CoreSight™ Program Flow Trace™

PFTv1.0 and PFTv1.1

Architecture Specification

ARM®

ARM IHI 0035B (ID060811)

CoreSight Program Flow Trace
Architecture Specification

Release Information

The following changes have been made to this document.

Change history

Date	Issue	Confidentiality	Change
09 April 2008	A	Non-Confidential	First release for v1.0
31 March 2011	B	Non-Confidential	First release for v1.1

Some information in this document was published previously in the *Embedded Trace Macrocell Architecture Specification*.

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

CoreSight Program Flow Trace Architecture Specification

Preface

This preface introduces the *Program Flow Trace* Architecture Specification, that defines the architecture for a *Program Trace Macrocell* (PTM). It contains the following sections:

About this specification

This is the *Architecture Specification* for Program Flow Trace. It defines the architecture for a *Program Trace Macrocell* (PTM).

Product revision status

The r*n*p*n* identifier indicates the revision status of some of the products referenced in this specification, where:

r*n*	Identifies the major revision of the product.
p*n*	Identifies the minor revision or modification status of the product.

Intended audience

This specification is written for the following target audiences:

- Designers of development tools providing support for PFT functionality. All chapters of this specification are relevant to these users.
- Advanced users of development tools providing support for PFT functionality. Chapter 3 and Chapter 5 of this specification are particularly relevant to these users.
- Designers of Trace Port Analyzers, or other trace capture devices. Chapter 4 and Appendix B of this specification are particularly relevant to these users.
- Designers of ARM processor products that include a PTM. Chapter 4 and Chapter 5 of this specification are particularly relevant to these users.

If you are a hardware engineer who is incorporating a PTM into an ARM processor design you must also refer to the relevant PTM *Technical Reference Manual*.

ARM Limited recommends that all users of this specification have experience of the ARM processor architecture. The ARM *Architecture Reference Manual* describes the ARM processor architecture.

Using this specification

This specification is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter for an introduction to:
- the Program Flow Trace architecture
- a *Program Trace Macrocell* (PTM) implementation of the architecture.

Chapter 2 *Program Flow Tracing*

Read this chapter for information about how the PFT architecture traces software execution.

Chapter 3 *Program Trace Macrocell Programmers Model*

Read this chapter for information about the programmers model for the Program Flow Trace architecture, including descriptions of the PTM registers.

Chapter 4 *Program Flow Trace Protocol*

Read this chapter for a description of the PFT protocol and the information output by the PTM.

Chapter 5 *Tracing Exceptions*

Read this chapter for a description of how a PTM traces exceptions.

Appendix A *PTM Quick Reference Information*

Read this appendix for quick reference information about a PTM implementation, including a summary of the parts of a PTM specification that are IMPLEMENTATION DEFINED.

Appendix B *Trace Decompressor Operation*

Read this appendix for summary of how a PTM trace decompressor must operate.

Appendix C *Software Issues for PFT*

Read this appendix for information about software issues with Program Flow Tracing. It gives information about tracing dynamically-loaded code, and about software and hardware support for Context IDs.

Appendix D *Architecture Version Information*

Read this appendix for a summary of information about the different architecture versions.

Glossary Read the glossary for definitions of some of the terms used in this specification.

Conventions

Conventions that this specification can use are described in:

- *Typographical*
- *Numbering*.

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** >	Enclose replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0 <Rd>, <CRn>, <CRm>, <Opcode_2>`

Numbering

A numbering convention used in this specification is:

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

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

- 'h7B4 is an unsized hexadecimal value.
- 8'd9 is an 8-bit wide decimal value of 9.
- 8'h3F is an 8-bit wide hexadecimal value of `0x3F`. This is equivalent to b0011 1111.
- 8'b1111 is an 8-bit wide binary value of b0000 1111.

Further reading

This section lists publications by ARM.

See `http://infocenter.arm.com/` for access to ARM documentation.

ARM publications

This specification defines the Program Flow Trace architecture. See the following documents for other relevant information:

- *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition* (ARM DDI 0406)
- *Embedded Trace Macrocell Architecture Specification* (ARM IHI 0014)
- *CoreSight Architecture Specification* (ARM IHI 0029)
- *AMBA AHB Trace Macrocell (HTM) Technical Reference Manual* (ARM DDI 0328)
- *ARM Debug Interface v5 Architecture Specification* (ARM IHI 0031)
- *RealView® ICE and RealView Trace User Guide* (ARM DUI 0155).

Feedback

ARM welcomes feedback on the Program Flow Trace architecture, and on its documentation.

Feedback on the Program Flow Trace architecture

If you have any comments or suggestions about this architecture, contact your supplier and give:

- the product name
- a concise explanation.

Feedback on this specification

If you have any comments on this specification, send an e-mail to `errata@arm.com`. Give:

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

ARM also welcomes general suggestions for additions and improvements.

Chapter 1
Introduction

This chapter contains a brief introduction to the Program Flow Trace architecture and to a *Program Trace Macrocell* (PTM) that implements the architecture. It contains the following section:

- *About the Program Trace Macrocell* on page 1-14.

1.1 About the Program Trace Macrocell

A *Program Trace Macrocell* (PTM) is a real-time trace module providing instruction tracing of a processor. A PTM is an integral part of an ARM RealView debug solution. The following sections describe the PTM:

- *Structure of a PTM*
- *The debug environment*
- *Thumb, ThumbEE, and Java support* on page 1-15
- *Connections to a PTM* on page 1-15
- *Trace compression* on page 1-16
- *Resets* on page 1-17.

1.1.1 Structure of a PTM

The main features of a PTM are:

Trace generation Outputs information that helps you to understand the operation of the processor. The trace protocol provides a real-time trace capability for processors that are deeply embedded in much larger ASIC designs.

Note

You cannot determine how the processor is operating internally by observing the pins of the ASIC, because the ASIC typically includes significant amounts of on-chip memory.

Triggering and filtering facilities

The PFT architecture specification defines a range of triggering and filtering resources that can be used to control tracing. For a number of these, it is IMPLEMENTATION DEFINED:

- whether a PTM implementation includes the resource
- how many instances of the resource are included.

If you are designing a PTM you can specify the exact set of triggering and filtering resources required by your particular application.

Resources include address comparators, counters, and sequencers.

1.1.2 The debug environment

A software debugger provides the user interface to the PTM. The debugger can configure all the PTM facilities, typically using a JTAG or Serial Wire interface. The debugger also displays the captured trace information.

The PTM compresses the trace information and exports it to a CoreSight trace capture subsystem. The CoreSight trace capture subsystem either:

- Exports the trace through a trace port. An external *Trace Port Analyzer* (TPA) captures the trace information, as Figure 1-1 on page 1-15 shows.
- Writes the trace directly to an on-chip *Embedded Trace Buffer* (ETB). When the trace capture is complete, the trace is read out at low speed, typically using a JTAG or Serial Wire interface.

When the trace has been captured the debugger extracts the information from the TPA or ETB and decompresses it to provide a full disassembly, with symbols, of the code that was executed. The debugger can also link this back to the original high-level source code, showing you how the code executed on the target system.

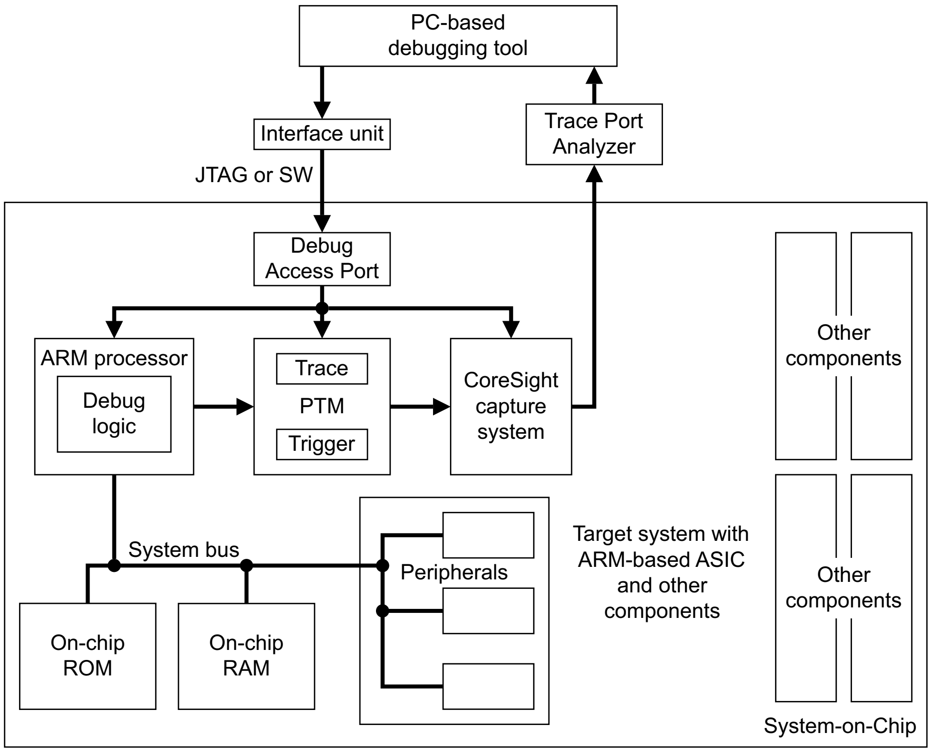


Figure 1-1 Example debugging environment

A system that includes an ETB in the CoreSight capture system might not include a trace port and separate TPA. Such a system must download the captured trace data through the *Debug Access Port* (DAP).

1.1.3 Thumb, ThumbEE, and Java support

ARM, Thumb®, and ThumbEE instructions can be fully traced. In processors that include a nontrivial Jazelle® implementation, Java bytecodes are not traced. However the trace always contains information about when the processor changes its instruction set or security state.

1.1.4 Connections to a PTM

Figure 1-2 shows the main connections to a PTM.

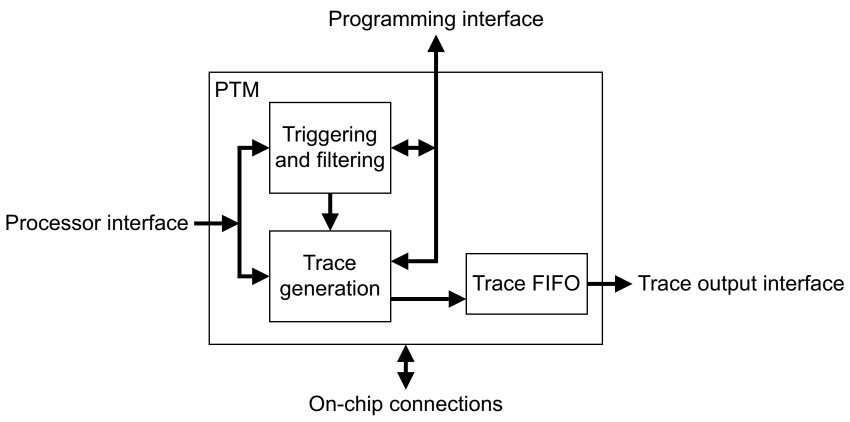


Figure 1-2 Main connections to a PTM

The PTM interfaces are:

Processor interface

This connects the PTM to the processor it is tracing, and carries all the information that is to be traced.

Programming interface

Use this interface to program the PTM registers, to configure and enable tracing and triggering. Typically, this interface is driven by a CoreSight DAP, that conforms to the *ARM Debug Interface v5 Architecture Specification*. The off-chip debugging tool drives the DAP through a JTAG or similar interface.

You can also drive the programming interface through a coprocessor interface or a memory-mapped interface. You can do this from the processor, or from on-chip bus fabric. This means software running on the system can configure and control tracing.

Trace output interface

The PTM uses this interface to output its trace data, for export to an off-chip Trace Port Analyzer or capture in an on-chip buffer. Typically, this interface is based on the *AMBA Trace Bus* (ATB) standard, for connection to a CoreSight trace capture device.

On-chip connections

These consist of additional IMPLEMENTATION DEFINED interfaces supported by a PTM. Typically, these include:

- external inputs, for connections from a cross-triggering network
- extended external inputs, typically used for connections from a performance monitor unit on the processor
- external outputs, for connections to a cross-triggering network
- a trigger output, to signal trigger events to trace capture devices
- permitted debug control signals, for example a signal that disables noninvasive debug
- timestamps, for use in correlating multiple trace streams.

1.1.5 Trace compression

The trace produced by the PTM is compressed to reduce the number of additional pins required on the ASIC, or to reduce the amount of memory required by the ETB.

The trace is compressed using the following techniques:

- the PTM outputs address information only when the processor branches to a location that cannot be directly inferred from the source code
- when the PTM outputs an address, it does not output any high-order bytes that have not changed since the previous trace address
- the PTM outputs trace only when it can use the full width of the trace output interface.

Note

For the debugger to be able to decode the trace, you must supply a static image of the code being executed. Self-modifying code cannot be traced because of this restriction.

1.1.6 Resets

This document refers to the following resets and reset operations:

Processor reset

This resets the processor, making it start execution from the reset vector address. It does not reset any PTM registers. The PTM indicates the processor reset by inserting an exception packet in the trace stream. The exception packet indicates that the exception was a processor reset.

PTM reset This is the main reset for the entire PTM, and resets all resettable PTM registers. The PTM register descriptions in *PTM register descriptions* on page 3-75 define the registers that can be reset.

Power-on reset

Whether a PTM supports a power-on reset is IMPLEMENTATION DEFINED. If power-on reset is supported, a power-on reset must perform a PTM reset.

Writing to the Programming bit

Writing to the Programming bit of the Main Control Register is a reset operation that resets parts of the PTM to their PTM reset state. You reset some parts of the PTM by writing a 1 to this bit, and reset other parts by writing 0 to this bit. For more information see *Programming bit and associated state* on page 3-70.

On a PTM reset, the state of the Main Control Register is reset to the state described in Table 3-17 on page 3-75. In particular, the Power-down bit and the Programming bit are set to 1.

On a PTM reset, the status of registers or individual bits is UNKNOWN if not specified in the register description.

Note

See *Programming bit and associated state* on page 3-70 for details of how the value of the Programming bit affects the register reset values on a PTM reset.

Chapter 2
Program Flow Tracing

This chapter describes how the PFT architecture traces software execution. It contains the following sections:

2.1 About Program Flow Tracing

When tracing processor execution, other ARM trace architectures generate trace for every instruction that the processor commits for execution. This generates trace that is easy to interpret, but it normally requires a high trace bandwidth. The *Program Flow Trace* (PFT) architecture assumes that any trace decompressor has a copy of the program being traced, and generally outputs only enough trace for the decompressor to reconstruct the program flow. However, its trace output also enables a decompressor to reconstruct the program flow when it does not have a copy of parts of the program, for example because the program uses self-modifying code.

The PFT architecture also provides full information about exceptions, and the instruction set state, security state, and current Context ID of the processor. It can also provide cycle count information, and timestamping.

A trace macrocell that implements the PFT architecture is called a *Program Trace Macrocell* (PTM).

PFT identifies certain instructions in the program, and certain events, as *waypoints*. A waypoint is a point where instruction execution by the processor might involve a change in the program flow. A PTM only traces these waypoints.

PFT waypoints include:

- all indirect branches
- conditional and unconditional direct branches
- all exceptions
- any instruction that changes the instruction set state or security state of the processor
- when Halting debug-mode is enabled, entering or leaving Debug state
- synchronization primitives.

When a waypoint occurs, the PTM generates trace data that describes the waypoint. From this data, a trace decompressor can determine how many instructions have been executed since the previous waypoint, and therefore can reconstruct the execution stream. In effect, the PTM outputs an indicator at each waypoint, and the decompressor matches these indicators with the waypoints in the program code, to reconstruct the program flow.

To provide this waypoint matching, certain instructions are treated as *waypoint instructions*, regardless of whether:

- they pass or fail their condition code check
- they are part of an IT block, if they are Thumb instructions.

Waypoint instructions are a subset of the waypoints that a PTM traces:

- A waypoint instruction is an instruction that might cause a change in the program flow, and that can be statically determined from the program image.
- Some instructions might cause a waypoint, but cannot be statically determined from the program image. These are not waypoint instructions. Typically, these are instructions that cause an exception or are affected by an exception. The PTM traces the exception as a waypoint, but that trace does not correspond to a waypoint instruction.

PFT only traces execution at waypoints. Tracing a waypoint implies the execution of all instructions from the target address of the previous waypoint up to the current waypoint. Nonwaypoint instructions are not traced explicitly, and a debugger can infer the execution of a block of contiguous instructions between two waypoints only when the waypoint at the end of the block is traced. The concept of a block of instructions is used throughout this document, and refers to the contiguous block of instructions between two waypoints.

The following subsections give more information about PTM tracing:

- *Tracing branches* on page 2-21
- *Tracing exceptions* on page 2-21
- *Nonwaypoint instructions* on page 2-21
- *PFT trace example* on page 2-21.

2.1.1 Tracing branches

When the processor executes a direct branch instruction, if a trace decompressor knows whether the instruction passed its condition code check it can determine the destination address of the instruction, because it knows the instruction address and opcode. Therefore, by default, a PTM traces a direct branch by generating an *atom*:

- an E atom indicates that the branch instruction passed its condition code check
- an N atom indicates that the branch instruction failed its condition code check.

The PTM can assemble a number of atoms, and output them as a single byte of trace in an *atom header*.

For an indirect branch instruction, when the return stack is disabled or not implemented:

- If the instruction fails its condition code check there is no branch and the PTM generates an N atom.
- If the instruction passes its condition code check the PTM must trace the destination address explicitly, and it generates a *branch address packet* to do this. A branch address packet implies an E atom.

Note

- A return stack is an optional PFT feature that reduces the number of branch packets generated by the PTM. and therefore significantly reduces the amount of trace output. For more information see *Use of a return stack* on page 4-195.
- A trace decompressor can configure the PTM to use branch address packets to trace direct branches, by configuring it for *branch broadcasting*. For example, this might be useful for tracing a `BLX <immed>` branch that changes to Thumb state. Generating a branch address packet means that the change to Thumb state is indicated explicitly in the trace stream.

2.1.2 Tracing exceptions

When an exception occurs, the PTM must output the destination address. This is similar to the branch address packet used to trace an indirect branch. However, the PTM must output extra address information, to indicate where execution had reached when the exception was taken. This address is related to the value the processor stores in R14 when taking the exception. The PTM does not generate any atoms on tracing an exception.

2.1.3 Nonwaypoint instructions

When the processor executes a nonwaypoint instruction, no information is included in the trace stream, because the trace decompressor can determine the execution of these instructions by analyzing the program image. However, the decompressor cannot directly determine any condition code results for nonwaypoint instructions from its analysis of the trace stream.

2.1.4 PFT trace example

Table 2-1 shows an instruction flow, executing ARM instructions, that includes examples of branch execution, and an exception, and the PFT trace generated, for a PTM that is not using a return stack.

Table 2-1 PTM trace example

Address	Instruction	Trace, if any, with explanation
`0x1000`	`MOV`	-
`0x1004`	`ADD`	-
`0x1008`	`B 0x1100`	Direct branch taken. E atom generated.
`0x1100`	`MOV`	-
`0x1104`	`LDR`	-

Table 2-1 PTM trace example (continued)

Address	Instruction	Trace, if any, with explanation
`0x1108`	`ADD`	-
`0x110C`	`CMP`	-
`0x1110`	`BNE 0x1104`	Direct branch taken. E atom generated.
`0x1104`	`LDR`	-
`0x1108`	`ADD`	-
`0x110C`	`CMP`	-
`0x1110`	`BNE 0x1104`	Direct branch taken. E atom generated.
`0x1104`	`LDR`	-
`0x1108`	`ADD`	-
`0x110C`	`CMP`	-
`0x1110`	`BNE 0x1104`	Direct branch not taken. N atom generated.
`0x1114`	`LDR`	-
`0x1118`	`STR`	-
`0x111C`	`MOV PC, Rn`	Indirect branch taken. Generate branch address packet indicating new address, `0x2000`.
`0x2000`	`MOV`	-
`0x2004`	`ADD`	-
`0x2008`	`MOVEQ PC, Rm`	Indirect branch not taken. N atom generated.
`0x200C`	`MOV`	-
`0x2010`	`ADD`	-
`0x2014`	`LDR PC`	Indirect branch taken. Generate branch address packet indicating new address, `0x3000`.
`0x3000`	`MOV`	-
`0x3004`	`ADD`	-
`0x3008`	`SUB`	-
`0x300C`	`MOV`	-
`0x3010`	`ADD`	-
`0x3014`	`SUB`	-
`0x3018`	`MOV`	-
`0x301C`	`ADD`	-
`0x3020`	`SUB`	-
`0x3024`	`MOV`	-
`0x3028`	`ADD`	-
`0x302C`	`SUB`	-

Table 2-1 PTM trace example (continued)

Address	Instruction	Trace, if any, with explanation
`0x3030`	`MOV`	-
`0x3034`	`ADD`	-
-	IRQ taken	Exception taken. Generate packets indicating address of last instruction executed, `0x3034`, and destination address, `0x0018`.
`0x0018`	`ADD`	No trace. This is the start of the IRQ exception handler.
Processor continues execution of the IRQ handler.		

2.2 Waypoint instructions

The following tables list the ARMv7 instructions that are always classified as waypoint instructions:

- Table 2-2, *Direct branches, ARM instruction set*
- Table 2-3, *Direct branches, Thumb and ThumbEE instruction sets*
- Table 2-4 on page 2-25, *Indirect branches, ARM instruction set* on page 2-25
- Table 2-5 on page 2-25, *Indirect branches, Thumb and ThumbEE instruction sets* on page 2-25.

Unpredictable encodings on page 2-26 describes the PFT requirements when the processor attempts to execute an instruction encoding that the architecture defines as UNPREDICTABLE.

Table 2-2 Direct branches, ARM instruction set

Instruction	Description
`B`	Unconditional branch
`B<cc>`	Conditional Branch
`BL`	Branch and Link
`BLX <immed>`	Branch with link and exchange
`ISB`	Instruction Synchronization Barrier (ISB), including CP15 encodings[a]
`DMB`	Data Memory Barrier (DMB), including CP15 encodings[a], if supported[b]
`DSB`	Data Synchronization Barrier (DSB), including CP15 encodings[a], if supported[b]

a. Includes the CP15 encodings in ARMv7 and in earlier versions of the ARM architecture.
b. Bit [24] of the ETMCCER determines whether the PTM treats DMB and DSB operations as waypoint instructions, see *Configuration Code Extension Register, ETMCCER* on page 3-103.

Table 2-3 Direct branches, Thumb and ThumbEE instruction sets

Instruction	Description
16-bit instruction space	
`B`	Unconditional branch
`B<cc>`	Conditional Branch
`CZB` or `CNZB`	Compare with zero and branch
32-bit instruction space	
`B`	Unconditional branch
`B<cc>`	Conditional Branch
`BL <immed>`	Branch and Link
`BLX <immed>`	Branch with link and exchange
`ISB`	Instruction Synchronization Barrier (ISB), including CP15 encodings[a]
`DMB`	Data Memory Barrier (DMB), including CP15 encodings[a], if supported[b]
`DSB`	Data Synchronization Barrier (DSB), including CP15 encodings[a], if supported
`ENTERX`	Enter ThumbEE state
`LEAVEX`	Leave ThumbEE state

a. Includes the CP15 encodings in ARMv7 and in earlier versions of the ARM architecture.

b. Bit [24] of the ETMCCER determines whether the PTM treats DMB and DSB operations as waypoint instructions, see *Configuration Code Extension Register, ETMCCER* on page 3-103.

Table 2-4 Indirect branches, ARM instruction set

Instruction	Description
`RFE`	Return from Exception
Data processing instructions that modify the PC	
`BX`	Branch and exchange
`BLX <reg>`	Branch with link and exchange
`BXJ`	Branch and exchange to Jazelle
`LDR` or `LDRT` to the PC	Load a word to the PC
`LDM` including the PC	Load multiple to the PC
`ERET`	Exception return, when Virtualization is supported

Table 2-5 Indirect branches, Thumb and ThumbEE instruction sets

Instruction	Description
16-bit instruction space	
`ADD` or `MOV` to the PC	Data processing instruction that modifies the PC
`BX`	Branch and exchange
`BLX <reg>`	Branch with link and exchange
`POP` including the PC	Pop from the stack including the PC
`HB`, `HBL`, `HBP`, or `HBLP`	Handler branches, ThumbEE instruction set only
32-bit instruction space	
`RFE`	Return from Exception
`LDM` including the PC	Load multiple to the PC
`TBB` or `TBH`	Table branch
`BXJ`	Branch and exchange to Jazelle
`SUBS PC, LR`	Data processing instruction that modifies the PC
`LDR` to the PC	Load a word to the PC
`ERET`	Exception return, when Virtualization is supported

——— Note ———

There are waypoint instructions which are permitted in other modes but are UNDEFINED in Hyp mode. When in Hyp mode, the following instructions are not waypoints:

- `RFE`
- `SUBS PC, LR, #N`, where N is not zero, when executing in Thumb state
- all flag setting data processing operations with the PC as the target register, when executing in ARM state

- `LDM` (exception return).

Where these instructions might also have been classified as exception return instructions, they are not classified as exception return instructions when in Hyp mode.

2.2.1 UNPREDICTABLE encodings

The ARM and Thumb instruction sets include many instruction encodings that the ARM architecture defines as UNPREDICTABLE. In particular, in the Thumb instruction set there are many instructions where the use of the PC as a destination register results in UNPREDICTABLE behavior. The PFT architecture does not specify UNPREDICTABLE instruction encodings as waypoint instructions. However the UNPREDICTABLE behavior of such an instruction might require the PTM to trace it as a waypoint instruction.

Because UNPREDICTABLE encodings are never executed in valid program code, trace decompression tools can safely ignore any UNPREDICTABLE encodings. However, ARM recommends that, if a decompression tool encounters an UNPREDICTABLE encoding, it indicates this to the user to ensure they are aware that some UNPREDICTABLE behavior might have occurred.

Note

Trace decompression might not accurately represent the real execution of the instruction stream, particularly if an UNPREDICTABLE encoding causes a branch that is not explicitly traced.

2.3 Upgrading a nonwaypoint instruction on an exception

A PTM must trace an exception by upgrading the last successfully executed instruction to be a waypoint instruction, if it was not already a waypoint instruction. How the PTM traces the exception depends on whether the last successfully executed instruction was already a waypoint instruction:

If the last instruction executed was already a waypoint instruction

The PTM outputs an exception branch address packet, indicating the exception taken and the exception vector address.

If the last instruction executed was not already a waypoint instruction

The PTM outputs a packet that indicates the address of this instruction, followed by an exception branch address packet that indicates the exception taken and the exception vector address.

The rest of this specification gives more information about tracing exceptions. In particular, Chapter 5 *Tracing Exceptions* gives a detailed description of how each type of exception is traced, with examples.

2.4 Timestamping

The PFT architecture supports timestamping. This is a mechanism where a time value is inserted into the trace stream periodically. The PTM inserts additional timestamps in the trace stream at points where you are likely to find them useful. A system that implements timestamping must include a counter to provide the source of the timestamp values, and must broadcast the same value to all compatible trace sources in the system. Each trace source samples the timestamp value and inserts it as an absolute value in the trace stream.

This timestamping mechanism provides the following features:

- Correlation of multiple independent trace sources in a system, for example, multiple PTMs in a multi-processor environment.
- Simple analysis of code performance, with a coarse granularity.
- Faster searching of large trace buffers when looking for points in multiple trace streams where code was executed in close proximity.

The PTM inserts timestamps in the trace stream with optional cycle accuracy. It also inserts timestamps:

- when it traces an ISB operation
- when it traces a DMB or DSB operation, if this option is supported
- when an exception is traced
- when a return from exception is traced.

This enables close temporal correlation of code around spinlocks and other inter-processor communications.

Note

Bit [25] of the ETMCCER determines whether the PTM inserts timestamps for DMB and DSB operations, see *Configuration Code Extension Register, ETMCCER* on page 3-103.

2.5 Virtualization

From PFTv1.1 the PTM supports the Virtualization Extensions, that provide hardware support for virtual machine operation. A virtualized system involves:

- The hypervisor, that runs in a new Non-secure mode, called Hyp mode. The hypervisor is responsible for switching Guest Operating Systems (Guest OS).
- A number of Guest OSes, that run in the Non-secure privileged and non-privileged modes.

This model is based on providing virtualization support for Guest OSes that do not make use of the ARM Security Extensions other than, optionally, making calls to the secure side. See the *ARM Architecture Reference Manual* for more information on virtualization.

Virtualization includes a mechanism to distinguish between multiple virtual machines using a *Virtual Machine ID* (VMID). From PFTv1.1 you can use this VMID for tracing and matching. See *VMID Comparator Value Register, ETMVMIDCVR* on page 3-109.

Chapter 3
Program Trace Macrocell Programmers Model

This chapter describes the programmers model for a *Program Trace Macrocell* (PTM) that implements the *Program Flow Trace* (PFT) architecture. It contains the following sections:

3.1 About the PTM programmers model

A PTM is a macrocell that implements the PFT architecture. The programmers model for a PTM defines:

- How a PTM is a component in the CoreSight architecture. See *CoreSight support* on page 3-33.
- The operation and control of the TraceEnable mechanism, that controls when trace is generated. See *TraceEnable* on page 3-34.
- The behavior of the address comparators, that are one of the TraceEnable inputs, and how to define address comparators. See *Address comparators* on page 3-39.
- The PTM event resources, and how they can be used to define PTM events. See *Event resources and PTM events* on page 3-49.
- The behavior and control of the PTM counters, that can be used as PTM resources. See *PTM counters* on page 3-56.
- The behavior and control of the PTM sequencer, that can be used to provide PTM resources. See *The PTM sequencer* on page 3-57.
- The behavior and control of the Instrumentation resources. See *Instrumentation resources* on page 3-58.
- How to program and read the PTM registers, and how these registers are organized. See *About the PTM registers* on page 3-65.
- The PTM registers. See *PTM register descriptions* on page 3-75.
- The access controls on the PTM registers. See *About the access permissions for PTM registers* on page 3-138.
- How you can save the PTM state of the macrocell before a power-down, and restore the state when you restore power. See *Power-down support* on page 3-132.

3.2 CoreSight support

CoreSight is a system-level debug and trace solution that enables debug and trace components to share resources and work together. It defines a Visible Component Architecture that specifies requirements of all CoreSight components that are visible to development tools. The following sections describe the features of the Visible Component Architecture:

- *Programmers model requirements*
- *Topology detection requirements*.

See the *CoreSight Architecture Specification* for more information about the CoreSight Architecture.

3.2.1 Programmers model requirements

The CoreSight programmers model specifies that the registers of each CoreSight component are memory-mapped in a 4KB region. The top 256 bytes of this space, registers `0x3C0`-`0x3FF`, are reserved for management registers that must be present in any CoreSight component. The section *PTM register descriptions* on page 3-75 includes descriptions of these registers.

3.2.2 Topology detection requirements

A PTM must implement the logical interfaces that Table 3-1 lists. Each logical interface must implement registers to support topology detection. See the *CoreSight Architecture Specification*.

Table 3-1 Required PTM logical interfaces

Interface	Type	Number
Trace output	Master	1
Processor	Slave	1 + (value of bits [14:12] of ETMSCR
External output	Master	Value of bits [22:20] of ETMCCR
External input	Slave	Value of bits [19:17] of ETMCCR
Trigger output	Master	1

For more information about the registers referred to in Table 3-1, see:

- *Configuration Code Register, ETMCCR* on page 3-78
- *System Configuration Register, ETMSCR* on page 3-82.

In any CoreSight component, registers `0x380`-`0x3BF` are reserved for topology detection and integration registers, and use of these registers for this purpose is IMPLEMENTATION DEFINED.

3.3 TraceEnable

TraceEnable is the mechanism that controls the regions of code that are traced. It enables you to:

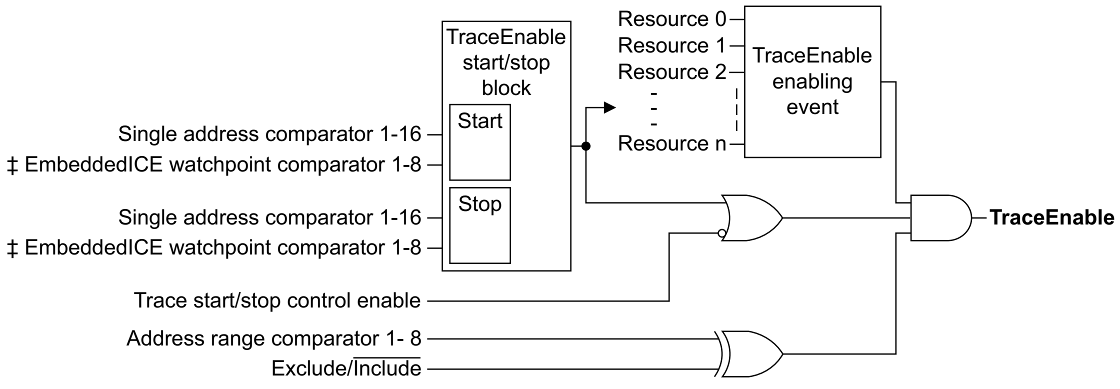
- start tracing when the processor executes a particular instruction
- stop tracing when the processor executes a particular instruction
- define a range of instructions to be included in the trace
- define a range of instructions to be excluded from the trace
- use other events in the system to control tracing.

About TraceEnable gives a general description of the TraceEnable mechanism, and *TraceEnable rules* on page 3-36 summarizes the operation of TraceEnable. The following sections then describe the TraceEnable components in more detail:

- *The TraceEnable start/stop block* on page 3-36
- *TraceEnable Include/exclude control* on page 3-37
- *Address comparators* on page 3-39
- *Context ID comparators* on page 3-44
- *EmbeddedICE watchpoint comparator inputs* on page 3-46
- *Event resources and PTM events* on page 3-49.

3.3.1 About TraceEnable

Figure 3-1 shows the TraceEnable components and the logical connections between them. The final TraceEnable signal is asserted to enable tracing, and deasserted to disable tracing.


