight system topology, and which devices are connected to which other devices.

The basic register structure is taken from the peripheral ID register structure used by PrimeCells. This defines two identification registers:

- A component identification register, that indicates that the identification registers are present. It extends the PrimeCell specification by permitting you to specify a component class, that can in turn specify that additional registers are present.
- A peripheral identification register that uniquely identifies the component.

The remainder of the programmer's model depends on the component classes. The classes are:

- class `0x1` ROM table
- class `0x9` CoreSight component
- class `0xF` PrimeCell or system component.

Each component occupies at least 4KB of address space. Where a component occupies more than 4KB, these registers must appear in the highest 4KB block, see *Spanning multiple 4KB blocks* on page 3-24. Every component on the Debug APB must support this programmer's model.

Table 3-1 shows the address offsets for the CoreSight component registers within a 4KB window.

Table 3-1 CoreSight component register address offsets

Address offset	Type	Width in bits	Name	Comment
`0xFFC`	Read only	8	Component ID3 Register	Preamble. See page 3-7
`0xFF8`	Read only	8	Component ID2 Register	Preamble. See page 3-7
`0xFF4`	Read only	8	Component ID1 Register	Component class. See page 3-7
`0xFF0`	Read only	8	Component ID0 Register	Preamble. See page 3-7
`0xFEC`	Read only	8	Peripheral ID3 Register	See page 3-8
`0xFE8`	Read only	8	Peripheral ID2 Register	See page 3-8

Table 3-1 CoreSight component register address offsets

Address offset	Type	Width in bits	Name	Comment
0xFE4	Read only	8	Peripheral ID1Register	See page 3-8
0xFE0	Read only	8	Peripheral ID0 Register	See page 3-8
0xFDC	Read only	8	Peripheral ID7 Register	Reserved. See page 3-8
0xFD8	Read only	8	Peripheral ID6 Register	Reserved. See page 3-8
0xFD4	Read only	8	Peripheral ID5 Register	Reserved. See page 3-8
0xFD0	Read only	8	Peripheral ID4 Register	See page 3-8
0xFCC	Read only	8	Device Type Identifier Register	See page 3-11
0xFC8	Read/Write	32 to 8	Device Configuration Register	See page 3-13
0xFB8	Read only	8	Authentication Status Register	See page 3-14
0xFB4	Read only	3	Lock Status Register	See page 3-16
0xFB0	Write only	32 or 8	Lock Access Register	See page 3-17
0xFA4	Read/Write	variable	Claim Tag Clear Register	See page 3-19
0xFA0	Read/Write	variable	Claim Tag Set Register	See page 3-19
0xF00	Read/Write	1	Integration Mode Control Register	See page 3-21

Figure 3-1 on page 3-4 shows the offsets of registers within the 4KB window for CoreSight components.

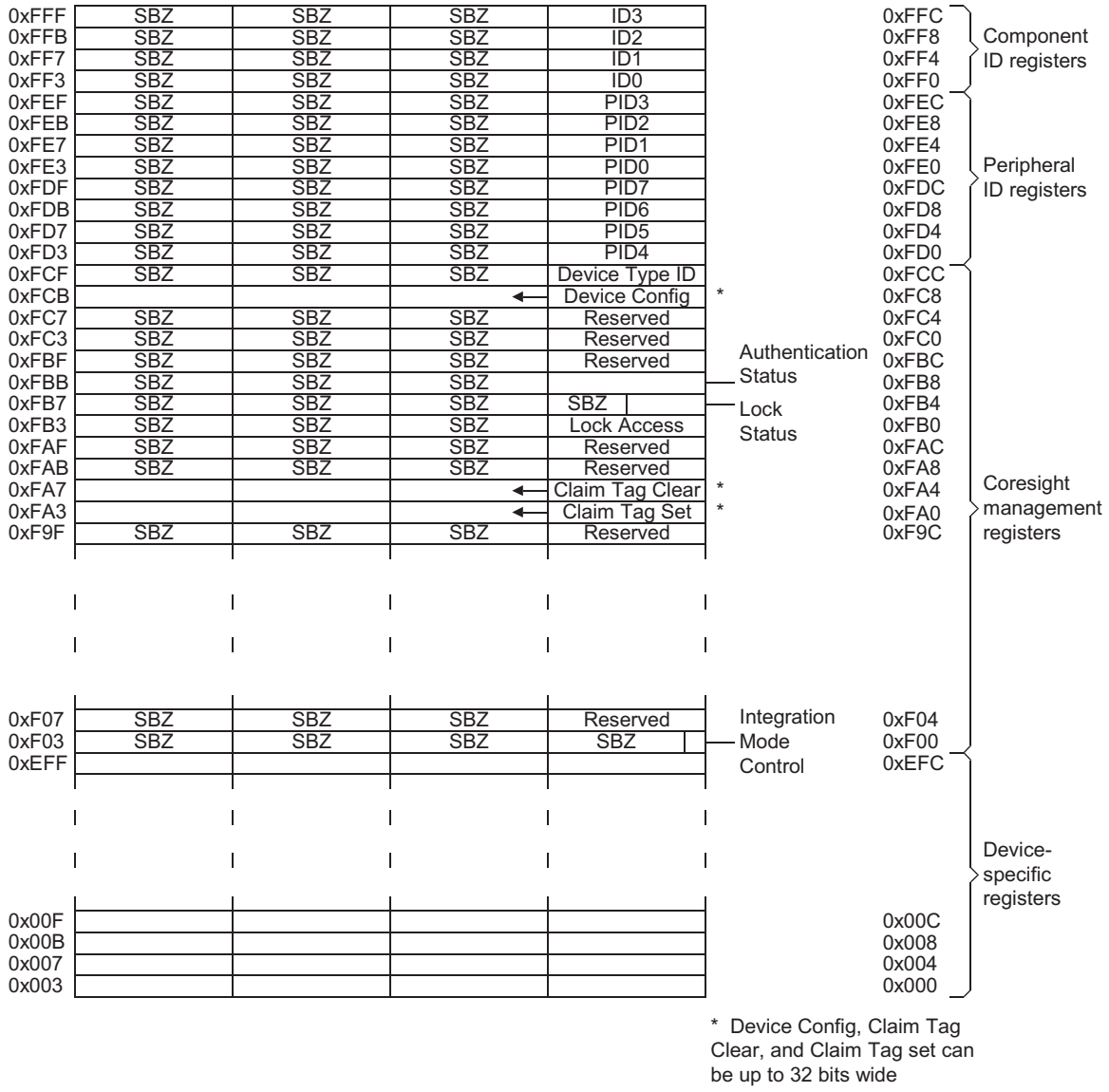


Figure 3-1 Contents of the 4KB window for a CoreSight component

The upper words are referred to as CoreSight management registers. These are common to all CoreSight components. This area must be viewed as reserved specifically for CoreSight registers and must not be used by device-specific control registers.

The remaining words can be used for component specific registers. It is recommended that:

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

3.1.1 Support for 8-bit AMBA 3 APB interfaces

To reduce the cost of the AMBA 3 APB interface, a component can implement an 8-bit AMBA 3 APB interface. If you do this, only the bottom 8 bits of each 32-bit word are visible. Every register in the CoreSight programmer's model can be implemented using only the bottom 8 bits of each 32-bit word, see *AMBA 3 APB interface width* on page 6-5.

3.2 Reserved locations

To be compatible with future versions of the CoreSight architecture, you must treat reserved locations as follows:

- when read, components must return zero when a reserved location is read
- components must ignore writes to reserved locations
- tools must not write to a reserved register
- when writing to a register with reserved bits, tools must preserve the value read from those bits or write back the read value.

In all specified registers, unspecified bits are reserved.

3.3 Component identification registers

Table 3-2 shows the component identification registers

Table 3-2 Component identification registers

Name	Offset	Bits	Value	Description
Component ID3	0xFFC	[7:0]	0xB1	Preamble
Component ID2	0xFF8	[7:0]	0x05	Preamble
Component ID1	0xFF4	[7:4]	-	Component class
		[3:0]	0x0	Preamble
Component ID0	0xFF0	[7:0]	0x0D	Preamble

Table 3-3 shows the component class values that you can use in the Component ID1 Register.

Table 3-3 Component class encodings

Value	Class description
0x0	Reserved.
0x1	ROM table. See *Class 0x1 ROM table* on page 3-10.
0x2-0x8	Reserved.
0x9	CoreSight component. See *Class 0x9 CoreSight component* on page 3-11
0xA-0xE	Reserved.
0xF	PrimeCell or system component with no standardized register layout, for backwards compatibility. See *Class 0xF PrimeCell or system component* on page 3-23

3.4 Peripheral identification registers

Table 3-4 shows the peripheral identification registers.

Table 3-4 Peripheral identification registers

Name	Offset	Bits	Value	Description
Peripheral ID7	0xFDC	[7:0]	0x00	Reserved
Peripheral ID6	0xFD8	[7:0]	0x00	Reserved
Peripheral ID5	0xFD4	[7:0]	0x00	Reserved
Peripheral ID4	0xFD0	[7:4]	-	4KB count
		[3:0]	-	JEP106 continuation code
Peripheral ID3	0xFEC	[7:4]	0x0	RevAnd
		[3:0]	0x0	Customer Modified
Peripheral ID2	0xFE8	[7:4]	-	Revision
		[3]	0x1	Always set. Indicates that a JEDEC assigned value is used
		[2:0]	-	JEP106 identity code [6:4]
Peripheral ID1	0xFE4	[7:4]	-	JEP106 identity code [3:0]
		[3:0]	-	Part Number [11:8]
Peripheral ID0	0xFE0	[7:0]	-	Part Number [7:0]

A component is uniquely identified by the following fields:

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

The meaning of the fields is as follows:

JEP106 continuation code, JEP106 identity code

These indicate the designer of the component and not the implementor, 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 identity code is bits [6:0] of the final number. For example, for ARM Limited this is `0x3B`.

Part Number

This is selected by the designer of the component.

Revision The Revision field is an incremental value starting at `0x0` for the first design of this 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/minor revision.

Customer Modified

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 This field indicates minor errata fixes specific to this design, for example metal fixes after implementation. In most cases this field is zero. It is recommended 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 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 3-24 and Table 3-8 on page 3-24. For ROM Tables, this value must be zero, see *Expansion above 960 entries* on page 15-5.

3.5 Class 0x1 ROM table

For the definition of the ROM table format see Chapter 15 *ROM table*.

3.6 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 3-6.

3.6.1 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.

Figure 3-2 shows the register bit assignments.

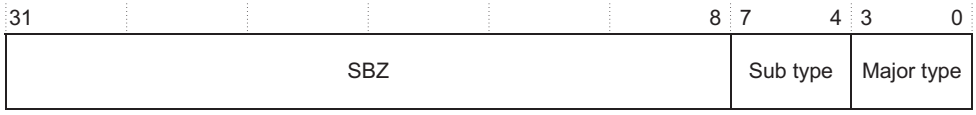


Figure 3-2 Device Type Identifier Register

Description Indicates the type of functionality the component supports.

Bits [31:8] Reserved, RAZ SBZP

[7:4] Sub Type

[3:0] Major Type and Class

Short Name `DEVTYPE`

Location `0xFCC`

Table 3-5 shows the device type encoding for the Device Type Identifier Register.

Table 3-5 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
		0x3-0xF	Reserved
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
		0x5-0xF	Reserved

Table 3-5 Device type encoding (continued)

Major Type [3:0]		Sub Type [7:4]	
Value	**Description**	**Value**	**Description**
0x4	Debug Control	0x0	Other
		0x1	Trigger Matrix, for example ECT
		0x2	Debug Authentication Module, see *Control of authentication interfaces* on page 12-5
		0x3-0xF	Reserved
0x5	Debug Logic	0x0	Other Type
		0x1	Processor core
		0x2	DSP
		0x3	Data Engine or Coprocessor
		0x4-0xF	Reserved
0x6-0xF	Reserved	-	-

3.6.2 Device Configuration Register, DEVID

The Device Configuration Register is read-only. Figure 3-3 shows the register bit assignments.

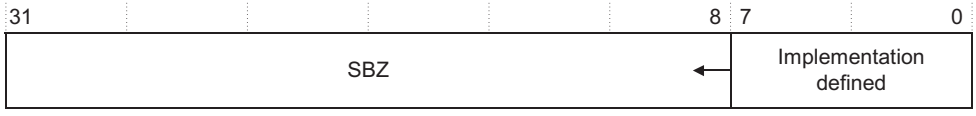


Figure 3-3 Device Configuration Register

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 read as zero.

If the component is configurable then it is recommended that this register reflects any changes to a standard configuration.

Bits [31:0] Implementation Defined. An 8-bit component can implement only the lower 8-bits.

Short Name DEVID

Location 0xFC8

3.6.3 Authentication Status Register, AUTHSTATUS

The Authentication Status Register is read-only, Figure 3-4 shows the register bit assignments. For details of authentication, see *Control of authentication interfaces* on page 12-5.

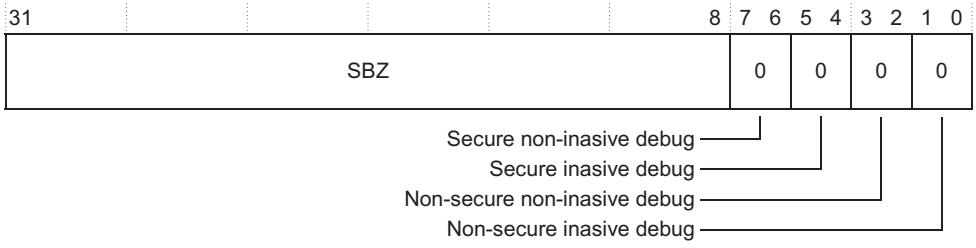


Figure 3-4 Authentication Status Register

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] Reserved, RAZ SBZP

[7:6] Secure Non-Invasive Debug

[5:4] Secure Invasive Debug

[3:2] Non-Secure Non-Invasive Debug

[1:0] Non-Secure Invasive Debug

Short Name AUTHSTATUS

Location 0xFB8

Table 3-6 shows the description for each pair of bits in each debug set.