TraceEnable start/stop block
Start
Stop
Single address comparator 1-16
‡ EmbeddedICE watchpoint comparator 1-8
Single address comparator 1-16
‡ EmbeddedICE watchpoint comparator 1-8
Trace start/stop control enable
Address range comparator 1- 8
Exclude/Include
Resource 0
Resource 1
Resource 2
Resource n
TraceEnable enabling event
TraceEnable

Notes: ‡ indicates optional components.
Most resource inputs are not shown, see text for details.
The number of address comparators, and the number of EmbeddedICE watchpoint comparators, are IMPLEMENTATION DEFINED. The diagram shows the maximum values.

Figure 3-1 The TraceEnable mechanism

Figure 3-1 shows that three controls are required to activate TraceEnable:

The TraceEnable enabling event

The PTM recognizes a number of resources, that you can use to define PTM events. See *Event resources and PTM events* on page 3-49. You must program the TraceEnable Event Register to define the event that enables TraceEnable activation.

The TraceEnable start/stop block

The TraceEnable start/stop block has separate start and stop controls. The inputs to each control are:

- selected single address comparators (SACs)
- selected Embedded ICE watchpoints, from the processor.

When an input to the start control is activated, the block turns on and asserts its output.

When an input to the stop control is activated, the block turns off and deasserts its output, if necessary.

The Trace start/stop control enable bit in the TraceEnable Control Register controls whether the output of the TraceEnable start/stop block must be asserted for TraceEnable to be asserted. However, the output of the start/stop block is always available as a PTM resource, regardless of the state of the Trace start/stop control enable bit.

You program the:

- TraceEnable Start/Stop Control Register to define the SACs that trigger the start control, and the SACs that trigger the stop control
- TraceEnable Start/Stop EmbeddedICE Control Register to define the EmbeddedICE watchpoints that trigger the start control, and the EmbeddedICE watchpoints that trigger the stop control.

For more information about the TraceEnable Start/Stop block see *The TraceEnable start/stop block* on page 3-36.

Address range comparators

You use address range comparators (ARCs) to define either the code regions that are included in the PTM trace, or the code regions that are excluded from the trace. The Include/exclude bit in the TraceEnable Control Register controls whether the ARCs are used to include code regions, or to exclude code regions.

For information about the address comparators, including how they are used to define ARCs, see *Address comparators* on page 3-39. For more information about how ARCs are used by the TraceEnable mechanism to include or exclude regions of code from the PTM trace see *TraceEnable Include/exclude control* on page 3-37.

This means that the PTM generates trace when all the following conditions are met:

- the TraceEnable enabling event is active
- either:
  - the TraceEnable start/stop block is in the on state
  - the Trace start/stop control enable bit indicates that the start/stop block is not controlling TraceEnable
- either:
  - the Include/exclude bit is set to include code, and the ARCs match for inclusion
  - the Include/exclude bit is set to exclude code, and the ARCs do not match for exclusion.

Figure 3-2 summarizes how you program the TraceEnable logic.

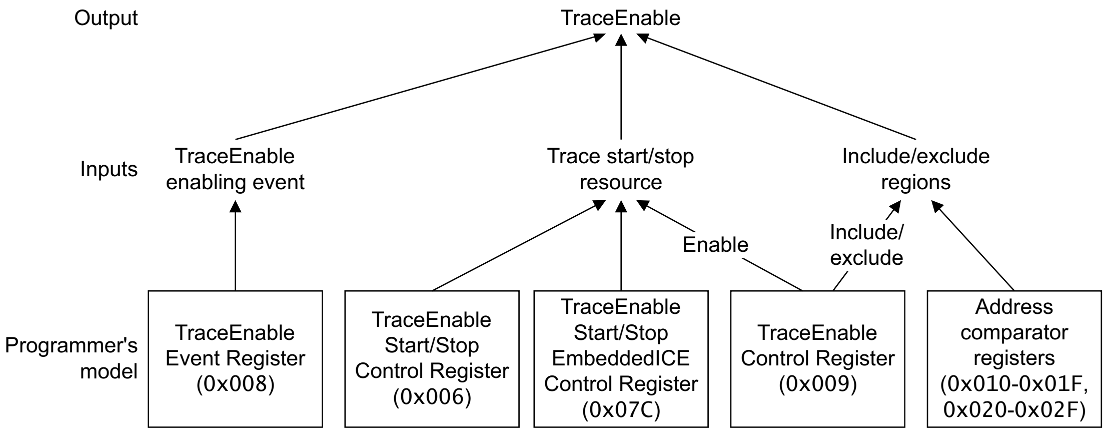

Output
TraceEnable
Inputs
TraceEnable enabling event
Trace start/stop resource
Include/exclude regions
Enable
Include/ exclude
Programmer's model
TraceEnable Event Register (0x008)
TraceEnable Start/Stop Control Register (0x006)
TraceEnable Start/Stop EmbeddedICE Control Register (0x07C)
TraceEnable Control Register (0x009)
Address comparator registers (0x010-0x01F, 0x020-0x02F)

Figure 3-2 Programming the TraceEnable logic

For descriptions of the registers used to program TraceEnable, see:

- *TraceEnable Start/Stop Control Register, ETMTSSCR* on page 3-83
- *TraceEnable Event Register, ETMTEEVR* on page 3-83

- *TraceEnable Control Register, ETMTECR1* on page 3-84
- *Address Comparator Value Registers, ETMACVRn* on page 3-86
- *Address Comparator Access Type Registers, ETMACTRn* on page 3-87
- *TraceEnable Start/Stop EmbeddedICE Control Register, ETMTESSEICR* on page 3-106.

Imprecise TraceEnable events

If TraceEnable is imprecise for any reason, any of the following might occur:

- tracing might not turn on in time to trace the required instruction
- tracing might not turn off in time to avoid tracing a specific instruction
- trace might be missing at the start of a trace region
- extra trace might appear at the end of a trace region.

Except for some IMPLEMENTATION DEFINED configurations, the **TraceEnable** signal is imprecise only if the resource that causes it to change is one of the following:

- a resource that is selected by the enabling event
- an address comparator linked to a Context ID comparator, and the Context ID changes.

See the appropriate PTM *Technical Reference Manual* for details of any IMPLEMENTATION DEFINED configurations for which the **TraceEnable** signal is not imprecise.

3.3.2 TraceEnable rules

TraceEnable can become active at any time. Usually, you use an address comparator to activate TraceEnable. This means the activation coincides with a waypoint, because address comparisons are performed only when a waypoint occurs. However, you can program the TraceEnable Event Register, so that any internal or external event activates TraceEnable, and the activation might not coincide with a waypoint.

Normally, when TraceEnable becomes active, tracing starts immediately, with the PTM generating an I-sync packet indicating the destination address of the last executed waypoint. The only cases where tracing does not start are:

- the PTM is disabled, because ProgBit is set to 1
- the processor is in Jazelle state
- the processor is in Debug state, and has Halting debug-mode enabled
- the processor is signaling that tracing is not permitted.

If an event activates TraceEnable between two waypoints, this change only takes effect at the next waypoint. When the processor executes the next waypoint, the PTM starts generating trace.

Normally, TraceEnable can become inactive only at a waypoint. This guarantees that the PTM traces the block of instructions from one waypoint to the next. The only exceptions to this, where the PTM stops tracing immediately, are:

- You set ProgBit to 1, disabling the PTM.
- You set the OS lock.
- The PTM is powered down. Usually this happens because the processor executes a `WFI` or `WFE` instruction.

A general rule for using TraceEnable is that, if two waypoints define the start and end of a block of instructions and you have to trace any instructions in that block, then you must trace the whole block.

3.3.3 The TraceEnable start/stop block

The TraceEnable start/stop block generates an enable input for the TraceEnable control. The block is turned on by a start input, and remains on until it receives a stop input.

The start inputs for the TraceEnable start/stop block are:

- one or more SACs, selected by the TraceEnable Start/Stop Control Register
- if supported, one or more EmbeddedICE watchpoint comparator inputs, selected by the TraceEnable Start/Stop EmbeddedICE Control Register.

Similarly, the stop inputs for the block are one or more SACs, and possibly one or more EmbeddedICE watchpoint comparator inputs, as defined in the same registers.

The SACs are defined by the address comparators. See *Address comparators* on page 3-39. When an SAC defines a stop input to the start/stop block, the block remains on for that instruction and is turned off at the next waypoint after the instruction.

It is IMPLEMENTATION DEFINED whether a PTM implements EmbeddedICE watchpoint comparator inputs, but if it does:

- it implements between one and eight watchpoint comparator inputs
- signals from the processor drive the comparator inputs, typically the **RANGEOUT** signals
- it is IMPLEMENTATION DEFINED whether the comparator inputs are inputs to the start/stop block.

For more information about the EmbeddedICE watchpoint comparator inputs, including checking whether and how they are implemented, see *EmbeddedICE watchpoint comparator inputs* on page 3-46.

The instructions between one waypoint and the next waypoint make up a block of instructions. The general operation of the start/stop block is:

- if a block of instructions includes a trace start/stop block start point then trace that block of instructions
- if a block of instructions includes a trace start/stop block stop point then:
  - — trace that block of instructions
  - — do not trace the next block of instructions, unless it includes a new start point.

Note

The way that the PTM traces the block of instructions that includes the trace start/stop block stop point makes it easy to use the block to control tracing a function, where:

- an SAC defines the start point, corresponding to a single entry point to the function
- multiple SACs define multiple stop points, each corresponding to a different function exit point.

If a start point and a stop point occur in the same block of instructions, then the start/stop block is:

- on for that entire block of instructions
- off after the last instruction in the block.

This behavior is the same regardless of whether the start point comes before the stop point, or after the stop point. This means that, when programming the SACs that are selected for TraceEnable start/stop block control, you must ensure that the stop SAC for one section of code that you want to trace is not in the same block of instructions as the start SAC for the next section of code you want to trace. If you fail to do this, the second section of code is not traced.

Note

- Multiple start points in a block of instructions are treated as a single start point.
- Multiple stop points in a block of instructions are treated as a single stop point.

3.3.4 TraceEnable Include/exclude control

The Include/exclude bit in the TraceEnable Control Register controls how the TraceEnable mechanism uses ARC. It selects one of two modes of operation:

Include In include mode, you select ARCs to define regions of code to include in the trace. When an ARC matches, all instructions in the address range are traced. Because PTM trace only explicitly traces waypoints, this means that the PTM traces all blocks of instructions that include any instruction in the address range.

Exclude In exclude mode, you select ARCs to define regions of code to exclude from the trace. The ARC is tested at each waypoint. At that point, the PTM considers the block of instructions between the previous waypoint and the current waypoint:

- if the ARC excludes all of that block of instructions from tracing then the PTM does not generate any trace
- otherwise, the PTM traces that block of instructions.

This means that, in exclude mode, you might find some instructions traced unexpectedly. A block of instructions between two waypoints is excluded from tracing only if the excluded ARC covers the entire block.

You can select multiple ARCs for Include/exclude control, to include or exclude multiple blocks of code. However, you cannot mix include blocks and exclude blocks in a single trace run.

3.4 Address comparators

It is IMPLEMENTATION DEFINED whether a PTM implements address comparators. If a PTM does implement address comparators then:

- it must implement an even number of comparators
- the maximum number of comparators is sixteen (eight pairs)
- the address comparators are numbered from one.

You can use address comparators:

- Individually, as *single address comparators* (SACs).
- In pairs, as *address range comparators* (ARCs). In this case, two adjacent address comparators form the ARC, so you can use address comparators 1 and 2 to define the first ARC.

You can use the same address comparator simultaneously as an SAC and as half of an ARC.

An SAC matches when an instruction at the specified address is committed for execution, regardless of whether the instruction passes its condition code test.

An ARC matches when any instruction in the specified range is committed for execution, regardless of whether the instruction passes its condition code test.

The following sections give more information about the address comparators:

- *General behavior of address comparators*
- *Single address comparators (SACs)* on page 3-41
- *Address range comparators (ARCs)* on page 3-42.

3.4.1 General behavior of address comparators

To define an address comparator:

- Program an Address Comparator Value Register with the address to be matched. See *Address Comparator Value Registers, ETMACVRn* on page 3-86.
- Program the corresponding Address Comparator Access Type Register with additional information about the required comparison. See *Address Comparator Access Type Registers, ETMACTRn* on page 3-87. If you want your address comparison to match only if the Context ID also matches you must:
  - Program the Address Comparator Access Type register to define the Context ID comparator to use.
  - Program the appropriate Context ID Comparator Value Register, and check the programming of the Context ID Comparator Mask Register. See *About the Context ID comparator registers* on page 3-98.

In addition, to define an ARC:

- Program the lower-numbered address comparator with the low address of the required range.
- Program the higher-numbered address comparator with the high address of the required range.
- Program both Address Comparator Access Type Registers with the same values.

The PTM keeps a record of the destination address of the last waypoint, and treats this waypoint destination address as the start address of a block of instructions. Between two waypoints the program flow is continuous. The next waypoint marks the end of the block of instructions, and at that point the PTM can check whether any address comparators matched any address in that block of instructions.

For example, consider the following sequence of code execution:

1. The instruction at `0x1000` is a direct branch to `0x2000`.
2. The program executes from `0x2000` to `0x2100`.
3. The instruction at `0x2100` is a direct branch to `0x3000`.

If an address comparator matches on address `0x2050`, then when the program flow reaches the waypoint at `0x2100`, the comparators must determine whether the instruction at address `0x2050` executed. Because the PTM can determine that the program executed from `0x2000` to `0x2100`, it also determines that the comparator matched.

Because the program executed from `0x2000` to `0x2100`, there was no change in security state or instruction set state in this block of instructions, and any change in state does not take effect until after execution of the block is complete.

Note

When processing a new waypoint, the PTM defines the block of instructions executed up to the waypoint from:

- The target address of the previous waypoint. This is the start address of the block of instructions.
- The address of either:
  - the current waypoint, if no exception has occurred
  - the last instruction before the exception.

  This is the address of the last instruction successfully executed, and is the end address of the block of instructions.

An instruction block end address might not be the exact address of the waypoint instruction that defines the end of the block, but:

- it must not be less than the address of the last successfully executed instruction
- it must be less than the address of the instruction that follows the last successfully executed instruction.

This means that a PTM implementation has some flexibility in the addresses used for comparison. See Table 3-2 for more information.

Table 3-2 Permitted instruction block end addresses

Instruction type[a]	Address: Instruction[b]	Address: Block end	Permitted?
ARM	`0x1000`	<`0x1000`	No, this is less than the address of the last instruction, `0x1000`.
	`0x1000`	`0x1000`	Yes
	`0x1000`	`0x1001`	Yes
	`0x1000`	`0x1002`	Yes
	`0x1000`	`0x1003`	Yes
	`0x1000`	≥`0x1004`	No, this is not less than the address of the next instruction, `0x1004`.
32-bit Thumb	`0x1000`	<`0x1000`	No, this is less than the address of the last instruction, `0x1000`.
	`0x1000`	`0x1000`	Yes
	`0x1000`	`0x1001`	Yes
	`0x1000`	`0x1002`	Yes
	`0x1000`	`0x1003`	Yes
	`0x1000`	≥`0x1004`	No, this is not less than the address of the next instruction, `0x1004`.
16-bit Thumb	`0x1000`	<`0x1000`	No, this is less than the address of the last instruction, `0x1000`.
	`0x1000`	`0x1000`	Yes
	`0x1000`	`0x1001`	Yes
	`0x1000`	≥`0x1002`	No, this is not less than the address of the next instruction, `0x1002`.

a. Instruction set, and instruction length for Thumb instructions.

b. Address of the last instruction executed, see text.

Note

If instruction execution wraps round the top of the memory space then processor operation is UNPREDICTABLE. If the current waypoint address is less than the previous waypoint address, and no branch or exception has occurred, then instruction execution has wrapped in this way. In this situation, the behavior of address range comparators is IMPLEMENTATION SPECIFIC.

If the processor does not execute any instructions between two waypoints then the comparators are not affected by the second waypoint. Examples of where no instructions are executed between two waypoints are:

- if an exception follows immediately after another exception
- if a processor reset occurs.

Terms used to describe address comparator behavior

The PTM tests all the address comparators whenever it processes a waypoint. The following terms are used when describing the behavior of SACs and ARCs:

Current waypoint	The waypoint currently being executed, for which the PTM is testing the comparators.
Previous waypoint	The last waypoint executed by the processor, before the current waypoint.
Instruction block	The block of instructions executed between the previous waypoint and the current waypoint. This block is defined by two addresses, START and END, defined as: **START** The destination address of the previous waypoint. **END** The address of the current waypoint.
CompAddr	When testing a SAC, the address programmed in the Address Comparator Value Register.
CompAddrLow	When testing an ARC, the address programmed in the lower-numbered Address Comparator Value Register, that defines the low address of the address range. For example, if you are using Address Comparator Value Registers 1 and 2 to define an ARC, **CompAddrLow** is the address held in Address Comparator Value Register 1.
CompAddrHigh	When testing an ARC, the address programmed in the higher-numbered Address Comparator Value Register, that defines the high address of the address range. For example, if you are using Address Comparator Value Registers 1 and 2 to define an ARC, **CompAddrHigh** is the address held in Address Comparator Value Register 2.

3.4.2 Single address comparators (SACs)

For a single address comparator, the address comparison is successful if:

(**CompAddr**≥**START**)AND(**CompAddr**≤**END**)

See *Terms used to describe address comparator behavior* for descriptions of the terms used in this definition.

The SAC matches only if the address comparison is successful and the conditions defined in the corresponding Address Comparator Access Type Register are met.

The PTM holds the result of a successful SAC match only for one cycle, that corresponds to the current waypoint. The PTM event resource corresponding to the SAC is TRUE for this one cycle only.

At each waypoint, the PTM always tests each Address Comparator Value Register as a SAC. To avoid unexpected trace output, you must make sure you use only the SACs that you have programmed.

3.4.3 Address range comparators (ARCs)

The PTM has two modes of operation for ARCs:

Include mode

In this mode, a match occurs when the instruction block overlaps the address range defined in the ARC. This means that the ARC matches if the processor executed an instruction in the range defined by the ARC. The PTM uses this mode for:

- TraceEnable include control
- the address range comparator events.

In include mode, the address comparator matches if:

$$(\mathbf{CompAddrLow} \leq \mathbf{END})\text{AND}(\mathbf{CompAddrHigh} > \mathbf{START})$$

Exclude mode

In this mode, a match occurs if the instruction block is completely within the address range defined in the ARC. This means that the ARC matches only if all of the instructions in the instruction block are in the exclude range defined by the ARC. The PTM uses this mode for:

- TraceEnable exclude control.

In exclude mode, the address comparator matches if:

$$(\mathbf{CompAddrLow} \leq \mathbf{START})\text{AND}(\mathbf{CompAddrHigh} > \mathbf{END})$$

See *Terms used to describe address comparator behavior* on page 3-41 for descriptions of the terms used in this definition.

The ARC matches only if the address comparison is successful and the conditions defined in the corresponding Address Comparator Access Type Register are met.

The PTM holds the result of a successful ARC match until it processes another waypoint. The PTM event resource corresponding to the ARC is TRUE for this time.

On a processor reset, the ARCs maintain their values.

At each waypoint, the PTM always tests every pair of Address Comparator Value Registers as an ARC. For example, if it implements six Address Comparator Value Registers it always tests:

- Address Comparator Value Registers 1 and 2 as ARC 1
- Address Comparator Value Registers 3 and 4 as ARC 2
- Address Comparator Value Registers 5 and 6 as ARC 3.

To avoid unexpected trace output, you must make sure you:

- program the TraceEnable Control Register to select, for trace include or exclude, only the ARCs you have programmed
- use only the ARC event resources that correspond to ARCs that you have programmed.

For a PTM that implements the maximum of 16 Address Comparator Value Registers, Table 3-3 lists the Address Comparator Value Registers that define each ARC.

Table 3-3 Definition of ARCs by Address Comparator Value Registers

ARC	Address Comparator Value Registers (ETMACVRs)	
	Range start address	Range end address
1	ETMACVR1	ETMACVR2
2	ETMACVR3	ETMACVR4
3	ETMACVR5	ETMACVR6
4	ETMACVR7	ETMACVR8
5	ETMACVR9	ETMACVR10

Table 3-3 Definition of ARCs by Address Comparator Value Registers (continued)

ARC	Address Comparator Value Registers (ETMACVRs)	
	Range start address	**Range end address**
6	ETMACVR11	ETMACVR12
7	ETMACVR13	ETMACVR14
8	ETMACVR15	ETMACVR16

3.5 Context ID comparators

It is IMPLEMENTATION DEFINED whether a PTM implements Context ID comparators. If it does, it implements:

- between one and three Context ID comparators
- a Context ID Comparator Value Register for each comparator
- a single Context ID Comparator Mask Register.

Each Context ID Comparator Value Register can hold a Context ID value, for comparison with the current Context ID. The Context ID Comparator Mask can hold a mask value, that is used to mask all Context ID comparisons. If you program the Context ID Comparator Mask Register to zero then no mask is applied to the Context ID comparisons. For more information, see *About the Context ID comparator registers* on page 3-98.

The PTM uses the Context ID comparators in two ways:

- You can make any address comparison conditional on the Context ID also matching, by programming the Address Comparator Access Type register with the number of the Context ID comparator to use.
- Each Context ID comparator provides an event resource. At each waypoint, the PTM compares the current Context ID with each of the Context ID comparators. If there is a match then the corresponding event resource is TRUE, otherwise it is FALSE. These values are held until the PTM processes the next waypoint.

3.6 Virtual Machine ID comparator

It is IMPLEMENTATION DEFINED whether a PTM implements the *Virtual Machine ID* (VMID) comparator.

The VMID Comparator Value Register can hold a VMID value, for comparison with the current VMID. For more information, see *VMID Comparator Value Register, ETMVMIDCVR* on page 3-109.

The PTM uses the VMID comparator in two ways:

- You can make any address comparison conditional on the VMID also matching, by programming the Address Comparator Access Type register to use the VMID comparator.
- The VMID comparator provides an event resource. At each waypoint, the PTM compares the current Virtual Machine ID with the VMID comparator. If there is a match then the corresponding event resource is TRUE, otherwise it is FALSE. These values are held until the PTM processes the next waypoint.

3.7 EmbeddedICE watchpoint comparator inputs

It is IMPLEMENTATION DEFINED whether a PTM implements any Embedded ICE watchpoint comparator inputs. If it does:

- it implements between one and eight EmbeddedICE watchpoint comparator inputs
- it might implement a single EmbeddedICE Watchpoint Behavior Control Register.

To find the number of EmbeddedICE comparator inputs implemented, read bits [19:16] of the Configuration Code Extension Register. See *Configuration Code Extension Register, ETMCCER* on page 3-103. If this field reads-as-zero then the implementation does not include any EmbeddedICE comparator inputs.

When a PTM implements one or more EmbeddedICE comparator inputs:

- Each comparator input connects to a signal from the processor, that is asserted when the corresponding EmbeddedICE watchpoint is triggered. Typically, these are the **RANGEOUT** signals from the processor.
- It is IMPLEMENTATION DEFINED whether the EmbeddedICE comparator inputs are inputs to the TraceEnable start/stop block. To check whether they are, read bit [20] of the Configuration Code Extension Register. This bit is 1 if the EmbeddedICE comparator inputs are inputs to the start/stop block.
- Each Embedded ICE comparator input provides a PTM trace resource.
- The PTM might implement the Embedded ICE Behavior Control Register. See *EmbeddedICE Behavior Control Register, ETMEIBCR* on page 3-106. To check whether this register is implemented, read bit [21] of the Configuration Code Extension Register. This bit is 1 if the EmbeddedICE Behavior Control Register is implemented. In this case, you can use this register to configure the behavior of each of the Embedded ICE comparator inputs.

The following sections give additional information about the behavior of the EmbeddedICE watchpoint comparator inputs:

- *EmbeddedICE watchpoint comparator input behavior*
- *Default behavior of EmbeddedICE watchpoint comparator inputs* on page 3-47
- *Pulse and latch behavior of EmbeddedICE watchpoint comparator inputs* on page 3-47
- *Examples of using EmbeddedICE watchpoint comparator inputs* on page 3-47.

3.7.1 EmbeddedICE watchpoint comparator input behavior

If you can control the behavior of the EmbeddedICE watchpoint comparator inputs, and specify different behavior of these inputs in different contexts, these signals are more useful for controlling tracing.

For example, if you are controlling the TraceEnable event, you want the controlling input held between comparisons, so that TraceEnable is active through a range of addresses.

In contrast, if you want to use an EmbeddedICE watchpoint comparator input as an input to the trace start/stop block, it is preferable if the input is pulsed for a single cycle. This avoids the possibility, for example, that a stop signal might be missed because a start signal from an EmbeddedICE watchpoint comparator input is still asserted.

To take account of these different requirements, if the EmbeddedICE Behavior Control Register is implemented, a debugger can program this register to select the default behavior of each EmbeddedICE watchpoint comparator input as either pulsed or latched, depending on the required use of each input. For more information see *EmbeddedICE Behavior Control Register, ETMEIBCR* on page 3-106.

If the EmbeddedICE Behavior Control Register is not implemented then the behavior of the EmbeddedICE watchpoint comparator inputs differs for different resources. See *Default behavior of EmbeddedICE watchpoint comparator inputs* on page 3-47.

3.7.2 Default behavior of EmbeddedICE watchpoint comparator inputs

When the EmbeddedICE Behavior Control Register is not implemented, Table 3-4 defines the behavior of any EmbeddedICE watchpoint comparator input connection to the different PTM resources.

Table 3-4 Default behavior of EmbeddedICE watchpoint comparator inputs

Resource driven by EmbeddedICE watchpoint comparator input	Behavior of input
Trigger event	Pulse
TraceEnable event	Latch
Trace start/stop block input	Pulse
Counter enable or reload event	Pulse
Sequencer state change event	Pulse
External output	Pulse

Note

A debugger can read bit [21] of the Configuration Code Extension Register to find whether the EmbeddedICE Behavior Control Register is implemented:

- If the register is not implemented the debugger can assume the EmbeddedICE watchpoint comparator inputs behave as Table 3-4 shows.
- If the register is implemented the debugger must write to the register to configure the behavior of each EmbeddedICE watchpoint comparator input, as required for the use it is making of the input.

3.7.3 Pulse and latch behavior of EmbeddedICE watchpoint comparator inputs

Correct implementation of configurable EmbeddedICE watchpoint comparator inputs requires control signals that indicate the sampling point for each input. For each input, the input is sampled at the appropriate point indicated by the control signals. The sampling depends on the value of the corresponding bit in the EmbeddedICE Behavior Control Register. When this bit is:

0	Pulse operation. When the signal is sampled HIGH, the EmbeddedICE watchpoint comparator input is asserted for a single cycle from the point where it is sampled.
1	Latch operation. The EmbeddedICE watchpoint comparator input is latched to the sampled value, and held in that state until the cycle before the next sample point.

3.7.4 Examples of using EmbeddedICE watchpoint comparator inputs

These are examples of how you could use the EmbeddedICE watchpoint comparator inputs, and how you must configure the appropriate input for each use:

- To use an EmbeddedICE watchpoint comparator input to count the number of instructions executed at a particular address, the PTM must *pulse* the EmbeddedICE watchpoint comparator input for one cycle each time the EmbeddedICE logic matches the required address. Therefore, you must set the appropriate bit of the EmbeddedICE Behavior Control Register to 0, to indicate that the EmbeddedICE watchpoint comparator input must be pulsed.
- To use an EmbeddedICE watchpoint comparator input to count the number of cycles spent in a particular range of instruction addresses, the PTM must hold the EmbeddedICE watchpoint comparator input HIGH from the first to the last clock cycle for which the EmbeddedICE logic compares an instruction address in the required range. Therefore, you must set the appropriate bit of the EmbeddedICE Behavior Control Register to 1, to indicate that the EmbeddedICE watchpoint comparator input must be latched.

- To use an EmbeddedICE watchpoint comparator input to include a particular range of trace addresses using **TraceEnable**, the EmbeddedICE watchpoint comparator input must latch between each comparison. Therefore, you must set the appropriate bit of the EmbeddedICE Behavior Control Register to 1, to indicate that the EmbeddedICE watchpoint comparator input must be latched.

For details of configuring the EmbeddedICE Behavior Control Register, see *EmbeddedICE Behavior Control Register, ETMEIBCR* on page 3-106.

3.8 Event resources and PTM events

A PTM includes a number of *event resources*. At any time, each event resource is either active or inactive. An active event resource generates a logical TRUE signal, and an inactive resource generates a logical FALSE signal.

A PTM also includes a number of events, each of the is defined by an event register. You can program each event register to define the corresponding event as the result of a logical operation involving one or two event resources. Therefore, at any time, each PTM event is either TRUE or FALSE.

The simplest PTM event definition is one where you program the event to equal a PTM event resource.

The following sections give more information about PTM event resources and events:

- *The PTM event resources*
- *Example PTM resource configuration* on page 3-51
- *Defining a PTM event* on page 3-52
- *Summary of the PTM events* on page 3-54.

3.8.1 The PTM event resources

A PTM event is identified by:

- a 3-bit resource type value
- a 4-bit index value, within the given resource type.

The 7-bit value obtained by concatenating a resource type and an associated index value is the resource number.

Table 3-5 shows the defined PTM resources. All combinations of resource type and index values not shown in the table are reserved.

Table 3-5 Event resource definitions

Resource type	Index values[a]	Resource description	Number of resources of this type
b000	0-15	Single address comparators 1-16	Depends on number of SACs, 0-16
b001	0-7	Address range comparators 1-8	Half the number of SACs, so 0-8
	8-11	Instrumentation resources 1-4	Depends on number of Instrumentation resources, 0-4
b010	0-7	EmbeddedICE watchpoint comparator inputs 1-8	Depends on number of EmbeddedICE watchpoint comparator inputs, 0-8
b100	0-3	Counter 1-4 at zero	Depends on number of counters, 0-4
b101	0-2	Sequencer in state 1-3	Three if sequencer implemented, otherwise none
	8-10	Context ID comparators 1-3	Depends on number of Context ID comparators, 0-3
	11	VMID Comparator[b]	0 or 1
	15	TraceEnable start/stop resource	0 or 1
b110	0-3	External inputs 1-4	Depends on number of external inputs, 0-4
	8-11	Extended external input selectors 1-4	Depends on number of extended external input selectors, 0-4
	13	Processor in Non-secure state	One, only if the processor implements the Security Extensions[c]
	14	Trace prohibited	One, always present
	15	Always TRUE	One, always present

a. Index ranges show the maximum range. The actual range depends on the number of resources of that type implemented by the PTM.

b. From PFTv1.1, and only for implementations that support the Virtualization Extension.

c. This resource is not implemented if the processor does not implement the Security Extensions.

Note

The PTM uses the same event type and index value encodings as ETMv3.4. However, it does not support all of the event resources that are defined in the ETM specification.

How each event resource is asserted depends on the PTM feature that generates the resource. For example:

- A SAC matches only for the single cycle when the comparison is made. Therefore, an SAC resource is TRUE only for that cycle.
- An ARC matches until the next waypoint. Therefore, an ARC resource is TRUE until the next waypoint.
- The behavior of an EmbeddedICE comparator resource depends on how you have programmed the EmbeddedICE Behavior Control Register for the corresponding comparator. See *EmbeddedICE Behavior Control Register, ETMEIBCR* on page 3-106.
- An external input resource is TRUE when the corresponding input signal is asserted HIGH.

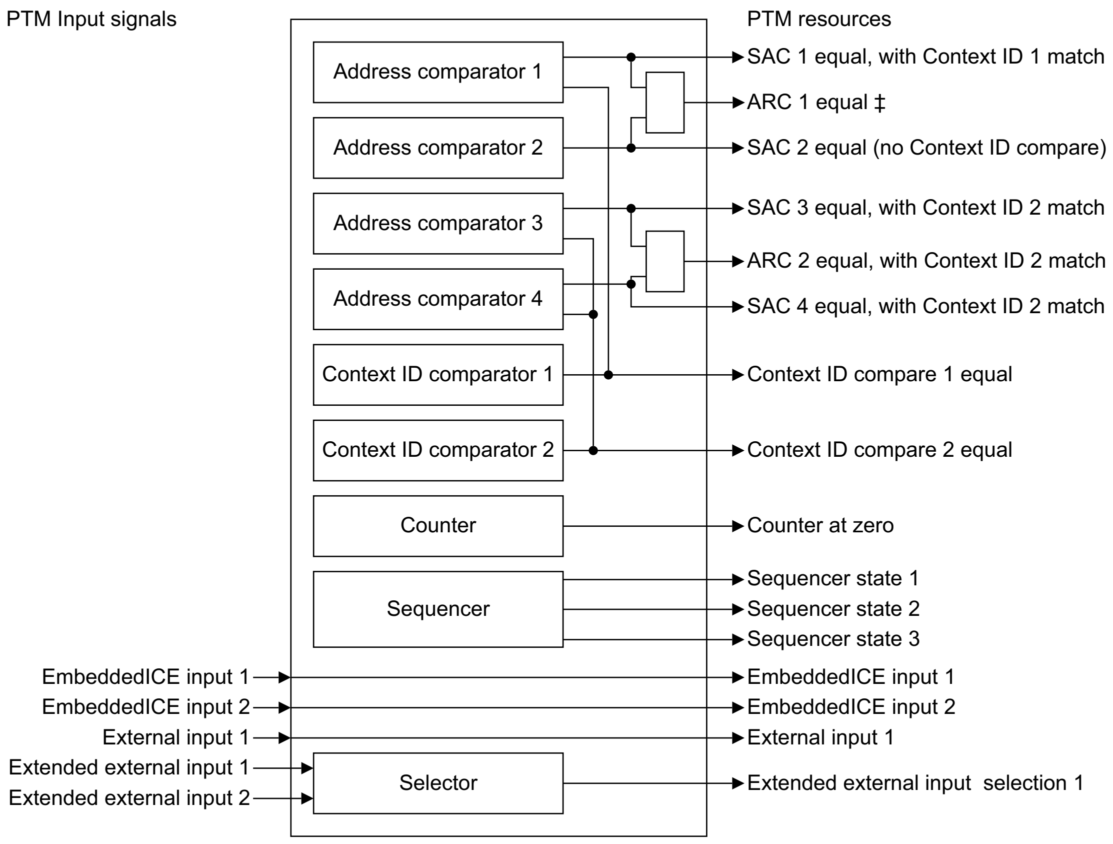
Some event resources are outputs from functional blocks of the PTM, such as the address comparators. Others act only as event resources. Table 3-6 shows where you can find more information about each of the event resources.