Table 3-6 Authentication status bits values and meanings

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

Table 3-7 shows how you must set the fields in Table 3-6.

Table 3-7 Recommended authentication status bit field settings

Field	Debug level not supported	Debug level supported
Secure invasive debug	00	if (**SPIDEN** and **DBGEN**) 11 else 10
Non-secure invasive debug	00	if (**DBGEN**) 11 else 10
Secure non-invasive debug	00	if (**SPNIDEN** or **SPIDEN**) and (**NIDEN** or **DBGEN**) 11 else 10
Non-secure non-invasive debug	00	if (**NIDEN** or **DBGEN**) 11 else 10

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.

3.6.4 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 programmer's model includes a Lock Status Register and a Lock Access Register 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 Lock Status Register return zero. For information on how CoreSight components can distinguish between internal and external accesses, see *Debug APB interface memory map* on page 12-6.

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, LOCKSTATUS

This indicates the status of the lock control mechanism, Figure 3-4 on page 3-14 shows the register bit assignments.

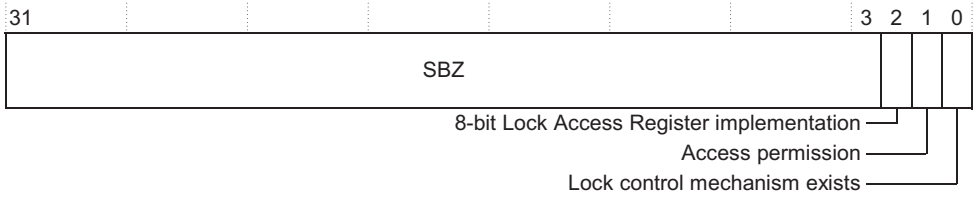


Figure 3-5 Lock Status Register

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 Lock Access Register. For most components this covers all registers except for the Lock Access Register `0xFB0`.

Bits [31:3] reserved, RAZ SBZP

[2] This component implements an 8-bit Lock Access Register.

[1] The values of this bit mean:

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

[0] Indicates that a lock control mechanism exists for this device

Short name `LOCKSTATUS`

Location `0xFB4`

Note

- If no lock control mechanism exists then this register must read as zero.
- Typically 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 masked to read as zero, because no lock access mechanism is required for that view.

Lock Access Register, LOCKACCESS

Figure 3-6 on page 3-18 shows the 32-bit wide format for the Lock Access Register.

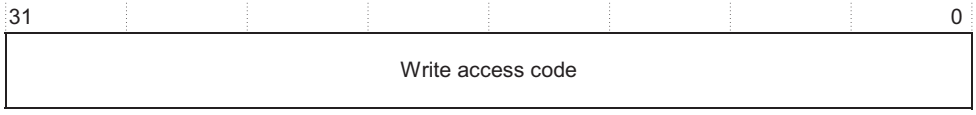


Figure 3-6 32-bit Lock Access Register

Figure 3-7 shows the 8-bit wide format for the Lock Access Register.

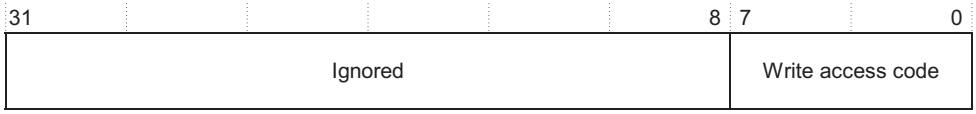


Figure 3-7 8-bit Lock Access Register

If `LOCKSTATUS[0] == 0x0` then this register is not present.

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

Bits [31:0] Write Access Code. A write of `0xC5ACCE55` enables further write access to this device. An invalid write will have the affect of removing write access.

If LOCKSTATUS[2] is set, then only bits [7:0] of this register are implemented and lock access is obtained by consecutively writing `0xC5`, `0xAC`, `0xCE`, `0x55`. Bits [31:8] are unused and any writes to them ignored.

Short name `LOCKACCESS`

Location `0xFB0`

Note

If your system has a mixture of 8 and 32-bit Lock Access Registers then it is possible to use the same routine by writing the value `0xC5CCE55` repeated with different amounts of rotation. Initially writing `0xACCE55C5`, then `0xCE55C5AC`, `0x55C5ACCE` and finally `0xC5ACCE55` causes either the 32-bit lock access register or the 8-bit version to become unlocked.

3.6.5 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 3-20 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*.

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 the Claim Tag Set Register.

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] Reserved, RAZ SBZP

[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*.

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 4 bits are implemented, a read of this register will return `0x0000000F`. If no claim tag is implemented, then a read of this register will return `0x00000000`.

It is recommended that a minimum of 4 claim bits are implemented.

Bits [31:n] Reserved, RAZ SBZ. Unimplemented locations return a zero on read.

[n-1:0] A bit programmable register bank which sets the Claim Tag Value. A read will return 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, 4 bits can control 4 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, 4 bits can control 2 areas of functionality each for 2 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.

3.6.6 Integration Mode Control Register

The Integration Mode Control Register is Read/Write. This register is used to enable topology detection. Figure 3-8 on page 3-22 shows the register bit assignments. For more information see Chapter 4 *Topology Detection Registers*.

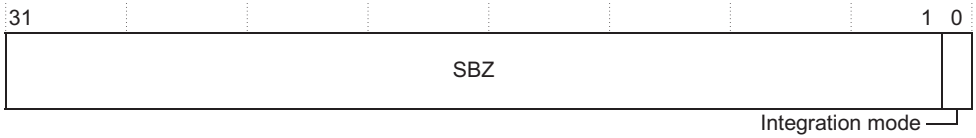


Figure 3-8 Integration Mode Control Register

Description This register enables the component to switch from a functional mode, the default behavior, to integration mode where the inputs and outputs of the component can be directly controlled for the purpose of integration testing and topology solving.

Bits [31:1] Reserved, RAZ SBZP

[0] 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 must read as zero.

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.

3.7 Class 0xF 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 and component ID register are specified for this class of component.

3.8 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 programmer's model in the last 4KB block. You do not have to implement the programmer's model in the other 4KB blocks. Each 4KB window must be allocated consecutively without gaps. Table 3-8 shows the full list of memory block requirements and the impacts to the component in terms of available registers and required address lines.

Table 3-8 Spanning multiple 4KB windows

Number of 4KB blocks	Value of PeripheralID4[7:4]	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 6-6.

Figure 3-9 shows how a component that requires four 4KB windows appears in the memory map.

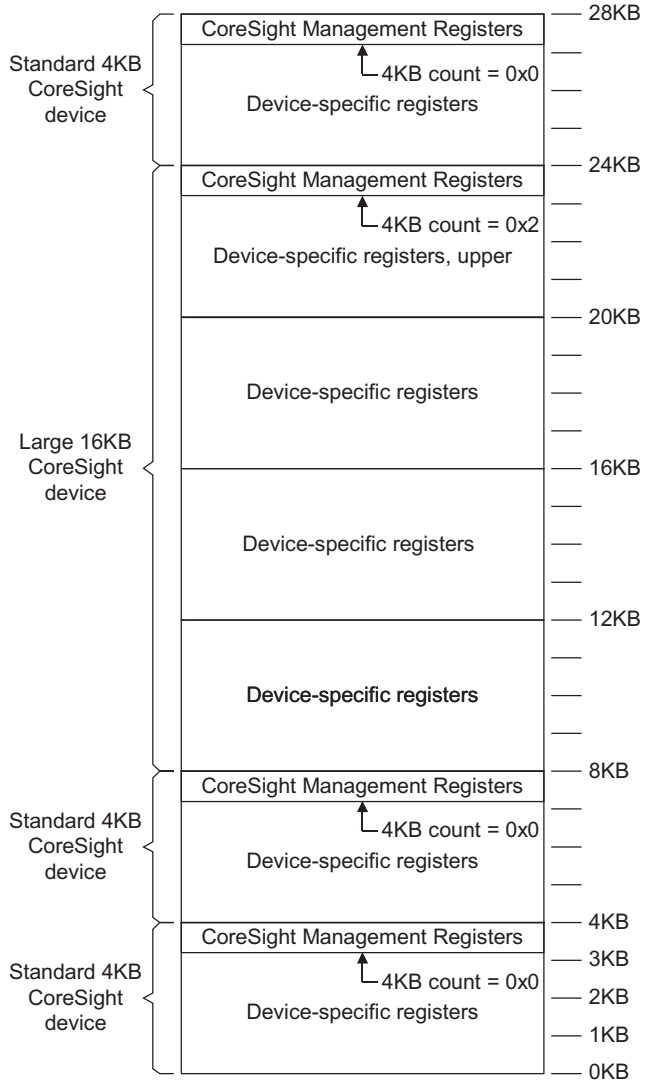


Figure 3-9 How multiple 4KB windows are spanned

3.8.1 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 programmer's model. If you do this, you must provide a means for determining the address of the additional space from the programmer's model of the component. This limits the system design options and is not recommended.

Chapter 4
Topology Detection Registers

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

4.1 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 10 *Topology Detection at the Component Level*. Interface vendors must define the requirements for other interfaces, following the rules in Chapter 16 *Topology Detection at the System Level*.

4.2 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 16 *Topology Detection at the System Level*.

4.2.1 Recommended method

It is recommended 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.

4.3 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 test bench 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 Integration Mode Control Register, see *Integration Mode Control Register* on page 3-21, to select integration mode as well as topology detection mode.

4.4 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 programmer's 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 programmer's model does not need to indicate whether the interface is connected but, if it does, some time can be saved during topology detection.

4.5 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 programmer's model of the affected component controls the configuration by selecting between a number of alternative connections for that interface.
- The programmer's 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.

4.5.1 External multiplexing

Figure 4-1 on page 4-7 shows and example of how variable connections can be implemented using an external multiplexor. 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.

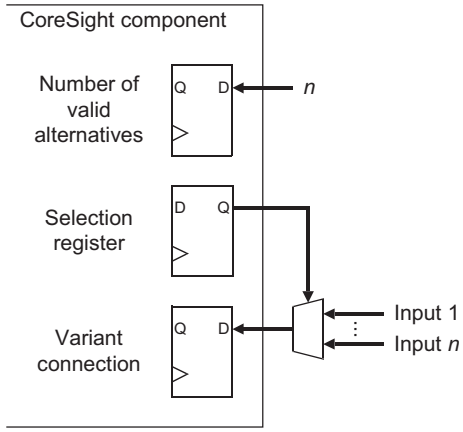


Figure 4-1 External multiplexing of connections

4.6 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 16 *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 16 *Topology Detection at the System Level*.

4.6.1 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 programmer's model.

Part C

CoreSight Reusable Component Architecture

Chapter 5
Reusable Component Architecture

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

5.1 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. All components delivered with the CoreSight Design Kit comply to this specification.

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 6
AMBA 3 APB Interface

This chapter describes the AMBA 3 APB interface that is used to program CoreSight components. It contains the following sections:

6.1 About the AMBA 3 APB interface

The AMBA 3 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.

In addition, all reusable CoreSight components must implement the **PCLKEN** signal, that permits the use of a single clock for low speed AMBA 3 APB interface logic and high-speed core logic.

Most existing debug components implement a JTAG TAP controller to access their functionality. The rationale for the change to memory-mapped access on-chip is discussed in *Component access using memory mapped interfaces instead of JTAG* on page 1-10.

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

6.2 AMBA 3 APB interface signals

All signals are suffixed with **DBG** to indicate that this is the Debug APB interface used for accessing CoreSight components. Table 6-1 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.

For further details, see the *AMBA 3 APB Protocol Specification*.

Table 6-1 Signals on the Debug APB interface

Name	Master input or output	Slave input or output	Clamp value	Description
PCLKDBG	In	In	-	The rising edge of **PCLKDBG** times all transfers on the AMBA 3 APB interface.
PCLKENDBG	In	In	-	This signal qualifies the rising edges of **PCLKDBG**.
PRESETDBGn	In	In	-	This signal resets the interface and is active LOW.
PADDRDBG[31:2]	Out	In	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 6-6
PSELDBG	Out	In	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	Out	In	0	This signal indicates the second and subsequent cycles of an AMBA 3 APB interface transfer.
PWRITEDBG	Out	In	0	When HIGH this signal indicates a write access and when LOW a read access.
PWDATADBG[31:0]	Out	In	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.

Table 6-1 Signals on the Debug APB interface (continued)

Name	Master input or output	Slave input or output	Clamp value	Description
PREADYDBG	In	Out	1	The ready signal is used by the slave to extend an AMBA 3 APB interface transfer.
PRDATADBG[31:0]	In	Out	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	In	Out	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.

6.3 AMBA 3 APB interface width

The AMBA 3 APB interface is 32-bits wide. However, a component implementing the AMBA 3 APB interface can implement only 8 or 16 bits of **PRDATADBG** and **PWDATADBG** provided that the programmer's model has been designed to accommodate this. For example, a device implementing an 8-bit interface cannot implement bytes at addresses `0x01`, `0x02`, `0x03`, `0x05`, and so on.

Chapter 3 *CoreSight Programmer's Model* describes the model compatible with an 8-bit, 16-bit or 32-bit AMBA 3 APB interface. Most CoreSight components implement a 32-bit interface, see *Support for 8-bit AMBA 3 APB interfaces* on page 3-5.

6.4 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 12-6. A component can use **PADDRDBG[31]** to distinguish between internal and external accesses. Most components use this to control the Lock Registers defined in *Lock registers* on page 3-16. Most components treat all other registers identically regardless of if they are accessed internally or externally.

6.5 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 possible to access the debug functionality of the component using the AMBA 3 APB interface.

Chapter 7
AMBA 3 ATB Interface

This chapter describes the AMBA 3 ATB interface. It contains the following chapters:

7.1 About the AMBA 3 ATB interface

The AMBA 3 ATB interface carries trace around an SoC.

Any CoreSight component, or platform, that has trace capabilities has an AMBA 3 ATB interface. An interface that generates trace data is a master on the AMBA 3 ATB interface and an interface that receives trace data is a slave on the AMBA 3 ATB interface.

The AMBA 3 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.

For all reusable components, the addition of **ATCLKEN** permits the use of a single clock with lower speed AMBA 3 ATB interfaces and high-speed core logic.