Table 3-6 Additional information about the PTM event resources

Event resource	For more information, see
Single address comparators	*Single address comparators (SACs)* on page 3-41
Address range comparators	*Address range comparators (ARCs)* on page 3-42
Instrumentation resources	*Instrumentation resources* on page 3-58
EmbeddedICE comparator inputs	*EmbeddedICE watchpoint comparator inputs* on page 3-46
Counter at zero	*PTM counters* on page 3-56
Sequencer in state 1-3	*The PTM sequencer* on page 3-57
Context ID comparators	*Context ID comparators* on page 3-44
VMID Comparator	*Virtual Machine ID comparator* on page 3-45
TraceEnable start/stop resource	*The TraceEnable start/stop block* on page 3-36
External inputs	*External inputs* on page 3-61
Extended external inputs	*Extended external inputs* on page 3-61
Processor in Non-secure state	*Non-secure state resource* on page 3-62
Trace prohibited	*Trace prohibited resource* on page 3-62
Always TRUE	*Hard-wired TRUE resource* on page 3-62

3.8.2 Example PTM resource configuration

Figure 3-3 shows an example PTM resource configuration. It also shows how the outputs from some blocks of the PTM can be combined to generate particular resources.



‡ The ARC 1 resource output is not valid, because the Context ID comparison settings for Address comparators 1 and 2 are different. With this resource configuration, do not use this resource.

Figure 3-3 Example resource configuration

You can filter address comparator matches by Context ID and processor state. From PFTv1.1:

- You can also filter based on processor mode.
- For implementations that support virtualization, you can also filter based on the VMID.

Depending on the value of the Address Comparator Access Type Registers, any of these values, alone or in combination, can be used to filter an address comparator match. See *Address Comparator Access Type Registers, ETMACTRn* on page 3-87.

Figure 3-4 on page 3-52 shows the inputs available to generate resource signals with PFTv1.1.

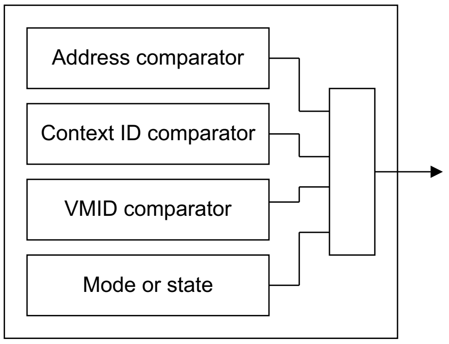


Figure 3-4 Address comparator match filtering from PFTv1.1

3.8.3 Defining a PTM event

You define a PTM event as a boolean operation that is applied to one or two of the event resources. If A and B are two event resources, Table 3-7 shows the supported operations and their encodings.

Table 3-7 Boolean operations for defining PTM events

Encoding	Function
b000	A
b001	NOT(A)
b010	A AND B
b011	NOT(A) AND B
b100	NOT(A) AND NOT(B)
b101	A OR B
b110	NOT(A) OR B
b111	NOT(A) OR NOT(B)

To define a PTM event, you program 17 bits of an event register with the encodings for:

- The required operation, as Table 3-7 shows.
- The first event resource, A.
- The second event resource, B, if required. If only one resource is required this field is ignored.

You identify PTM resources by their resource numbers. To form a resource number, concatenate the appropriate resource type and index value from Table 3-5 on page 3-49. For example, Context ID comparator 2 is event type b101, with index value b1001 (9), so its resource number is b1011001.

Table 3-8 shows the bitfields used to define an event.

Table 3-8 Defining a PTM event

Bits	Description
[16:14]	Boolean function
[13:7]	Resource B
[6:0]	Resource A

Figure 3-5 shows how an event is defined, including how the resource numbers are split into the resource type and index values.

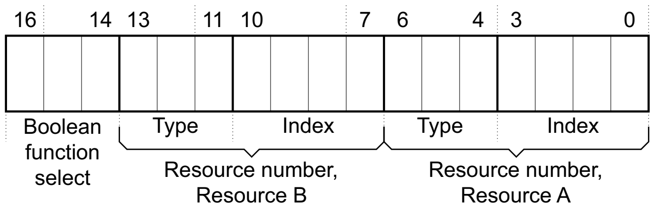


Figure 3-5 Defining a PTM event

Note

To permanently enable or disable an event, you must specify:

- Resource A as the hard-wired resource, resource type b110 with index b1111 (15)
- the boolean function as either:
  - A, to enable the event
  - Not (A), to disable the event.

To define a PTM event you program an event register with the information in Table 3-8 on page 3-52 and Figure 3-5. Write this information to bits [16:0] of the register. See *Summary of the PTM events* on page 3-54 for a list of the event registers.

Read values of event registers

A PTM contains a number of event registers which are used to select event resources for controlling tracing or triggering. These registers include the Trigger Event Register, see *Trigger Event Register, ETMTRIGGER* on page 3-80 and the TraceEnable Event Register, see *TraceEnable Event Register, ETMTEEVR* on page 3-83. These registers are read/write if bit [11] of the ETMCCER is set.

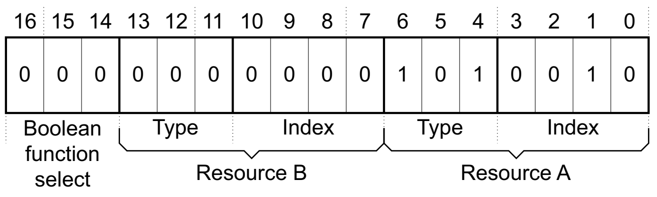
From PFTv1.1, if an invalid resource is programmed, such as one which is architecturally Reserved or a resource which is not supported by the specific implementation, the read value returned is UNKNOWN and the behavior of the event is UNPREDICTABLE.

This enables implementations with a small number of event resources to minimize the number of register bits used to store the event resource selection.

Examples of event programming

Example 3-1 shows how to program an event to occur when the sequencer reaches state 3.

Example 3-1 Encoding an event based on a single resource

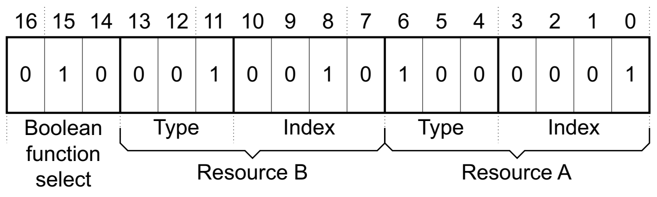


- Set bits [16:14] to select the Boolean A function, b000.
- Define Resource A as sequencer state 3:
  - resource type b101, for the sequencer
  - index value b0010, for state 3.

Because the selected Boolean function does not use Resource B the value of the Resource B register field is ignored.

Example 3-2 shows how to program an event to occur when counter 2 reaches zero while the processor is executing instructions in the in address range defined by ARC 3.

Example 3-2 Encoding an event based on a combination of resources



- Set bits [16:14] to select the Boolean A AND B function, b010.
- Define Resource B as ARC 3:
  - resource type b001, for an ARC
  - index value b0010, for ARC 3.
- Define Resource A as counter 2:
  - resource type b100, for a counter
  - index value b0001, for counter 2.

3.8.4 Summary of the PTM events

Table 3-9 shows all of the registers that you can program to define PTM events. In some cases, it is IMPLEMENTATION DEFINED whether a register is implemented.

Table 3-9 The PTM event registers

Event	Register	
	Location	**Description, see**
Trigger	`0x02`	*Trigger Event Register, ETMTRIGGER* on page 3-80
TraceEnable	`0x08`	*TraceEnable Event Register, ETMTEEVR* on page 3-83
Counter 1 enable	`0x54`	*Counter Enable Event Registers, ETMCNTENRn* on page 3-92
Counter 2 enable	`0x55`	
Counter 3 enable	`0x56`	
Counter 4 enable	`0x57`	
Counter 1 reload	`0x58`	*Counter Reload Event Registers, ETMCNTRLDEVRn* on page 3-93
Counter 2 reload	`0x59`	
Counter 3 reload	`0x5A`	
Counter 4 reload	`0x5B`	

Table 3-9 The PTM event registers (continued)

Event	Register Location	Description, see
Sequencer transition, state 1 to state 2	`0x60`	*Sequencer State Transition Event Registers, ETMSQabEVR* on page 3-96
Sequencer transition, state 2 to state 1	`0x61`	
Sequencer transition, state 2 to state 3	`0x62`	
Sequencer transition, state 3 to state 1	`0x63`	
Sequencer transition, state 3 to state 2	`0x64`	
Sequencer transition, state 1 to state 3	`0x65`	
External output 1 event	`0x68`	*External Output Event Registers, ETMEXTOUTEVRn* on page 3-97
External output 2 event	`0x69`	
External output 3 event	`0x6A`	
External output 4 event	`0x6B`	
Timestamp event	`0x7E`	*Timestamp Event Register, ETMTSEVR* on page 3-107

3.9 PTM counters

A PTM can include one or more counters, that you control using PTM events, and configure using the counter registers. For a PTM implementation, the number of counters is IMPLEMENTATION DEFINED. A PTM implementation:

- is not required to provide any counters
- can provide as many as four counters.

Each implemented counter is controlled by two events, and has two other associated registers:

The counter enable event

While the counter enable event is TRUE, the counter is enabled and counts down.

Each counter is 16-bit, so it can count from 1 to 65535. While a counter is enabled, it is clocked by the system clock, even on cycles when the processor is stalled, and decrements on each clock cycle.

Use the Counter Enable Event Register to define the event that enables the counter. See *Counter Enable Event Registers, ETMCNTENRn* on page 3-92.

The counter reload event

If the counter reload event occurs, the counter is reloaded from the Counter Reload Value Register. See *Counter Reload Value Registers, ETMCNTRLDVRn* on page 3-91.

The counter reload event takes priority over the counter enable event.

Use the Counter Reload Event Register to define the event that causes a counter reload. See *Counter Reload Event Registers, ETMCNTRLDEVRn* on page 3-93.

The Counter Reload Value Register

This register holds the value that is loaded into the counter when the Counter Reload Event is TRUE. See *Counter Reload Value Registers, ETMCNTRLDVRn* on page 3-91.

The Counter Value Register

You can:

- read this register at any time to find the current value of the counter
- write a new value to this register when you are programming the PTM.

For more information see *Counter Value Registers, ETMCNTVRn* on page 3-94.

When a counter reaches zero it remains at zero and the associated resource becomes active, and remains active until the counter is reloaded.

3.9.1 Use of PTM counters

Possible uses of the PTM counters include:

- Periodic sampling of the PC, using a counter to define the sampling period. Each time the counter reaches zero you enable tracing briefly to output the last waypoint target address. This gives you a low bandwidth PC sampler, that you can use as a simple profiling tool. The cost of each sample point is an I-sync packet and a single atom header or branch address packet, which is between 7 and 11 bytes, or between 8 and 16 bytes if you have enabled cycle-accurate tracing.

- Extending the performance counters provided by the processor.

- Counting the number of occurrences of a single instruction, for example by programming a SAC to match on the address of the required instruction, and programming a counter to decrement when that SAC matches.

3.10 The PTM sequencer

The PTM includes a three-state sequencer. To create a multiple-stage trigger scheme, use a trigger event based on a sequencer state transition. However, when you want a trigger to be derived from a single event, you do not require the sequencer.

Figure 3-6 shows the sequencer state diagram. The sequencer can change state on every clock cycle, and a different event causes each of the possible sequencer transitions. For more information about PTM events see *Event resources and PTM events* on page 3-49.

On every cycle the sequencer does one of the following:
- remains in the current state
- moves to one of the other two states.

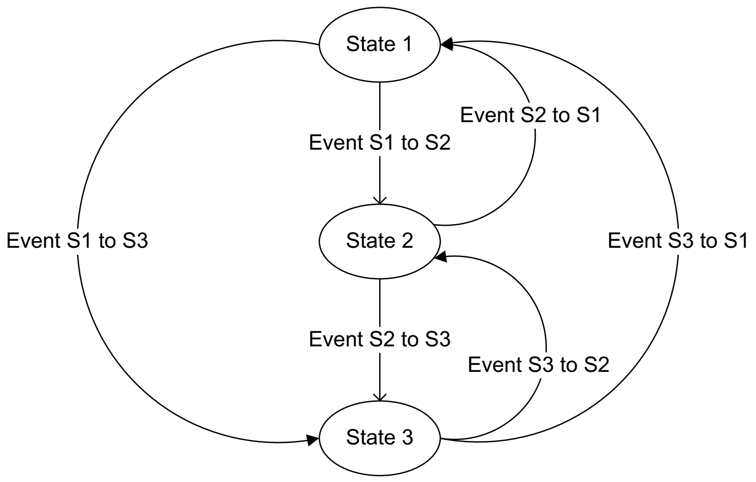


Figure 3-6 Sequencer state diagram

A PTM reset puts the sequencer into State 1. See *Resets* on page 1-17.

Whatever the current state of the sequencer, there are two state transition events that change its state, and:

- if both of these state transition events are active the sequencer remains in its current state
- if neither of these state transition events is active the sequencer remains in its current state
- the behavior of the sequencer is UNPREDICTABLE if either of these state transition events has not been programmed.

You can read and write the current state of the sequencer. See *About the sequencer registers* on page 3-95 and *Programming bit and associated state* on page 3-70.

3.10.1 Use of the PTM sequencer

Possible uses of the PTM sequencer include:
- defining a complex triggering sequence
- extending the performance counters provided by the processor.

3.11 Instrumentation resources

Instrumentation resources provide a simple, low-overhead method of controlling tracing from software, based on:

- The provision of up to four PTM event resources, Instrumentation resource 1 to Instrumentation resource 4. For more information, see *The Instrumentation resource event resources*.
- Instructions in the ARM and Thumb instruction sets that control these resources. These instructions can:
  - Set a specified Instrumentation resource, for the current and following cycles.
  - Clear a specified Instrumentation resource, for the current and following cycles.
  - Pulse a specified Instrumentation resource. This means the resource is set for the current cycle, cleared for the next cycle, and remains clear for following cycles. If the resource is already set then it remains set for the current cycle and is cleared from the next cycle.

  For more information, see *Instructions for controlling the Instrumentation resources*.
- Using these PTM event resources to define PTM events. See *Event resources and PTM events* on page 3-49.

For a PTM implementation, the number of Instrumentation resources is IMPLEMENTATION DEFINED. A PTM implementation:
- is not required to provide any Instrumentation resources
- can provide as many as four Instrumentation resources.

Bits [15:13] of the Configuration Code Extension Register specify the number of Instrumentation resources implemented. See *Configuration Code Extension Register, ETMCCER* on page 3-103.

In addition, if the PTM implements at least one Instrumentation resource, you can use the Instrumentation resource control bit, bit [24], of the Main Control Register to control the availability of Instrumentation resource programming. If this bit is set to 1, the Instrumentation resources can be programmed only when the processor is in a privileged mode. For more information see *Main Control Register, ETMCR* on page 3-75.

3.11.1 The Instrumentation resource event resources

Table 3-10 shows the possible Instrumentation resource event resources.

Table 3-10 The Instrumentation resource event resources

Resource type[a]	Index value[a]	Description of resource type
b001	8	Instrumentation resource 1
b001	9	Instrumentation resource 2
b001	10	Instrumentation resource 3
b001	11	Instrumentation resource 4

a. See *The PTM event resources* on page 3-49 for more information about these fields.

Table 3-5 on page 3-49 also includes these event resources.

3.11.2 Instructions for controlling the Instrumentation resources

Both the ARM and Thumb instruction sets reserve twelve instructions for use as Instrumentation instructions. These instructions are part of the Debug hint (`DBG`) part of the NOP-compatible hint space. Figure 3-7 on page 3-59 shows the Thumb encodings for these instructions.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	1	1	1	1	Hint			

Figure 3-7 Thumb encodings

Figure 3-8 shows the ARM encodings for these instructions.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	1	0	0	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	1	1	1	1	Hint			

Figure 3-8 ARM encodings

For more information see the *ARM Architecture Reference Manual*.

In the DBG instruction, the value of the Hint field determines the Instrumentation resource operation.

Hint field encodings for the DBG Instrumentation instructions

Table 3-11 shows the hint field encodings for the DBG Instrumentation instructions. These encodings are the same for the ARM and Thumb instruction sets.

Table 3-11 Hint field encodings for the Instrumentation instructions

Hint value	Effect of instruction
0x0	Set Resource 1
0x1	Set Resource 2
0x2	Set Resource 3
0x3	Set Resource 4
0x4	Clear Resource 1
0x5	Clear Resource 2
0x6	Clear Resource 3
0x7	Clear Resource 4
0x8	Pulse Resource 1
0x9	Pulse Resource 2
0xA	Pulse Resource 3
0xB	Pulse Resource 4

Note

The DBG hint instructions are defined in the ARMv7 architecture specification. See the *ARM Architecture Reference Manual*. This Architecture Specification only defines the twelve field values given in Table 3-11 for use with PTM implementations. These definitions are the same as those used in version 3.4 of the *ETM Architecture Specification*.

3.11.3 Instrumentation resource behavior when tracing parallel execution

If multiple Instrumentation resource instructions are executed in parallel, the Instrumentation resource must behave as if the instructions were executed in sequence. However, if a resource is activated, by a set or pulse instruction, at any point in the resulting sequence then it must be active for the current cycle. Table 3-12 shows examples of this, for the case where two instructions are executed in parallel.

Table 3-12 Instrumentation resource parallel execution examples, for two instructions

Equivalent instruction sequence		Instrumentation resource behavior[a]
First instruction	**Second instruction**	
No effect	Set	Set on current cycle, continues[b] on future cycles.
No effect	Clear	Hold previous state for current cycle, clear[c] from next cycle.
Clear	Set	Set on current cycle, continues[b] on future cycles.
Clear	Pulse	Set on current cycle, clear[c] from next cycle.
No effect	Pulse	Set on current cycle, clear[c] from next cycle.

a. In the descriptions of resource behavior, the *current cycle* is the cycle when the parallel execution is performed.

b. *Continues* means that the resource remains set until another instruction clears the resource. The instruction immediately following the parallel execution might clear the resource. In this case the resource is only active during the current cycle.

c. Unless the instruction decoded in the next cycle sets the resource.

3.12 PTM input resources

This section gives a short description of the PTM resources that are not described anywhere else. It contains the following sections:

- *External inputs*
- *Extended external inputs*
- *Non-secure state resource* on page 3-62
- *Trace prohibited resource* on page 3-62
- *Hard-wired TRUE resource* on page 3-62.

3.12.1 External inputs

The number of external inputs implemented by a PTM is IMPLEMENTATION DEFINED, between zero and four. Each implemented external input provides a resource that you can use to define PTM events.

Read bits [19:17] of the Configuration Code Register to find the number of implemented external inputs. This field reads-as-zero if the PTM does not implement any external inputs.

Note

The external inputs, typically implemented as **EXTIN** signals, are not related directly to memory accesses. Tracing is imprecise if you use them in any way to enable or disable tracing. For more information about imprecise tracing, see *Imprecise TraceEnable events* on page 3-36.

3.12.2 Extended external inputs

The number of extended external inputs implemented by a PTM is IMPLEMENTATION DEFINED, and can be zero. If a PTM implements extended external inputs, it must implement:

- An extended external input bus, that provides signal inputs to the PTM.
- Between one and four extended external input selectors. Each selector provides a resource that you can use to define PTM events.
- A control register for the extended external input selectors. See *Extended External Input Selection Register, ETMEXTINSELR* on page 3-105. You program this register to define the bit of the extended external input bus that drives each of the extended external input selectors.

If you read the Configuration Code Extension Register:

- bits [10:3] specify the width of the extended external input bus
- bits [2:0] specify the number of extended external input selectors.

Both of these fields read-as-zero if the PTM does not implement any extended external inputs.

Figure 3-9 on page 3-62 shows an implementation with an 8-bit wide extended external input bus and two extended external input selectors.

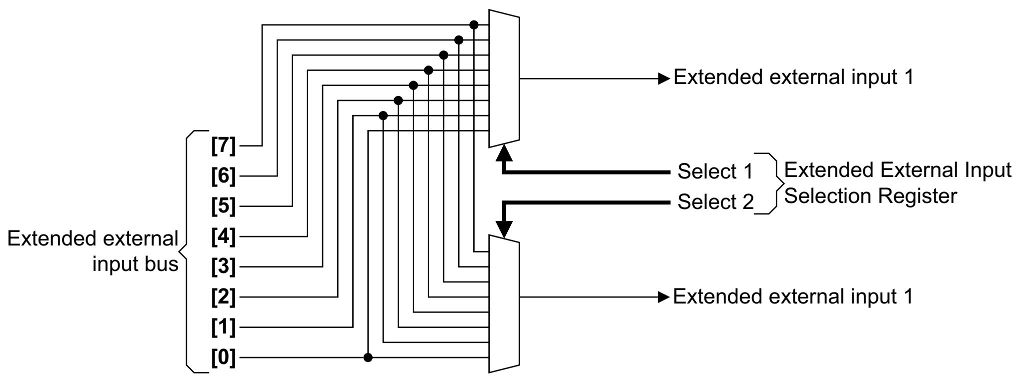

Extended external input bus
[7]
[6]
[5]
[4]
[3]
[2]
[1]
[0]
Extended external input 1
Select 1
Select 2
Extended External Input Selection Register
Extended external input 1

Figure 3-9 Extended external inputs implementation example

3.12.3 Non-secure state resource

Whether the PTM implements this resource depends only on whether the processor that implements the PTM also implements the Security Extensions:

- if the processor implements the Security Extensions then this resource is always implemented, and is TRUE when the processor is in the Non-secure state
- if the processor does not implement the Security Extensions then this resource is not implemented, and cannot be used.

If the PTM is enabled, but has not traced any waypoints since it was powered up, this resource is FALSE and remains FALSE until the PTM recalculates it when it traces the first waypoint.

3.12.4 Trace prohibited resource

The PTM always implements this resource. It is TRUE when tracing is prohibited.

If the PTM is enabled, but has not traced any waypoints since it was powered up, this resource is TRUE and remains TRUE until the PTM recalculates it when it traces the first waypoint.

3.12.5 Hard-wired TRUE resource

The PTM always implements this resource. The resource is always TRUE, and you can use this resource to permanently enable or permanently disable a PTM event. See *Defining a PTM event* on page 3-52.

3.13 PTM external outputs

The number of external inputs implemented by a PTM is IMPLEMENTATION DEFINED, between zero and four. You can use PTM events to control any implemented external outputs. For each implemented external output, the PTM implements an External Output Event Register. See *External Output Event Registers, ETMEXTOUTEVRn* on page 3-97.

Read bits [22:20] of the Configuration Code Register to find the number of implemented external outputs. This field reads-as-zero if the PTM does not implement any external outputs.

To use a PTM external output you program the associated External Output Event Register to define the event that drives the output. Then, when the event is TRUE, the output is driven HIGH.

When the Programming bit is set to 1, the external outputs are forced LOW.

3.14 Triggering a trace run

A PTM includes a trigger event to specify an important point in a trace run. You determine the trigger condition by using the event logic to configure the event resources. See *Event resources and PTM events* on page 3-49, and *Trigger Event Register, ETMTRIGGER* on page 3-80.

The trigger event specifies the conditions that must be met to generate a trigger. When a trigger occurs:

- a trigger packet is generated in the trace
- a trigger can be indicated separately to the system
- a request can be made for the processor to enter debug state.

In a CoreSight system, the trigger indication to the system is normally connected to a *Cross Trigger Interface* (CTI) to enable it to be routed to a trace capture device such as an *Embedded Trace Buffer* (ETB) or *Trace Port Interface Unit* (TPIU).

A simple trigger can be based on memory access address or data matches, for example the execution of an instruction from a particular address. However, a more complicated set of trigger conditions is possible, such as executing a particular instruction several times, or a particular sequence of events occurring before the trigger is asserted.

In any trace run, only a single trigger can be generated by a PTM. When the trigger has been asserted you must set the ETM Programming bit of the ETMCR to 1, and then clear it to 0, before another trigger can occur.

3.15 About the PTM registers

The following sections describe the PTM registers, in general:

- *Register short names*
- *PTM trace and PTM management registers* on page 3-66
- *Accessing the PTM registers* on page 3-66
- *Use of the Programming bit* on page 3-69
- *Synchronization of PTM register updates* on page 3-70
- *Organization of the PTM registers* on page 3-71.

PTM register descriptions on page 3-75 then describes each of the registers, and *About the access permissions for PTM registers* on page 3-138 describes how a system can control access to the registers.

3.15.1 Register short names

All of the PTM registers have short names. Most of these are mnemonics for the full name of the register, except that the short name starts with the letters ETM, indicating that the register is defined by an ARM trace architecture. The ETM architecture is the original ARM trace architecture, and because register assignments are consistent across the trace architectures the register short names always take the ETM prefix. Table 3-13 gives some examples of the register short names.

Table 3-13 Examples of register short names

PTM register name	Register short name	Explanation of short name
Main Control Register	ETMCR	Trace Control Register
Trigger Event Register	ETMTRIGGER	Trace Trigger (Register)
Address Comparator Value Register 3	ETMACVR3	Trace Address Comparator Value Register 3

The use of the ETM prefix for the register short names means that the short names are distinct from the short names used for other registers, such as the processor control coprocessor registers and the debug registers.

3.15.2 PTM trace and PTM management registers

The PTM register map is divided into two areas, as Table 3-14 shows.

Table 3-14 Split of PTM register map into trace and management registers

Register number	Register address	Area
0x000-0x0BF	0x000-0x0BF	Trace
0x0C0-0x0C5	0x300-0x314	Management
0x0C6-0x3BF	0x318-0xEFF	Trace
0x3C0-0x3E7	0xF00-0xF9F	Management
0x3E8-0x3E9	0xFA0-0xFA4	Management in PFTv1.0 Trace in PFTv1.1
0x3EA-0x3FF	0xFA8-0xFFF	Management

Note

Any PTM register numbers not specified in Table 3-16 on page 3-71 are reserved and might be used in the future as either trace or management registers.

This split of the register map is made for register save/restore purposes. For more information see:

- *About the Operating System Save and Restore registers* on page 3-110
- *Power-down support* on page 3-132.

3.15.3 Accessing the PTM registers

The following subsections describe the possible methods of accessing the PTM registers:

- *Coprocessor access*
- *Memory-mapped access* on page 3-68.

An implementation must include at least one interface to the registers, but whether each interface is supported is IMPLEMENTATION DEFINED. If more than one interface is implemented, the architecture supports concurrent access from multiple interfaces.

The description of each method of access includes details of any restrictions that apply to that method, but see also:

- *Restrictions on the type of access to PTM registers* on page 3-68 for information about restrictions that apply to all access methods
- *PTM register access models* on page 3-68.

Coprocessor access

A coprocessor interface to the PTM registers enables you to use the PTM as an extended breakpoint unit, to test for unit failure when testing multiple devices. You can do the following without external hardware:

- program the PTM
- collect trace, in conjunction with the *Embedded Trace Buffer* (ETB)
- examine the contents of the ETB, if implemented.

The following subsections describes how a coprocessor register interface changes the programmers model:

- *The coprocessor access model* on page 3-67
- *Determination of support* on page 3-67
- *Behavior of other CP14 accesses with Opcode_1 equal to 1* on page 3-68.

The coprocessor access model

Where a coprocessor model is supported, a single coprocessor gives access to all the accessible PTM registers. All instructions in Coprocessor 14 (CP14) with Opcode_1 equal to 1 are reserved for PTM use.

The registers you can access using the CP14 interface include the CoreSight management registers and the OS Save/Restore registers. See *Organization of the PTM registers* on page 3-71 for a list of all the PTM registers, in register-number order.

Note

When accessed through the coprocessor interface, the Lock Access and Lock Status registers (registers `0x3EC` and `0x3ED`) read-as-zero. You do not have to set a lock to access the PTM registers through the coprocessor interface.

Use the following instructions to read and write the PTM registers:

```
MRC p14, 1, <Rd>, reg[9:7], reg[3:0], reg[6:4]
MCR p14, 1, <Rd>, reg[9:7], reg[3:0], reg[6:4]
```

In these instructions, `reg[9:0]` is the PTM register number, as Table 3-16 on page 3-71 shows.

Figure 3-10 shows the mapping between the bits of the PTM register number and the fields of the CP14 instruction.

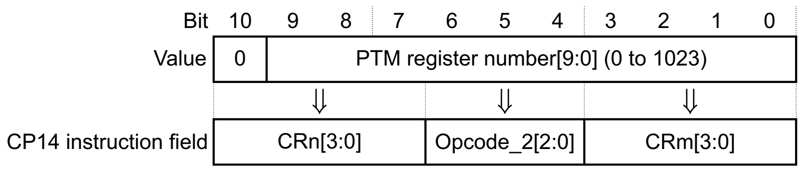

Bit 10 9 8 7 6 5 4 3 2 1 0
Value
0
PTM register number[9:0] (0 to 1023)
CP14 instruction field
CRn[3:0]
Opcode_2[2:0]
CRm[3:0]

Figure 3-10 Mapping from register number to CP14 `MRC` or `MCR` instruction fields

You can access the PTM registers through the coprocessor interface only when the processor is in a privileged mode. Any attempt to read or write a PTM register using coprocessor instructions when the processor is in User mode generates an Undefined Instruction exception. However, any coprocessor accesses initiated by a debug tool when the processor is halted in Debug state is always privileged regardless of the state of the CPSR.

A read from a nonexistent register returns zero, and a write to a nonexistent register is ignored. The tools must determine what registers are valid from the programmers model. For more information on access permissions see *About the access permissions for PTM registers* on page 3-138.

Access to PTM registers might also be prevented by controls in the processor. The CPACR, NSACR, and HCPTR (for Virtualization) include controls to trap accesses to PTM registers.

Note

This behavior is different to coprocessor access to debug registers, where attempting to access a nonexistent register usually results in an Undefined Instruction exception.

Determination of support

To determine whether coprocessor access is supported, read the ID Register. See *ID Register, ETMIDR* on page 3-101:

```
MRC p14, 1, <Rd>, c0, c9, 7
```

If this instruction does not generate an Undefined Instruction exception, and returns a nonzero value, then coprocessor access is supported.

Behavior of other CP14 accesses with Opcode_1 equal to 1

All CP14 access instructions with Opcode_1 equal to 1 are reserved for PTM use. However, only instructions with CRn values from b0000 to b0111 are used for PTM register accesses. `MRC` and `MCR` accesses to Coprocessor 14 with a CRn value of b1000 or greater are:

- UNDEFINED in User mode
- UNPREDICTABLE in privileged modes.

Memory-mapped access

A PTM can provide memory-mapped access to its registers. This is usually required in a CoreSight system, and provides all the benefits of coprocessor access, along with other benefits described in the *CoreSight Architecture Specification*.

Memory-mapped access provides a 4KB address space, and gives access to all registers. Each register occupies 4 bytes, making a total of 1024 registers available in this way. For example, register `0x080` is at offset `0x200` from the base address of the PTM.

A read from a nonexistent register returns zero, and a write to a nonexistent register is ignored. Debug tools must use the programmers model to determine the registers that are valid.

For more information about register access permissions see *About the access permissions for PTM registers* on page 3-138.

The PTM can distinguish between memory-mapped accesses from:

- on-chip software
- a debugger, for example using the CoreSight *Debug Access Port* (DAP).

Software accesses require you to first unlock the PTM, using the lock registers described in *About the lock registers* on page 3-118.

Restrictions on the type of access to PTM registers

In *About the PTM registers* on page 3-65, Table 3-16 on page 3-71 shows the access type, read-only, write-only, or read/write, of each PTM register. Debug software must take account of these access types. Behavior is UNPREDICTABLE if you attempt a read access to a write-only register, or a write access to a read-only register. This is true for all methods of accessing the registers.

PTM register access models

Table 3-15 summarizes the more common PTM register access models, with an indication of the situations when they are likely to be appropriate.

Table 3-15 Typical PTM register access implementations

Register interfaces	Access requirements
Coprocessor only	Debugger access is through an ARM Debug Interface v5[a]. In addition, software access is possible through the coprocessor interface.
Memory-mapped only	Debugger access is through an ARM Debug Interface v5[a]. It is IMPLEMENTATION DEFINED whether software access is possible through the system memory map.
Coprocessor and memory-mapped	Debugger access is through an ARM Debug Interface v5[a]. In addition, software access is possible through the coprocessor interface, and it is IMPLEMENTATION DEFINED whether software access is possible through the system memory map.

a. For more information see the *ARM Debug Interface v5 Architecture Specification*.

For information about the access controls that can apply to PTM register accesses see *About the access permissions for PTM registers* on page 3-138.

3.15.4 Use of the Programming bit

When programming the PTM registers you must enable all the changes at the same time. For example, if you reprogram the counter, it might start to count based on incorrect events, before you have set up required counter enable condition. Also, the programming interface clock can be asynchronous to the PTM clock.

You must use the Programming bit, ProgBit, in the Main Control Register, to disable all PTM operations during programming. For more information see *Main Control Register, ETMCR* on page 3-75.

Figure 3-11 shows the procedure for using ProgBit to control the programming of the PTM registers. When the Programming bit is set to 0 you must not write to any trace register other than the Main Control Register, because this can lead to UNPREDICTABLE behavior.

When setting the Programming bit to 1, you must not change any other bits of the Main Control Register. You must only change the value of Main Control Register bits other than the Programming bit when bit [1] of the Status Register is set to 1. ARM recommends that you use a read-modify-write procedure when modifying the Main Control Register.

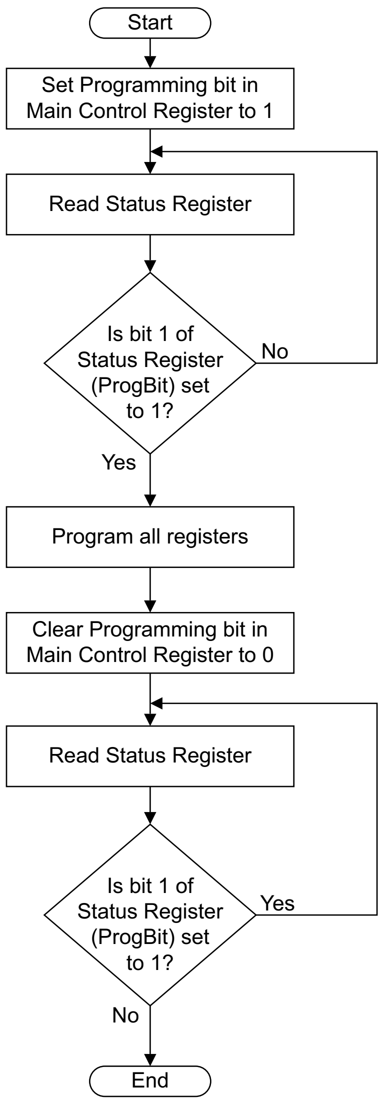

Start
Set Programming bit in Main Control Register to 1
Read Status Register
Is bit 1 of Status Register (ProgBit) set to 1?
No
Yes
Program all registers
Clear Programming bit in Main Control Register to 0
Read Status Register
Is bit 1 of Status Register (ProgBit) set to 1?
Yes
No
End

Figure 3-11 Programming PTM registers

You can program the PTM registers regardless of whether the processor is in Debug state.

Programming bit and associated state

The Programming bit disables all PTM functions. See Figure 3-11 on page 3-69 for the procedure to change this bit. While the Programming bit is set to 1:

- Trace generation is disabled. The FIFO is emptied and then no more trace is produced.
- The counters, sequencer, and start/stop block are held in their current state.
- The external outputs are forced LOW.

PTM state items

The following items of PTM state are affected by the Programming bit:

- the value of the counters, held in the Counter Value Registers, `0x5C-0x5F`
- the sequencer
- the start/stop resource status, bit [2] of the Status Register
- the trigger flag, bit [3] of the Status Register.

You can read and write the PTM state information directly. This means you can save this state information when powering down the ARM processor, and restore it when restarting the system.

The PTM retains its state while the Programming bit is set to 1 and tracing is disabled. The state is reset when the Programming bit is cleared to 0, unless written to since the Programming bit was last set to 1. You must set the Programming bit to 1 before reading the state. This holds the state stable, ensuring you obtain a consistent result. See *Use of the Programming bit* on page 3-69 for more information.

3.15.5 Synchronization of PTM register updates

Software running on the processor can program the debug registers through at least one of:

- a CP14 coprocessor interface
- the memory-mapped interface, if it is implemented.

It is IMPLEMENTATION DEFINED the interfaces are implemented.

For the CP14 coprocessor interface, the following synchronization rules apply:

- All changes to CP14 PTM registers that appear in program order after any explicit memory operations are guaranteed not to affect those memory operations.
- Any change to CP14 PTM registers is guaranteed to be visible to subsequent instructions only after one of:
  - — performing an ISB operation
  - — taking an exception
  - — returning from an exception.

  However, the following rules apply to coprocessor PTM register accesses:

  - — when an `MRC` instruction directly reads a register using the same register number as an `MCR` instruction used to write it, the `MRC` is guaranteed to observe the value written, without requiring any context synchronization between the `MCR` and `MRC` instructions
  - — when an `MCR` instruction directly writes a register using the same register number as a previous `MCR` instruction used to write it, the final result is the value of the second `MCR`, without requiring any context synchronization between the two `MCR` instructions.

  This is important when changing the value of the Programming bit in the Main Control Register. After writing to the Main Control Register to change the value of the Programming bit you must make at least one read of the Status Register before you program any other registers. See *Use of the Programming bit* on page 3-69 for more information. You must execute an ISB instruction writing to the Main Control Register and reading the Status Register.

  ARM recommends that, after programming the PTM registers, you always execute an `ISB` instruction to ensure that all updates are committed to the PTM before you restart normal code execution.

For the memory-mapped interface, the following synchronization rules apply:

- Changes to memory-mapped PTM registers that appear in program order after an explicit memory operation are guaranteed not to affect that previous memory operation only if the order is guaranteed by the memory order model or by the use of a DMB or DSB operation between the memory operation and the register change.
- A DSB operation causes all writes to memory-mapped PTM registers appearing in program order before the DSB to be completed.

  However, the following rules apply to memory-mapped PTM register accesses:

  - when a load directly reads a register using the same address as a store used to write it, the load is guaranteed to observe the value written, without requiring any context synchronization between the store and load
  - when a store directly writes a register using the same address as a previous store used to write it, the final result is the value of the second store, without requiring any context synchronization between the two stores.

  ARM recommends that, after programming the PTM registers, you always execute a `DSB` instruction followed by an `ISB` instruction, to ensure that all updates are committed to the PTM before you restart normal code execution.
- All accesses to the memory-mapped PTM registers by a particular processor have their effect in the order that the processor performs the accesses, as determined by the memory order model and the use of DSB and DMB operations.

Some memory-mapped PTM registers are not idempotent for reads or writes. Therefore, the region of memory occupied by the PTM registers must not be marked as Normal memory, because the memory order model permits accesses to Normal memory locations that are not appropriate for such registers. Memory used for memory-mapped PTM registers must have the Strongly-ordered or Device attribute, otherwise the effects of accesses to the registers are UNPREDICTABLE.

Synchronization between register updates made through the external debug interface and updates made by software running on the processor is IMPLEMENTATION DEFINED. However, if the external debug interface is implemented through the same port as the memory-mapped interface, then updates made through the external debug interface have the same properties as updates made through the memory-mapped interface.

3.15.6 Organization of the PTM registers

Table 3-16 lists the PTM registers, in register order. In the table, access type is described as follows:

RW	Read and write
RO	Read only
WO	Write only.

Table 3-16 PTM registers summary

Register[a]	Name	Type	Description
`0x000-0x0BF`, Trace registers[b]			
`0x000`	Main Control	RW	See *Main Control Register, ETMCR* on page 3-75
`0x001`	Configuration Code	RO	See *Configuration Code Register, ETMCCR* on page 3-78
`0x002`	Trigger Event	RW	See *Trigger Event Register, ETMTRIGGER* on page 3-80
`0x003`	Reserved	-	-
`0x004`	Status	RW	See *Status Register, ETMSR* on page 3-81
`0x005`	System Configuration	RO	See *System Configuration Register, ETMSCR* on page 3-82

Table 3-16 PTM registers summary (continued)

Register[a]	Name	Type	Description
TraceEnable configuration. See *About the TraceEnable control registers* on page 3-82:			
`0x006`	TraceEnable Start/Stop Control	RW	See *TraceEnable Start/Stop Control Register, ETMTSSCR* on page 3-83
`0x007`	Reserved	-	-
`0x008`	TraceEnable Event	RW	See *TraceEnable Event Register, ETMTEEVR* on page 3-83
`0x009`	TraceEnable Control	RW	See *TraceEnable Control Register, ETMTECR1* on page 3-84
`0x00A`	Reserved	-	-
FIFOFULL configuration:			
`0x00B`	FIFOFULL Level	RW	See *FIFOFULL Level Register, ETMFFLR* on page 3-85
`0x00C-0x00F`	Reserved	-	-
Address comparators. See *About the address comparator registers* on page 3-86:			
`0x010- 0x01F`	Address Comparator Value 1-16	RW	See *Address Comparator Value Registers, ETMACVRn* on page 3-86
`0x020- 0x02F`	Address Comparator Access Type 1-16	RW	See *Address Comparator Access Type Registers, ETMACTRn* on page 3-87
`0x030- 0x04F`	Reserved	-	-
Counters. See *About the counter registers* on page 3-91:			
`0x050-0x053`	Counter Reload Value 1-4	RW	See *Counter Reload Value Registers, ETMCNTRLDVRn* on page 3-91
`0x054-0x057`	Counter Enable Event 1-4	RW	See *Counter Enable Event Registers, ETMCNTENRn* on page 3-92
`0x058-0x05B`	Counter Reload Event 1-4	RW	See *Counter Reload Event Registers, ETMCNTRLDEVRn* on page 3-93
`0x05C-0x05F`	Counter Value 1-4	RW	See *Counter Value Registers, ETMCNTVRn* on page 3-94
Sequencer. See *About the sequencer registers* on page 3-95:			
`0x060-0x065`	Sequencer State Transition Event 1-6	RW	See *Sequencer State Transition Event Registers, ETMSQabEVR* on page 3-96
`0x066`	-	-	Reserved
`0x067`	Current Sequencer State	RW	See *Current Sequencer State Register, ETMSQR* on page 3-97
External output event:			
`0x068-0x06B`	External Output Event 1-4	RW	See *External Output Event Registers, ETMEXTOUTEVRn* on page 3-97
Context ID comparators. See *About the Context ID comparator registers* on page 3-98:			
`0x06C-0x06E`	Context ID Comparator Value	RW	See *Context ID Comparator Value Registers, ETMCIDCVRn* on page 3-98
`0x06F`	Context ID Comparator Mask	RW	See *Context ID Comparator Mask Register, ETMCIDCMR* on page 3-99
Other trace[b] registers:			
`0x070-0x077`	IMPLEMENTATION SPECIFIC	RW	See *Implementation specific registers, ETMIMPSPEC0 to ETMIMPSPEC7* on page 3-100

Table 3-16 PTM registers summary (continued)

Register[a]	Name	Type	Description
`0x078`	Synchronization Frequency	RW	See *Synchronization Frequency Register, ETMSYNCFR* on page 3-101
`0x079`	ID	RO	See *ID Register, ETMIDR* on page 3-101
`0x07A`	Configuration Code Extension	RO	See *Configuration Code Extension Register, ETMCCER* on page 3-103
`0x07B`	Extended External Input Selection	RW	See *Extended External Input Selection Register, ETMEXTINSELR* on page 3-105
`0x07C`	TraceEnable Start/Stop EmbeddedICE Control	RW	See *TraceEnable Start/Stop EmbeddedICE Control Register, ETMTESSEICR* on page 3-106
`0x07D`	Embedded ICE Behavior Control	RW	See *EmbeddedICE Behavior Control Register, ETMEIBCR* on page 3-106
`0x07E`	Timestamp Event	RW	See *Timestamp Event Register, ETMTSEVR* on page 3-107
`0x07F`	Auxiliary Control	RW	See *Auxiliary Control Register, ETMAUXCR* on page 3-108
`0x080`	CoreSight Trace ID	RW	See *CoreSight Trace ID Register, ETMTRACEIDR* on page 3-108
`0x081-0x08F`	-	-	Reserved
`0x090`	VMID Comparator Value Register	RW	See *VMID Comparator Value Register, ETMVMIDCVR* on page 3-109
`0x091-0x0BF`	-	-	Reserved
`0x0C0-0x0C5`, Management registers[b]			
Operating system save and restore registers. See *About the Operating System Save and Restore registers* on page 3-110:			
`0x0C0`	OS Lock Access	WO	See *OS Lock Access Register, ETMOSLAR* on page 3-110
`0x0C1`	OS Lock Status	RO	See *OS Lock Status Register, ETMOSLSR* on page 3-110
`0x0C2`	OS Save/Restore	RW	See *OS Save and Restore Register, ETMOSSRR* on page 3-112
Other management registers:			
`0x0C3-0x0C3`	-	-	Reserved
`0x0C4`	Power-Down Control	RW	See *Device Power-Down Control Register, ETMPDCR* on page 3-113
`0x0C5`	Power-Down Status	RW	See *Device Power-Down Status Register, ETMPDSR* on page 3-114
`0x0C6-0x3BF`, Trace registers[b]			
`0x0C6-0x37F`	-	-	Reserved
`0x380-0x3BF`	Integration registers	-	Reserved for IMPLEMENTATION DEFINED topology detection and integration test registers
`0x3C0-0x3FF`, Management registers[b]			
`0x3C0`	Integration Mode Control	RW	See *Integration Mode Control Register, ETMITCTRL* on page 3-116
`0x3E8`	Claim Tag Set	RW	See *Claim Tag Set Register, ETMCLAIMSET* on page 3-116
`0x3E9`	Claim Tag Clear	RW	See *Claim Tag Clear Register, ETMCLAIMCLR* on page 3-117

Table 3-16 PTM registers summary (continued)

Register[a]	Name	Type	Description
`0x3EC`	Lock Access	WO	See *Lock Access Register, ETMLAR* on page 3-118
`0x3ED`	Lock Status	RO	See *Lock Status Register, ETMLSR* on page 3-119
`0x3EE`	Authentication Status	RO	See *Authentication Status Register, ETMAUTHSTATUS* on page 3-119
`0x3F2`	Device Configuration	RO	See *Device Configuration Register, ETMDEVID* on page 3-121
`0x3F3`	Device Type	RO	See *Device Type Register, ETMDEVTYPE* on page 3-122
Peripheral and Component ID registers:			
`0x3F4`	Peripheral ID4	RO	See *Peripheral ID4 Register, ETMPIDR4* on page 3-127
`0x3F5`	Peripheral ID5	RO	Reserved in current implementations. See *Peripheral ID5 to Peripheral ID7 Registers, ETMPIDR5 to ETMPIDR7* on page 3-128
`0x3F6`	Peripheral ID6	RO	
`0x3F7`	Peripheral ID7	RO	
`0x3F8`	Peripheral ID0	RO	See *Peripheral ID0 Register, ETMPIDR0* on page 3-124
`0x3F9`	Peripheral ID1	RO	See *Peripheral ID1 Register, ETMPIDR1* on page 3-125
`0x3FA`	Peripheral ID2	RO	See *Peripheral ID2 Register, ETMPIDR2* on page 3-126
`0x3FB`	Peripheral ID3	RO	See *Peripheral ID3 Register, ETMPIDR3* on page 3-126
`0x3FC`	Component ID0	RO	See *Component ID0 Register, ETMCIDR0* on page 3-129
`0x3FD`	Component ID1	RO	See *Component ID1 Register, ETMCIDR1* on page 3-130
`0x3FE`	Component ID2	RO	See *Component ID2 Register, ETMCIDR2* on page 3-130
`0x3FF`	Component ID3	RO	See *Component ID3 Register, ETMCIDR3* on page 3-131

a. The Register column gives the *register number*. Registers are numbered sequentially from zero. Where registers are accessed in a memory-mapped scheme, the offset of a register is (4×register number).

b. For more information about the division into Trace and Management registers, see *PTM trace and PTM management registers* on page 3-66.

For details of the access controls on the PTM registers see *About the access permissions for PTM registers* on page 3-138.

3.16 PTM register descriptions

The following subsections describe the PTM registers, in register number order. See Table 3-16 on page 3-71 for a list of these registers, with references to the subsections that describe them.

3.16.1 Main Control Register, ETMCR

The ETMCR characteristics are:

Purpose	Controls general operation of the PTM, such as whether tracing is enabled or is cycle-accurate.
Usage Constraints	There are no usage constraints.
Configurations	Available in all PTM implementations.
Attributes	See the register summary in Table 3-16 on page 3-71.

Figure 3-12 shows the ETMCR bit assignments for PFTv1.1.

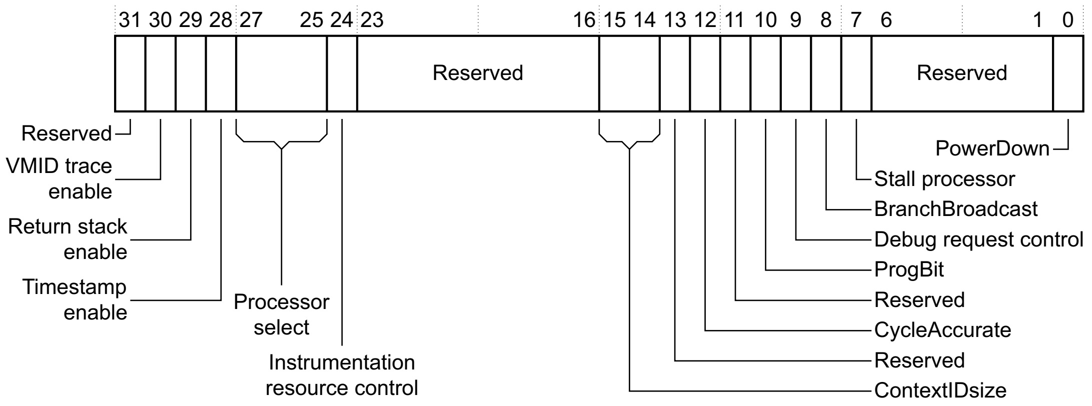


Figure 3-12 ETMCR bit assignments

Table 3-17 shows the ETMCR bit assignments.

Table 3-17 ETMCR bit assignments

Bits	Name	Function
[31]	Reserved	SBZP.
[30]	Reserved	For PFTv1.0 this bit is Reserved, SBZP.
	VMID trace enable	For PFTv1.1: • This bit controls VMID tracing. Set this bit to 1 to enable VMID tracing. See *Virtualization* on page 2-29. • If bit [26] of the Configuration Code Extension Register is zero, indicating that Virtualization support is not implemented, this bit is RAZ/WI. • A PTM reset sets this bit to 0.
[29]	Return stack enable	Set this bit to 1 to enable use of the return stack. See *Use of a return stack* on page 4-195. You must not set this bit to 1 if bit [8] of this register is set to 1 to enable branch broadcasting. Behavior is UNPREDICTABLE if you enable both use of the return stack and branch broadcasting. If bit [23] of the Configuration Code Extension Register is zero, indicating that the return stack is not implemented, this bit is RAZ/WI. A PTM reset sets this bit to 0.

Table 3-17 ETMCR bit assignments (continued)

Bits	Name	Function
[28]	Timestamp enable	Set this bit to 1 to enable timestamping. See *Timestamping* on page 4-198. If bit [22] of the Configuration Code Extension Register is zero, indicating that timestamping is not implemented, this bit is RAZ/WI. A PTM reset sets this bit to 0.
[27:25]	Processor select	If a PTM is shared between multiple processors, selects the processor to trace. For the maximum value permitted, see bits [14:12] of ETMSCR. See *ETMSCR bit assignments* on page 3-82. To guarantee that the PTM is correctly synchronized to the new processor, you must update these bits as follows: 1. Set bit [10], Programming bit, to b1. 2. Poll bit [1] of the Status Register until it is set to b1, as described in *Use of the Programming bit* on page 3-69. 3. Set bit [0], PowerDown, to b1. 4. Change the Processor select bits. 5. Clear bit [0], PowerDown, to b0. 6. Perform other programming required as normal. You must not set this field to a value greater than bits [14:12] of the System Configuration Register. On a PTM reset these bits are all zero.
[24]	Instrumentation resources access control	When this bit is set to 1, the Instrumentation resources can only be controlled when the processor is in a privileged mode. When this bit is set to 0, the Instrumentation resources can be accessed in both privileged and User modes. If no Instrumentation resources are implemented this bit is RAZ/WI. Bits [15:13] of the Configuration Code Extension Register indicate the number of implemented Instrumentation resources. A PTM reset sets this bit to 0.
[23:16]	Reserved	SBZP.
[15:14]	ContextIDsize	The possible values of this field are: **b00** No Context ID tracing. **b01** One byte traced, Context ID bits [7:0]. **b10** Two bytes traced, Context ID bits [15:0]. **b11** Four bytes traced, Context ID bits [31:0]. **Note** The PTM traces only the number of bytes specified, even if the new Context ID value is larger than this. If Context ID tracing is not supported these bits are RAZ/WI. A PTM reset sets these bits to zero.
[13]	Reserved	SBZP.
[12]	CycleAccurate	Set this bit to 1 to enable cycle-accurate tracing. See *Cycle-accurate tracing* on page 4-160. If the implementation does not support cycle-accurate tracing this bit is RAZ/WI. A PTM reset sets this bit to 0.
[11]	Reserved	SBZP.

Table 3-17 ETMCR bit assignments (continued)

Bits	Name	Function
[10]	ProgBit	Programming bit. You must set this bit to 1 to program the PTM, and clear it to 0 when programming is complete. See *Programming bit and associated state* on page 3-70. A PTM reset sets this bit to 1.
[9]	Debug request control[a]	When this bit is set to 1 and the trigger event occurs, the **DBGRQ** output is asserted until **DBGACK** is observed. This enables a debugger to force the ARM processor into Debug state. A PTM reset sets this bit to 0. **Note** If the Programming Bit or the OS Lock are set before the processor has entered debug state, it is IMPLEMENTATION DEFINED whether the PTM sustains this request. The processor might or might not enter debug state.
[8]	BranchBroadcast	Set this bit to 1 to enable branch broadcasting. Branch broadcasting traces the addresses of direct branch instructions. See *Branch broadcasting* on page 4-185. You must not set this bit to 1 if bit [29] of this register is set to 1 to enable use of the return stack. Behavior is UNPREDICTABLE if you enable both use of the return stack and branch broadcasting. If the implementation does not support branch broadcasting this bit is RAZ/WI. A PTM reset sets this bit to 0.
[7]	Stall processor	Set this bit to 1 to enable the use of the **FIFOFULL** output to stall the processor to prevent overflow: • when this bit is 0, **FIFOFULL** remains LOW at all times and the FIFO overflows if there are too many trace packets • when this bit is 1, **FIFOFULL** is asserted when the trace FIFO is nearly full, and can be used to stall the processor. See *FIFOFULL Level Register, ETMFFLR* on page 3-85 for more information. If the **FIFOFULL** signal is not implemented then this bit is RAZ/WI. A PTM reset sets this bit to 0.
[6:1]	Reserved	SBZP.
[0]	PowerDown	A pin controlled by this bit enables the PTM power to be controlled externally. The external pin is often **PTMPWRDOWN**, or inverted as **PTMPWRUP**. This bit must be cleared by the trace software tools at the beginning of a debug session. When this bit is set to 1, the PTM must be powered down and disabled, and then operated in a low power mode with all clocks stopped. When this bit is set to 1, writes to some registers and fields might be ignored. You can always write to the following registers and fields: • ETMCR, bit [0] and bits [27:25] • ETMLAR • ETMCLAIMSET • ETMCLAIMCLR • ETMOSLAR. When ETMCR is written with this bit set to 1, writes to bits other than bits [27:25, 0] might be ignored. A PTM reset sets this bit to 1.

a. When this bit is set, the PTM requests the processor to enter debug state when a PTM trigger occurs. If the Programming Bit or the OS Lock are set before the processor has entered debug state, it is IMPLEMENTATION DEFINED whether the PTM sustains this request. The processor might or might not enter debug state.

Checking support for IMPLEMENTATION DEFINED features

The values of ETMCR bits specify whether the PTM supports the following IMPLEMENTATION DEFINED features:

- VMID tracing, controlled by bit [30]
- the return stack, controlled by bit [29]
- timestamping, controlled by bit [28]
- Context ID tracing, controlled by bits [15:14]
- cycle-accurate tracing, controlled by bit [12]
- branch broadcasting, controlled by bit [8]
- FIFOFULL stalling, controlled by bit [7].

You can read the System Configuration Register, the Configuration Code Register, and the Configuration Code Extension Register to determine whether some of these features are supported. However, if a feature is not supported then its control field in the Main Control Register is RAZ/WI. Therefore, debug tools can perform a sequence of read, modify, write, and re-read on the Main Control Register to find the features that are supported. To avoid changing other PTM control settings, the test process is:

1. Ensure that, in the ETMCR, the Programming bit, bit [10], is set to 1, and the PowerDown bit, bit [0], is set to 0.
2. Read the ETMCR, and save a copy of the returned value.
3. Modify the returned data, setting bits [29:28,15:14,12,8,7] to 1.
4. Write the modified value back to the ETMCR.
5. Read the ETMCR again, and check each of bits [29:28,15:14,12,8,7] to see whether the feature is supported. For fields other than the ContextIDsize field, the meaning of the value returned is:

bit == 0	feature not supported
bit == 1	feature is supported.

   For the ContextIDsize field, bits [15:14], the meaning of the value returned is:

bits [15:14] == 0	Context ID tracing not supported
bits [15:14] > 0	Context ID tracing is supported.

6. Write the value from step 2 back to the ETMCR to restore its original value.

3.16.2 Configuration Code Register, ETMCCR

The ETMCCR characteristics are:

Purpose	Enables software to read the IMPLEMENTATION DEFINED configuration of the PTM, giving the number of each type of resource. Where a value indicates the number of instances of a particular resource, zero indicates that there are no implemented resources of that resource type.
Usage constraints	There are no usage constraints.
Configurations	Available in all PTM implementations.
Attributes	See the register summary in Table 3-16 on page 3-71.

Figure 3-13 on page 3-79 shows the ETMCCR bit assignments.

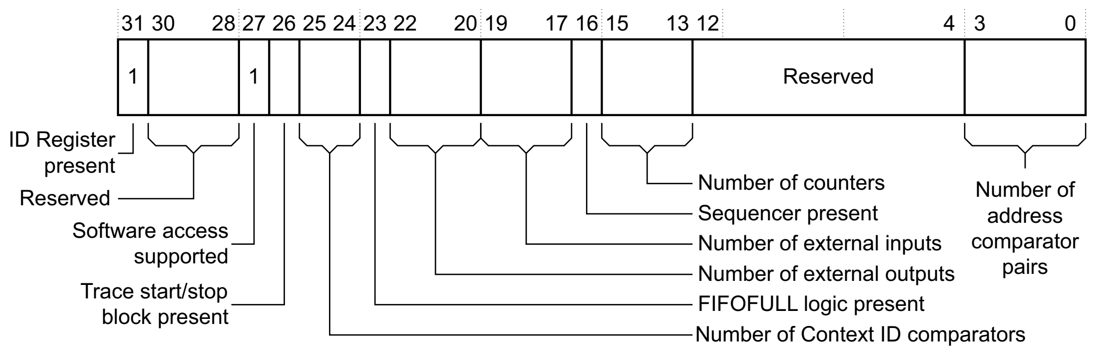

31 30 28 27 26 25 24 23 22 20 19 17 16 15 13 12 4 3 0
1
1
Reserved
ID Register present
Reserved
Software access supported
Trace start/stop block present
Number of counters
Sequencer present
Number of external inputs
Number of external outputs
FIFOFULL logic present
Number of Context ID comparators
Number of address comparator pairs

Figure 3-13 ETMCCR bit assignments

Table 3-18 shows the ETMCCR bit assignments.

Table 3-18 ETMCCR bit assignments

Bits	Maximum value	Description
[31]	1	This bit is Read-as-One, indicating that the ETMIDR is present and defines the PFT architecture version in use. For more information see *ID Register, ETMIDR* on page 3-101.
[30:28]	-	Reserved, SBZP.
[27]	1	This bit is Read-as-One, indicating that the PTM supports software accesses to the PTM registers. See *Accessing the PTM registers* on page 3-66.
[26]	1	This bit indicates whether the trace start/stop block is present: **0** Trace start/stop block not implemented. **1** Trace start/stop block implemented. If no address comparators are implemented, this bit is RAZ.
[25:24]	3	The number of Context ID comparators implemented.
[23]	1	This bit indicates whether the FIFOFULL logic is present: **0** FIFOFULL logic not implemented. **1** FIFOFULL logic implemented. **Note** You can use FIFOFULL only if it is supported by both your PTM and your system. Some processors do not support FIFOFULL, so it cannot be used by the system. Therefore, a debugger must consider this bit in conjunction with bit [8] of the System Configuration Register, ETMSCR, of the processor connected to the PTM. If this bit is 0, the FIFOFULL Region Register, `0x00A`, is not implemented and is Reserved, RAZ. In this case, the Processor stall bit, bit [7], of the ETMCR is RAZ/WI.
[22:20]	4	The number of external outputs, supplied by the ASIC. A system that includes a PTM might not connect all of the external outputs provided by the PTM. This field indicates the number of connected external outputs.
[19:17]	4	The number of external inputs, supplied by the ASIC. A system that includes a PTM might not connect all of the external inputs provided by the PTM. This field indicates the number of connected external inputs.
[16]	1	This bit indicates whether the sequencer is present: **0** Sequencer not implemented. **1** Sequencer implemented.

Table 3-18 ETMCCR bit assignments (continued)

Bits	Maximum value	Description
[15:13]	4	The number of counters.
[12:4]	-	Reserved, SBZP.
[3:0]	8	The number of pairs of address comparators.

3.16.3 Trigger Event Register, ETMTRIGGER

The ETMTRIGGER register characteristics are:

Purpose	Defines the event that controls the trigger.
Usage constraints	There are no usage constraints.
Configurations	Available in all PTM implementations.
Attributes	See the register summary in Table 3-16 on page 3-71.

Figure 3-14 shows the ETMTRIGGER bit assignments.

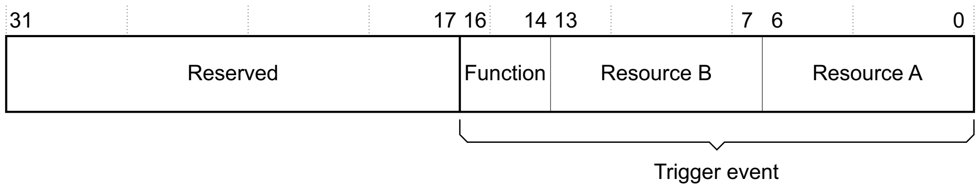


Figure 3-14 ETMTRIGGER bit assignments

Table 3-19 shows the ETMTRIGGER bit assignments.

Table 3-19 ETMTRIGGER bit assignments

Bits	Description
[31:17]	Reserved, SBZP.
[16:0]	Trigger event. Subdivided as: **Function, bits [16:14]** Specifies the logical operation that combines the two resources that define the event. **Resource B, bits [13:7] and Resource A, bits [6:0]** Specify the two resources that are combined by the logical operation specified by the Function field. For more information see *Event resources and PTM events* on page 3-49.

Defining a PTM event on page 3-52 describes how you define a trigger event.

3.16.4 Status Register, ETMSR

The ETMSR characteristics are:

Purpose	Provides information about the current status of the trace and trigger logic.
Usage constraints	This is a Read/Write register with some bits that are Read Only.
Configurations	Available in all PTM implementations.
Attributes	See the register summary in Table 3-16 on page 3-71.

Figure 3-15 shows the ETMSR bit assignments.

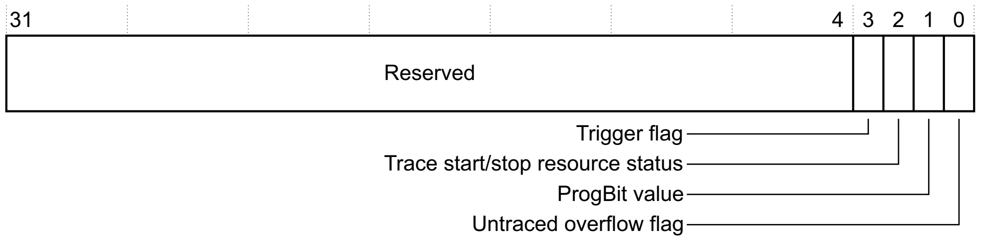


Figure 3-15 ETMSR bit assignments

Table 3-20 shows the ETMSR bit assignments.

Table 3-20 ETMSR bit assignments

Bits	Type	Description
[31:4]	-	Reserved, SBZP.
[3]	RW	Trigger bit. Set when the trigger occurs, and prevents the trigger from being output until the PTM is next programmed. For more information see *Programming bit and associated state* on page 3-70.
[2]	RW	Holds the current status of the trace start/stop resource. If set to 1, it indicates that a *trace start* address has been matched, without a corresponding *trace stop* address match.
[1]	RO	Indicates the current effective value of ProgBit, bit [10] of the ETMCR. You must wait for this bit to go to 1 before you start to program the PTM. See *Programming bit and associated state* on page 3-70. If you read other bits in the ETMSR while this bit is 0, some instructions might not have taken effect. ARM recommends that you set the Programming bit to 1 and then wait for this bit to go to 1 before interpreting any other values in this register or reading any other register. This bit remains 0 while there is any data in the FIFO. This ensures that the FIFO is empty before you can reprogram the PTM. **Note** From PFT v1.1, this bit is set when OS Lock is also set.
[0]	RO	If set to 1, there is an overflow that has not yet been traced. This bit is cleared to 0 when either: • trace is restarted • the PTM Power Down bit, bit [0] of the Main Control Register, `0x000`, is set to 1. **Note** Setting or clearing the Programming bit, ProgBit, does not clear this bit to 0.

3.16.5 System Configuration Register, ETMSCR

The ETMSCR characteristics are:

Purpose	Shows the PTM features supported by the PTM. The contents of this register are based on inputs provided by the ASIC.
Usage constraints	There are no usage constraints.
Configurations	Available in all PTM implementations.
Attributes	See the register summary in Table 3-16 on page 3-71.

Figure 3-16 shows the ETMSCR bit assignments.

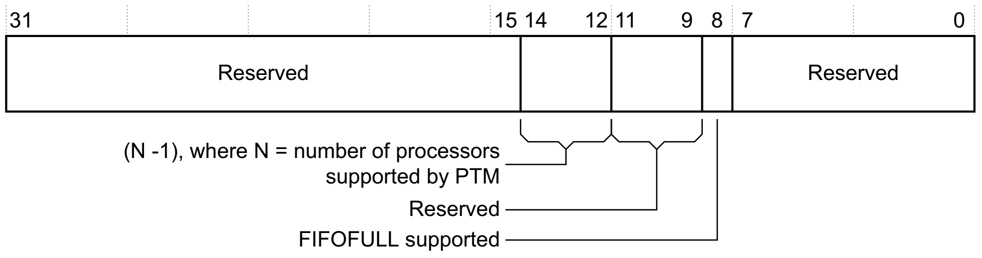


Figure 3-16 ETMSCR bit assignments

Table 3-21 shows the ETMSCR bit assignments.

Table 3-21 ETMSCR bit assignments

Bits	Description
[31:15]	Reserved, SBZP.
[14:12]	Number of supported processors minus 1. The value given here is the maximum value that can be written to bits [27:25] of the ETMCR.
[11:9]	Reserved, SBZP.
[8]	If set to 1, **FIFOFULL** is supported. This bit is used in conjunction with bit [23] of the ETMCCR. **Note** You can use **FIFOFULL** only if it is supported by both your PTM and your system. Some processors do not support **FIFOFULL**, so it cannot be used by the system.
[7:0]	Reserved, SBZP.

3.16.6 About the TraceEnable control registers

TraceEnable on page 3-34 describes the TraceEnable mechanism that controls when trace is generated. Three registers control and configure the operation of TraceEnable. They are described in the following sections:

- *TraceEnable Start/Stop Control Register, ETMTSSCR* on page 3-83
- *TraceEnable Event Register, ETMTEEVR* on page 3-83
- *TraceEnable Control Register, ETMTECR1* on page 3-84.

The following sections also describe registers that control TraceEnable operation:

- *TraceEnable Start/Stop EmbeddedICE Control Register, ETMTESSEICR* on page 3-106
- *EmbeddedICE Behavior Control Register, ETMEIBCR* on page 3-106.