7.2 AMBA documentation

The AMBA 3 ATB interface is part of the AMBA 3 protocol family. It is documented here because, at the time of writing, it is not specified in the official AMBA 3 documentation. Consult the latest AMBA 3 documentation to determine if a more recent version of the AMBA 3 ATB protocol specification is available.

7.3 AMBA 3 ATB interface Signals

Table 7-1 shows all the signals for a component implementing a trace data interface. Signals are prefixed **AT** for signals relating to data flow and **AF** for flush control signals, clock and reset signals apply to both. The clamp value is the value must be imposed on an output when the component is powered down or disabled.

Table 7-1 AMBA 3 ATB interface signals

Name	Master input or output	Slave input or output	Clamp value	Description
ATCLK	In	In	-	Trace bus clock.
ATCLKEN	In	In	-	Enable signal for **ATCLK** domain
ATRESETn	In	In	-	AMBA 3 ATB interface reset when LOW. Applies to **AF*** signals as well as **AT***. This signal is asserted LOW asynchronously, and deasserted HIGH synchronously.
ATVALID	Out	In	0	Indicates that a transfer is being attempted to this cycle. If low, all other **AT** signals must be ignored this cycle. See *ATVALID and ATREADY* on page 7-7.
ATREADY	In	Out	1	Slave is ready to accept data.See *ATVALID and ATREADY* on page 7-7.
ATID[6:0]	Out	In	0	An ID that uniquely identifies the source of the trace. See *ATID* on page 7-8.
ATBYTES[m:0]	Out	In	0	The number of bytes on **ATDATA** to be captured minus 1.
ATDATA[n:0]	Out	In	0	Trace data.
AFVALID	In	Out	0	Flush. All buffers must be flushed, as trace capture is about to stop.See *AFVALID and AFREADY* on page 7-9.
AFREADY	Out	In	1	Flush Acknowledge. Asserted when buffers have been flushed. See *AFVALID and AFREADY* on page 7-9.

7.4 AMBA 3 ATB interface rules

The AMBA 3 ATB interface rules are as follows:

1. On each rising edge of **ATCLK** all other signals must be sampled.
2. If **ATREADY** is LOW and **ATVALID** is HIGH on a cycle, **ATVALID**, **ATDATA**, **ATID** and **ATBYTES** must retain the same value on the next cycle.

   If **ATVALID** is LOW, **ATREADY** must be ignored.
3. If **ATREADY** is HIGH and **ATVALID** is HIGH, the bottom (**ATBYTES** + 1) bytes of **ATDATA** must be captured.

   Note

   This means it is not possible for zero bytes to be captured.

   The data must be aligned to the least significant byte.
4. The width of **ATDATA** must be a power of two equal to or greater than 8 bits.

   It is recommended that components use at least a 32-bit implementation.
5. $m = \log_2(n+1) - 4$

   The width of **ATBYTES[m:0]** and **ATDATA[n:0]** are related. Table 7-2 shows this relationship.

Table 7-2 Width of ATDATA[n:0] and ATBYTES[m:0]

n	m
7	**ATBYTES** not required
15	0
31	1
63	2
127	3

   For example, if **ATDATA** is 32 bits wide, **ATBYTES** will be 2 bits wide.
6. Whenever bytes on **ATDATA** are captured, the value of **ATID** must also be captured.
7. The order of the bytes of trace must be preserved, even between trace with different **ATIDs**.

The CoreSight trace funnel might order trace from different sources in any order, but when funneled onto a single bus the order cannot be changed, see *ATVALID and ATREADY* on page 7-7.

Note

This differs from ID signals used in AMBA 3 AXI interfaces, that only preserve the ordering between transactions with the same ID.

8. When **AFVALID** is asserted, it must remain asserted until **AFREADY** is asserted.

   AFREADY is ignored while **AFVALID** is deasserted.

9. When **AFVALID** is asserted, buffered trace that has already been generated must be output immediately.

10. When **AFREADY** is asserted, **AFVALID** must be deasserted on the following cycle unless another flush is intended.

11. When **AFVALID** is asserted, **AFREADY** must be asserted one cycle after all trace has been output that was already generated and stored in internal buffers on the cycle in which **AFVALID** was first asserted.

    See *AFVALID and AFREADY* on page 7-9.

12. **ATRESETn** can be asserted, LOW, asynchronously, but deassertion must be synchronous after the rising edge of **ATCLK**.

13. A trace source can only start driving **ATVALID** HIGH after **ATRESETn** has been HIGH at a rising edge of **ATCLK**.

14. When an AMBA 3 ATB interface slave is incapable of responding (because of its programming, or being powered down, or not being present) it must drive **ATREADY** HIGH and **AFVALID** LOW.

    This ensures that a replicator does not block because one of its two outputs is disabled.

15. When an AMBA 3 ATB interface master is incapable of responding, because of programming, or being powered down, or not being present, it must drive **AFREADY** HIGH and **ATVALID** LOW.

    ATID, **ATBYTES** and **ATDATA** can be any value. This can be used to tie off unused CoreSight trace funnel slave ports.

16. The **ATID** values `0x00` and `0x70` to `0x7F` are reserved and must not be used.

    See *Special Trace Source IDs* on page 14-7.

7.5 ATVALID and ATREADY

ATVALID and **ATREADY** provide basic flow control. The master indicates that it has trace available to output by asserting **ATVALID**. If the slave can accept a trace, it responds by asserting **ATREADY**, otherwise the same trace must be output again on the following cycle. Figure 7-1 shows an example.

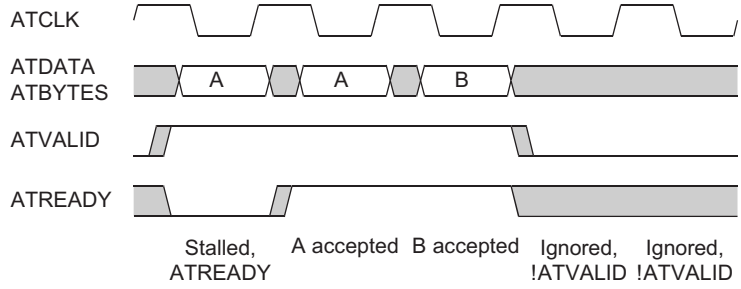


Figure 7-1 Normal ATVALID and ATREADY flow control

If a slave is unable to respond with **ATREADY** in the same cycle as **ATVALID** is asserted, it is recommended that it implements sufficient internal buffering to store one or more cycles of trace. The slave can then assert **ATREADY** whenever there is space in the buffer, even when **ATVALID** is not asserted.

7.6 ATID

Trace is split into different streams with separate IDs to:

- distinguish between trace from different sources
- distinguish between high and low bandwidth components of a trace, so that components downstream of the trace source can perform selective filtering
- provide alignment synchronization information, by changing the ID at an alignment synchronization point, such as the beginning of a trace packet.

Most sources use a single, static ID, although several IDs might be used.

The ID for each trace stream must be unique. It is therefore important that:

- The IDs for all AMBA 3 ATB interface sources are chosen when designing the system, and no AMBA 3 ATB interfaces are exported from the system.
- The ID for each AMBA 3 ATB interface source is programmable by the debugger, because this enables components to be reused in larger systems. This is the recommended option and is mandatory for reusable CoreSight components.

7.7 AFVALID and AFREADY

In a typical trace source there is a fixed amount of time, a number of pipeline stages, between an event to be traced occurring, such as an instruction being executed on a processor, and trace being generated for that event. Figure 7-2 shows this situation. The trace is then written to a FIFO, and output a word at the time. The period between trace being generated and being output is in theory unbounded, especially if the trace source waits until a whole word of trace is available before outputting it.

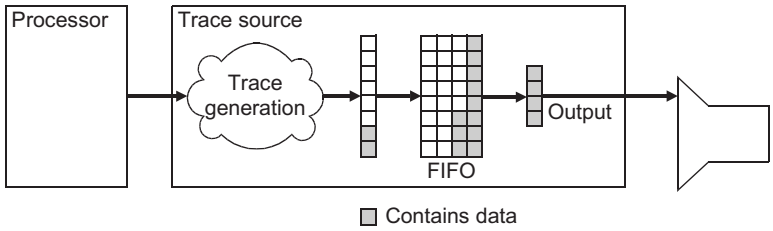


Figure 7-2 Trace generation and trace output

There are two scenarios addressed by this feature:

1. The system, or a portion of the system, is about to be powered down or its clock is about to be stopped. Any trace remaining in buffers or FIFOs in trace sources must be output before this is done, so a flush is issued. Normally no more trace is generated, because processor and memory activity has already stopped. If trace is still being generated after the flush, for example in a processor idle loop, it can be ignored.

2. The trace capture device, an off-chip *Trace Port Analyzer* (TPA) or an on-chip *Embedded Trace Buffer* (ETB), is about to stop capturing, usually because of a trigger point. Either:
   - The on-chip logic is aware that capture is about to stop. A flush can be issued at this point.
   - The on-chip logic is only aware of the trigger and does not know how much trace after this is to be captured. A flush can be issued at the point of the trigger. This ensures all trace generated before the trigger is captured, although it does not make any difference to trace generated after the trigger.
   - A flush is issued periodically. This is the simplest solution. The period must be such that a flush always occurs between a trigger occurring and trace capture stopping.

When a flush occurs, indicated by **AFVALID** HIGH, it is expected that CoreSight trace funnels give the highest priority to trace sources that have not yet asserted **AFREADY**.

A flush sequence, when started, cannot be cancelled by the trace slave (sink).

Note

AFREADY is asserted when all trace that was already generated when **AFVALID** was asserted has been output, not when the FIFO is next empty. Trace that is generated after **AFVALID** is asserted and stored in the FIFO is output as normal after **AFREADY** is asserted.

Figure 7-3 shows an example flush.

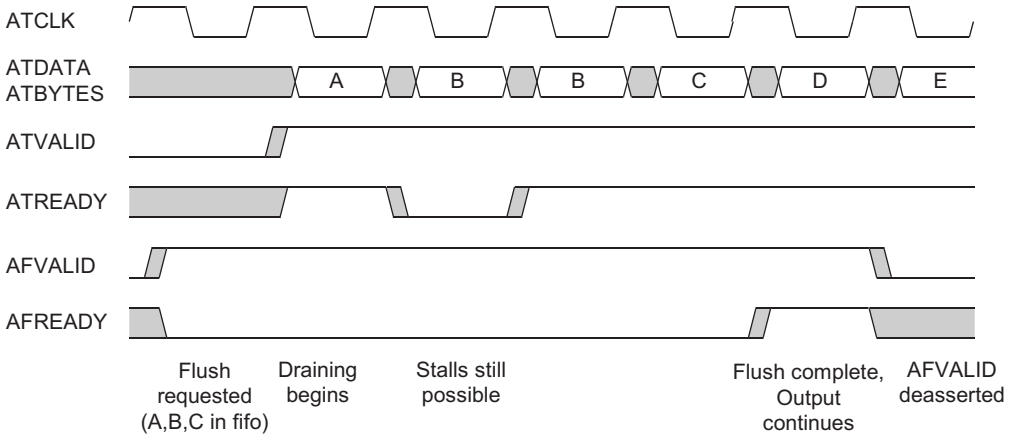


Figure 7-3 Flush procedure

7.7.1 Behavior of trace sources with no external storage

If an AMBA 3 ATB interface master is a trace source and does not store any trace internally, it can use the algorithm shown in Table 7-3 to control **AFREADY**.

Table 7-3 Basic AFREADY control algorithm

ATVALID	ATREADY	AFREADY next cycle
0	-	1
1	0	0
1	1	1

Figure 7-4 on page 7-11 shows an example of this algorithm.

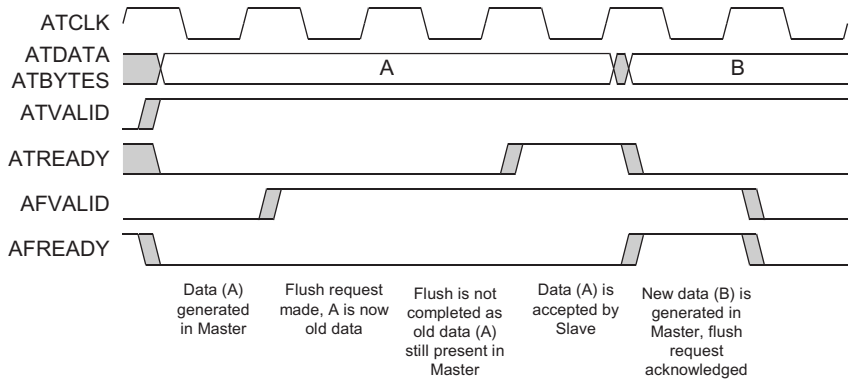


Figure 7-4 Flushing from a master with no internal storage

7.8 AMBA 3 ATB interface signal naming conventions

The interfaces conforms to the following signal naming conventions:

- AMBA 3 ATB interface signals connected to a master port are appended with M:
  - master outputs:
    ATIDM[6:0]
    ATDATAM[31:0]
    ATVALIDM
    ATBYTESM[1:0]
    AFREADYM
  - master inputs:
    ATREADYM
    AFVALIDM
- AMBA 3 ATB interface signals connected to a slave port are appended with S:
  - slave outputs:
    ATREADYS
    AFVALIDS
  - slave inputs:
    ATIDS[6:0]
    ATDATAS[31:0]
    ATVALIDS
    ATBYTESS[1:0]
    AFREADYS.

7.9 AMBA 3 ATB interface timing parameters

Because of the small size of CoreSight components, it is anticipated that individual components are flattened along with additional SoC logic. The timing specifications are intended as recommended guidelines only, and are primarily for development purposes.

7.9.1 AMBA 3 ATB interface master timing parameters

Figure 7-5 shows typical AMBA 3 ATB interface master timing parameters.