3.16.7 TraceEnable Start/Stop Control Register, ETMTSSCR

The ETMTSSCR characteristics are:

Purpose	Specifies the single address comparators that hold the trace start and stop addresses.
Usage constraints	There are no usage constraints.
Configurations	Available in all PTM implementations.
Attributes	See the register summary in Table 3-16 on page 3-71.

Figure 3-17 shows the ETMTSSCR bit assignments.

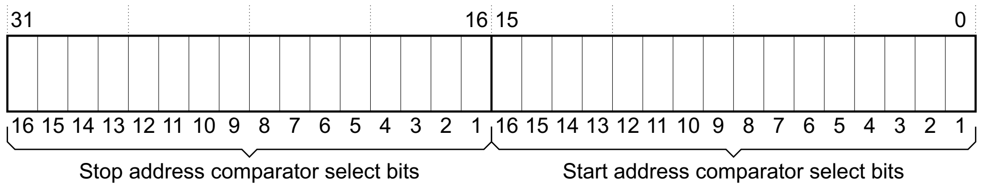


Figure 3-17 ETMTSSCR bit assignments

Table 3-22 shows the ETMTSSCR bit assignments.

Table 3-22 ETMTSSCR bit assignments

Bits	Description
[31:16]	When a bit is set to 1, it selects a single address comparator 16-1 as a stop address for the TraceEnable start/stop block. For example, if you set bit [16] to 1 it selects single address comparator 1 as a stop address.
[15:0]	When a bit is set to 1, it selects a single address comparator 16-1 as a start address for the TraceEnable start/stop block. For example, if you set bit [0] to 1 it selects single address comparator 1 as a start address.

3.16.8 TraceEnable Event Register, ETMTEEVR

The ETMTEEVR characteristics are:

Purpose	Defines the **TraceEnable** enabling event.
Usage constraints	There are no usage constraints
Configurations	Available in all PTM implementations.
Attributes	See the register summary in Table 3-16 on page 3-71.

Figure 3-18 shows the ETMTEEVR bit assignments.

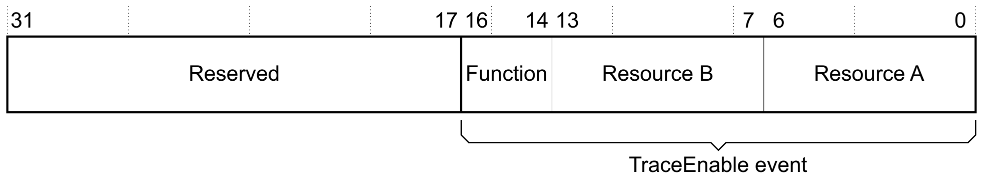


Figure 3-18 ETMTEEVR bit assignments

Table 3-23 shows the ETMTEEVR bit assignments.

Table 3-23 ETMTEEVR bit assignments

Bits	Description
[31:17]	Reserved, SBZP.
[16:0]	TraceEnable event. Subdivided as: **Function, bits [16:14]** Specifies the logical operation that combines the two resources that define the event. **Resource B, bits [13:7] and Resource A, bits [6:0]** Specify the two resources that are combined by the logical operation specified by the Function field. For more information see *Event resources and PTM events* on page 3-49.

Defining a PTM event on page 3-52 describes how you define a TraceEnable event.

3.16.9 TraceEnable Control Register, ETMTECR1

The ETMTECR1 characteristics are:

Purpose	• enables the start/stop logic • determines whether the resources specified in this register are used for include or exclude control • specifies the address range comparators used for include/exclude control.
Usage constraints	There are no usage constraints.
Configurations	Available in all PTM implementations.
Attributes	See the register summary in Table 3-16 on page 3-71.

Figure 3-19 shows the ETMTECR1 bit assignments.

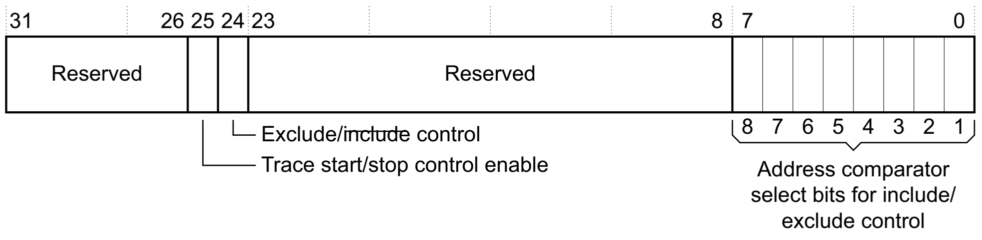

31
26 25 24 23
8 7
0
Reserved
Reserved
Exclude/~~include~~ control
Trace start/stop control enable
8 7 6 5 4 3 2 1
Address comparator select bits for include/ exclude control

Figure 3-19 ETMTECR1 bit assignments

Table 3-24 shows the ETMTECR1 bit assignments.

Table 3-24 ETMTECR1 bit assignments

Bits	Description
[31:26]	Reserved, SBZP.
[25]	Trace start/stop control enable. The possible values of this bit are: **0** Tracing is unaffected by the trace start/stop logic. **1** Tracing is controlled by the trace on and off addresses configured for the trace start/stop logic. See *The TraceEnable start/stop block* on page 3-36. The trace start/stop event resource, resource `0x5F`, is not affected by the value of this bit.
[24]	Include/exclude control. The possible values of this bit are: **0** Include. The specified address range comparators indicate the regions where tracing can occur. When outside this region tracing is prevented. **1** Exclude. The specified address range comparators indicate regions to be excluded from the trace. When outside an exclude region, tracing can occur.
[23:8]	Reserved, SBZP.
[7:0]	When a bit is set to 1, it selects an address range comparator, 8-1, for include/exclude control. For example, bit [0] set to 1 selects address range comparator 1.

Tracing all memory

To trace all memory:

- set bit [24] in the ETMTECR1 to 1
- set all other bits in the ETMTECR1 to 0
- set the ETMTEEVER to `0x6F` (TRUE).

This has the effect of excluding nothing, that is, tracing everything.

3.16.10 FIFOFULL Level Register, ETMFFLR

The ETMFFLR characteristics are:

Purpose	Defines the level below which the FIFO is considered full.
Usage constraints	There are no usage constraints.
Configurations	Available in all PTM implementations.
Attributes	See the register summary in Table 3-16 on page 3-71.

Figure 3-20 shows the ETMFFLR bit assignments.

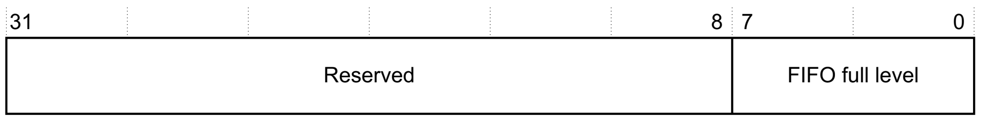


Figure 3-20 ETMFFLR bit assignments

Table 3-25 shows the ETMFFLR bit assignments.

Table 3-25 ETMFFLR bit assignments

Bits	Description
[31:8]	Reserved.
[7:0]	The number of bytes left in the FIFO, below which the **FIFOFULL** signal is asserted. For example, setting this value to 15 causes processor stalling, if enabled, when there are less than 15 free bytes in the FIFO.

The maximum valid value for this register is the size of the FIFO. This causes **FIFOFULL** to be asserted whenever the FIFO is not empty. Behavior is UNPREDICTABLE if the value 0 is written to this register and Stall processor is selected in the Main Control Register, `0x000`.

If a value larger than the FIFO size is written to the ETMFFLR, the FIFO size itself is selected, and is the value returned when the register is read.

You can use FIFOFULL control only if it is supported by both the PTM and the processor it is connected to:

- Bit [23] in the ETMCCR indicates when **FIFOFULL** is supported by the PTM. See *Configuration Code Register, ETMCCR* on page 3-78.
- Bit [8] in the ETMSCR indicates when **FIFOFULL** is supported by the processor. See *System Configuration Register, ETMSCR* on page 3-82.

You can use **FIFOFULL** control only when both these bits are set to 1.

If the PTM does not implement processor stalling, this register is reserved, and you cannot use the programmers model to determine the FIFO size. If a PTM does not implement processor stalling, bit [23] of the ETMCCR is RAZ. See *Configuration Code Register, ETMCCR* on page 3-78.

3.16.11 About the address comparator registers

The PTM has two registers defined for each of the single address comparators. The following sections describe these registers:

- *Address Comparator Value Registers, ETMACVRn*
- *Address Comparator Access Type Registers, ETMACTRn* on page 3-87.

When a pair of address comparator registers is used to define an address range, the upper Address Comparator Value Register must always contain an address that is greater than the lower Address Comparator Value Register. Otherwise, the behavior of the address range comparators is UNPREDICTABLE.

Address comparators on page 3-39 describes the use of the address comparator registers.

3.16.12 Address Comparator Value Registers, ETMACVR*n*

The ETMACVR characteristics are:

Purpose An ETMACVR holds an address for comparison.

Usage constraints Each ETMACVR is used with the corresponding ETMACTR. See *Address Comparator Access Type Registers, ETMACTRn* on page 3-87.

Configurations The number of ETMACR pairs:

- is IMPLEMENTATION DEFINED
- is specified by ETMCCR bits [3:0]
- can be zero.

See *Configuration Code Register, ETMCCR* on page 3-78.

Unimplemented ETMACVRs are RAZ/WI.

Attributes See the register summary in Table 3-16 on page 3-71.

Figure 3-21 shows the ETMACVR bit assignments.

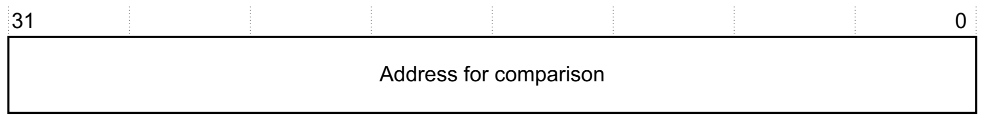


Figure 3-21 ETMACVR bit assignments

Table 3-26 shows the ETMACVR bit assignments.

Table 3-26 ETMACVR bit assignments

Bits	Description
[31:0]	Address value

Each ETMACVR has the same bit assignments.

3.16.13 Address Comparator Access Type Registers, ETMACTR*n*

The ETMACTR characteristics are:

Purpose Specifies the type of access, for example instruction or data, and other comparator configuration information for an address comparison.

Usage constraints Each ETMACTR is used with the corresponding ETMACVR. See *Address Comparator Value Registers, ETMACVRn* on page 3-86.

Configurations The number of ETMACTR pairs:

- is IMPLEMENTATION DEFINED
- is specified by ETMCCR bits [3:0].
- can be zero.

See *Configuration Code Register, ETMCCR* on page 3-78.

Unimplemented ETMACTRs are RAZ/WI.

Attributes See the register summary in Table 3-16 on page 3-71.

Figure 3-22 shows the ETMACTR bit assignments from PFTv1.1.

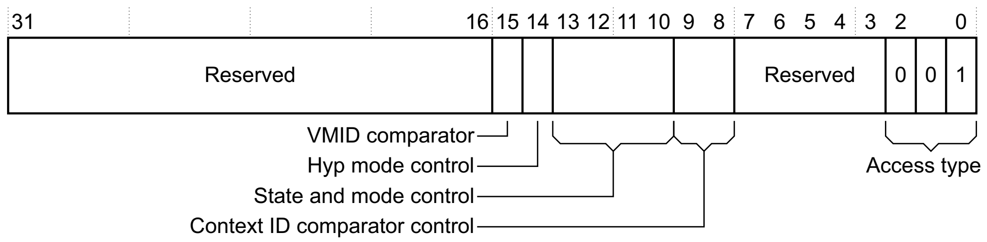


Figure 3-22 ETMACTR bit assignments from PFTv1.1

Figure 3-23 on page 3-88 shows the ETMACTR bit assignments for PFTv1.0.

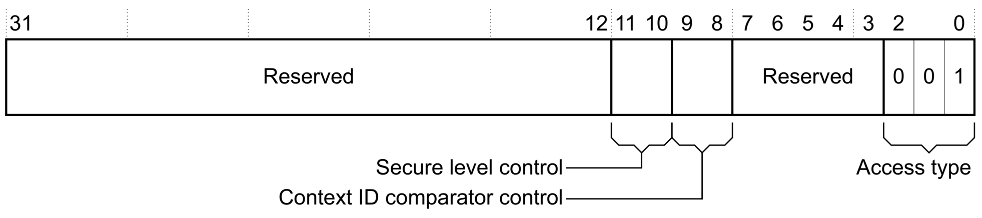


Figure 3-23 ETMACTR bit assignments, PFTv1.0

Table 3-27 shows the ETMACTR bit assignments.

Table 3-27 ETMACTR bit assignments

Bits	Description
[31:16]	Reserved, SBZP.
[15]	For PFTv1.0 this bit is Reserved, SBZP.
	From PFTv1.1 this bit sets the *Virtual Machine ID* (VMID) comparison enable, if the processor implements the Virtualization Extensions.[a] A value of 1 means that the address comparator matches only if the current VMID matches the value stored in the ETMVMIDCVR. See *VMID Comparator Value Register, ETMVMIDCVR* on page 3-109. This bit is reserved, RAZ if the processor does not implement the Virtualization extensions.
[14]	For PFTv1.0 this bit is Reserved, SBZP.
	From PFTv1.1 this bit sets the Hyp mode comparison enable, if the processor implements the Virtualization Extensions.[a] A value of 1 means that the address comparator also matches if the processor is operating in Hyp mode. See *Virtualization* on page 2-29. This bit is reserved, RAZ if the processor does not implement the Virtualization extensions.
[13:10]	For PFTv1.0: • bits [13:12] are Reserved, SBZP • bits [11:10] are described elsewhere in this table.
	From PFTv1.1 these bits set state and mode comparison control. The assignment of these bits is: **Bit [13, 11]** Non-secure state comparison control. **Bit [12, 10]** Secure state comparison control. For each pair of bits, the encoding is: **b00** Match in all modes in this state. **b01** Do not match in any modes in this state. **b10** Match in all modes except User mode in this state. **b11** Match only in User mode in this state. If the processor does not implement the Security Extensions, bits [13, 11] are reserved, RAZ/WI. See *Filtering by state and mode, from PFTv1.1* on page 3-89
[11:10]	Security level control. The permitted values of this field are: **b00** Security level ignored. **b01** Match only if in Non-secure state. **b10** Match only if in Secure state. The value of b11 is reserved and must not be used. This field is available only if the connected processor implements the Security Extensions. If the Security Extensions are not implemented then these bits are RAZ/WI. This description is valid only for PFTv1.0. For PFTv1.1 and later, see the description of bits [13:10] in this table.

Table 3-27 ETMACTR bit assignments (continued)

Bits	Description
[9:8]	Context ID comparator control. The permitted values of this field are: **b00** Ignore Context ID comparator. **b01** Address comparator matches only if Context ID comparator value 1 matches. **b10** Address comparator matches only if Context ID comparator value 2 matches. **b11** Address comparator matches only if Context ID comparator value 3 matches.
[7:3]	Reserved, SBZP.
[2:0]	Access type. This field is read-only, and ignores writes. It returns the value: **b001** Instruction execute. **Note** Other ARM trace architectures permit address matching conditional on the type of access and support different values for this field.

a. If bit [26] of the ETMCCER is zero, it indicates that Virtualization support is not implemented. See *Virtualization* on page 2-29 for more information.

Each of ETMACTR has the same bit assignments.

Filtering by state and mode, from PFTv1.1

From PFTv1.1 a PTM can base address matching on:

- any mode or state
- any mode in Secure state
- any mode in Non-secure state.

ETMACTR[13:10], the State and mode control field, defines the conditions for an address match, as:

- Table 3-28 shows for an implementation that includes the Security Extensions
- Table 3-29 on page 3-90 shows for an implementation that does not include the Security Extensions.

In addition, when the Virtualization Extensions are implemented, ETMACTR[15:14] control matching in Hyp mode and on VMID, see Table 3-29 on page 3-90. For these bits:

- When bit [15] VMID comparison enable is set, no match is recognized unless the current VMID matches the value stored in the ETMVMIDCVR.
- When bit [14] Hyp mode comparison enable is set, Hyp mode is considered as an additional criterion with the modes in Table 3-28. That is, operation in Hyp mode always produces a match.

Table 3-28 Address comparator filtering by state and mode, PFTv1.1 with Security Extensions

Bits [13:10]	Secure state		Non-secure state	
	Kernel modes	User mode	Kernel modes	User mode
b0000	Yes	Yes	Yes	Yes
b0001	-	-	Yes	Yes
b0010	Yes	Yes	-	-
b0011	-	-	-	-
b0100	Yes	-	Yes	Yes

Table 3-28 Address comparator filtering by state and mode, PFTv1.1 with Security Extensions

Bits [13:10]	Secure state		Non-secure state	
	Kernel modes	User mode	Kernel modes	User mode
b0101	-	Yes	Yes	Yes
b0110	Yes	-	-	-
b0111	-	Yes	-	-
b1000	Yes	Yes	Yes	-
b1001	-	-	Yes	-
b1010	Yes	Yes	-	Yes
b1011	-	-	-	Yes
b1100	Yes	-	Yes	-
b1101	-	Yes	Yes	-
b1110	Yes	-	-	Yes
b1111	-	Yes	-	Yes

Table 3-29 Address comparator filtering by state and mode, PFTv1.1, no Security Extensions

Bits [13:10][a]	Privileged modes	User mode	Matches in
b0000	Yes	Yes	All modes
b0001	-	-	Never matches
b0100	Yes	-	Privileged modes only
b0101	-	Yes	User mode only

a. Bits [13,11] are RAZ/WI if the processor does not implement the Security Extensions.

Access types for address range comparators

If you are using two address comparators as an address range comparator, the access type must be identical for each, otherwise the behavior of the comparator is UNPREDICTABLE. The only exceptions to this are:

- Bits [6:5] must be set only for the first comparator in the pair. These bits control data value comparisons.
- The special case where the range includes the address `0xFFFFFFFF`. See *Selecting a range to include address 0xFFFFFFFF*.

Note

This information is also given in *Address comparators* on page 3-39.

Selecting a range to include address 0xFFFFFFFF

Ranges are defined to be exclusive of the upper address, so if you specify an upper address of `0xFFFFFFFF`, only addresses up to and including `0xFFFFFFFE` match. To specify a data address to include `0xFFFFFFFF`, configure the upper address comparator as follows:

- Set the comparator value in the ETMACVR to `0xFFFFFFFF`
- set the size mask, bits [4:3] of the ETMACTR, to b11.

This is the only case where the size mask can be different between the two address comparators of an address range comparator.

For more information, see *Address comparators* on page 3-39.

3.16.14 About the counter registers

There are between zero and four 16-bit counters. Four registers are used to define the operation of each counter. The following sections describe the counter registers:

- *Counter Reload Value Registers, ETMCNTRLDVRn*
- *Counter Enable Event Registers, ETMCNTENRn* on page 3-92
- *Counter Reload Event Registers, ETMCNTRLDEVRn* on page 3-93
- *Counter Value Registers, ETMCNTVRn* on page 3-94.

Table 3-30 summarizes the Counter registers:

Table 3-30 Summary of Counter registers

Counter	Counter Registers:			
	Reload Value[a]	Enable[a]	Reload Event[a]	Value[a]
1	`0x050`	`0x054`	`0x058`	`0x05C`
2	`0x051`	`0x055`	`0x059`	`0x05D`
3	`0x052`	`0x056`	`0x05A`	`0x05E`
4	`0x053`	`0x057`	`0x05B`	`0x05F`

a. Register numbers are listed. Where registers are accessed in a memory-mapped scheme, the register offset is 4×(Register number).

See *PTM counters* on page 3-56 for more information about the counter registers.

Reduced function counter, from PFTv1.1

From PFTv1.1, counter 1 can be implemented as a counter with reduced functionality. The reduced function counter has the following attributes:

- 16-bit reload value, configured by ETMCNTRLDVR1.
- Decrements on every cycle. ETMCNTENR1 is Reserved.
- Reloads every time the counter reaches zero. ETMCNTRLDEVR1 is Reserved.
- The counter value cannot be read. ETMCNTVR1 is Reserved.
- The counter always starts at the reload value when the PTM programming bit is cleared.
- The value cannot be saved or restored.

Bit [27] of the ETMCCER identifies whether counter 1 is a reduced function counter. See *Configuration Code Extension Register, ETMCCER* on page 3-103.

If more than 1 counter is implemented, the counters other than counter 1 are always full function counters.

3.16.15 Counter Reload Value Registers, ETMCNTRLDVR*n*

The ETMCNTRLDVR characteristics are:

Purpose	An ETMCNTRLDVR specifies the starting value of the corresponding counter.
Usage constraints	Each ETMCNTRLDVR is used with a corresponding ETMCNTENR, ETMCNTRLDEVR, and ETMCNTVR. See *About the counter registers*.

Configurations The number of ETMCNTRLDVRs:

- is IMPLEMENTATION DEFINED
- is specified by ETMCCR bits [15:13]
- can be zero.

See *Configuration Code Register, ETMCCR* on page 3-78.

Unimplemented ETMCNTRLDVRs are RAZ/WI.

Attributes See the register summary in Table 3-16 on page 3-71.

Figure 3-24 shows the ETMCNTRLDVR bit assignments.

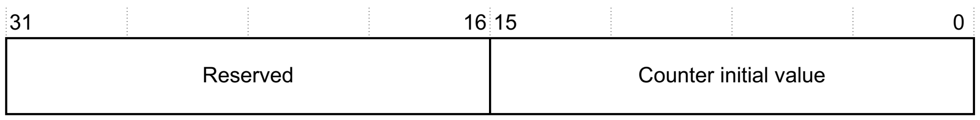


Figure 3-24 ETMCNTRLDVR bit assignments

Table 3-31 shows the ETMCNTRLDVR bit assignments.

Table 3-31 ETMCNTRLDVR bit assignments

Bits	Description
[31:16]	Reserved
[15:0]	Initial count

Each ETMCNTRLDVR has the same bit assignments.

When the programming bit is cleared, the counter is loaded with the value in the ETMCNTRLDVR register unless the ETMCNTVR has been programmed while the programming bit is set.

The counter is reloaded with the value in ETMCNTRLDVR whenever the corresponding counter reload event, specified by ETMCNTRLDEVR, is active.

3.16.16 Counter Enable Event Registers, ETMCNTENR*n*

The ETMCNTENR characteristics are:

Purpose An ETMCNTENR:

- defines the event that enables the corresponding counter
- can be used to configure the counter for continuous operation.

Usage constraints Each ETMCNTENR is used with a corresponding ETMCNTRLDVR, ETMCNTRLDEVR, and ETMCNTVR. See *About the counter registers* on page 3-91.

Configurations The number of ETMCNTENRs:

- is IMPLEMENTATION DEFINED
- is specified by ETMCCR bits [15:13]
- can be zero.

See *Configuration Code Register, ETMCCR* on page 3-78.

Unimplemented ETMCNTENRs are RAZ/WI.

Attributes See the register summary in Table 3-16 on page 3-71.

Figure 3-25 on page 3-93 shows the ETMCNTENR bit assignments.

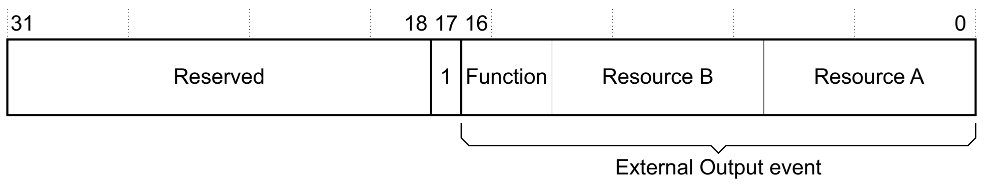


Figure 3-25 ETMCNTENR bit assignments

Table 3-32 shows the ETMCNTENR bit assignments.

Table 3-32 ETMCNTENR bit assignments

Bits	Description
[31:18]	Reserved, SBZP.
[17]	Reserved, RAO/WI. **Note** This bit is RAO/WI for consistency with previous ARM trace architectures.
[16:0]	Count enable event. Subdivided as: **Function, bits [16:14]** Specifies the logical operation that combines the two resources that define the event. **Resource B, bits [13:7] and Resource A, bits [6:0]** Specify the two resources that are combined by the logical operation specified by the Function field. For more information see *Event resources and PTM events* on page 3-49. To configure a continuous counter, program the function field as b000, to select Resource A, and the Resource A field to select the always TRUE resource. See *The PTM event resources* on page 3-49.

Each of the Counter Enable Event Registers has the same bit assignments.

Defining a PTM event on page 3-52 describes how you define a counter enable event.

3.16.17 Counter Reload Event Registers, ETMCNTRLDEVR*n*

The ETMCNTRLDEVR characteristics are:

Purpose	An ETMCNTRLDEVR defines the event that causes the corresponding counter to be reloaded with the value held in the corresponding ETMCNTRLDVR.
Usage constraints	Each ETMCNTRLDEVR is used with a corresponding ETMCNTRLDVR, ETMCNTENR, and ETMCNTVR. See *About the counter registers* on page 3-91.
Configurations	The number of ETMCNTRLDEVRs: • is IMPLEMENTATION DEFINED • is specified by ETMCCR bits [15:13] • can be zero. See *Configuration Code Register, ETMCCR* on page 3-78. Unimplemented ETMCNTRLDEVRs are RAZ/WI.
Attributes	See the register summary in Table 3-16 on page 3-71.

Figure 3-26 on page 3-94 shows the ETMCNTRLDEVR bit assignments.

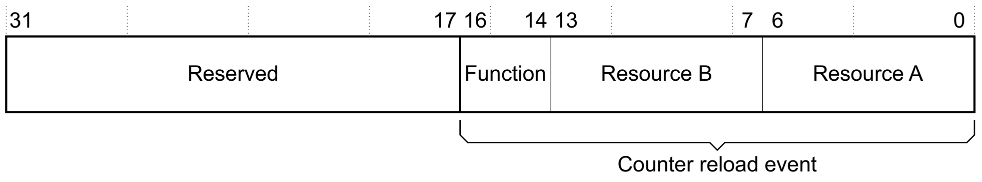


Figure 3-26 ETMCNTRLDEVR bit assignments

Table 3-33 shows the ETMCNTRLDEVR bit assignments.

Table 3-33 ETMCNTRLDEVR bit assignments

Bits	Description
[31:17]	Reserved
[16:0]	Count reload event. Subdivided as: **Function, bits [16:14]** Specifies the logical operation that combines the two resources that define the event. **Resource B, bits [13:7] and Resource A, bits [6:0]** Specify the two resources that are combined by the logical operation specified by the Function field. For more information see *Event resources and PTM events* on page 3-49.

Each ETMCNTRLDEVR has the same bit assignments.

Defining a PTM event on page 3-52 describes how you define a counter reload event.

3.16.18 Counter Value Registers, ETMCNTVR*n*

The ETMCNTVR characteristics are:

Purpose An ETMCNTVR holds the current value of the corresponding counter.

Usage constraints Each ETMCNTVR is used with a corresponding ETMCNTRLDVR, ETMCNTENR, and ETMCNTRLDEVR. See *About the counter registers* on page 3-91.

Configurations The number of ETMCNTVRs:
- is IMPLEMENTATION DEFINED
- is specified by ETMCCR bits [15:13]
- can be zero.

See *Configuration Code Register, ETMCCR* on page 3-78.

Unimplemented ETMCNTVRs are RAZ/WI.

Attributes See the register summary in Table 3-16 on page 3-71.

Figure 3-27 shows the ETMCNTVR bit assignments.

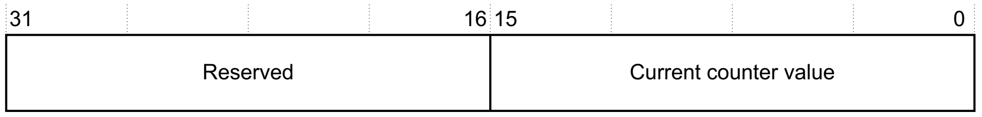


Figure 3-27 ETMCNTVR bit assignments

Table 3-34 shows the ETMCNTVR bit assignments.

Table 3-34 ETMCNTVR bit assignments

Bits	Description
[31:16]	Reserved.
[15:0]	Current counter value. See *Programming bit and associated state* on page 3-70 for information on programming this register.

Each ETMCNTVR has the same bit assignments.

If the ETMCNTVR is written while the programming bit is set, the counter does not load the value from the ETMCNTRLDVR but retains the value written to the ETMCNTVR.

3.16.19 About the sequencer registers

A PTM implementation can include a sequencer. If it does, software controls the sequencer by defining the events that cause the sequencer to move between its different states.

Each sequencer state transition event has its own register, and these registers are programed to control the state transitions. An additional register holds the current state of the sequencer. Table 3-35 lists the sequencer registers, with the register number and address offset of each register.

When programming the sequencer, you must program a valid encoding into each ETMSQ*ab*EVR, otherwise the behavior of the sequencer is UNPREDICTABLE. For example, if you want the sequencer only to perform transitions between states 1 and 2, you must:

- program ETMSQ12EVR and `ETMSQ21EVR` to control transitions between these two states
- program ETMSQ23EVR and `ETMSQ13EVR` to ensure that state 3 is never entered
- program ETMSQ31EVR and `ETMSQ32EVR`, typically with the value `0x0000406F`, to ensure these transitions never occur.

If the sequencer must be in a particular state when the Programming bit is cleared, the Sequencer State Register (`0x067`) should be written after programming the Sequencer State Transition Event Registers to ensure this value is used. If the Sequencer State Register is not written then the sequencer state resets to 1 when the ETM Programming bit is cleared.

Programming any of the Sequencer State Transition Event Registers when the Programming bit is set to 1 resets the sequencer to State 1. This behavior is IMPLEMENTATION DEFINED.

Table 3-35 Sequencer register allocation

Register		Short name	Description
Number	Offset[a]		
`0x060`	`0x180`	ETMSQ12EVR	State 1 to State 2 Transition Event Register[b]
`0x061`	`0x184`	ETMSQ21EVR	State 2 to State 1 Transition Event Register[b]
`0x062`	`0x188`	ETMSQ23EVR	State 2 to State 3 Transition Event Register[b]
`0x063`	`0x18C`	ETMSQ31EVR	State 3 to State 1 Transition Event Register[b]
`0x064`	`0x190`	ETMSQ32EVR	State 3 to State 2 Transition Event Register[b]

Table 3-35 Sequencer register allocation (continued)

Register Number	Register Offset[a]	Short name	Description
0x065	0x194	ETMSQ13EVR	State 1 to State 3 Transition Event Register[b]
0x066	0x198	-	Reserved
0x067	0x19C	ETMSQR	Current Sequencer State Register. See *Current Sequencer State Register, ETMSQR* on page 3-97

a. Register offset where the registers are accessed in a memory-mapped scheme. The register offset is always 4×(Register number).

b. Each of the Sequencer State Transition Event Registers has the same bit assignments. See *Sequencer State Transition Event Registers, ETMSQabEVR* for details.

3.16.20 Sequencer State Transition Event Registers, ETMSQ*ab*EVR

The ETMSQ*ab*EVR characteristics are:

Purpose ETMSQ*ab*EVR defines the event that causes the sequencer to transition from state a to state b.

Usage constraints There are no usage constraints.

Configurations Whether the PTM includes a sequencer is IMPLEMENTATION DEFINED, and is specified by ETMCCR bit [16]. See *Configuration Code Register, ETMCCR* on page 3-78.

If the PTM does not include a sequencer the ETMSQ*ab*EVRs are RAZ/WI.

Attributes See the register summary in Table 3-16 on page 3-71.

Figure 3-28 shows the ETMSQ*ab*EVR bit assignments.

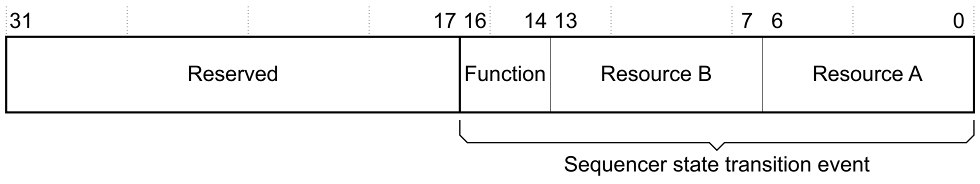


Figure 3-28 ETMSQ*ab*EVR bit assignments

Table 3-36 shows the ETMSQ*ab*EVR bit assignments.

Table 3-36 ETMSQ*ab*EVR bit assignments

Bits	Description
[31:17]	Reserved
[16:0]	Sequencer state transition event. Subdivided as: **Function, bits [16:14]** Specifies the logical operation that combines the two resources that define the event. **Resource B, bits [13:7] and Resource A, bits [6:0]** Specify the two resources that are combined by the logical operation specified by the Function field. For more information see *Event resources and PTM events* on page 3-49.

Each ETMSQ*ab*EVR has the same bit assignments. Table 3-35 on page 3-95 shows all of the ETMSQ*ab*EVRs.

Defining a PTM event on page 3-52 describes how you define a sequencer state transition event.

3.16.21 Current Sequencer State Register, ETMSQR

The ETMSQR characteristics are:

Purpose	Holds the current state of the sequencer.
Usage constraints	There are no usage constraints.
Configurations	Whether the PTM includes a sequencer is IMPLEMENTATION DEFINED, and is specified by ETMCCR bit [16]. See *Configuration Code Register, ETMCCR* on page 3-78. If the PTM does not include a sequencer ETMSQR is RAZ/WI.
Attributes	See the register summary in Table 3-16 on page 3-71.

Figure 3-29 shows the ETMSQR bit assignments.

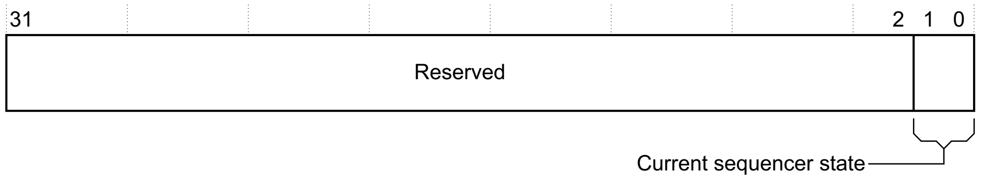


Figure 3-29 ETMSQR bit assignments

Table 3-37 shows the ETMSQR bit assignments.