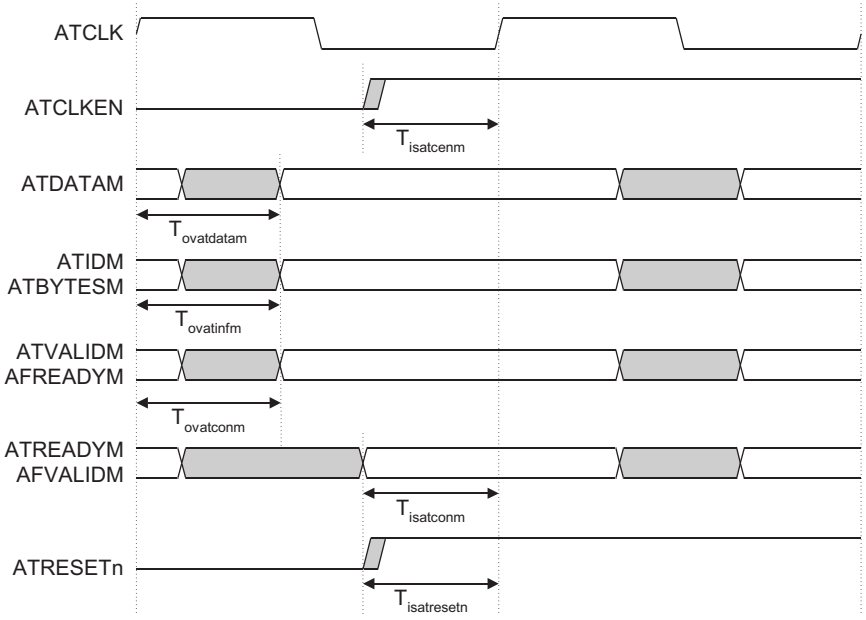


Figure 7-5 AMBA 3 ATB interface master timing

Table 7-4 shows the timing parameters for the interface.

Table 7-4 AMBA 3 ATB interface master timing parameters

Parameter	Description	Max	Min
$T_{isatcenm}$	**ATCLKEN** input setup to rising **ATCLK**	-	30%
$T_{ovatdatam}$	Rising **ATCLK** to **ATDATAM** valid	40%	-
$T_{ovatinfm}$	Rising **ATCLK** to **ATBYTESM** and **ATID** and ID outputs valid	40%	-

Table 7-4 AMBA 3 ATB interface master timing parameters

Parameter	Description	Max	Min
$T_{ovatconm}$	Rising **ATCLK** to AMBA 3 ATB interface control outputs valid	40%	-
$T_{isatconm}$	AMBA 3 ATB interface control inputs setup to rising **ATCLK**	-	30%
$T_{isatresetn}$	**ATRESETn** input setup to rising **ATCLK**	-	30%

7.9.2 AMBA 3 ATB interface slave timing parameters

Figure 7-6 shows typical AMBA 3 ATB interface slave timing parameters.

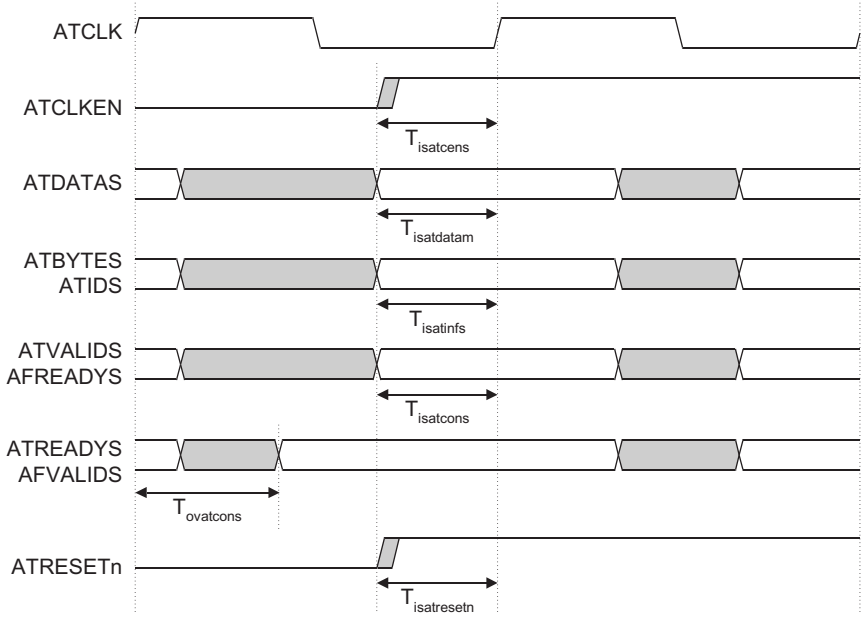


Figure 7-6 AMBA 3 ATB interface slave timing

Table 7-5 shows the timing parameters for the interface.

Table 7-5 AMBA 3 ATB interface slave timing parameters

Parameter	Description	Max	Min
$T_{isatcens}$	**ATCLKEN** input setup to rising **ATCLK**	-	30%
$T_{isatdatas}$	**ATDATAS** input setup to rising **ATCLK**	-	30%
$T_{isatinfs}$	**ATBYTESM** and **ATID** inputs to rising **ATCLK**	-	30%
$T_{isatcons}$	AMBA 3 ATB interface control inputs setup to rising **ATCLK**	-	30%
$T_{ovatcons}$	Rising **ATCLK** to AMBA 3 ATB interface control outputs valid	40%	-
$T_{isatresetn}$	**ATRESETn** input setup to rising **ATCLK**	-	30%

Chapter 8
Channel Interface

This chapter describes the Channel Interface. It contains the following sections:

8.1 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.

The CoreSight Design Kit provides the *Cross Trigger Interface* (CTI), that enables events to be passed between components. However:

- some systems require more trigger signals than are supported by a CTI
- in a platform-oriented system it is necessary to connect these trigger 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. Figure 8-1 shows how you can use the channel interface.

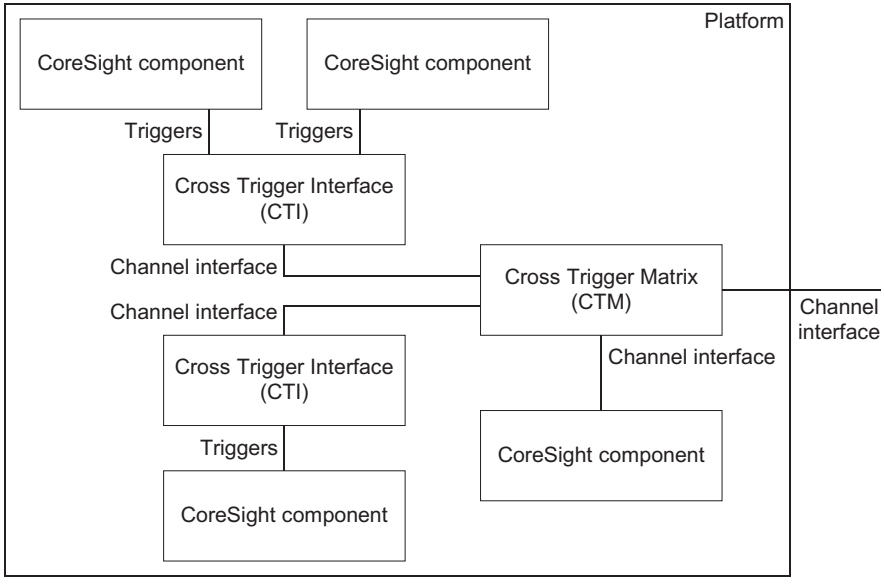


Figure 8-1 Use of the Channel Interface

The Channel Interface supports:

- four trigger channels
- bidirectional communication
- synchronous or asynchronous variants.

8.1.1 Channel interface limitations

The Channel Interface is designed to pass events between components in different clock domains with minimum overhead. If multiple events are presented to the Channel Interface in close succession, these can be interpreted as a single event.

Figure 8-2 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.

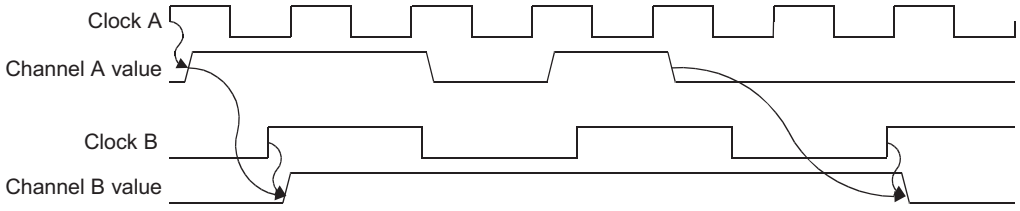


Figure 8-2 Event merging in the Channel Interface

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.

8.2 Channels

The channel interface consists of two components:

- Channel outputs. These signal events generated by the component.
- Channel inputs. These signal 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 four channels. If more channels are required, they must be treated as two or more parallel four-channel interfaces.

8.3 Channel interface signals

Table 8-1 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 8-1 Channel signals

Name	Input or output	Clamp value	Description
CHIN[3:0]	In	-	Channel input
CHINACK[3:0]	Out	1	Channel input acknowledge
CHOUT[3:0]	Out	0	Channel output
CHOUTACK[3:0]	In	-	Channel output acknowledge

Figure 8-3 shows how the asynchronous interface uses a basic 4-phase handshaking protocol. The same protocol is used by **CHOUT** and **CHOUTACK**.

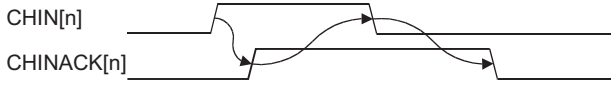


Figure 8-3 Channel interface handshaking

Table 8-2 shows the set of signals required by a synchronous channel interface.

Table 8-2 Synchronous channel interface signals

Name	Input or output	Clamp value	Description
CHCLK	In	-	Clock
CHIN[3:0]	In	-	Channel input
CHOUT[3:0]	Out	0	Channel output

8.4 Channel connections

The channel interface is bidirectional, and therefore you must take care to connect the correct signals together. Figure 8-4 shows how to connect two asynchronous channel interfaces together.

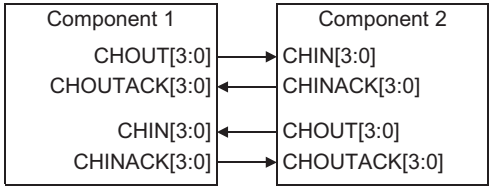


Figure 8-4 Asynchronous channel interface connection

Figure 8-5 shows how to connect two synchronous channel interfaces together.

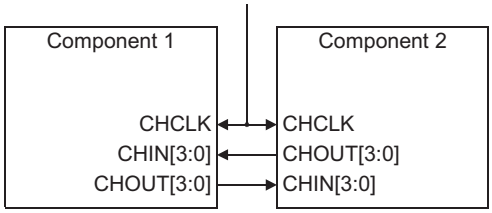


Figure 8-5 Synchronous channel interface connection

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 8-1 on page 8-5 and Table 8-2 on page 8-5.

8.5 Synchronous and asynchronous conversions

Figure 8-6 shows a circuit that makes it possible to convert between synchronous and asynchronous versions of this interface. The circuit also includes a **BYPASS** signal to permit the converter to be bypassed if not required.

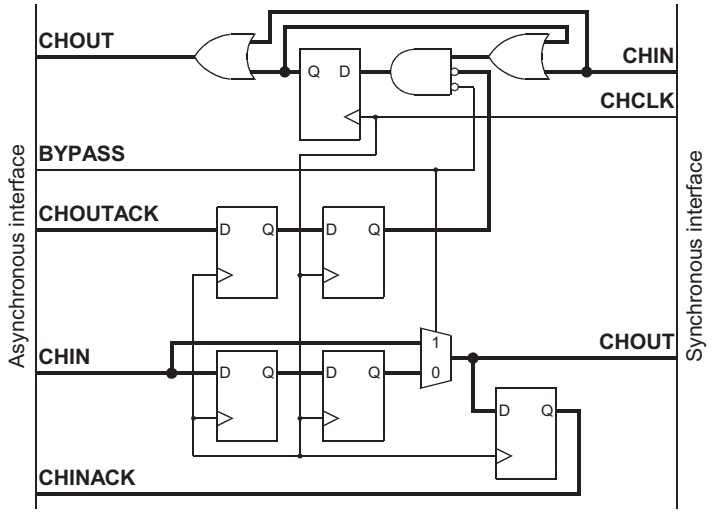


Figure 8-6 Asynchronous to synchronous converter

If you implement a synchronous to asynchronous converter, you increase the likelihood of events being merged as described in *Channel interface limitations* on page 8-3.

Chapter 9
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:

9.1 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 lock access register that is optional in all CoreSight components, see section *Lock Access Register, LOCKACCESS* on page 3-17, and by the software lockout feature of the DAP.

9.2 Definitions of secure and invasive debug

This section defines Secure debug and invasive debug.

9.2.1 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.

9.2.2 Definition of invasive debug

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

This includes any changes to the contents of memory, insertion of instructions into a processor pipeline. It also includes effects that change the number of cycles taken to perform an operation.

The presence of software-accessible registers in a debug component does not cause it to be an invasive component, because accesses to the registers are caused by the processor, and not by the debug component. Operations which affect the behavior of software which reads registers in debug components are not invasive.

9.3 Authentication interface signals

Table 9-1 shows 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 9-1 Authentication interface signals

Name	Input or output	Description
DBGEN	In	Invasive debug enable
NIDEN	In	Non-invasive debug enable
SPIDEN	In	Secure invasive debug enable
SPNIDEN	In	Secure non-invasive debug enable

9.4 Authentication rules

The authentication rules are as follows:

1. All signals must be sampled asynchronously. 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.

   It is recommended 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. It is recommended 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. It is recommended that these signals are not driven in this way.

   Rules 5 to 7 are similar to rules 2 to 6, 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 the value of any of the authentication signals is changed, 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.

Table 9-2 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 9-2 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	Yes	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	Yes	No (2)	No (2)	No (3)	No (3)

Table 9-2 Authentication signal restrictions (continued)

SPIDEN	DBGEN	SPNIDEN	NIDEN	Legal signal combination	Invasive debug permitted		Non-invasive debug permitted	
					S	NS	S	NS
1	0	1	1	Yes	No (2)	No (2)	Yes	Yes
1	1	0	0	No (4,7)	Yes (7)	Yes (4)	Yes (4,7)	Yes (4)
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

9.5 User mode debugging

Individual components might offer greater control over the level of debug permitted. For example, 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 *Embedded Trace Macrocell Architecture Specification*.