Table 3-37 ETMSQR bit assignments

Bits	Description
[31:2]	Reserved
[1:0]	The permitted values of this field are: **b00** Sequencer currently in state 1. **b01** Sequencer currently in state 2. **b10** Sequencer currently in state 3. The value of b11 is reserved and must not be used.

See *Programming bit and associated state* on page 3-70 for information on programming this register.

3.16.22 External Output Event Registers, ETMEXTOUTEVR*n*

The ETMEXTOUTEVR characteristics are:

Purpose An ETMEXTOUTEVR defines the event that controls the corresponding external output.

Usage constraints There are no usage constraints.

Configurations The number of external outputs:

- is IMPLEMENTATION DEFINED
- is specified by ETMCCR bits [22:20]
- can be zero.

See *Configuration Code Register, ETMCCR* on page 3-78.

Unimplemented ETMEXTOUTEVRs are RAZ/WI.

Attributes See the register summary in Table 3-16 on page 3-71.

Figure 3-30 on page 3-98 shows the ETMEXTOUTEVR bit assignments.

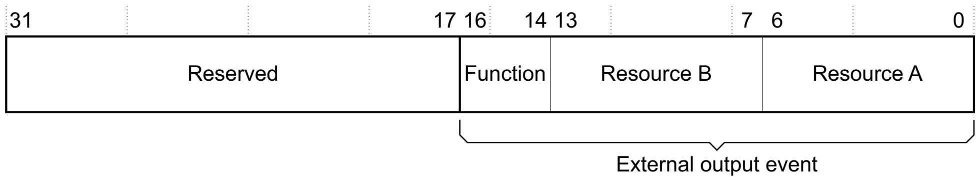


Figure 3-30 ETMEXTOUTEVR bit assignments

Table 3-38 shows the ETMEXTOUTEVR bit assignments.

Table 3-38 ETMEXTOUTEVR bit assignments

Bits	Description
[31:17]	Reserved
[16:0]	External output event. Subdivided as: **Function, bits [16:14]** Specifies the logical operation that combines the two resources that define the event. **Resource B, bits [13:7] and Resource A, bits [6:0]** Specify the two resources that are combined by the logical operation specified by the Function field. For more information see *Event resources and PTM events* on page 3-49.

Each ETMEXTOUTEVR has the same bit assignments.

Defining a PTM event on page 3-52 describes how you define an external output event.

3.16.23 About the Context ID comparator registers

A PTM can implement up to three Context ID comparators. The PTM implements a value register for each Context ID comparator, and a single mask register that applies to all of the Context ID comparators. Table 3-39 shows these registers.

Table 3-39 Summary of the Context ID comparator registers

Description	Short name	Register number	Offset[a]
Context ID Comparator Value 1 Register	ETMCIDCVR1	`0x06C`	`0x1B0`
Context ID Comparator Value 2 Register	ETMCIDCVR2	`0x06D`	`0x1B4`
Context ID Comparator Value 3 Register	ETMCIDCVR3	`0x06E`	`0x1B8`
Context ID Comparator Mask Register	ETMCIDCMR	`0x06F`	`0x1BC`

a. Register offset where the registers are accessed in a memory-mapped scheme. The register offset is always 4×(Register number).

The following sections describe these registers:

- *Context ID Comparator Value Registers, ETMCIDCVRn*
- *Context ID Comparator Mask Register, ETMCIDCMR* on page 3-99.

3.16.24 Context ID Comparator Value Registers, ETMCIDCVR*n*

The ETMCIDCVR characteristics are:

Purpose	An ETMCIDCVR holds a 32-bit Context ID value for comparison.
Usage constraints	There are no usage constraints.

Configurations The number of Context ID comparators:

- is IMPLEMENTATION DEFINED
- is specified by ETMCCR bits [25:24]
- can be zero.

See *Configuration Code Register, ETMCCR* on page 3-78.

Unimplemented ETMCIDCVRs are RAZ/WI.

Attributes See the register summary in Table 3-16 on page 3-71.

Figure 3-31 shows the ETMCIDCVR bit assignments.

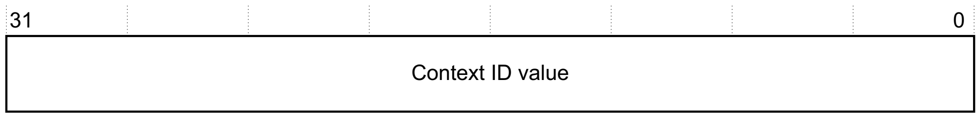


Figure 3-31 ETMCIDCVR bit assignments

Table 3-40 shows the ETMCIDCVR bit assignments.

Table 3-40 ETMCIDCVR bit assignments

Bits	Description
[31:0]	Context ID value

Each ETMCIDCVR has the same bit assignments.

3.16.25 Context ID Comparator Mask Register, ETMCIDCMR

The ETMCIDCMR characteristics are:

Purpose Holds a 32-bit mask for use for all Context ID comparisons.

Usage constraints There are no usage constraints.

Configurations The number of Context ID comparators:

- is IMPLEMENTATION DEFINED
- is specified by ETMCCR bits [25:24]
- can be zero.

See *Configuration Code Register, ETMCCR* on page 3-78.

If the PTM does not implement any Context ID comparators then the ETMCIDCMR is RAZ/WI.

Attributes See the register summary in Table 3-16 on page 3-71.

Figure 3-32 shows the ETMCIDCMR bit assignments.

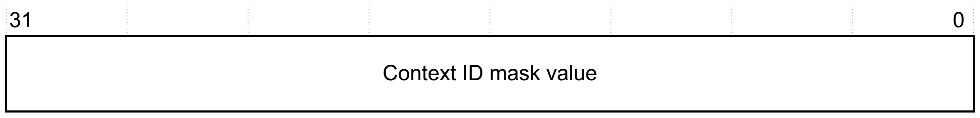


Figure 3-32 ETMCIDCMR bit assignments

Table 3-41 shows the ETMCIDCMR bit assignments.

Table 3-41 ETMCIDCMR bit assignments

Bits	Description
[31:0]	Context ID mask value

The same mask is used for all of the Context ID comparators. When a Context ID mask bit is set to 1, the corresponding bit in the Value Register is disregarded in the comparison and should be zero.

3.16.26 IMPLEMENTATION SPECIFIC registers, ETMIMPSPEC0 to ETMIMPSPEC7

Register numbers `0x70`-`0x77` in the register map are reserved for up to eight IMPLEMENTATION SPECIFIC registers. Even when a PTM does not implement these registers, IMPLEMENTATION SPECIFIC Register 0, register number `0x70`, must be partially defined, so that a debugger can implement a general mechanism for detecting IMPLEMENTATION SPECIFIC extensions.

See Table 3-16 on page 3-71 for details of access to this register area.

IMPLEMENTATION SPECIFIC Register 0

The IMPLEMENTATION SPECIFIC Register 0 characteristics are:

Purpose	Shows the presence of any IMPLEMENTATION SPECIFIC features, and enables any features that are provided.
Usage constraints	There are no usage constraints.
Configurations	Available in all PTM implementations.
Attributes	See the register summary in Table 3-16 on page 3-71.

Figure 3-33 shows the IMPLEMENTATION SPECIFIC Register 0 default bit assignments.

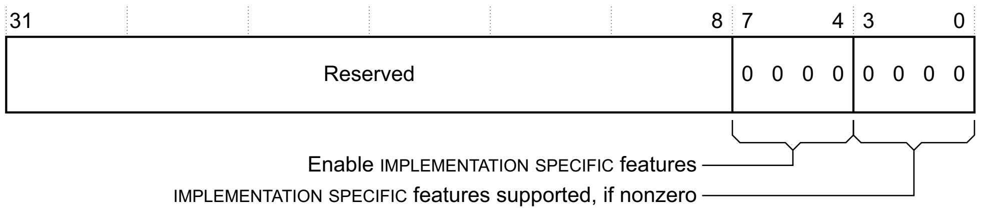

31
8
7
4
3
0
Reserved
0 0 0 0
0 0 0 0
Enable IMPLEMENTATION SPECIFIC features
IMPLEMENTATION SPECIFIC features supported, if nonzero

Figure 3-33 IMPLEMENTATION SPECIFIC Register 0 bit assignments

Table 3-42 shows the IMPLEMENTATION SPECIFIC Register 0 default bit assignments.

Table 3-42 IMPLEMENTATION SPECIFIC Register 0 bit assignments

Bits	Access	Description
[31:8]	-	Reserved
[7:4]	RW	Enable IMPLEMENTATION SPECIFIC extensions. When these bits are b0000 the PTM must behave as if the IMPLEMENTATION SPECIFIC extensions are not implemented. The behavior of the PTM is IMPLEMENTATION DEFINED when these bits are set to any value other than b0000. A PTM reset sets these bits to b0000.
[3:0]	RO	If this field is b0000 then the PTM does not support any IMPLEMENTATION SPECIFIC extensions. Other values are for use only as permitted in writing by ARM Limited.

3.16.27 Synchronization Frequency Register, ETMSYNCFR

The ETMSYNCFR characteristics are:

Purpose	Holds the trace synchronization frequency value.
Usage constraints	There are no usage constraints.
Configurations	Available in all PTM implementations.
Attributes	See the register summary in Table 3-16 on page 3-71.

Figure 3-34 shows the ETMSYNCFR bit assignments, with the default value of the register.

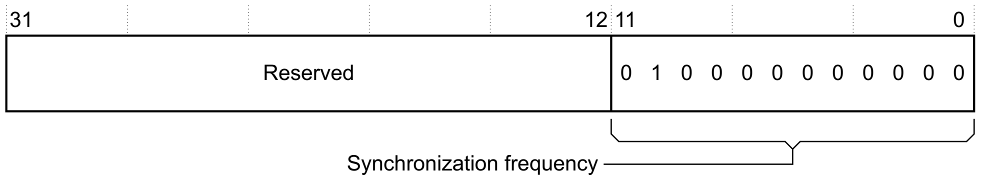


Figure 3-34 ETMSYNCFR bit assignments

Table 3-43 shows the ETMSYNCFR bit assignments.

Table 3-43 ETMSYNCFR bit assignments

Bits	Description
[31:12]	Reserved
[11:0]	Synchronization frequency. A PTM reset sets this field to the default value of 1024.

If you write a value of zero to this register the PTM disables the synchronization frequency counter, and does not generate periodic synchronization requests. This does not affect the other sources of periodic synchronization requests. For more information see *Periodic synchronization* on page 4-191.

Note

The Synchronization Frequency Register must be programmed to a value greater than the size of the FIFO, or to zero.

An implementation might not implement the bottom bits of this register, because of limitations in the accuracy of the synchronization frequency. In this case, a value read from this register might be different from the value written to it.

3.16.28 ID Register, ETMIDR

The ETMIDR characteristics are:

Purpose
- Holds the PTM architecture variant.
- Defines the programmers model for the PTM.

Usage constraints	There are no usage constraints.
Configurations	Available in all PTM implementations.
Attributes	See the register summary in Table 3-16 on page 3-71.

Figure 3-35 on page 3-102 shows the ETMIDR bit assignments.

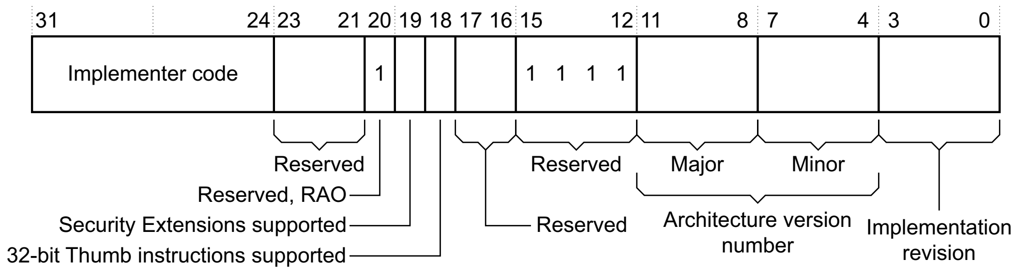


Figure 3-35 ETMIDR bit assignments

Table 3-44 shows the ETMIDR bit assignments.

Table 3-44 ETMIDR bit assignments

Bits	Description
[31:24]	Implementer code. The following codes are defined[a], and all other values are reserved by ARM Limited: `0x41` ASCII code for A, indicating ARM Limited. `0x44` ASCII code for D, indicating Digital Equipment Corporation. `0x4D` ASCII code for M, indicating Motorola, Freescale Semiconductor Inc. `0x51` ASCII code for Q, indicating QUALCOMM Inc. `0x56` ASCII code for V, indicating Marvell Semiconductor Inc. `0x69` ASCII code for i, indicating Intel Corporation.
[23:21]	Reserved, RAZ.
[20]	Reserved, This bit is RAO.
[19]	Support for Security Extensions. The possible values of this bit are: **0** The PTM behaves as if the processor is in Secure state at all times. **1** The ARM architecture Security Extensions are implemented by the processor.
[18]	Support for 32-bit Thumb instructions. The possible values of this bit are: **0** A 32-bit Thumb instruction is traced as two instructions, and exceptions might occur between these two instructions. **1** A 32-bit Thumb instruction is traced as a single instruction. See *32-bit Thumb instructions* on page 4-200 for more information.
[17:16]	Reserved, RAZ.
[15:12]	Reserved. This field reads as b1111.
[11:8]	Major architecture version number. See *Product revision status* on page viii. A value of b0011 is used for PFTv1.

[7:4]	Minor architecture version number. See *Product revision status* on page viii. The possible values for this field are:
	b0000 = PFTv1.0
	b0001 = PFTv1.1.
	All other values are Reserved.

[3:0]	Implementation revision. This matches the Revision field in the Peripheral ID2 register. See *Peripheral ID2 Register, ETMPIDR2* on page 3-126.

a. The Implementer code list applies to all ARM processor and trace architectures. This list does not indicate the implementer of PFT architectures.

3.16.29 Configuration Code Extension Register, ETMCCER

The ETMCCER characteristics are:

Purpose	This register holds PTM configuration information additional to that in the ETMCCR. See *Configuration Code Register, ETMCCR* on page 3-78.
Usage constraints	Software uses this register with the ETMCCR.
Configurations	Available in all PTM implementations.
Attributes	See the register summary in Table 3-16 on page 3-71.

Figure 3-36 shows the ETMCCER bit assignments for PFTv1.1.

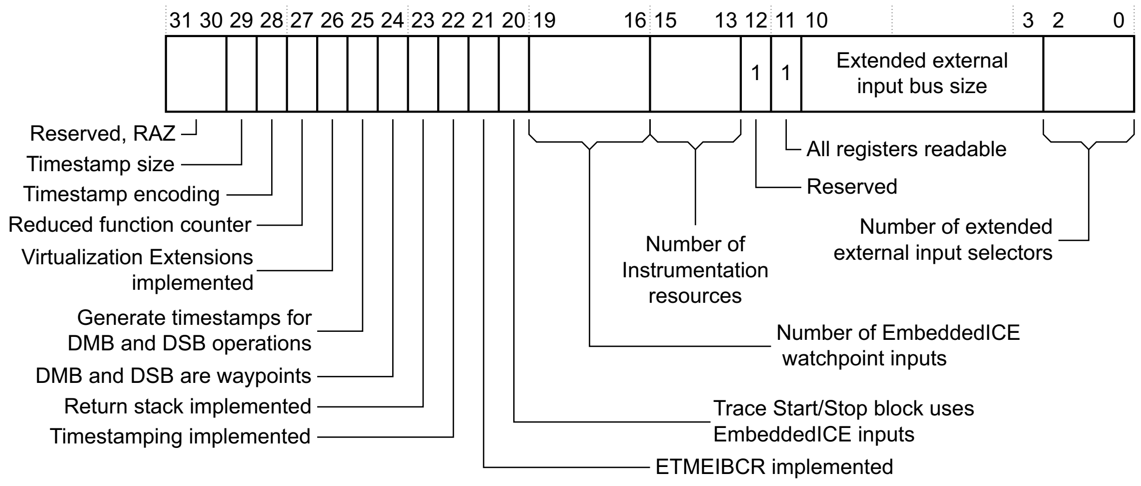

31 30 29 28 27 26 25 24 23 22 21 20 19 16 15 13 12 11 10 3 2 0
1
1
Extended external input bus size
Reserved, RAZ
Timestamp size
Timestamp encoding
Reduced function counter
Virtualization Extensions implemented
Generate timestamps for DMB and DSB operations
DMB and DSB are waypoints
Return stack implemented
Timestamping implemented
ETMEIBCR implemented
Trace Start/Stop block uses EmbeddedICE inputs
Number of EmbeddedICE watchpoint inputs
Number of Instrumentation resources
All registers readable
Reserved
Number of extended external input selectors

Figure 3-36 ETMCCER bit assignments

Table 3-45 shows the ETMCCER bit assignments.

Table 3-45 ETMCCER bit assignments

Bits	Description
[31:30]	Reserved, RAZ.
[29]	For PFTv1.0 this bit is Reserved, RAZ.
	Timestamp size. From PFTv1.1 this bit specifies the maximum size, in bits, of the timestamp for the timestamp packet. **0** Maximum size of the timestamp is 48 bits. **1** Maximum size of the timestamp is 64 bits.
[28]	For PFTv1.0 this bit is Reserved, RAZ.
	Timestamp encoding From PFTv1.1 this bit specifies the encoding used for the timestamp value in the timestamp packet. **0** The timestamp is Gray coded. **1** The timestamp is encoded as a natural binary number. See *Encoding of the timestamp value* on page 4-183.

Table 3-45 ETMCCER bit assignments (continued)

Bits	Description
[27]	For PFTv1.0 this bit is Reserved, RAZ.
	Reduced function counter From PFTv1.1 this bit specifies whether counter 1 is implemented as a reduced function counter. **0** All counters are implemented as full-function counters. **1** Counter 1 is implemented as a reduced-function counter. See *Reduced function counter, from PFTv1.1* on page 3-91.
[26]	For PFTv1.0 this bit is Reserved, RAZ.
	Virtualization Extensions implemented. From PFTv1.1, this bit is 1 if the Virtualization Extensions are implemented, and 0 if they are not implemented. When this bit is 1: • The ETMVMIDCVR is implemented. See *VMID Comparator Value Register, ETMVMIDCVR* on page 3-109. • The VMID trace enable bit in the ETMCR is writable. See *Main Control Register, ETMCR* on page 3-75. • The VMID Comparator bit in every ETMACTR is implemented. See *Address Comparator Access Type Registers, ETMACTRn* on page 3-87. • The VMID Comparator Event Resource encoding is permitted.
[25]	Generate timestamps for DMB and DSB operations.[a] The meaning of the possible values of this bit are: **0** The PTM never generates a timestamp for a DMB or DSB operation. **1** If timestamping is enabled the PTM generates a timestamp for any DMB or DSB operation.
[24]	DMB and DSB operations are waypoint instructions.[a] The meaning of the possible values of this bit are: **0** The PTM never treats a DMB or DSB operation as a waypoint instruction, and the decompressor must not treat a DMB or DSB as a waypoint instruction. **1** The PTM treats any DMB or DSB operation as a waypoint instruction.
[23]	Return stack implemented. This bit is 1 if the stack is implemented, and 0 if it is not implemented. When this bit is 1, the Return stack enable bit in the ETMCR is writable.
[22]	Timestamping implemented. This bit is 1 if timestamping is implemented, and 0 if it is not implemented. When this bit is 1: • the ETMTSEVR is implemented • the Timestamp enable bit in the ETMCR is writable.
[21]	EmbeddedICE Behavior Control Register implemented. This bit is 1 if the register is implemented, and 0 if it is not implemented.
[20]	Trace Start/Stop block can use EmbeddedICE watchpoint comparator inputs. This bit is 1 if the Trace Start/Stop block can used these inputs, and is 0 otherwise.
[19:16]	Number of EmbeddedICE watchpoint comparator inputs implemented. This field can take any value from b0000 (0 inputs) to b1000 (8 inputs).
[15:13]	Number of Instrumentation resources supported. The maximum value of this field is b100, for four Instrumentation resources. For more information see *Instrumentation resources* on page 3-58.
[12]	Reserved, RAO.

Table 3-45 ETMCCER bit assignments (continued)

Bits	Description
[11]	All registers readable. This bit is RAO, indicating that all registers are readable.
[10:3]	Size of extended external input bus. This field is zero if bits [2:0] are b000.
[2:0]	Number of extended external input selectors.

a. The PTM always treats ISB operations as waypoint instructions, and if timestamping is enabled it always generates a timestamp for an ISB operation.

3.16.30 Extended External Input Selection Register, ETMEXTINSELR

The ETMEXTINSELR characteristics are:

Purpose Selects the extended external inputs. See *Extended external inputs* on page 3-61.

Usage constraints There are no usage constraints.

Configurations Available in all PTM implementations.

The number of extended external input selectors:

- is IMPLEMENTATION DEFINED
- is specified by ETMCCER bits [2:0]
- can be zero.

See *Configuration Code Extension Register, ETMCCER* on page 3-103:

If the PTM implements fewer than four extended external input selectors then the unused extended external input selector fields in the ETMEXTINSELR are RAZ/WI.

If the PTM does not implement any extended external input selectors then the ETMEXTINSELR is RAZ/WI.

Attributes See the register summary in Table 3-16 on page 3-71.

Figure 3-37 shows the ETMEXTINSELR bit assignments.

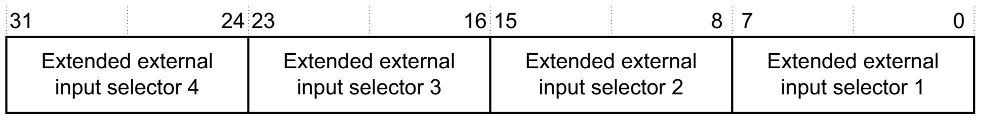

31
24 23
16 15
8 7
0
Extended external input selector 4
Extended external input selector 3
Extended external input selector 2
Extended external input selector 1

Figure 3-37 ETMEXTINSELR bit assignments

Table 3-46 shows the ETMEXTINSELR bit assignments.

Table 3-46 ETMEXTINSELR bit assignments

Bits	Description
[31:24]	Extended external input selector 4.
[23:16]	Extended external input selector 3.
[15:8]	Extended external input selector 2.
[7:0]	Extended external input selector 1.

3.16.31 TraceEnable Start/Stop EmbeddedICE Control Register, ETMTESSEICR

The ETMTESSEICR characteristics are:

Purpose Specifies the EmbeddedICE watchpoint comparator inputs that are used as trace start and stop resources.

Usage constraints There are no usage constraints.

Configurations Available in all PTM implementations.

The number of EmbeddedICE watchpoint comparators:

- is IMPLEMENTATION DEFINED
- is specified by ETMCCER bits [19:16]
- can be zero.

See *Configuration Code Extension Register, ETMCCER* on page 3-103.

If the PTM does not implement any EmbeddedICE watchpoint comparators then the ETMTESSEICR is RAZ/WI.

Attributes See the register summary in Table 3-16 on page 3-71.

Figure 3-38 shows the ETMTESSEICR bit assignments.

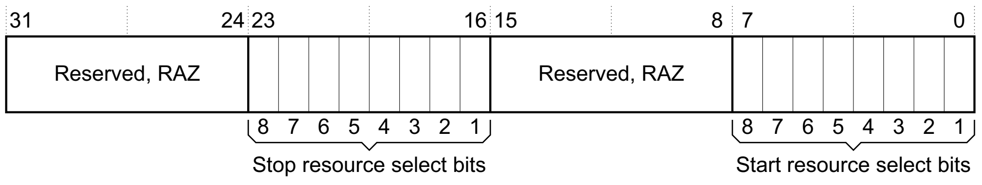


Figure 3-38 ETMTESSEICR bit assignments

Table 3-47 shows the ETMTESSEICR bit assignments.

Table 3-47 ETMTESSEICR bit assignments

Bits	Description
[31:24]	Reserved, SBZP.
[23:16]	TraceEnable stop control selection. Setting a bit in this field to 1 selects the corresponding EmbeddedICE watchpoint comparator input as a TraceEnable stop control, for the Start/Stop block. Bit [16] corresponds to input 1, bit [17] to input 2, and this pattern continues up to bit [23] corresponding to input 8.
[15:8]	Reserved, SBZP.
[7:0]	TraceEnable start control selection. Setting a bit in this field to 1 selects the corresponding EmbeddedICE watchpoint comparator input as a TraceEnable start control, for the Start/Stop block. Bit [0] corresponds to input 1, bit [1] to input 2, and this pattern continues up to bit [7] corresponding to input 8.

3.16.32 EmbeddedICE Behavior Control Register, ETMEIBCR

The ETMEIBCR characteristics are:

Purpose Controls the sampling behavior of the EmbeddedICE watchpoint comparator inputs.

Usage constraints There are no usage constraints.

Configurations This is an optional register. Bit [21] of the ETMCCER is set to 1 if the ETMEIBCR is implemented. See *Configuration Code Extension Register, ETMCCER* on page 3-103.

Attributes See the register summary in Table 3-16 on page 3-71.

Figure 3-39 shows the ETMEIBCR bit assignments.

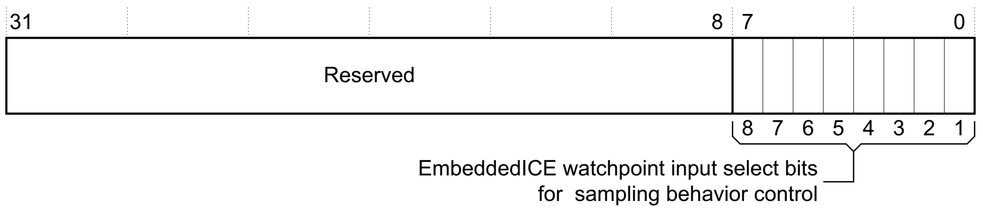

31
8
7
0
Reserved
8 7 6 5 4 3 2 1
EmbeddedICE watchpoint input select bits
for sampling behavior control

Figure 3-39 ETMEIBCR bit assignments

Table 3-48 shows the ETMEIBCR bit assignments.

Table 3-48 ETMEIBCR bit assignments

Bits	Description
[31:8]	Reserved, SBZP.
[7:0]	EmbeddedICE watchpoint comparator input sampling behavior. Each bit controls the sampling behavior of one of the EmbeddedICE watchpoint comparator inputs: **Bit == 0** When sampled, the corresponding input is pulsed for a single sample. **Bit == 1** When sampled, the corresponding input is latched and held until one cycle before the next sampling point. Bit [0] corresponds to input 1, bit [1] to input 2, and this pattern continues up to bit [7] corresponding to input 8.

For more information about the behavior of the EmbeddedICE watchpoint comparator inputs see *EmbeddedICE watchpoint comparator input behavior* on page 3-46.

Note

If the EmbeddedICE Behavior Control Register is not implemented, the EmbeddedICE watchpoint comparator inputs must behave as described in *Default behavior of EmbeddedICE watchpoint comparator inputs* on page 3-47.

3.16.33 Timestamp Event Register, ETMTSEVR

The ETMTSEVR characteristics are:

Purpose Defines an event that requests the insertion of a timestamp into the trace stream.

Usage constraints There are no usage constraints.

Configurations This register is implemented only when bit [22] of the ETMCCER is set to 1. See *Configuration Code Extension Register, ETMCCER* on page 3-103. If this register is not implemented, this register is RAZ/WI.

Attributes See the register summary in Table 3-16 on page 3-71.

Figure 3-40 shows the ETMTSEVR bit assignments.

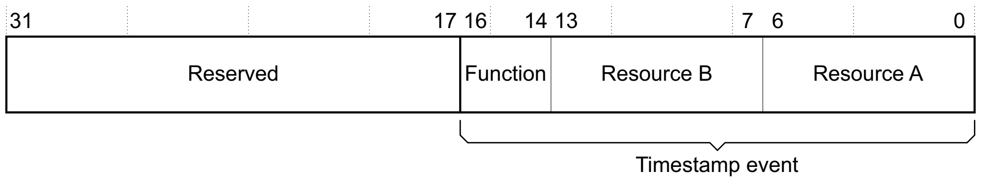


Figure 3-40 ETMTSEVR bit assignments

Table 3-49 shows the ETMTSEVR bit assignments.

Table 3-49 ETMTSEVR bit assignments

Bits	Description
[31:17]	Reserved
[16:0]	Timestamp event. Subdivided as: **Function, bits [16:14]** Specifies the logical operation that combines the two resources that define the event. **Resource B, bits [13:7] and Resource A, bits [6:0]** Specify the two resources that are combined by the logical operation specified by the Function field. For more information see *Event resources and PTM events* on page 3-49.

You can program this register so that an external device or a programmable event causes the PTM to insert a timestamp in the trace stream. For example, you might program it so that the execution of a `DMB` instruction on another processor causes the insertion of a timestamp.

Defining a PTM event on page 3-52 describes how you define a timestamp event.

ARM strongly recommends that you do not program this register with the Always true event, event `0x6F`. If you program the Timestamp event to be always true the PTM inserts many timestamps into the trace stream, and the trace FIFO is likely to overflow.

Typically, you program the Timestamp Event Register to cause the PTM to insert a timestamp in the trace stream periodically. You can do this by programming one of the PTM counters to decrement every cycle, and programming the ETMTSEVR so that the timestamp event occurs each time the counter reaches zero.

3.16.34 Auxiliary Control Register, ETMAUXCR

The ETMAUXCR characteristics are:

Purpose	Provides additional IMPLEMENTATION DEFINED PTM controls.
Usage constraints	There are no usage constraints.
Configurations	Available in all PTM implementations.
Attributes	See the register summary in Table 3-16 on page 3-71. The reset value of this register is zero.

The contents of the ETMAUXCR are IMPLEMENTATION DEFINED.

Tools must be aware that changing the value of this register might cause the PTM to behave in a way that contradicts this architecture specification. See the documentation of the specific PTM implementation for details of the IMPLEMENTATION DEFINED support for this register.

3.16.35 CoreSight Trace ID Register, ETMTRACEIDR

The ETMTRACEIDR characteristics are:

Purpose	Defines the 7-bit Trace ID, for output to the trace bus.
Usage constraints	There are no usage constraints.
Configurations	Available in all PTM implementations.
Attributes	See the register summary in Table 3-16 on page 3-71. The reset value of this register is zero.

Figure 3-41 on page 3-109 shows the ETMTRACEIDR bit assignments.

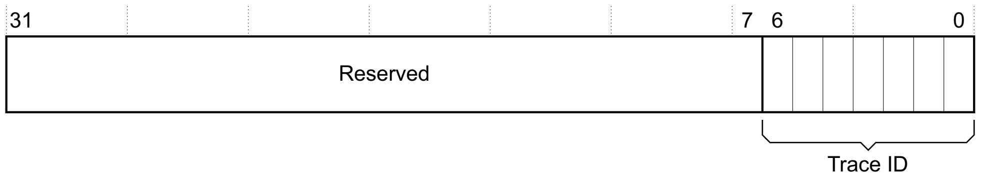


Figure 3-41 ETMTRACEIDR bit assignments

Table 3-50 shows the ETMTRACEIDR bit assignments.

Table 3-50 ETMTRACEIDR bit assignments

Bits	Description
[31:7]	Reserved.
[6:0]	Trace ID to output onto the trace bus.

This register is used in systems where multiple trace sources are present and tracing simultaneously. For example, where a PTM outputs trace onto the AMBA 3 *Advanced Trace Bus* (ATB), a unique ID is required for each trace source so that the trace can be uniquely identified as coming from a particular trace source. For more information about the AMBA 3 ATB, see the *CoreSight Architecture Specification*.

3.16.36 VMID Comparator Value Register, ETMVMIDCVR

The ETMVMIDCVR characteristics are:

Purpose Holds a value that the current *Virtual Machine ID* (VMID) can be compared to.

Usage constraints There are no usage constraints.

Configurations This register is only available in PFTv1.1 or later.

This is an optional register. Bit [26] of the ETMCCER reads as 1 if the ETMVMIDCVR is implemented. See *Configuration Code Extension Register, ETMCCER* on page 3-103. If this register is not implemented, the ETMVMIDCVR is RAZ/WI.

Attributes See the register summary in Table 3-16 on page 3-71.

Figure 3-42 shows the ETMVMIDCVR bit assignments.

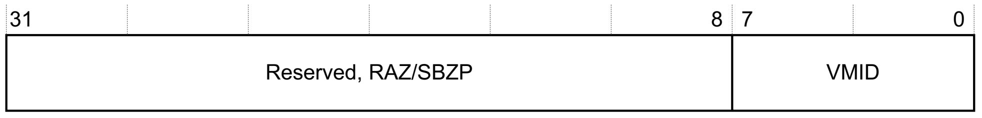


Figure 3-42 ETMVMIDCVR bit assignments

Table 3-51 shows the ETMVMIDCVR bit assignments.

Table 3-51 ETMVMIDCVR bit assignments

Bits	Description
[31:8]	Reserved.
[7:0]	Virtual Machine ID.

There is no mask for VMID comparators.

3.16.37 About the Operating System Save and Restore registers

You can use these registers to save the entire PTM state of the processor before it is powered down. *Power-down support* on page 3-132 describes the use of these registers. The following sections describe the registers:

- *OS Lock Access Register, ETMOSLAR*
- *OS Lock Status Register, ETMOSLSR*
- *OS Save and Restore Register, ETMOSSRR* on page 3-112.

3.16.38 OS Lock Access Register, ETMOSLAR

The ETMOSLAR characteristics are:

Purpose	Locks access to the PTM trace registers.
Usage constraints	There are no usage constraints.
Configurations	Available in all PTM implementations.
Attributes	See the register summary in Table 3-16 on page 3-71.

Figure 3-43 shows the ETMOSLAR bit assignments.

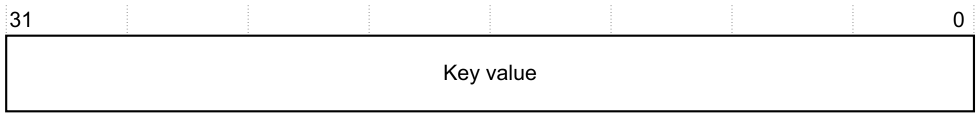


Figure 3-43 ETMOSLAR bit assignments

Table 3-52 shows the ETMOSLAR bit assignments.

Table 3-52 ETMOSLAR bit assignments

Bits	Description
[31:0]	Write `0xC5ACCE55` to this field to lock the PTM trace registers. Write any other value to this field to unlock the PTM trace registers.