Figure 9-1 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.

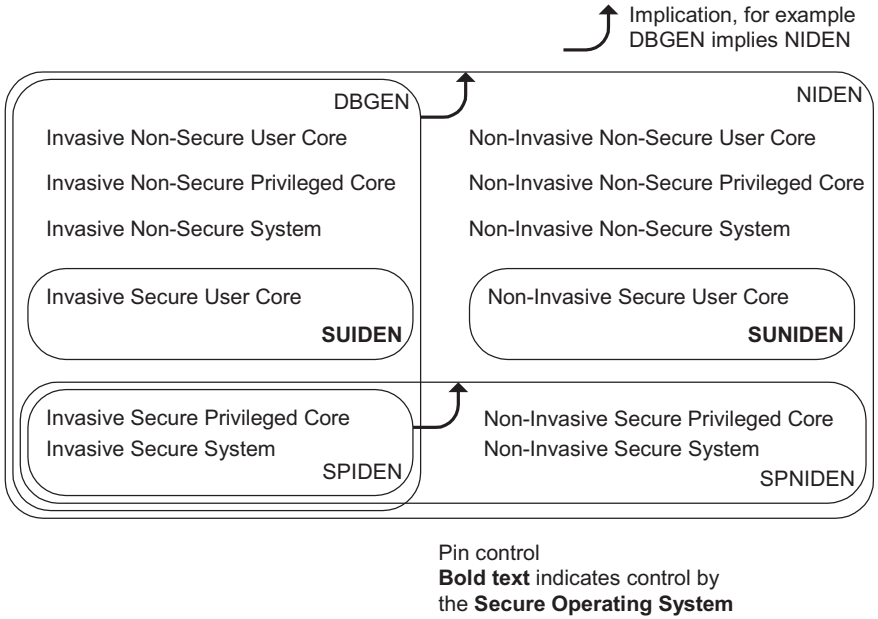


Figure 9-1 Interaction between CoreSight and ARM Security Extensions

9.6 Control of the authentication interface

The authentication interface is controlled at the system level. For more information, see *Control of authentication interfaces* on page 12-5.

9.7 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 10
Topology Detection at the Component Level

This chapter describes how to detect components connected to the AMBA 3 ATB interface and where they are logically located in any corresponding hierarchical connection. It contains the following sections:

10.1 About topology detection at the component level

This chapter describes how to perform topology detection on each interface type. Chapter 4 *Topology Detection Registers* describes the topology detection requirements of CoreSight components. Chapter 16 *Topology Detection at the System Level* describes how debuggers can use this information to detect the topology of a target system.

10.2 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.

10.2.1 Interfaces on standard components

Table 10-1 on page 10-4 shows the interfaces present on standard components in the CoreSight Design Kit. It is provided here for easy reference and might be superseded by a later version of the CoreSight Design Kit. For more details see the *CoreSight Design Kit Technical Reference Manual*.

Table 10-1 Interfaces on standard components

Programmable component	Interfaces
CoreSight ETM	AMBA 3 ATB interface, master 2x Trigger, master. **EXTOUT[1:0]** 4x Trigger, slave. **EXTIN[3:0]** Trigger, master. **TRIGOUT** Variant: CoreETM, slave The number of core interfaces can be read from the ETM System Configuration register. The state of **DBGACK** can be driven directly in all ARM cores, including those that are not CoreSight compliant.
CoreSight ETB	AMBA 3 ATB interface, slave Trigger, master. **ACQCOMP** Trigger, master. **FULL** Trigger, slave. **TRIGIN** Trigger, slave. **FLUSHIN**.
TPIU	AMBA 3 ATB interface, slave Trigger, slave. **TRIGIN** Trigger, slave. **FLUSHIN**.
DAP	No topology detection interfaces.
HTM	AMBA 3 ATB interface, master 2x Trigger, master. **HTMEXTOUT[1:0]** 2x Trigger, master. **HTMEXTIN[1:0]** Trigger, master. **HTMTRIGGER** Variant: AHB, probe The number of AHB interfaces can be read from the HTMCFGCODE2 register. The method to perform topology detection of this interface is not defined.
CoreSight Funnel	8x AMBA 3 ATB interface, slave AMBA 3 ATB interface, master
CTI	8x Trigger, slave. **TRIGIN[7:0]** 8x Trigger, master. **TRIGOUT[7:0]** Channel, bidirectional
VIC (PL190/192)	32x Trigger, master: **VICINTSOURCE[n]**

10.3 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 16 *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 4 *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*

10.3.1 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 3 ATB interfaces, **ATVALID** must be controllable, because if LOW an intermediate bridge will not pass any control signals through it.

10.3.2 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.

10.4 Signals for topology detection

Table 10-2 shows the controllable signals for each interface type, see Table 10-1 on page 10-4.

Table 10-2 Controllable signals for each interface type

Interface	Master wire(s)	Slave wire(s)
AMBA 3 ATB interface	**ATVALID**	**ATVALID**, **ATREADY**
CoreETM	**DBGACK**	**DBGACK**
Trigger	**TRIGOUT**	**TRIGIN**, **TRIGINACK**, if present
Channel, bidirectional	**CHOUT[0]**	**CHIN[0]**, **CHINACK[0]**, if asynchronous

In ARM processor cores with JTAG access the value of **DBGACK** can be controlled only from a JTAG debugger.

The Trigger interface is defined for miscellaneous point-to-point connections carrying a one-bit signal. A Trigger interface can optionally implement an acknowledge signal, that if implemented must be controllable. Substitute **TRIGIN** and **TRIGINACK** 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 3 ATB interface signals for topology detection* on page 10-8
- *Core ETM signals for topology detection* on page 10-8
- *Trigger signals for topology detection* on page 10-9
- *Channel signals for topology detection* on page 10-9.

Refer to these sections in conjunction with the algorithm given in *Detection algorithm* on page 16-5.

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.

10.4.1 AMBA 3 ATB interface signals for topology detection

Table 10-3 shows the topology detection sequence for an AMBA 3 ATB interface.

Table 10-3 Topology detection for the AMBA 3 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
Master deassert	**ATVALID** ← 0
Slave check deasserted	**ATVALID** == 0
Slave post-deassert	**ATREADY** ← 0

10.4.2 Core ETM signals for topology detection

Table 10-4 shows the topology detection sequence for a core ETM interface.

Table 10-4 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

10.4.3 Trigger signals for topology detection

Table 10-5 shows the topology detection sequence for a trigger interface.

Table 10-5 Topology detection for the trigger interface

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

10.4.4 Channel signals for topology detection

Table 10-6 shows the topology detection sequence for a channel interface.

Table 10-6 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 11
System Architecture

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

- *About the system architecture* on page 11-2.

11.1 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 12
System Design

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

12.1 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.

12.2 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 altogether, 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 clock frequencies in this domain must not vary over time. The power can be removed or the clocks can be stopped when debug functionality is not required in order 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 allows 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 *CoreSight Design Kit 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.

12.3 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 9 *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.

It is recommended 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.

12.4 Memory system design

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

12.4.1 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 12-1 shows how the lower 2GB represents internal accesses, and the upper 2GB represents external accesses.

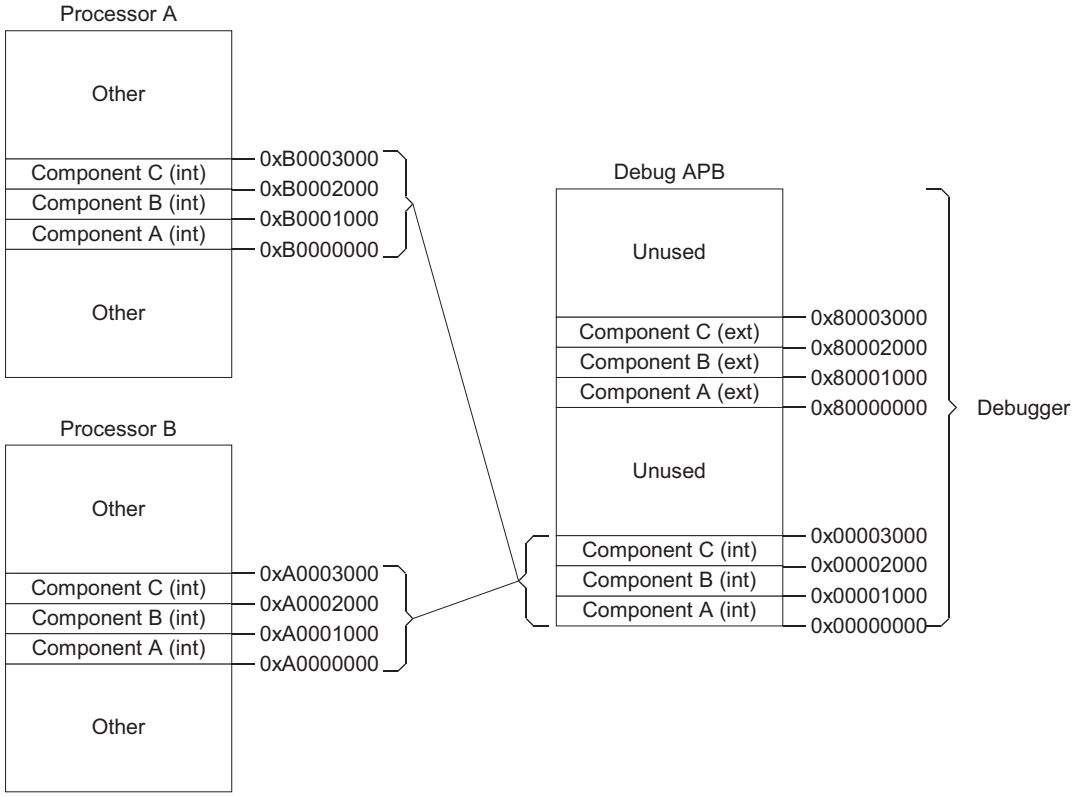


Figure 12-1 AMBA 3 APB interface memory map

It is not possible for system software to access the upper 2GB of the Debug APB interface address space. In the CoreSight Design Kit, 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.

12.4.2 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 13
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:

13.1 About the physical interface

This chapter defines the connector used for debug communication and trace output from a CoreSight system.

Systems that do not implement a trace port can use an alternative JTAG-only connector. This connector is not defined in this specification.

13.2 Target system connector

Because of the number of *Trace Port Analyzers* (TPAs) that are already developed for the ETM architecture, see the *Embedded Trace Macrocell Architecture Specification*, the ETM physical connector and pin assignment has been reused to assist tool compatibility.

The specified target system connector is the AMP Mictor connector. 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.
- 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 13-5 and *Dual target connector pinout* on page 13-6.

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 the run-time control, JTAG, signals on the connector, enabling a single debug connection to the target.

Table 13-1 lists the AMP part numbers for the four possible connectors.

Table 13-1 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.

13.3 Target connector description

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

- *Single target connector pinout*
- *Dual target connector pinout* on page 13-6.

13.3.1 Single target connector pinout

Figure 13-1 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.

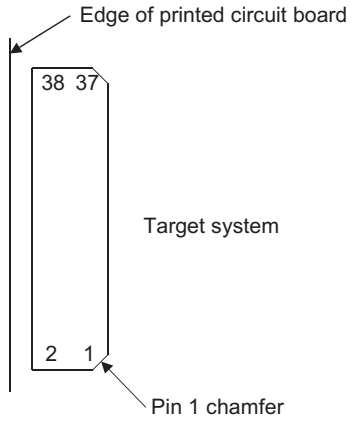


Figure 13-1 Recommended orientation for one connector

Table 13-2 shows all the connections on a single Mictor connector, showing where differences exist between this and the connections specified in the *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 13-2 Single target connector pinout

Pin	Signal name	Pin	Signal name
38	**TRACEDATA[0]**	37	**TRACEDATA[8]**
36	**TRACECTL**	35	**TRACEDATA[9]**
34	Logic 1	33	**TRACEDATA[10]**

Table 13-2 Single target connector pinout (continued)

Pin	Signal name	Pin	Signal name
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**
16	**TRACEDATA[7]**	15	**TCK**
14	**VSupply**	13	**RTCK**
12	**VTRef**	11	**TDO**
10	No connection, was **EXTTRIG**	9	**nSRST**
8	**TRIGOUT**, was **DBGACK**	7	**TRIGIN**, was **DBGRQ**
6	**TRACECLK**	5	**GND**
4	No connection	3	No connection
2	No connection	1	No connection

13.3.2 Dual target connector pinout

Figure 13-2 on page 13-7 shows the arrangement for two connectors. It is recommended that they are placed in line, with pins 1 separated by 1.35 inches.

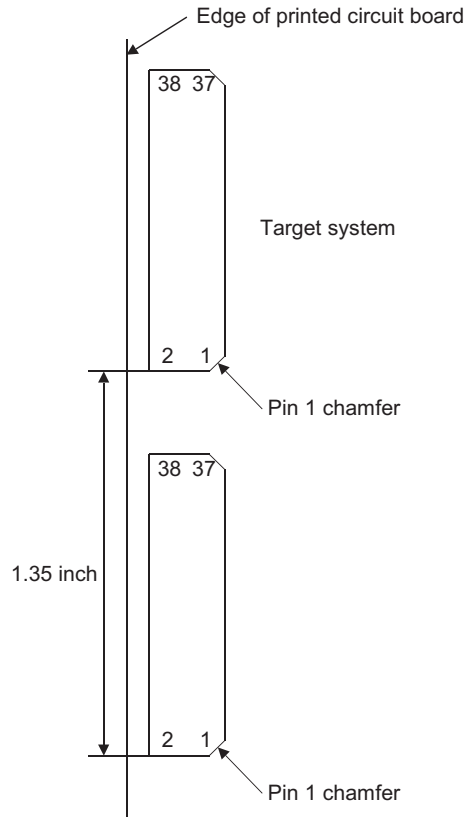


Figure 13-2 Recommended orientation for two connectors

Table 13-3 shows the connections for the secondary Mictor connector. For the primary connector see Table 13-2 on page 13-5. If a signal is not implemented on the target system then it must be replaced with a Logic 0 connection.

Table 13-3 Dual target connector pinout

Pin	Signal name	Pin	Signal name
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 13-3 Dual target connector pinout (continued)

Pin	Signal name	Pin	Signal name
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

13.4 Decoding requirements for trace capture devices

Table 13-4 shows the three conditions that must be decoded by *Trace Capture Devices* (TCDs), for example, a TPA or a logic analyzer.

Table 13-4 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

13.4.1 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.

13.4.2 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 *Embedded Trace Macrocell Architecture Specification*.

13.4.3 Trigger

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

13.4.4 Trace disable

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

13.5 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 as 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 exists 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 in order to maximize the speed at which trace capture is possible.

Figure 13-3 shows the timing for **TRACECLK**.

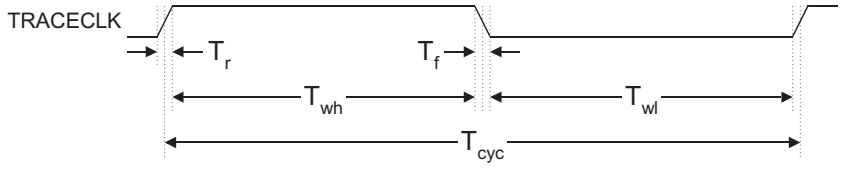


Figure 13-3 TRACECLK specification

It is recommended that trace ports provide a **TRACECLK** as symmetrical as possible, because both edges are used to capture trace. Figure 13-4 shows the setup and hold requirements of the trace data pins, **TRACEDATA[n:0]** and **TRACECTL**, with respect to **TRACECLK**.

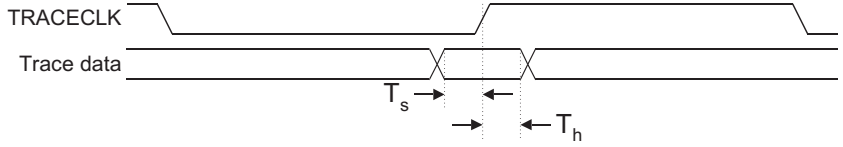


Figure 13-4 Trace data specification

It is recommended that 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. It is recommended that TPAs are able to delay each trace data signal individually by up to a whole clock period, to compensate for trace ports where Ts and Th are not balanced or vary between data signals.

13.6 Signal details

The signals on the target connector pins are:

- *VTRef output*
- *TRACECLK output* on page 13-13
- *TRACECTL output* on page 13-13
- *TRACEDATA[n:1] output* on page 13-13
- *Logic 1 input* on page 13-13
- *Logic 0 input* on page 13-13
- *TRIGIN input* on page 13-13
- *TRIGOUT output* on page 13-14
- *nSRST input* on page 13-14
- *nTRST input* on page 13-14
- *TDI input* on page 13-14
- *TMS input* on page 13-14
- *TCK input* on page 13-14
- *RTCK output* on page 13-15
- *TDO output* on page 13-15
- *VSupply output* on page 13-15
- *GND* on page 13-15
- *No connection* on page 13-15.

13.6.1 VTRef output

The **VTRef** signal is intended to supply a logic-level reference voltage to enable debug equipment to adapt to the signalling 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.

13.6.2 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.

13.6.3 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.

13.6.4 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 Chapter 7 *AMBA 3 ATB Interface*, or direct data from a single trace source.

13.6.5 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**.

13.6.6 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**.

13.6.7 TRIGIN input

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

Note

This pin is equivalent to the **DBGRQ** signal defined within the ETM architecture.

13.6.8 TRIGOUT 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.

Note

This pin is equivalent to the **DBGACK** signal defined within the ETM architecture.

13.6.9 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.

13.6.10 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.

13.6.11 TDI input

TDI is the Test Data In signal from the run control unit to the target JTAG port. It is recommended that you pull this pin to a defined state.

13.6.12 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.

13.6.13 TCK input

TCK is the Test Clock signal from the run control unit to the target JTAG port. It is recommended that this pin is pulled to a defined state.

13.6.14 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.

13.6.15 TDO output

TDO is the Test Data Out from the target JTAG port to the run control unit. This pin must be set to its inactive drive state, tri-state, when the JTAG state machine is not in the Shift-IR or Shift-DR states.

13.6.16 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.

13.6.17 GND

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

13.6.18 No connection

No connection must be made to this pin.

Chapter 14
Trace Formatter

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

14.1 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 3 ATB interface source IDs into a single trace stream. For more information about the AMBA 3 ATB protocol, see Chapter 7 *AMBA 3 ATB Interface*. 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, see *AMBA 3 ATB interface Signals* on page 7-4.

14.2 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 7 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 14-1 shows the structure of a formatter frame.

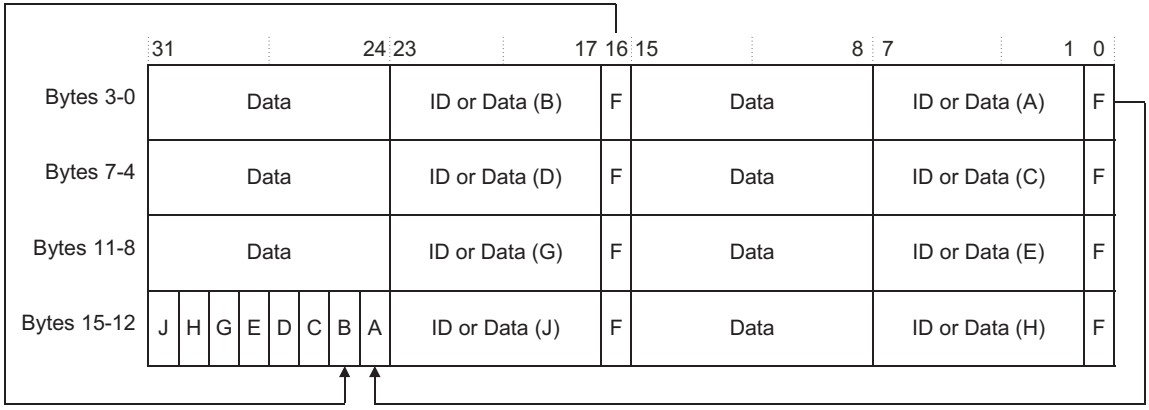


Figure 14-1 Formatter frame structure

Each time the ID changes, at least one byte of data must be output for that ID. Table 14-1 shows the meaning of each bit in a formatter frame. It is output least significant bit first, starting with bit 0.

Table 14-1 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 14-1 on page 14-3 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]
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 14-1 on page 14-3 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.

Table 14-1 Meaning of bits in a formatter frame

Byte number	Bits	Description
	1	Auxiliary bit for byte 2, see bit in Figure 14-1 on page 14-3 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 14-1 on page 14-3 marked C. See bit 0.
	3	Auxiliary bit for byte 6, see bit in Figure 14-1 on page 14-3 marked D. See bit 0.
	4	Auxiliary bit for byte 8, see bit in Figure 14-1 on page 14-3 marked E. See bit 0.
	5	Auxiliary bit for byte 10, see bit in Figure 14-1 on page 14-3 marked G. See bit 0.
	6	Auxiliary bit for byte 12, see bit in Figure 14-1 on page 14-3 marked H. See bit 0.
	7	Auxiliary bit for byte 14see bit in Figure 14-1 on page 14-3 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.

14.2.1 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 14-2 shows the frame this represents.

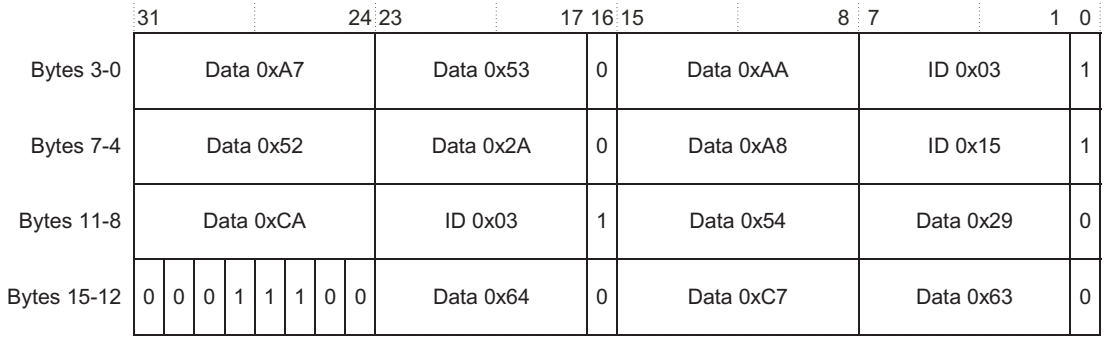


Figure 14-2 Example formatter frame

Table 14-2 shows how this frame is decoded.

Table 14-2 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 0x15
13	Data byte.	0xC7, ID 0x15
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 0x15
15	Auxiliary bits.	-

14.2.2 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 must discard all frame synchronization packets once the start of a frame has been found. See *Modes of operation* on page 14-9 for more information about continuous mode.

Figure 14-3 shows a frame synchronization packet.

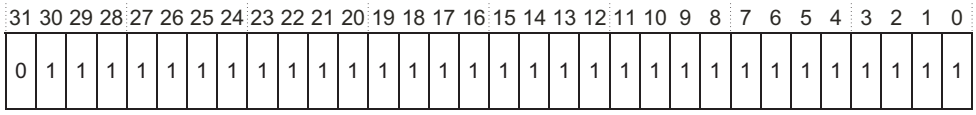


Figure 14-3 Full frame synchronization packet

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.

14.2.3 Half-word 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 14-9 for more information about continuous mode. Figure 14-4 shows a halfword synchronization packet.

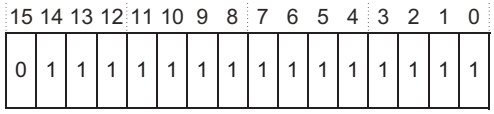


Figure 14-4 Halfword synchronization packet

14.2.4 Special Trace Source IDs

The following IDs are set aside 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. This is accompanied by one byte of data, that must be zero.

0x70-0x7C Reserved.

14.2.5 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 14-9.

14.2.6 Repeating the trace source ID

If a large amount of trace is generated consecutively by a single source ID, then 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.

It is recommended that the source ID is repeated approximately every 10 frames.

14.2.7 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 14-9.

14.3 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.

14.3.1 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 for 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 13-9.
- 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 be generated.

To ensure that all trace is output from a trace sink when stopping trace, extra data may be added to the end of the trace stream, see *Bypass mode* on page 14-10.

14.3.2 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.

14.3.3 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 there data for a trace port.

14.4 Flush of trace data at the end of operation

The formatter must ensure all trace is output at the end of the trace. This is requested by an AMBA 3 ATB protocol flush as described in *AFVALID and AFREADY* on page 7-9. 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.

14.4.1 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 illustrate sequences observed on a 32-bit trace port. Figure 14-5 shows an example of how the last AMBA 3 ATB protocol transaction left 3 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 14-5 End of session example 1

Figure 14-6 on page 14-11 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 14-6 End of session example 2

14.4.2 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 15
ROM table

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

- *About the ROM table* on page 15-2
- *ROM table format* on page 15-3
- *ROM hierarchy* on page 15-6
- *Location of ROM table* on page 15-8.

15.1 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.

The ROM table can be implemented using a 32-bit format or an 8-bit format.

15.2 ROM table format

This section describes the format of the ROM table.

15.2.1 0xFF0-0xFFC component identification registers

The ROM table has a specific class, specified in *Component identification registers* on page 3-7, `0xB105100D`, spread over the lower byte of each word, as specified in *Component identification registers* on page 3-7.

15.2.2 0xFD0-0xFEC Peripheral Identification Register

The Peripheral Identification Register uniquely identifies the SoC or platform. If any of the components pointed to by the ROM are changed or any of their connections are changed, then this register 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 identity 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 read as zero, 4'b0000.

15.2.3 0xF00-0xFE0 reserved

Registers in this region Read As Zero. They are reserved.

15.2.4 0x000-0xEFC ROM entries

There are two formats for ROM entries, 8-bit and 32-bit. All entries must be of the same format.

Table 15-1 shows how, in the 8-bit format, each entry takes 16 bytes of address space

Table 15-1 ROM entries

Offset in entry	Bits	Description
0	[7:0]	Entry[7:0]
4	[7:0]	Entry[15:8]
8	[7:0]	Entry[23:16]
12	[7:0]	Entry[31:24]

Table 15-2 shows how, in the 32-bit format, each entry takes 4 bytes of address space

Table 15-2 ROM entries

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

Table 15-3 shows the format that the entries take.

Table 15-3 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).
[11:2]	-	Reserved, Read As Zero.
[1]	Format	0 = 8-bit format 1 = 32-bit format.
[0]	Entry present	Always 1.

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

If the component occupies several consecutive 4KB blocks, see *Class 0xF PrimeCell or system component* on page 3-23, the base address of the highest block in memory is given. 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.

15.2.5 Expansion above 960 entries

If more than 960 entries are required, or 240 if the 8-bit format is used, 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.

15.3 ROM 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 15-1 shows two ROM tables pointing to the same component, in this case ROM Table D.

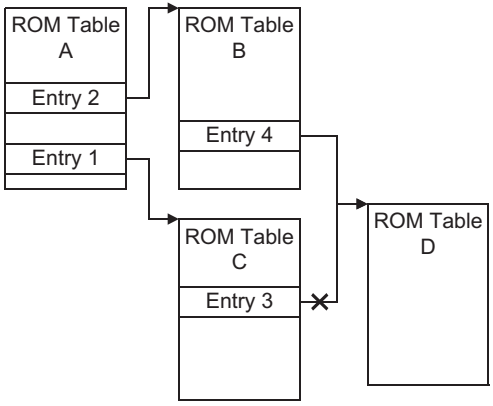


Figure 15-1 Prohibited duplicate ROM table references

A ROM table entry must not point to a ROM table that directly or indirectly points to it. Figure 15-2 shows an example of this prohibited circular reference.