When the PTM trace registers are locked, some attempts to access the locked registers returns a slave-generated error response. See *Power-down support* on page 3-132 for more information.

In PFTv1.0, accessing this register, to lock or unlock the PTM trace registers, also resets the internal save/restore counter. See *OS Save and Restore Register, ETMOSSRR* on page 3-112 for details of this counter. You must lock the PTM trace registers before you perform an OS save or restore. This prevents any changes to the trace registers during the save or restore process.

The OS Lock Status Register is read-only. To find out whether the PTM trace registers are locked you read the OS Lock Status register. See *OS Lock Status Register, ETMOSLSR*.

3.16.39 OS Lock Status Register, ETMOSLSR

The ETMOSLSR characteristics are:

Purpose	• Used to find whether the PTM trace registers are locked. • Can be used to find whether PTM trace register locking is implemented.
Usage constraints	There are no usage constraints.
Configurations	Available in all PTM implementations.
Attributes	See the register summary in Table 3-16 on page 3-71.

Figure 3-44 shows the ETMOSLSR bit assignments.

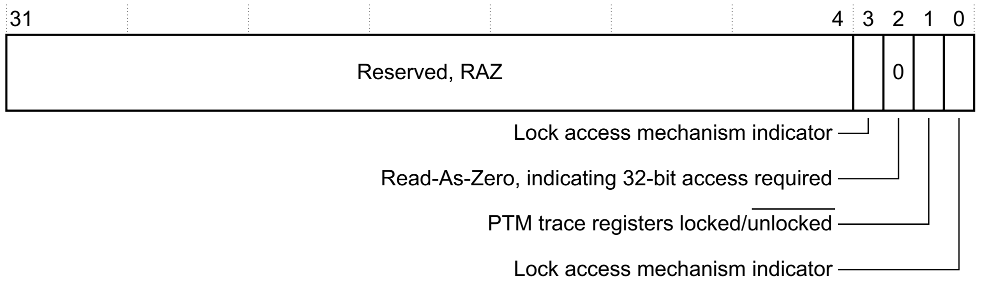

31
4
3
2
1
0
Reserved, RAZ
0
Lock access mechanism indicator
Read-As-Zero, indicating 32-bit access required
PTM trace registers locked/unlocked
Lock access mechanism indicator

Figure 3-44 ETMOSLSR bit assignments

Table 3-53 shows the ETMOSLSR bit assignments.

Table 3-53 ETMOSLSR bit assignments

Bits	Description
[31:4]	Reserved, RAZ.
[3]	Lock access mechanism indicator. In PFTv1.0, this bit is always Read-As-Zero. From PFTv1.1, this bit is used together with bit [0] to indicate whether the OS lock is implemented. SeeTable 3-54 and Table 3-81 on page 3-133 for information on how these bits are used.
[2]	32-bit access bit. This bit is always Read-As-Zero, indicating that a 32-bit access is required to operate the ETMOSLSR.
[1]	Locked bit. The possible values of this bit are: **0** PTM trace registers are not locked. **1** PTM trace registers are locked. Any access to these registers returns a slave-generated error response. In PFTv1.0, the reset value of this field is IMPLEMENTATION DEFINED. From PFTv1.1, if the OS Lock is implemented, this field is set from a PTM reset..
[0]	Lock access mechanism indicator. In PFTv1.0, the possible values of this bit are: **0** OS Lock and OS Save and Restore registers are not implemented. In this case, bits [31:0] of the OS Lock Status Register are RAZ. **1** OS Lock and OS Save and Restore registers are implemented and it is possible to set the OS Lock for this macrocell, to lock the PTM trace registers. From PFTv1.1, this bit is used together with bit [3]. SeeTable 3-54 and Table 3-81 on page 3-133 for information on how these bits are used. The reset value of this field is IMPLEMENTATION DEFINED.

Table 3-54 shows how bits [3] and [0] are used to determine whether the OS lock is implemented.

Table 3-54 OS lock implementation

ETMOSLSR[3]	ETMOSLSR[0]	OS lock implemented
0	0	No
0	1	Yes
1	0	Yes
1	1	Reserved

If a read of the OS Lock Status Register returns zero, OS Locking is not implemented.

See *Power-down support* on page 3-132 for more information about OS Locking.

3.16.40 OS Save and Restore Register, ETMOSSRR

The ETMOSSRR characteristics are:

Purpose	Used to save or restore the complete PTM trace register state of the macrocell.
Usage constraints	There are no usage constraints.
Configurations	This register is only available in PFTv1.0.
Attributes	See the register summary in Table 3-16 on page 3-71.

Figure 3-45 shows the ETMOSSRR bit assignments.

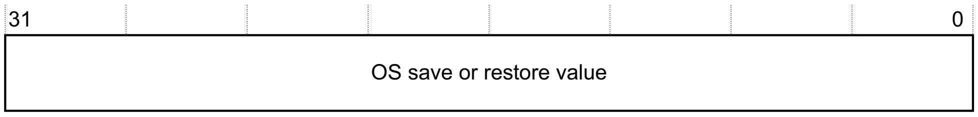


Figure 3-45 ETMOSSRR bit assignments

Table 3-55 shows the ETMOSSRR bit assignments.

Table 3-55 ETMOSSRR bit assignments

Bits	Description
[31:0]	The first access to the register must be a read. On this access: • on an OS save, this field returns the number of additional read accesses required to save the PTM trace register settings • on an OS restore, this field returns an UNKNOWN value, or an IMPLEMENTATION DEFINED value. **Note** On an OS restore, this read must be performed even if it returns an UNKNOWN value. On subsequent accesses the field holds the PTM trace register value being saved or restored.

This register works in conjunction with an internal sequence counter, to save or restore the contents of the PTM trace registers. See Table 3-14 on page 3-66 for details of how the PTM registers are split into trace and management registers.

Before accessing this register, you must write the key value, `0xC5ACCE55`, to the ETMOSLAR. This write resets the OS save and restore internal sequence counter. Locking the PTM trace registers in this way prevents any change to their contents during the save or restore process.

When you have locked access to the PTM trace registers you must read the ETMOSSRR. The significance of the result returned depends on whether you are performing an OS save or an OS restore:

OS save The value returned is the number of additional ETMOSSRR accesses required to save or restore the PTM trace registers. After reading this value, you must:

- save this value, for use for the OS restore
- perform this number of reads of the ETMOSSRR, saving the returned values to save the status of the PTM trace registers.
- to restore the PTM trace registers you must perform this number of writes to the ETMOSSRR.

OS restore The value returned is UNKNOWN, or might be IMPLEMENTATION DEFINED. After reading this value, you must:

- discard this value
- use the number of accesses value saved from the OS save operation to find how many accesses are required to restore the status of the PTM trace registers
- perform this number of writes to the ETMOSSRR, writing back the status information saved in the OS save operation.

The number of accesses required, and the order and interpretation of the save and restore data, is IMPLEMENTATION DEFINED. However, the restore sequence must observe the writable status of all bits of the registers being restored.

Behavior is UNPREDICTABLE if:

- you access the ETMOSSRR when the PTM trace registers are not locked
- after writing `0xC5ACCE55` to the OS Lock Access Register, your first access to the ETMOSSRR is a write
- after your first read of the ETMOSSRR, you mix read and write accesses to the ETMOSSRR
- after your first read of the ETMOSSRR, you perform more read or writes to the ETMOSSRR than are required to save or restore the PTM trace registers.

For more information on using the ETMOSSRR, see *Power-down support* on page 3-132.

3.16.41 Device Power-Down Control Register, ETMPDCR

The ETMPDCR characteristics are:

Purpose	This register is used to request power up of the PTM trace registers.
Usage constraints	There are no usage constraints.
Configurations	This register is only available from PFTv1.1.
Attributes	See the register summary in Table 3-16 on page 3-71.

Figure 3-46 shows the ETMPDCR bit assignments.

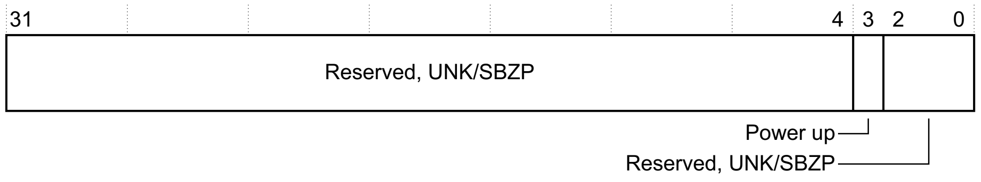


Figure 3-46 ETMPDCR bit assignments

Table 3-56 shows the ETMPDCR bit assignments.

Table 3-56 ETMPDCR bit assignments

Bits	Description
[31:4]	Reserved, UNK/SBZP
[3]	Power up bit. This bit drives an output signal, **ETMPWRUPREQ**. When this signal is HIGH, power must be provided to the PTM Trace Registers. Bit [0] of the ETMPDSR indicates when power is provided to the registers. The **ETMPWRUPREQ** output can be combined with DBGCOREPWRUPREQ if the PTM Trace Registers are in the same power domain as the CPU.
[2:0]	Reserved, UNK/SBZP

3.16.42 Device Power-Down Status Register, ETMPDSR

The ETMPDSR characteristics are:

Purpose	This register indicates the power-down status of the PTM.
Usage constraints	There are no usage constraints.
Configurations	Available in all PTM implementations.
Attributes	See the register summary in Table 3-16 on page 3-71.

Figure 3-47 shows the ETMPDSR bit assignments.

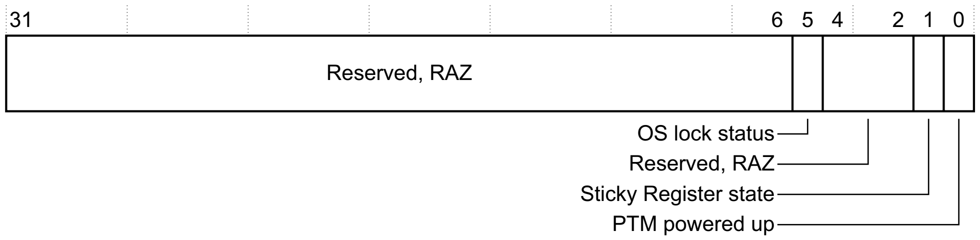


Figure 3-47 ETMPDSR bit assignments

Table 3-57 shows the ETMPDSR bit assignments.

Table 3-57 ETMPDSR bit assignments

Bits	Description
[31:6]	Reserved, RAZ.
[5]	For PFTv1.0 this bit is Reserved, RAZ.
	LK bit. From PFTv1.1 this bit indicates the status of the OS Lock. The possible values of this bit are: **0** PTM trace registers are not locked. **1** PTM trace registers are locked. Any access to these registers returns a slave-generated error response. See *Power-down support* on page 3-132 for more information If the OS Lock is implemented, this field is set from a PTM reset. The external debugger must always clear the OS Lock after system power on reset, before it can access trace registers.

Table 3-57 ETMPDSR bit assignments (continued)

Bits	Description
[4:2]	Reserved, RAZ.
[1]	Sticky Register state bit. The possible values of this bit are: **0** PTM trace registers have not been powered down since this register was last read. **1** PTM trace registers have been powered down since this register was last read, and have lost their state. This bit is cleared to 0 on reading this register if the core power domain of the PTM is currently powered up, indicated by bit [0] being set to 1. When the core power domain of the PTM is powered down, this bit is set to 1. Reads of this register when the core power domain is powered down return 1 for this bit, and do not change the value of this bit. Reads of this register when the core power domain is powered up return the current value of this bit, and then clear this bit to 0. If the Software Lock mechanism is locked and the ETMPDSR read is made through the memory mapped interface, this bit is not cleared. When this bit is set to 1, an access to any PTM Trace Register returns an error response.
[0]	PTM powered up bit. The value of this bit indicates whether you can access the PTM trace registers. The possible values are: **0** PTM trace registers cannot be accessed. **1** PTM trace registers can be accessed. Usually, an external input to the PTM, driven by the system power controller, controls the value of this bit. When this bit is set to 0, an access to any PTM trace register returns an error response.

Table 3-58 shows the different encodings of the ETMPDSR.

Table 3-58 ETMPDSR encodings

Bit [1] Sticky Register state	Bit [0] PTM powered up	Meaning
0	0	PTM trace registers are inaccessible. No state has been lost.
0	1	PTM trace registers are accessible.
1	0	PTM trace registers are powered down, inaccessible, and their state has been lost.
1	1	PTM trace registers are powered up. However, their state has been lost because of a power down.

This register must be implemented in:

- PTMs that supports multiple power domains where the power to the PTM trace registers can be removed
- implementations where access to the PTM trace registers is restricted because of other design limitations.

If the PTM only occupies a single power domain, this register might always read as `0x00000001`, indicating that the PTM is powered up and accessible. In this case, if the PTM is not powered up:

- no PTM registers are accessible
- the ETMPDSR does not indicate whether the PTM state has been lost.

3.16.43 Integration Mode Control Register, ETMITCTRL

The ETMITCTRL characteristics are:

Purpose	Enables topology detection or integration testing.
Usage constraints	There are no usage constraints.
Configurations	Available in all PTM implementations.
Attributes	See the register summary in Table 3-16 on page 3-71.

Figure 3-48 shows the ETMITCTRL bit assignments.

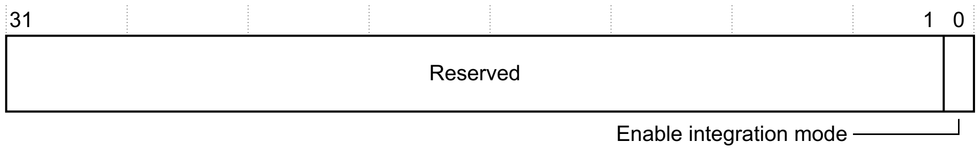


Figure 3-48 ETMITCTRL bit assignments

Table 3-59 shows the ETMITCTRL bit assignments.

Table 3-59 ETMITCTRL bit assignments

Bits	Description
[31:1]	Reserved.
[0]	When this bit is set to 1, the device enters an integration mode that permits topology detection or integration testing. A PTM reset sets this bit to 0.

3.16.44 About the claim tag registers

Software can use the claim tag to coordinate software and debugger access to PTM functionality.

The following sections describe the two claim tag registers:

- *Claim Tag Set Register, ETMCLAIMSET*
- *Claim Tag Clear Register, ETMCLAIMCLR* on page 3-117.

3.16.45 Claim Tag Set Register, ETMCLAIMSET

The ETMCLAIMSET characteristics are:

Purpose	Used to set bits in the claim tag and find the number of bits supported by the claim tag.
Usage constraints	There are no usage constraints.
Configurations	Available in all PTM implementations.
Attributes	See the register summary in Table 3-16 on page 3-71.

Figure 3-49 shows the ETMCLAIMSET bit assignments.

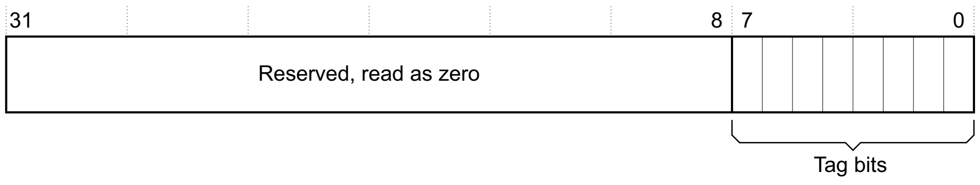


Figure 3-49 ETMCLAIMSET bit assignments

Table 3-60 shows the ETMCLAIMSET bit assignments.

Table 3-60 ETMCLAIMSET bit assignments

Bits	Description
[31:8]	Reserved.
[7:0]	On reads, returns `0xFF`. On writes, a 1 in a bit position causes the corresponding bit in the claim tag value to be set.

3.16.46 Claim Tag Clear Register, ETMCLAIMCLR

The ETMCLAIMCLR characteristics are:

Purpose	• Clear bits in the claim tag to 0 • Find the current value of the claim tag.
Usage constraints	There are no usage constraints.
Configurations	Available in all PTM implementations.
Attributes	See the register summary in Table 3-16 on page 3-71.

Figure 3-50 shows the ETMCLAIMCLR bit assignments.

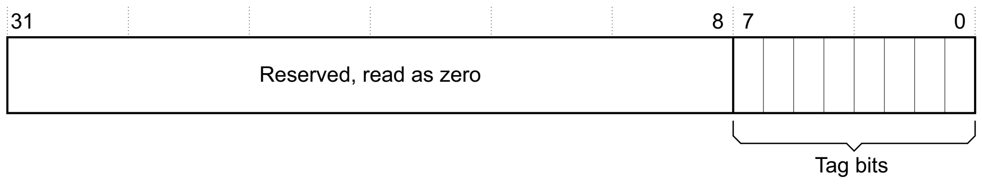


Figure 3-50 ETMCLAIMCLR bit assignments

Table 3-61 shows the ETMCLAIMCLR bit assignments.

Table 3-61 ETMCLAIMCLR bit assignments

Bits	Description
[31:8]	Reserved.
[7:0]	On reads, returns the current claim tag value. On writes, a 1 in a bit position causes the corresponding bit in the claim tag value to be cleared to 0. A PTM reset sets this field to `0x00`.

3.16.47 About the lock registers

The lock registers control memory-mapped software access to all other registers, including the ETMCR. If you lock the PTM using this feature, it ignores memory-mapped software writes. JTAG accesses, coprocessor accesses, memory-mapped debugger accesses, and all reads are unaffected.

Note

- Any implementation of the PFT architecture that implements memory-mapped access to PTM registers must implement the lock access mechanism.
- When the lock access mechanism is implemented, a PTM reset locks the PTM.

You can use this feature to prevent accidental modification of the PTM registers by software being debugged. For example, software that accidentally initializes unwanted areas of memory might disable the PTM, making it impossible to trace such software. To prevent this, on-chip software that accesses the PTM must access the PTM registers as follows:

1. Unlock the PTM by writing `0xC5ACCE55` to the ETMLAR.
2. Access the other PTM registers.
3. Lock the PTM by writing any other value, for example `0x0`, to the ETMLAR.

The following sections describe the lock registers:

- *Lock Access Register, ETMLAR*
- *Lock Status Register, ETMLSR* on page 3-119.

3.16.48 Lock Access Register, ETMLAR

The ETMLAR characteristics are:

Purpose	Locks and unlocks access to all other PTM registers.
Usage constraints	From PFTv1.1, accesses to this register from the coprocessor interface are UNPREDICTABLE.
Configurations	Available in all PTM implementations.
Attributes	See the register summary in Table 3-16 on page 3-71.

Figure 3-51 shows the ETMLAR bit assignments.

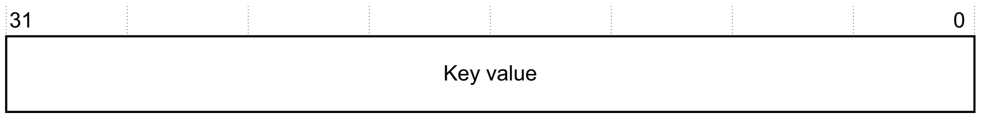


Figure 3-51 ETMLAR bit assignments

Table 3-62 shows the ETMLAR bit assignments.

Table 3-62 ETMLAR bit assignments

Bits	Description
[31:0]	Write `0xC5ACCE55` to this field to unlock the PTM. Write any other value to this field to lock the PTM. Writes to this register from an interface that ignores the lock registers are ignored.

3.16.49 Lock Status Register, ETMLSR

The ETMLSR characteristics are:

Purpose	The register has two uses: • software reading the ETMLSR from any interface can check bit [0] to find out whether the lock registers are implemented for that interface • software reading the ETMLSR from an interface for which the lock registers are implemented can check bit [1] to find out whether the registers are currently locked.
Usage constraints	From PFTv1.1, accesses to this register from the coprocessor interface are UNPREDICTABLE.
Configurations	Available in all PTM implementations.
Attributes	See the register summary in Table 3-16 on page 3-71.

Figure 3-52 shows the ETMLSR bit assignments.

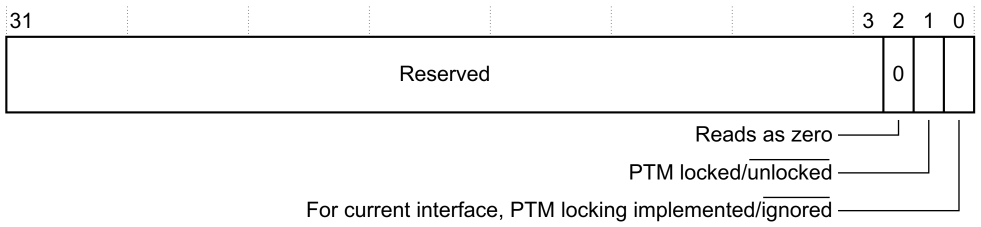


Figure 3-52 ETMLSR bit assignments

Table 3-63 shows the ETMLSR bit assignments.

Table 3-63 ETMLSR bit assignments

Bits	Description
[31:3]	Reserved.
[2]	Reads as b0. Indicates that the ETMLAR is 32 bits.
[1]	Indicates whether the PTM is locked. The possible values of this bit are: **0** Writes are permitted. **1** PTM locked. Writes are ignored. If this register is accessed from an interface where the lock registers are ignored, this field reads as 0 regardless of whether the PTM is locked.
[0]	Indicates whether the lock registers are implemented for this interface. The possible values of this bit are: **0** This access is from an interface that ignores the lock registers. **1** This access is from an interface that requires the PTM to be unlocked.

3.16.50 Authentication Status Register, ETMAUTHSTATUS

The ETMAUTHSTATUS characteristics are:

Purpose	Reports the level of tracing currently permitted by the authentication signals provided to the PTM.
Usage constraints	There are no usage constraints.
Configurations	Available in all PTM implementations.
Attributes	See the register summary in Table 3-16 on page 3-71.

Figure 3-53 shows the ETMAUTHSTATUS bit assignments.

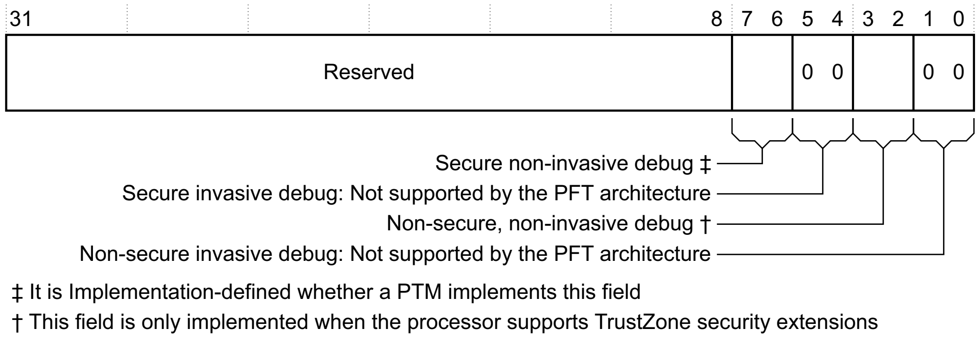

31
8 7 6 5 4 3 2 1 0
Reserved
0 0
0 0
Secure non-invasive debug ‡
Secure invasive debug: Not supported by the PFT architecture
Non-secure, non-invasive debug †
Non-secure invasive debug: Not supported by the PFT architecture
‡ It is Implementation-defined whether a PTM implements this field
† This field is only implemented when the processor supports TrustZone security extensions

Figure 3-53 ETMAUTHSTATUS bit assignments

Table 3-64 shows the ETMAUTHSTATUS bit assignments.

Table 3-64 ETMAUTHSTATUS bit assignments

Bits	Description
[31:8]	Reserved, RAZ.
[7:6]	Permission for Secure non-invasive debug See *Implementation of the Secure non-invasive debug field* for more information.
[5:4]	Reads as b00, Secure invasive debug not supported by the PFT architecture.
[3:2]	Permission for Non-secure non-invasive debug. This field is only implemented if the processor implemented with the PTM implements the Security Extensions. When this field is implemented the possible values of the field are: **b10** Non-secure non-invasive debug disabled. **b11** Non-secure non-invasive debug enabled. This field is a logical OR of the **NIDEN** and **DBGEN** signals. It takes the value b11 when the OR is TRUE, and b10 when the OR is FALSE. If the processor does not support the Security Extensions, bits [3:2] are reserved, RAZ.
[1:0]	Reads as b00, Non-secure invasive debug not supported by the PFT architecture.

Implementation of the Secure non-invasive debug field

It is IMPLEMENTATION DEFINED whether a PTM implements the Secure non-invasive debug field. If this field is implemented, its behavior depends on whether the processor implemented with the PTM supports the Security Extensions. If the processor does support the Security Extensions, then the behavior of this field depends on the of the following applies:

- the processor controls what trace is prohibited
- the PTM controls what trace is prohibited.

Table 3-65 summarizes this behavior.

Table 3-65 Implementation of the Secure non-invasive debug field

Processor includes Security Extensions?	Control of Secure tracing	Behavior of Secure non-invasive debug field, bits [7:6]
No	Not applicable when the processor does not support the Security Extensions	The processor is assumed to operate in a Secure state. The possible values of the field are: **b10** Secure non-invasive debug disabled. **b11** Secure non-invasive debug enabled. The value of this field is a logical OR of the **NIDEN** and **DBGEN** signals. It takes the value b11 when the OR is TRUE, and b10 when the OR is FALSE.
Yes	Not controlled by PTM	The field reads as b00, indicating that the PTM does not control when trace is prohibited.
Yes	Controlled by PTM	The possible values of the field are: **b10** Secure non-invasive debug disabled. **b11** Secure non-invasive debug enabled. The value of this field is a logical result of: (**SPNIDEN** OR **SPIDEN**) AND (**NIDEN** OR **DBGEN**) It takes the value b11 when the logical result is TRUE, and b10 when it is FALSE. Figure 3-54 shows the logic used to obtain the value of this field.

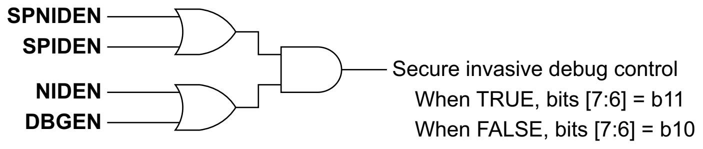

SPNIDEN
SPIDEN
NIDEN
DBGEN
Secure invasive debug control
When TRUE, bits [7:6] = b11
When FALSE, bits [7:6] = b10

Figure 3-54 Secure non-invasive debug enable logic when controlled by the PTM

3.16.51 Device Configuration Register, ETMDEVID

The ETMDEVID characteristics are:

Purpose	Returns an IMPLEMENTATION DEFINED CoreSight component capabilities field.
Usage constraints	There are no usage constraints.
Configurations	Available in all PTM implementations. The width of the data field in the register is IMPLEMENTATION DEFINED.
Attributes	See the register summary in Table 3-16 on page 3-71.

Figure 3-55 shows the ETMDEVID bit assignments.

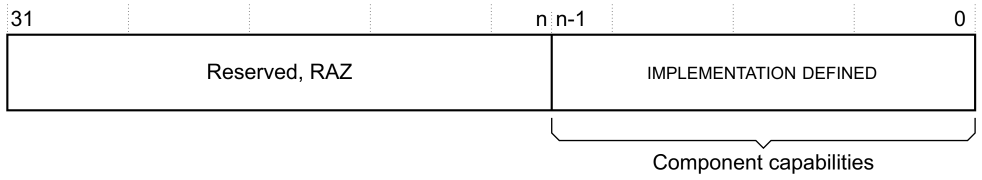


Figure 3-55 ETMDEVID bit assignments

Table 3-66 shows the ETMDEVID bit assignments.

Table 3-66 ETMDEVID bit assignments

Bits[a]	Description
[31:n]	Reserved. Read-as-zero.
[n-1:0]	Component capabilities. Bit assignments in this field are IMPLEMENTATION DEFINED.

a. The value of n is IMPLEMENTATION DEFINED.

The *CoreSight Architecture Specification* requires every CoreSight component to implement the Device Configuration Register. The data field in this register indicates the capabilities of the component. The width of the data field, and the meaning of the bits in the data field, are IMPLEMENTATION DEFINED. All unused bits must Read-As-Zero.

If a component is configurable, ARM recommends that this register is used to indicate any changes to the standard configuration.

3.16.52 Device Type Register, ETMDEVTYPE

The ETMDEVTYPE characteristics are:

Purpose	Returns the CoreSight device type of the PTM macrocell.
Usage constraints	There are no usage constraints.
Configurations	Available in all PTM implementations.
Attributes	See the register summary in Table 3-16 on page 3-71.

Figure 3-56 shows the ETMDEVTYPE bit assignments.

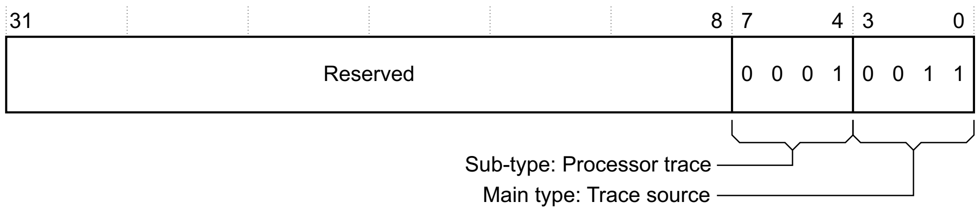


Figure 3-56 ETMDEVTYPE bit assignments

Table 3-67 shows the ETMDEVTYPE bit assignments.

Table 3-67 ETMDEVTYPE bit assignments

Bits	Description
[31:8]	Reserved
[7:4]	Sub type, `0x1`, processor trace
[3:0]	Main type, `0x3`, trace source

3.16.53 About the Peripheral Identification Registers

The Peripheral Identification Registers provide standard information required for all CoreSight components. They are a set of eight registers, listed in register number order in Table 3-68:

Table 3-68 Summary of the peripheral identification registers

Register	Description	Number	Offset[a]
Peripheral ID4	See *Peripheral ID4 Register, ETMPIDR4* on page 3-127	`0x3F4`	`0xFD0`
Peripheral ID5	See *Peripheral ID5 to Peripheral ID7 Registers, ETMPIDR5 to ETMPIDR7* on page 3-128	`0x3F5`	`0xFD4`
Peripheral ID6		`0x3F6`	`0xFD8`
Peripheral ID7		`0x3F7`	`0xFDC`
Peripheral ID0	See *Peripheral ID0 Register, ETMPIDR0* on page 3-124	`0x3F8`	`0xFE0`
Peripheral ID1	See *Peripheral ID1 Register, ETMPIDR1* on page 3-125	`0x3F9`	`0xFE4`
Peripheral ID2	See *Peripheral ID2 Register, ETMPIDR2* on page 3-126	`0x3FA`	`0xFE8`
Peripheral ID3	See *Peripheral ID3 Register, ETMPIDR3* on page 3-126	`0x3FB`	`0xFEC`

a. Register offset where the registers are accessed in a memory-mapped scheme. The register offset is always 4 × (Register number).

Only bits [7:0] of each Peripheral ID Register are used, with bits [31:8] reserved. Together, the eight Peripheral ID Registers define a single 64-bit Peripheral ID, as Figure 3-57 shows.

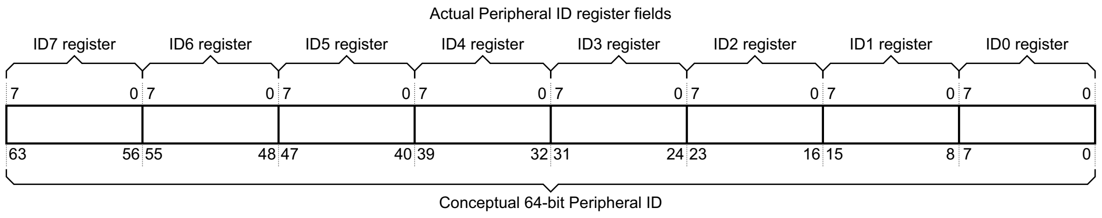


Figure 3-57 Mapping between the Peripheral ID Registers and the Peripheral ID value

Figure 3-58 shows the standard Peripheral ID fields in the single conceptual Peripheral ID.

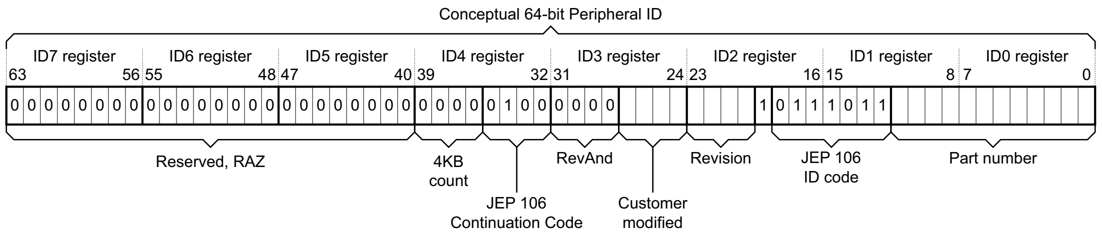


Figure 3-58 Peripheral ID fields for an ARM implementation

Table 3-69 shows the standard Peripheral ID fields, and shows where this information is held in the Peripheral ID Registers.

Table 3-69 Register fields for the Peripheral ID registers

Name	Size	Description	See Register
4KB Count	4 bits	Log_2 of the number of 4KB blocks occupied by the device. PTM implementations occupy a single 4KB block, so this field is always `0x0`.	Peripheral ID4
JEP 106 code	4+7 bits	Identifies the designer of the device. This consists of a 4-bit continuation code and a 7-bit identity code. For a PTM designed by ARM the continuation code is `0x4` and the identity code is `0x3B`, indicating ARM.	Peripheral ID1, Peripheral ID2, Peripheral ID4
Part Number	12 bits	Part number for the device.	Peripheral ID0, Peripheral ID1
Revision	4 bits	Revision of the peripheral.	Peripheral ID2
RevAnd	4 bits	Indicates a late modification to the device, usually as a result of an Engineering Change Order.	Peripheral ID3
Customer modified	4 bits	Indicates an endorsed modification to the device.	Peripheral ID3

For more information about these fields, see the *CoreSight Architecture Specification*.

In PFTv1.0, the revision field in the ETMIDR and ETMPIDR2 are identical. In PFTv1.1:
- ETMPIDR2 identifies the revision of the external debugger and memory mapped interfaces.
- ETMIDR identifies the revision of the Trace registers and the OS Save/Restore registers.