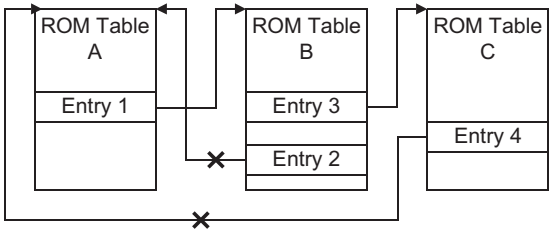


Figure 15-2 Prohibited circular ROM table references

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 implementor for ROM tables only accessed from other ROM tables.

15.4 Location of 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.

15.4.1 From DAP

Each memory *Access Port* (AP), AHB-AP or APB-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 *CoreSight Design Kit Technical Reference Manual*.

15.4.2 From CoreSight compliant processor cores

A CoreSight compliant processor must have a register containing a single ROM table entry. This entry must identify a ROM table that enables the CoreSight components, including the core debug logic, to be found.

An additional register can also be used to contain a second ROM table entry that directly identifies the core debug logic of the device.

15.4.3 From other 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.

Chapter 16
Topology Detection at the System Level

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

16.1 About topology detection at the system level

Chapter 4 *Topology Detection Registers* describes the topology detection requirements of CoreSight components. Chapter 10 *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.

16.2 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 *0xFD0-0xFEC Peripheral Identification Register* on page 15-3.
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 16-5.
   d. Saves the description for later use.

16.2.1 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.

16.3 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. The component must be otherwise ignored.

16.4 Detection algorithm

It is recommended that a debugger connecting to a system executes the following algorithm in order 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 10-7 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 Integration Mode Control Register at address `0xF00`.

Component postamble

Clear bit 0 of the Integration Mode Control Register at address `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.

Chapter 17
Compliance Criteria

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

17.1 About compliance classes

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

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.

17.2 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.

17.2.1 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) component the JTAG interface of which is accessible to the tools. For more information about implementing multiple systems containing DAPs, see *Multiple DAPs* on page 17-10.
- All CoreSight components must:
  - be accessible from the DAP through an AHB-AP or APB-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 shown in Figure 17-8 on page 17-10.

    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 programmer's 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 programmer's 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 programmer's model.

- For each CoreSight component and JTAG controlled processor, all inputs and outputs defined as type Trigger are connected to a *Cross Trigger Interface* (CTI) component, unless there is only one component in the system with trigger 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. The CoreSight Design Kit provides a Cross Trigger Matrix 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 4-6.
- It is recommended that the system is CoreSight compliant at all times, but it is recognized that this might be unachievable 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.

17.2.2 Optional debug functionality

CoreSight Debug systems can also implement the following:

- DAP visibility of system components through an AHB-AP or APB-AP.

Single-core debug

Figure 17-1 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.

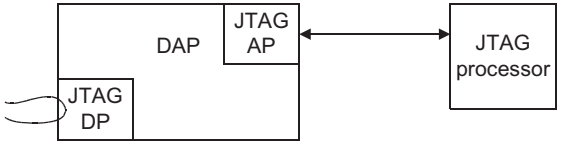


Figure 17-1 Single core with JTAG debug access

Multi-core debug

Figure 17-2 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.

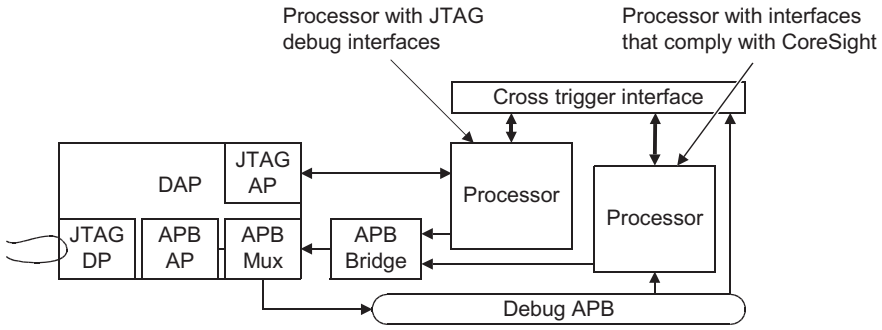


Figure 17-2 Multi-core system

17.3 CoreSight Trace

The section defines the CoreSight Trace compliance class.

17.3.1 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, where each trace sink is an ETB or a TPIU:
  - if a TPIU is implemented, its output is connected to a compliant connector as defined in Chapter 13 *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 17-3.

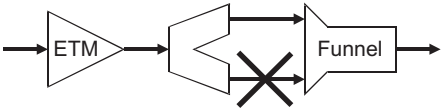


Figure 17-3 Non-compliant replicator and CoreSight trace funnel connection

A particular example that must be avoided is feedback, see example in Figure 17-4.

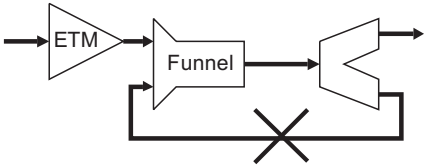


Figure 17-4 Non-compliant feedback loop

Note

The Replicator and Synchronous Bridge are invisible to the tools, and can therefore be used wherever required.

17.3.2 Optional trace functionality

CoreSight Debug systems can also implement CoreSight Debug optional functionality and any of the following:

- tracing of AHB buses using the ARM *AHB Trace Macrocell* (HTM).

Basic single core trace

Figure 17-5 shows an example system with single processor 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.

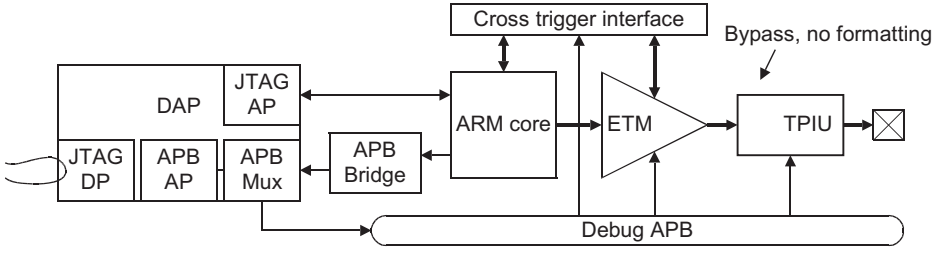


Figure 17-5 Single core trace with formatting bypass

Advanced single core trace

Figure 17-6 on page 17-8 shows an example system with full trace capabilities in a single processor system. The ETM provides ARM core instruction and data 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.

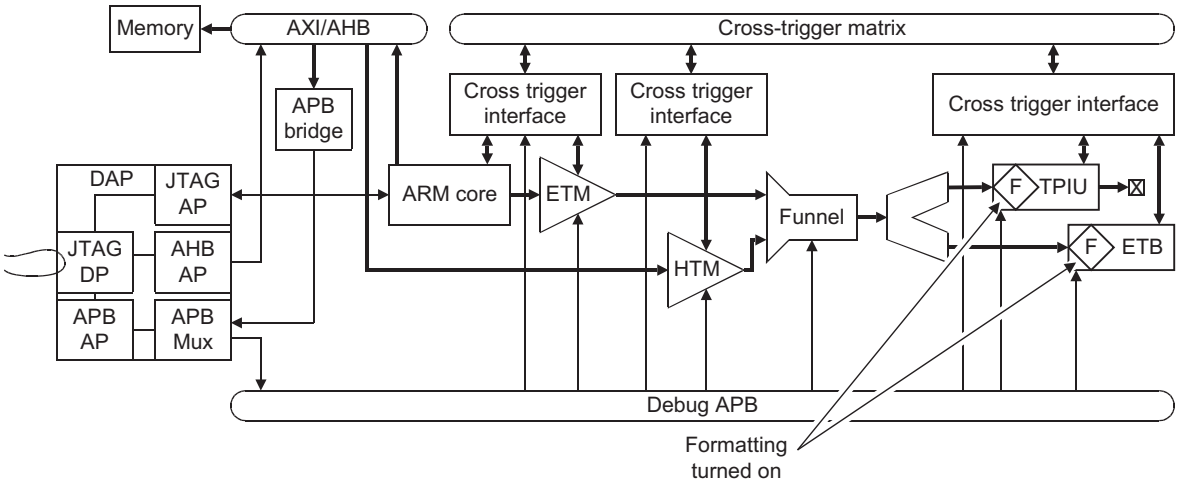


Figure 17-6 Full CoreSight trace with single core

Multi-core trace

Figure 17-7 on page 17-9 shows a system with an ARM processor and a DSP. A third smaller subsystem is added to support merging of multiple CoreSight AMBA 3 ATB interfaces into a single trace stream.

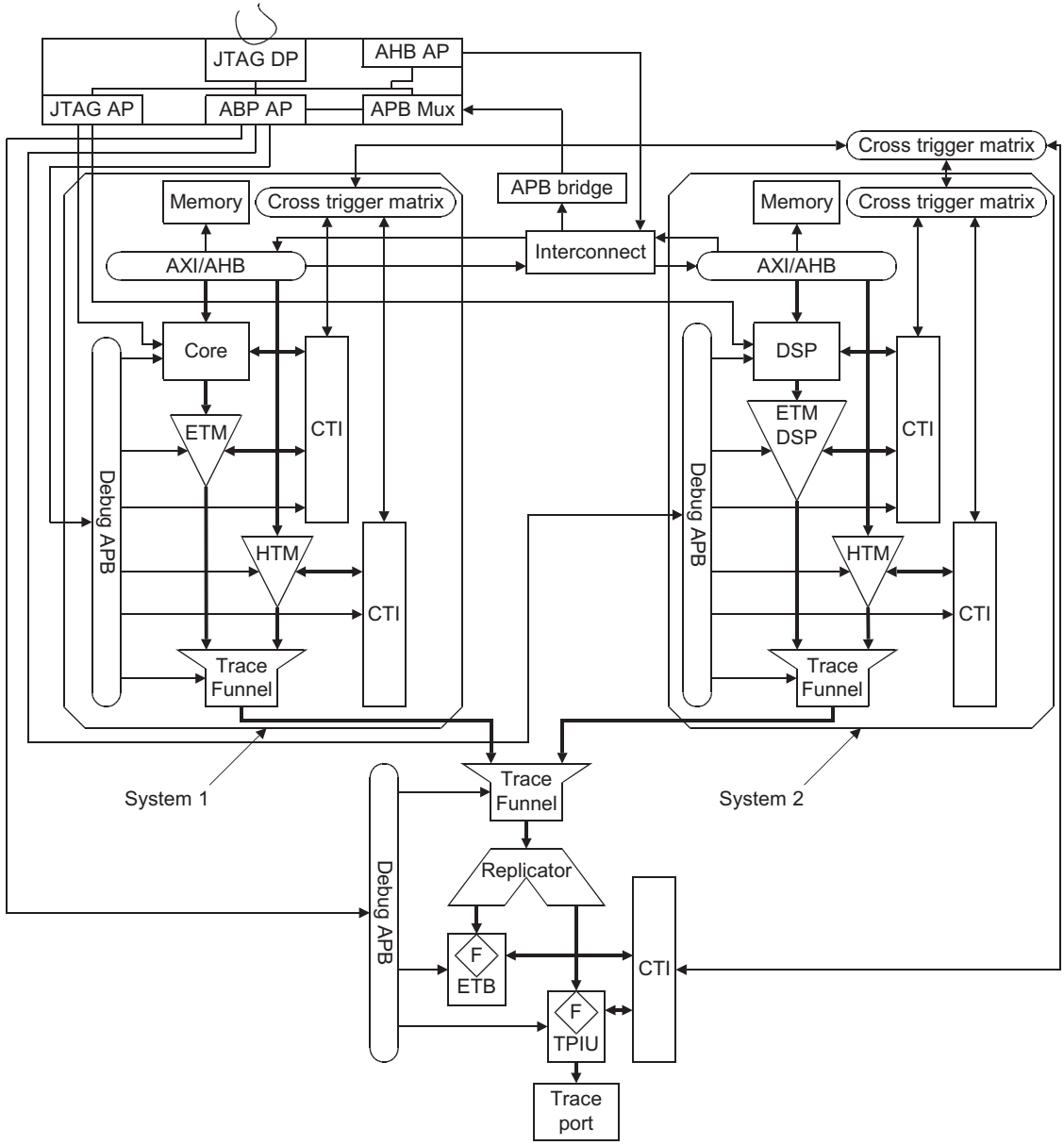


Figure 17-7 Full system trace with ARM core and CoreSight compliant DSP

17.4 Multiple DAPs

The only circumstance in which multiple DAPs can be implemented is where each DAP controls an independent system to be debugged. In particular:

- Each system must be able to be debugged by a debugger that has no support for the other systems.
- Topology detection is performed on a each system independently. Therefore connections between systems are not discovered by the topology detection process and are not supported by the CoreSight system. For example:
  - An AMBA 3 ATB interface trace master in one system must not be connected to an AMBA 3 ATB interface trace slave in another system.
  - Two systems can communicate using shared memory, because this connection is not used by the CoreSight system and is not visible as part of the CoreSight topology detection process.
- 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 shown in Figure 17-8 and must not be as shown in Figure 17-9 on page 17-11.

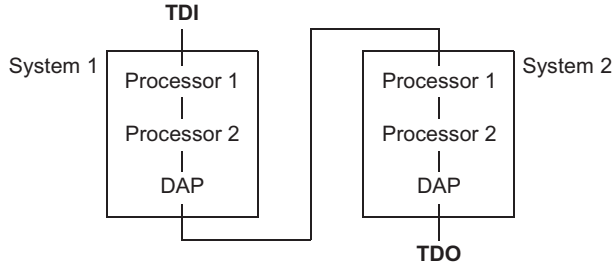


Figure 17-8 JTAG connections across systems

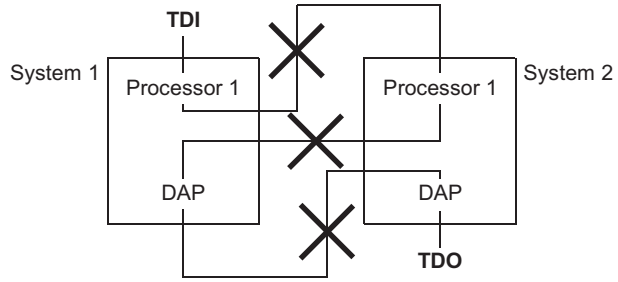


Figure 17-9 Non-compliant interleaved JTAG connections across systems

- Additional JTAG TAP controllers can be implemented in series with JTAG TAP controllers of the CoreSight systems. For example, in Figure 17-10, 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.

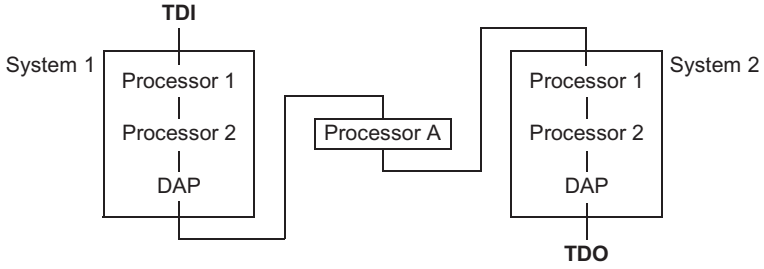


Figure 17-10 Systems with additional JTAG TAP controllers

- The DAP for one system cannot be accessed through the JTAG-AP of the DAP for another system, as shown in Figure 17-11.

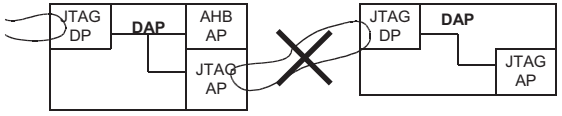


Figure 17-11 Non-compliant DAP connection

Glossary

This glossary describes some of the terms used in technical documents from ARM Limited.

Advanced eXtensible Interface (AXI)
A bus protocol that supports separate address/control and data phases, unaligned data transfers using byte strobes, burst-based transactions with only start address issued, separate read and write data channels to enable low-cost DMA, ability to issue multiple outstanding addresses, out-of-order transaction completion, and easy addition of register stages to provide timing closure.

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

AXI is targeted at high performance, high clock frequency system designs and includes a number of features that make it very suitable for high speed sub-micron interconnect.

Advanced High-performance Bus (AHB)
A bus protocol with a fixed pipeline between address/control and data phases. It only supports a subset of the functionality provided by the AMBA AXI protocol. The full AMBA AHB protocol specification includes a number of features that are not

commonly required for master and slave IP developments and ARM Limited recommends only a subset of the protocol is usually used. This subset is defined as the AMBA AHB-Lite protocol.

See also Advanced Microcontroller Bus Architecture and AHB-Lite.

Advanced Microcontroller Bus Architecture (AMBA)
A family of protocol specifications that describe a strategy for the interconnect. AMBA is the ARM open standard for on-chip buses. It is an on-chip bus specification that details a strategy for the interconnection and management of functional blocks that make up a *System-on-Chip* (SoC). It aids in the development of embedded processors with one or more CPUs or signal processors and multiple peripherals. AMBA complements a reusable design methodology by defining a common backbone for SoC modules.

Advanced Peripheral Bus (APB)
A simpler bus protocol than AXI and AHB. It is designed for use with ancillary or general-purpose peripherals such as timers, interrupt controllers, UARTs, and I/O ports. Connection to the main system bus is through a system-to-peripheral bus bridge that helps to reduce system power consumption.

AHB *See* Advanced High-performance Bus.

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

AHB-AP *See* AHB Access Port.

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. In most cases, the extra facilities provided by a full AMBA AHB interface are implemented more efficiently by using an AMBA AXI protocol interface.

Aligned A data item stored at an address that is divisible by the number of bytes that defines the data size is said to be aligned. Aligned words and halfwords have addresses that are divisible by four and two respectively. The terms word-aligned and halfword-aligned therefore stipulate addresses that are divisible by four and two respectively.

AMBA *See* Advanced Microcontroller Bus Architecture.

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

APB *See* Advanced Peripheral Bus.

Architecture The organization of hardware and/or software that characterizes a processor and its attached components, and enables devices with similar characteristics to be grouped together when describing their behavior, for example, Harvard architecture, instruction set architecture, ARMv6 architecture.

ATB *See* Advanced Trace Bus.

ATB bridge A synchronous ATB bridge provides a register slice to facilitate timing closure through the addition of a pipeline stage. It also provides a unidirectional link between two synchronous ATB domains.

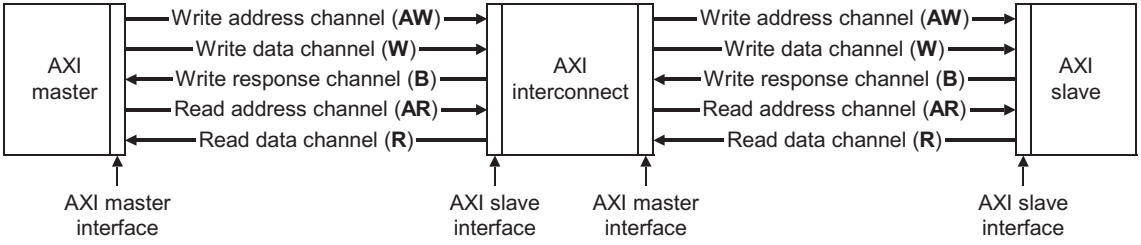
An asynchronous ATB bridge provides a unidirectional link between two ATB domains with asynchronous clocks. It is intended to support connection of components with ATB ports residing in different clock domains.

AXI *See* Advanced eXtensible Interface.

AXI channel order and interfaces

The block diagram shows:

- the order in which AXI channel signals are described
- the master and slave interface conventions for AXI components.



AXI terminology The following AXI terms are general. They apply to both masters and slaves:

Active read transaction

A transaction for which the read address has transferred, but the last read data has not yet transferred.

Active transfer

A transfer for which the **xVALID**[1] handshake has asserted, but for which **xREADY** has not yet asserted.

Active write transaction

A transaction for which the write address or leading write data has transferred, but the write response has not yet transferred.

Completed transfer

A transfer for which the **xVALID/xREADY** handshake is complete.

Payload The non-handshake signals in a transfer.

Transaction An entire burst of transfers, comprising an address, one or more data transfers and a response transfer (writes only).

Transmit An initiator driving the payload and asserting the relevant **xVALID** signal.

Transfer A single exchange of information. That is, with one **xVALID/xREADY** handshake.

The following AXI terms are master interface attributes. To obtain optimum performance, they must be specified for all components with an AXI master interface:

Combined issuing capability

The maximum number of active transactions that a master interface can generate. This is specified instead of write or read issuing capability for master interfaces that use a combined storage for active write and read transactions.

Read ID capability

The maximum number of different **ARID** values that a master interface can generate for all active read transactions at any one time.

Read ID width

The number of bits in the **ARID** bus.

Read issuing capability

The maximum number of active read transactions that a master interface can generate.

Write ID capability

The maximum number of different **AWID** values that a master interface can generate for all active write transactions at any one time.

---

1. The letter **x** in the signal name denotes an AXI channel as follows:

AW Write address channel.
W Write data channel.
B Write response channel.
AR Read address channel.
R Read data channel.

Write ID width

The number of bits in the **AWID** and **WID** buses.

Write interleave capability

The number of active write transactions for which the master interface is capable of transmitting data. This is counted from the earliest transaction.

Write issuing capability

The maximum number of active write transactions that a master interface can generate.

The following AXI terms are slave interface attributes. To obtain optimum performance, they must be specified for all components with an AXI slave interface

Combined acceptance capability

The maximum number of active transactions that a slave interface can accept. This is specified instead of write or read acceptance capability for slave interfaces that use a combined storage for active write and read transactions.

Read acceptance capability

The maximum number of active read transactions that a slave interface can accept.

Read data reordering depth

The number of active read transactions for which a slave interface can transmit data. This is counted from the earliest transaction.

Write acceptance capability

The maximum number of active write transactions that a slave interface can accept.

Write interleave depth

The number of active write transactions for which the slave interface can receive data. This is counted from the earliest transaction.

Byte An 8-bit data item.

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 enable the system to be used again. In other cases, only a warm reset is required.

See also Warm reset.

Core A core is that part of a processor that contains the ALU, the datapath, the general-purpose registers, the Program Counter, and the instruction decode and control circuitry.

Core reset *See* Warm reset.

Cross Trigger Interface (CTI)
Part of an Embedded Cross Trigger device. The CTI provides the interface between a core/ETM and the CTM within an ECT.

Cross Trigger Matrix (CTM)
The CTM combines the trigger requests generated from CTIs and broadcasts them to all CTIs as channel triggers within an Embedded Cross Trigger device.

CTI *See* Cross Trigger Interface.

CTM *See* Cross Trigger Matrix.

CoreSight The infrastructure for monitoring, tracing, and debugging a complete system on chip.

Debug Access Port (DAP)
A TAP block that acts as an AMBA, AHB or AHB-Lite, master for access to a system bus. The DAP is the term used to encompass a set of modular blocks that support system wide debug. The DAP is a modular component, intended to be extendable to support optional access to multiple systems such as memory mapped AHB and CoreSight APB through a single debug interface.

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

Debug Test Access Port (DBGTAP)
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 **DBGTDI**, **DBGTDO**, **DBGTMS**, and **TCK**. The optional terminal is **TRST**. This signal is mandatory in ARM cores because it is used to reset the debug logic.

DNM *See* Do Not Modify.

Do Not Modify (DNM)
In Do Not Modify fields, the value must not be altered by software. DNM fields read as Unpredictable values, and must only be written with the same value read from the same field on the same processor.
DNM fields are sometimes followed by RAZ or RAO in parentheses to show which way the bits must read for future compatibility, but programmers must not rely on this behavior.

Doubleword A 64-bit data item. The contents are taken as being an unsigned integer unless otherwise stated.

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

ECT *See* Embedded Cross Trigger.

Embedded Cross Trigger (ECT)
The ECT is a modular component to support the interaction and synchronization of multiple triggering events with an SoC.

EmbeddedICE logic An on-chip logic block that provides TAP-based debug support for ARM processor cores. It is accessed through the TAP controller on the ARM core using the JTAG interface.

EmbeddedICE-RT The JTAG-based hardware provided by debuggable ARM processors to aid debugging in real-time.

Embedded Trace Buffer
The ETB provides on-chip storage of trace data using a configurable sized RAM.

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

ETB *See* Embedded Trace Buffer.

ETM See *Embedded Trace Macrocell*.

Event 1 (Simple) An observable condition that can be used by an ETM to control aspects of a trace.

2 (Complex) A boolean combination of simple events that is used by an ETM to control aspects of a trace.

Formatter The formatter is an internal input block in the ETB and TPIU that embeds the trace source ID within the data to create a single trace stream.

Half-rate clocking (ETM)
Dividing the trace clock by two so that the TPA can sample trace data signals on both the rising and falling edges of the trace clock. The primary purpose of half-rate clocking is to reduce the signal transition rate on the trace clock of an ASIC for very high-speed systems.

Halfword A 16-bit data item.

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

IEM *See* Intelligent Energy Management.

IGN *See* Ignore.

Ignore (IGN) Must ignore memory writes.

IMB *See* Instruction Memory Barrier.

Implementation-defined
The behavior is not architecturally defined, but is defined and documented by individual implementations.

Implementation-specific
The behavior is not architecturally defined, and does not have to be documented by individual implementations. Used when there are a number of implementation options available and the option chosen does not affect software compatibility.

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

Intelligent Energy Management (IEM)
A technology that enables dynamic voltage scaling and clock frequency variation to be used to reduce power consumption in a device.

Joint Test Action Group (JTAG)
The name of the organization that developed standard IEEE 1149.1. This standard defines a boundary-scan architecture used for in-circuit testing of integrated circuit devices. It is commonly known by the initials JTAG.

JTAG *See* Joint Test Action Group.

JTAG Access Port (JTAG-AP)
An optional component of the DAP that provides JTAG access to on-chip components, operating as a JTAG master port to drive JTAG chains throughout a SoC.

JTAG-AP *See* JTAG Access Port.

JTAG Debug Port (JTAG-DP)
An optional external interface for the DAP that provides a standard JTAG interface for debug access.

JTAG-DP *See* JTAG Debug Port.

Macrocell A complex logic block with a defined interface and behavior. A typical VLSI system comprises several macrocells (such as a processor, an ETM, and a memory block) plus application-specific logic.

Power-on reset *See* Cold reset.

Processor A processor is the circuitry in a computer system required to process data using the computer instructions. It is an abbreviation of microprocessor. A clock source, power supplies, and main memory are also required to create a minimum complete working computer system.

RealView ICE A system for debugging embedded processor cores using a JTAG interface.

Replicator A replicator 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.

Reserved A field in a control register or instruction format is reserved if the field is to be defined by the implementation, or produces Unpredictable results if the contents of the field are not zero. These fields are reserved for use in future extensions of the architecture or are implementation-specific. All reserved bits not used by the implementation must be written as 0 and read as 0.

SBO *See* Should Be One.

SBZ *See* Should Be Zero.

SBZP *See* Should Be Zero or Preserved.

Should Be One (SBO)
Should be written as 1 (or all 1s for bit fields) by software. Writing a 0 produces Unpredictable results.

Should Be Zero (SBZ)
Should be written as 0 (or all 0s for bit fields) by software. Writing a 1 produces Unpredictable results.

Should Be Zero or Preserved (SBZP)
Should be written as 0 (or all 0s for bit fields) by software, or preserved by writing the same value back that has been previously read from the same field on the same processor.

TAP *See* Test access port.

TCD *See* Trace Capture Device.

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 **TRST**. This signal is mandatory in ARM cores because it is used to reset the debug logic.

TPA See *Trace Port Analyzer*.

TPIU *See* Trace Port Interface Unit.

Trace Capture Device (TCD)
: A generic term to describe Trace Port Analyzers, logic analyzers, and on-chip trace buffers.

Trace funnel
: A device that combines multiple trace sources onto a single bus.

Trace hardware
: A term for a device that contains an Embedded 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.

Warm reset
: Also known as a core reset. Initializes the majority of the processor excluding the debug controller and debug logic. This type of reset is useful if you are using the debugging features of a processor.

Word
: A 32-bit data item.