ARM recommends that implementations keep these values identical to ensure revision numbers can be managed easily. However, in cases where an ECO fix is required and changing both revisions is difficult, it is acceptable to change the revision fields independently. The next official implementation should re-align the revision numbers, potentially missing values from either field.

The following sections describe the fields present in each register. Registers are described in register name order, ID0 to ID7. Table 3-68 on page 3-123 shows the register numbers and offset addresses of these registers, that do not run in register name order.

3.16.54 Peripheral ID0 Register, ETMPIDR0

The ETMPIDR0 characteristics are:

Purpose Holds peripheral identification information.

Usage constraints
- Only bits [7:0] of this register are valid and they must be used with bits [7:0] of the other Peripheral ID registers to obtain the CoreSight Peripheral ID for the PTM.
- From PFTv1.1, accesses to this register from the coprocessor interface are UNPREDICTABLE.

Configurations Available in all PTM implementations.

Attributes See the register summary in Table 3-16 on page 3-71.

Figure 3-59 shows the ETMPIDR0 bit assignments.

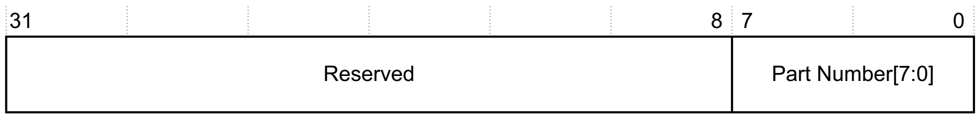


Figure 3-59 ETMPIDR0 bit assignments

Table 3-70 shows the ETMPIDR0 bit assignments.

Table 3-70 ETMPIDR0 bit assignments

Bits	Description[a]
[31:8]	Reserved
[7:0]	Part Number[7:0]

a. See Table 3-69 on page 3-124 for more information about the register fields.

3.16.55 Peripheral ID1 Register, ETMPIDR1

The ETMPIDR1 characteristics are:

Purpose Holds peripheral identification information.

Usage constraints
- Only bits [7:0] of this register are valid and they must be used with bits [7:0] of the other Peripheral ID registers to obtain the CoreSight Peripheral ID for the PTM.
- From PFTv1.1, accesses to this register from the coprocessor interface are UNPREDICTABLE.

Configurations Available in all PTM implementations.

Attributes See the register summary in Table 3-16 on page 3-71.

Figure 3-60 shows the ETMPIDR1 bit assignments.

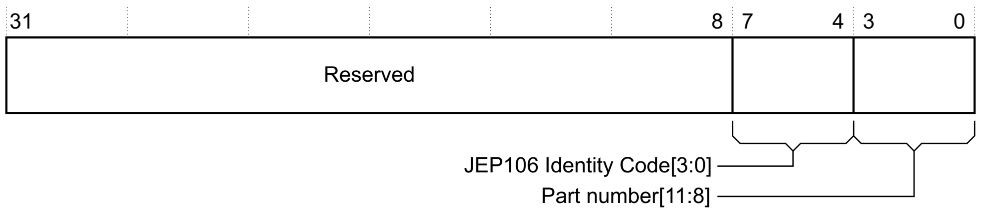


Figure 3-60 ETMPIDR1 bit assignments

Table 3-71 shows the ETMPIDR1 bit assignments.

Table 3-71 ETMPIDR1 bit assignments

Bits	Description[a]
[31:8]	Reserved
[7:4]	JEP106 Identity Code[3:0]
[3:0]	Part Number[11:8]

a. See Table 3-69 on page 3-124 for more information about the register fields.

3.16.56 Peripheral ID2 Register, ETMPIDR2

The ETMPIDR2 characteristics are:

Purpose Holds peripheral identification information.

Usage constraints
- Only bits [7:0] of this register are valid and they must be used with bits [7:0] of the other Peripheral ID registers to obtain the CoreSight Peripheral ID for the PTM.
- From PFTv1.1, accesses to this register from the coprocessor interface are UNPREDICTABLE.

Configurations Available in all PTM implementations.

Attributes See the register summary in Table 3-16 on page 3-71.

Figure 3-61 shows the ETMPIDR2 bit assignments.

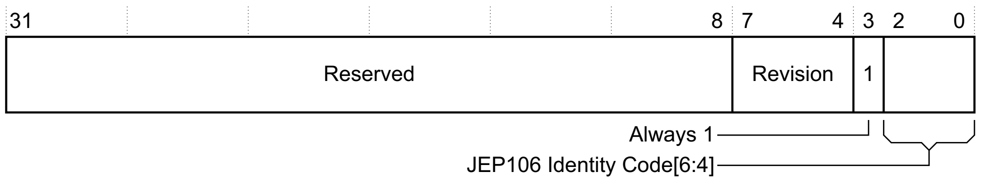


Figure 3-61 ETMPIDR2 bit assignments

Table 3-72 shows the ETMPIDR2 bit assignments.

Table 3-72 ETMPIDR2 bit assignments

Bits	Description[a]
[31:8]	Reserved.
[7:4]	Revision. This value matches the revision field in the ETMIDR. See *ID Register, ETMIDR* on page 3-101.
[3]	Always 1.
[2:0]	JEP106 Identity Code[6:4].

a. See Table 3-69 on page 3-124 for more information about the register fields.

3.16.57 Peripheral ID3 Register, ETMPIDR3

The ETMPIDR3 characteristics are:

Purpose Holds peripheral identification information.

Usage constraints
- Only bits [7:0] of this register are valid and they must be used with bits [7:0] of the other Peripheral ID registers to obtain the CoreSight Peripheral ID for the PTM.
- From PFTv1.1, accesses to this register from the coprocessor interface are UNPREDICTABLE.

Configurations Available in all PTM implementations.

Attributes See the register summary in Table 3-16 on page 3-71.

Figure 3-62 on page 3-127 shows the ETMPIDR3 bit assignments.

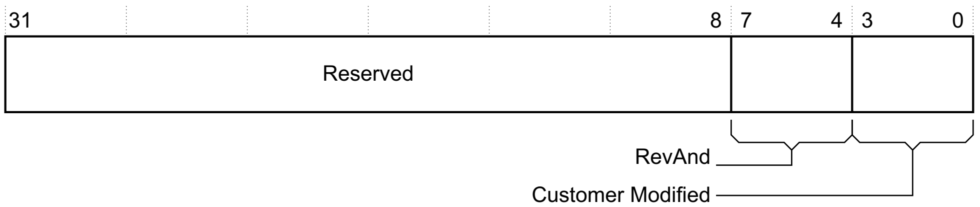


Figure 3-62 ETMPIDR3 bit assignments

Table 3-73 shows the ETMPIDR3 bit assignments.

Table 3-73 ETMPIDR3 bit assignments

Bits	Description[a]
[31:8]	Reserved
[7:4]	RevAnd
[3:0]	Customer Modified

a. See Table 3-69 on page 3-124 for more information about the register fields.

3.16.58 Peripheral ID4 Register, ETMPIDR4

The ETMPIDR4 characteristics are:

Purpose Holds peripheral identification information.

Usage constraints
- Only bits [7:0] of this register are valid and they must be used with bits [7:0] of the other Peripheral ID registers to obtain the CoreSight Peripheral ID for the PTM.
- From PFTv1.1, accesses to this register from the coprocessor interface are UNPREDICTABLE.

Configurations Available in all PTM implementations.

Attributes See the register summary in Table 3-16 on page 3-71.

Figure 3-63 shows the ETMPIDR4 bit assignments.

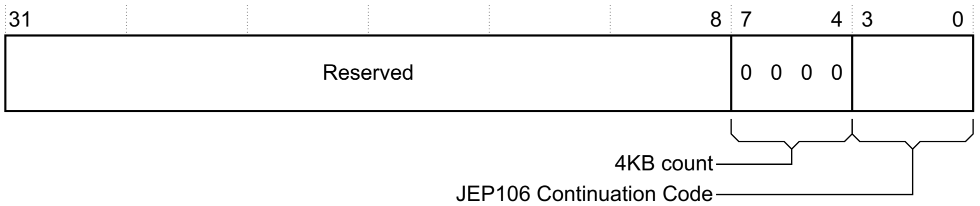


Figure 3-63 ETMPIDR4 bit assignments

Table 3-74 lists the ETMPIDR4 bit assignments.

Table 3-74 ETMPIDR4 bit assignments

Bits	Description[a]
[31:8]	Reserved
[7:4]	4KB count
[3:0]	JEP106 Continuation Code

a. See Table 3-69 on page 3-124 for more information about the register fields.

3.16.59 Peripheral ID5 to Peripheral ID7 Registers, ETMPIDR5 to ETMPIDR7

The characteristics for ETMPIDR5 to ETMPIDR7 are:

Purpose	Reserved for future expansion of the CoreSight peripheral identification information.
Usage constraints	From PFTv1.1, accesses to these registers from the coprocessor interface are UNPREDICTABLE.
Configurations	These registers are defined as reserved registers.
Attributes	See the register summary in Table 3-16 on page 3-71.

Figure 3-64 shows the ETMPIDR5 to ETMPIDR7 bit assignments.

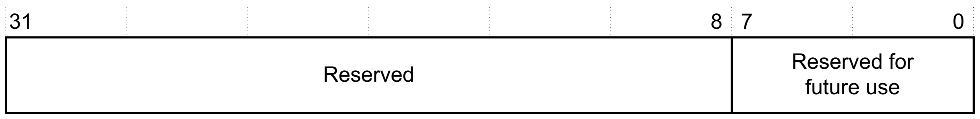


Figure 3-64 ETMPIDR5 to ETMPIDR7 bit assignments

Table 3-75 shows the ETMPIDR5 to ETMPIDR7 bit assignments.

Table 3-75 ETMPIDR5 to ETMPIDR7 bit assignments

Bits	Description
[31:8]	Reserved
[7:0]	Reserved

Table 3-68 on page 3-123 shows the register addresses and memory offsets for these registers.

3.16.60 About the Component Identification Registers

A PTM includes four read-only Component Identification Registers, ComponentID0 to ComponentID3. Table 3-76 shows these registers.

Table 3-76 Summary of the component identification registers

Register	Description	Number	Offset[a]
Component ID0	See *Component ID0 Register, ETMCIDR0*	`0x3FC`	`0xFF0`
Component ID1	See *Component ID1 Register, ETMCIDR1* on page 3-130	`0x3FD`	`0xFF4`
Component ID2	See *Component ID2 Register, ETMCIDR2* on page 3-130	`0x3FE`	`0xFF8`
Component ID3	See *Component ID3 Register, ETMCIDR3* on page 3-131	`0x3FF`	`0xFFC`

a. Register offset where the registers are accessed in a memory-mapped scheme. The register offset is always 4×(Register number).

The Component Identification Registers identify the PTM as a CoreSight component. For more information, see the *CoreSight Architecture Specification*.

Only bits [7:0] of each register are used. Figure 3-65 shows the concept of a single 32-bit component ID, obtained from the four Component Identification Registers.

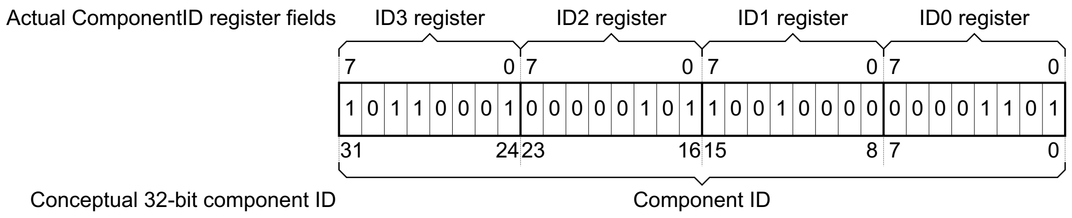


Figure 3-65 Mapping between the Component ID Registers and the Component ID value

3.16.61 Component ID0 Register, ETMCIDR0

The ETMCIDR0 characteristics are:

Purpose Holds byte 0 of the CoreSight preamble information.

Usage constraints
- Only bits [7:0] of this register are valid and they must be used with bits [7:0] of the other Component ID registers to obtain the CoreSight Component ID for the PTM.
- From PFTv1.1, accesses to this register from the coprocessor interface are UNPREDICTABLE.

Configurations Available in all PTM implementations.

Attributes See the register summary in Table 3-16 on page 3-71.

Figure 3-66 shows the ETMCIDR0 bit assignments.

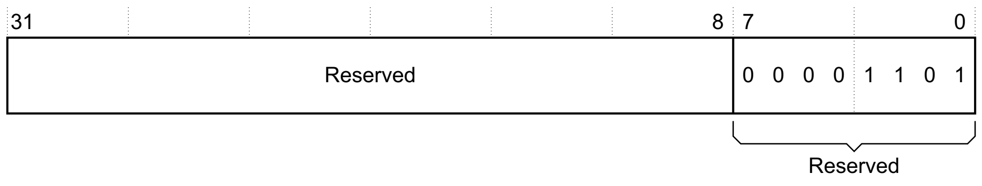


Figure 3-66 ETMCIDR0 bit assignments

Table 3-77 shows the ETMCIDR0 bit assignments.

Table 3-77 ETMCIDR0 bit assignments

Bits	Value	Description
[31:8]	-	Reserved
[7:0]	`0x0D`	Reserved

3.16.62 Component ID1 Register, ETMCIDR1

The ETMCIDR1 characteristics are

Purpose Holds byte 1 of the CoreSight preamble information.

Usage constraints
- Only bits [7:0] of this register are valid and they must be used with bits [7:0] of the other Component ID registers to obtain the CoreSight Component ID for the PTM.
- From PFTv1.1, accesses to this register from the coprocessor interface are UNPREDICTABLE.

Configurations Available in all PTM implementations.

Attributes See the register summary in Table 3-16 on page 3-71.

Figure 3-67 shows the ETMCIDR1 bit assignments.

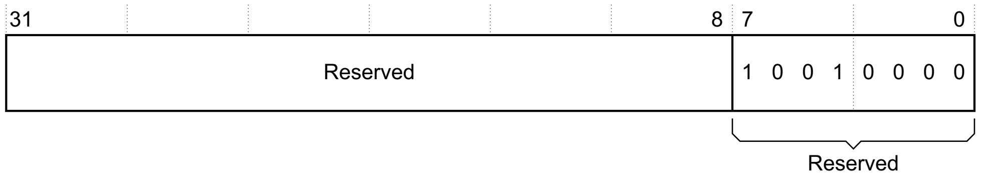


Figure 3-67 ETMCIDR1 bit assignments

Table 3-78 shows the ETMCIDR1 bit assignments.

Table 3-78 ETMCIDR1 bit assignments

Bits	Value	Description
[31:8]	-	Reserved
[7:0]	`0x90`	Reserved

3.16.63 Component ID2 Register, ETMCIDR2

The ETMCIDR2 characteristics are:

Purpose Holds byte 2 of the CoreSight preamble information.

Usage constraints
- Only bits [7:0] of this register are valid and they must be used with bits [7:0] of the other Component ID registers to obtain the CoreSight Component ID for the PTM.
- From PFTv1.1, accesses to this register from the coprocessor interface are UNPREDICTABLE.

Configurations Available in all PTM implementations.

Attributes See the register summary in Table 3-16 on page 3-71.

Figure 3-68 on page 3-131 shows the ETMCIDR2 bit assignments.

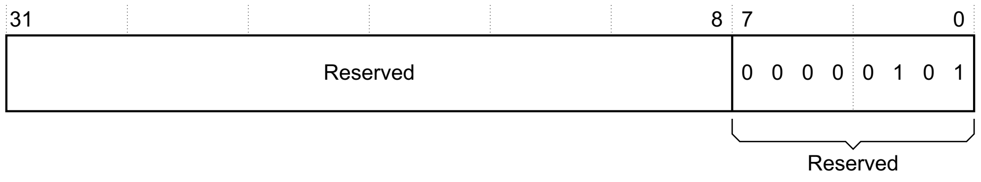


Figure 3-68 ETMCIDR2 bit assignments

Table 3-79 shows the ETMCIDR2 bit assignments.

Table 3-79 ETMCIDR2 bit assignments

Bits	Value	Description
[31:8]	-	Reserved
[7:0]	`0x05`	Reserved

3.16.64 Component ID3 Register, ETMCIDR3

The ETMCIDR3 characteristics are:

Purpose Holds byte 3 of the CoreSight preamble information.

Usage constraints
- Only bits [7:0] of this register are valid and they must be used with bits [7:0] of the other Component ID registers to obtain the CoreSight Component ID for the PTM.
- From PFTv1.1, accesses to this register from the coprocessor interface are UNPREDICTABLE.

Configurations Available in all PTM implementations.

Attributes See the register summary in Table 3-16 on page 3-71.

Figure 3-69 shows the ETMCIDR3 bit assignments.

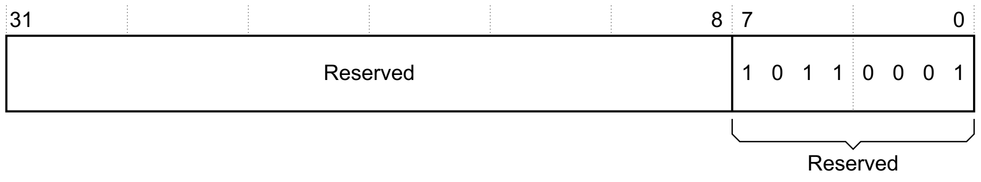


Figure 3-69 ETMCIDR3 bit assignments

Table 3-80 shows the ETMCIDR3 bit assignments.

Table 3-80 ETMCIDR3 bit assignments

Bits	Value	Description
[31:8]	-	Reserved
[7:0]	`0xB1`	Reserved

3.17 Power-down support

The PFT architecture includes power-down support for the macrocell. This support enables the entire PTM state of the macrocell to be saved before it is powered down, and restored when it is powered up again.

Note

The PTM state is held in the PTM trace registers. See *PTM trace and PTM management registers* on page 3-66.

The main features of the PTM power-down support are:

- The PTM registers are split between PTM trace registers and PTM management registers. It is the PTM trace registers that you can save and restore, to provide power-down support.
- You can lock the PTM trace registers, to prevent any change to their contents.
  - You lock or unlock the PTM trace registers by writing to the ETMOSLAR. See *OS Lock Access Register, ETMOSLAR* on page 3-110.
  - You can find the current lock status of the PTM trace registers by reading the ETMOSLSR. See *OS Lock Status Register, ETMOSLSR* on page 3-110.

  This section gives more information about using these registers.
- In PFTv1.0 you can save and restore the ETM trace registers using the ETMOSSRR, see *OS Save and Restore Register, ETMOSSRR* on page 3-112. The registers are saved as, or restored using, a continuous block of data. You do not have to save or restore the registers individually.
- From PFTv1.1 the ETMOSSRR is not included and the registers are saved and restored individually.
- From PFTv1.1 the ETMPDCR provides the ability to request power to the PTM trace registers. See *Device Power-Down Control Register, ETMPDCR* on page 3-113.
- The PTM implementation includes the ETMPDSR, that indicates when the PTM trace registers are powered up. See *Device Power-Down Status Register, ETMPDSR* on page 3-114. This register also indicates whether the state of the PTM trace registers has been lost because of a power-down, and prevents access to the PTM trace registers until after the debugger has read the ETMPDSR.

It is IMPLEMENTATION DEFINED whether this power-down support is included in a PTM implementation. However, you can always read the ETMOSLSR to find whether this feature is implemented.

When power down support is implemented, a Direct JTAG interface to the PTM registers is not permitted. Access to the PTM registers from an external debugger must use the ARM Debug Interface v5. For more information see the ARM Debug Interface v5 Architecture Specification.

In a system that supports multiple power domains, the PTM might be split into two domains:

- One domain, typically called the PTM core power domain, is where all the PTM trace registers are located. Typically, this domain is where most of the PTM resources and trace generation logic are found, because normally these must run at the same clock speed as the processor.
- The other domain, typically called the PTM Debug domain, is where the programming interface and PTM Management Registers are located.

This power domain split enables the processor and core domain of the PTM to be dynamically powered down while permitting a debugger to maintain communication with the PTM, and to determine the part of the PTM that is powered down.

Typically, the PTM core power domain is the same power domain as the processor. However, some implementations might separate the PTM core power domain from the processor power domain to enable the PTM core power domain to be powered down when the PTM is not in use.

In a typical CoreSight system, the PTM Debug domain is the same power domain as the other debug and trace components. This permits an implementation to power down all the debug and trace logic when not in use.

Table 3-81 shows how the ETMOSLSR indicates the power down support that is implemented.

Table 3-81 Determining the level of power down support

PFT Architecture	ETMOSLSR[3]	ETMOSLSR[0]	Power down support
PFTv1.0	0	0	SinglePower, see *SinglePower in PFTv1.0*
PFTv1.0	0	1	Full Support, see *Full Power Down Support in PFTv1.0*
PFTv1.1 and later	0	0	SinglePower, see *SinglePower from PFTv1.1* on page 3-134
PFTv1.1 and later	1	0	Full Support, see *Full Power Down Support from PFTv1.1* on page 3-135

Values not shown are Reserved.

3.17.1 Power down support in PFTv1.0

Two levels of power down support are provided in PFTv1.0, SinglePower and Full support.

SinglePower in PFTv1.0

A SinglePower implementation can be identified by reading the ETMOSLSR. If bit [3] and bit [0] of the ETMOSLSR are both 0 then this is a SinglePower implementation.

SinglePower implementations do not support tracing over a power down. To avoid losing PTM state when the processor is powered down, one of the following options must be used:

- Do not power down the processor. This can be achieved by setting the DBGNOPWRDWN bit in the processor debug registers.
- The PTM must remain powered when the processor is powered down. This involves implementing the PTM in a separate power domain from the processor.
- The PTM registers must be manually saved by software running on the processor. This mechanism does not guarantee that the processor and an external debugger will not conflict while the saving and restoring is taking place.

A SinglePower implementation has the following attributes:

- The OS Lock is not implemented:
  - — The ETMOSLAR is not implemented and ignores writes
  - — The ETMOSSRR is not implemented and accesses to the ETMOSSRR are Unpredictable
  - — The ETMOSLSR always reads as `0x00000000`
- The ETMPDSR always reads as `0x00000001`.

For more details on Access permissions in SinglePower implementations see *Access permissions for PFTv1.0 SinglePower implementations* on page 3-140.

Full Power Down Support in PFTv1.0

An implementation with Full Power Down support can be identified by reading the ETMOSLSR. If bit [3] is b0 and bit [0] is b1 then the implementation has full power down support.

Full power down support has the following attributes:

- The OS Lock is implemented:
  - — The ETMOSLAR is implemented
  - — The ETMOSSRR is implemented and is used to save and restore the PTM trace registers
  - — When the OS Lock is set, accesses to Trace registers return an Error.

- The ETMPDSR is fully implemented:
  - When the StickyState bit [1] is set, accesses to Trace registers return an Error.

For more details on Access permissions implementations with full power down support see *Access permissions for PFTv1.0 with multiple power implementations* on page 3-143.

To save the PTM trace registers, perform the following steps:

1. If you are using a memory-mapped interface, unlock the CoreSight Lock, if implemented. See *About the lock registers* on page 3-118.
2. Read the ETMPDSR to clear the StickyState bit if it is set. See *Device Power-Down Status Register, ETMPDSR* on page 3-114.
3. Set the OS Lock using the ETMOSLAR. See *About the Operating System Save and Restore registers* on page 3-110.
4. Use the ETMOSSRR to read out the ETM registers and save them to memory. See *OS Save and Restore Register, ETMOSSRR* on page 3-112.
5. The PTM core domain can now be powered down.

To restore the PTM trace registers, perform the following steps:

1. If you are using a memory-mapped interface, unlock the CoreSight Lock, if implemented. See *About the lock registers* on page 3-118.
2. Read the ETMPDSR to clear the StickyState bit. See *Device Power-Down Status Register, ETMPDSR* on page 3-114.
3. Set the OS Lock if it is not already set using the ETMOSLAR. See *About the Operating System Save and Restore registers* on page 3-110.
4. Use the ETMOSSRR to restore the ETM registers from memory. See *OS Save and Restore Register, ETMOSSRR* on page 3-112.
5. Clear the OS Lock using the ETMOSLAR. See *OS Save and Restore Register, ETMOSSRR* on page 3-112.

3.17.2 Power down support from PFTv1.1

Two levels of power down support are provided in PFTv1.1, SinglePower and Full support.

SinglePower from PFTv1.1

A SinglePower implementation can be identified by reading the ETMOSLSR. If bit [3] and bit [0] of the ETMOSLSR are both 0 then this is a SinglePower implementation.

SinglePower implementations do not support tracing over a power down. To avoid losing PTM state when the processor is powered down, one of the following options must be used:

- Do not power down the processor. This can be achieved by setting the DBGNOPWRDWN bit in the processor debug registers.
- The PTM must remain powered when the processor is powered down. This involves implementing the PTM in a separate power domain from the processor.
- The PTM registers must be manually saved by software running on the processor. This mechanism does not guarantee that the processor and an external debugger will not conflict while the saving and restoring is taking place.

A SinglePower implementation has the following attributes:

- The OS Lock is not implemented:
  - The ETMOSLAR is not implemented and ignores writes

    - The ETMOSSRR is not implemented and accesses to the ETMOSSRR are UNPREDICTABLE
    - The ETMOSLSR always reads as `0x00000000`.

- The ETMPDSR always reads as `0x00000001`.

For more details on Access permissions in SinglePower implementations see *Access permissions for PFTv1.1 SinglePower implementations* on page 3-146.

Full Power Down Support from PFTv1.1

An implementation with Full Power Down support can be identified by reading the ETMOSLSR. If bit [3] is 1 and bit [0] is 0 then the implementation has full power down support.

Full power down support from PFTv1.1 has the following attributes:

- The OS Lock is implemented:
    - The OS Lock is set from a PTM reset.
    - The ETMOSLAR is implemented.
    - The ETMOSSRR is not implemented. Trace registers must be manually saved and restored while the OS Lock is set.
    - When the OS Lock is set, accesses to Trace registers from an external debugger return an Error.
- The ETMPDSR is fully implemented:
    - The StickyState bit has no effect on accesses to any registers.

For more details on Access permissions implementations with full power down support see *Access permissions for PFTv1.1 with multiple power implementations* on page 3-149.

To save the PTM trace registers, perform the following steps:

1. If you are using a memory-mapped interface, unlock the CoreSight Lock, if implemented. See *About the lock registers* on page 3-118.
2. Set the OS Lock using the ETMOSLAR. See *About the Operating System Save and Restore registers* on page 3-110.
3. Poll ETMSR bit [1] until it becomes set, indicating the PTM is idle. See *Status Register, ETMSR* on page 3-81.
4. Manually read the PTM trace registers and save the contents to memory.
5. The PTM core domain can now be powered down.

If the procedure is terminated early, for example if the power down sequence is terminated before this procedure is complete, if the OS Lock is cleared before the ETMSR bit [1] is set then the PTM might not restart tracing immediately and the PTM resources might not become active immediately.

To restore the PTM trace registers, perform the following steps:

1. If you are using a memory-mapped interface, unlock the CoreSight Lock, if implemented. See *About the lock registers* on page 3-118.
2. The OS Lock must be set from an PTM reset. Check this by reading the ETMOSLSR. See *OS Lock Status Register, ETMOSLSR* on page 3-110.
3. Poll the ETMSR bit [1] until it becomes set, indicating the PTM is idle. See *Status Register, ETMSR* on page 3-81.
4. Manually restore the PTM trace registers from memory.
5. Clear the OS Lock using the ETMOSLAR. See *OS Lock Access Register, ETMOSLAR* on page 3-110.

Significant changes to power down support introduced in PFTv1.1

- The ETMOSSRR is never implemented.
- If implemented, the OS Lock is set from a PTM reset.
- If implemented, if the OS Lock is set it only causes an error response to debugger accesses to the PTM Trace registers.
- If implemented, the ETMPDSR bit [1], Sticky Register State, no longer has any effect on accesses to any PTM registers.
- The OS Lock status is visible in the ETMPDSR.
- The Claim tag registers are now PTM Trace registers and must be saved and restored manually.
- The ETMSR bit [1] becomes set when the PTM becomes idle after setting the OS Lock. This is used to indicate that the PTM is sufficiently idle for the PTM trace registers to be saved or restored.
- Access permissions to some registers are changed. See *About the access permissions for PTM registers* on page 3-138

3.17.3 PTM behavior when the OS Lock is set

You set the OS Lock by writing the lock key of `0xC5ACCE55` to the ETMOSLAR. See *OS Lock Access Register, ETMOSLAR* on page 3-110. When the OS Lock is set all PTM functions are disabled. This means that:

- Tracing becomes inactive. The FIFO is emptied and no more trace is produced. From PFTv1.1 the ETMSR bit [1] becomes set when the PTM has fully drained and has become idle.
- The PTM holds the counters, sequencer, Instrumentation resources, and start/stop block in their current state.
- The external outputs are forced LOW.
- A Trigger cannot be generated. Any trigger generated before the OS Lock is set must be output before ETMSR bit [1] is set and before power is removed.
- ETMSR bit [3] is unchanged. This is only cleared by an ETM reset or when the ETM programming bit is cleared. It is not cleared when the OS Lock is unset.
- ETMCR bit [10] maintains its current value. See *Main Control Register, ETMCR* on page 3-75.
- In PFTv1.0 any attempt to access the PTM trace registers causes an error response. See *About the access permissions for PTM registers* on page 3-138.
- From PFTv1.1 any attempt by an external debugger to access the PTM Trace Registers causes an error response.
- Whether the address range comparators maintain their state is IMPLEMENTATION DEFINED. See *Address comparators* on page 3-39.

You must use the WFI mechanism to ensure that the FIFO is empty before you remove power from the macrocell.

Note

The WFI mechanism must be present on any processor and PTM combination that provides power-down support. See the *Technical Reference Manual* (TRM) for your processor and PTM macrocell for more information.

When the OS Lock is cleared, you can restart tracing. The counters, sequencer, start/stop block and Instrumentation resources continue operating from their held state. If the implementation maintains the sticky state of the address range comparators during OS Lock then the address range comparators continue operating with this held sticky state. However, if the PTM has been powered down since the OS Lock was set:

- the state of the counters, sequencer, start/stop block is restored as part of the OS Save/Restore sequence
- the state of the Instrumentation resources, and the state of the address range comparators, is lost.

3.17.4 Guidelines for the PTM trace registers to be saved and restored

Table 3-14 on page 3-66 defined the split of PTM registers into trace and management registers.

The registers that are included in the save and restore mechanism is IMPLEMENTATION DEFINED. However, the mechanism must include all registers whose contents are lost in a power-down. Table 3-82 gives a list of the PTM registers typically included in the save and restore mechanism.

Table 3-82 Typical list of PTM registers to be saved and restored

Register number	Register offset	Register name
`0x000`	`0x000`	Main Control Register, ETMCR
`0x002`	`0x008`	Trigger Event Register, ETMTRIGGER
`0x004`	`0x010`	Status Register, ETMSR
`0x006`	`0x018`	TraceEnable Start/Stop Control Register, ETMTSSCR
`0x008`	`0x020`	TraceEnable Event Register, ETMTEEVR
`0x009`	`0x024`	TraceEnable Control Register, ETMTECR1
`0x00B`	`0x02C`	FIFOFULL Level Register, ETMFFLR
`0x010-0x01F`	`0x040-0x07C`	Address Comparator Value Registers, ETMACVR*n*
`0x020-0x02F`	`0x080-0x0BC`	Address Comparator Access Type Registers, ETMACTR*n*
`0x050-0x053`	`0x140-0x14C`	Counter Reload Value Registers, ETMCNTRLDVR*n*
`0x054-0x057`	`0x150-0x15C`	Counter Enable Event Registers, ETMCNTENR*n*
`0x058-0x05B`	`0x160-0x16C`	Counter Reload Event Registers, ETMCNTRLDEVR*n*
`0x05C-0x05F`	`0x170-0x17C`	Counter Value Registers, ETMCNTVR*n*
`0x060-0x065`	`0x180-0x194`	Sequencer State Transition Event Registers, ETMSQ*ab*EVR
`0x067`	`0x19C`	Current Sequencer State Register, ETMSQR
`0x068-0x06B`	`0x1A0-0x1AC`	External Output Event Registers, ETMEXTOUTEVR*n*
`0x06C-0x06E`	`0x1B0-0x1B8`	Context ID Comparator Value Registers, ETMCIDCVR*n*
`0x06F`	`0x1BC`	Context ID Comparator Mask Register, ETMCIDCMR
`0x078`	`0x1E0`	Synchronization Frequency Register, ETMSYNCFR
`0x07B`	`0x1EC`	Extended External Input Selection Register, ETMEXTINSELR
`0x07E`	`0x1F8`	Timestamp Event Register, ETMTSEVR
`0x07F`	`0x1FC`	Auxiliary Control Register, ETMAUXCR[a]
`0x080`	`0x200`	CoreSight Trace ID Register, ETMTRACEIDR
`0x090`	`0x240`	VMID Comparator Value Register, ETMVMIDCVR[a]
`0x3E8`	`0xFA0`	Claim Tag Set Register, ETMCLAIMSET[a]
`0x3E9`	`0xFA4`	Claim Tag Clear Register, ETMCLAIMCLR[a]

a. From PFTv1.1.

3.18 About the access permissions for PTM registers

A PTM implements controls on accesses to the PTM registers. These controls depend on the register access model implemented by the PTM. The usual access models are summarized in *PTM register access models* on page 3-68.

A PTM might be part of a system that is implemented with multiple power domains. A typical implementation might implement two domains that can be independently powered down, for example:

- the core power domain powers the processor that is being traced, and contains most of the PTM logic, including the trace registers
- a debug power domain contains the trace output logic, including the programming interface or interfaces.

However, the system that includes the PTM might be implemented in a single power domain. An implementation of this type is called a SinglePower system.

The access controls on memory-mapped accesses to PTM registers depend on whether the system is implemented with multiple power domains, or as a SinglePower system.

Note

If your PTM is accessed using an ARM Debug Interface v5, see the access permissions descriptions in the *ARM Debug Interface v5 Architecture Specification*.

The types of access that can be made to the PTM registers are described in *Access types*. The access permissions that control PTM register accesses, and restrictions on PTM register accesses, are described in:

- *Restrictions on accesses using a Direct JTAG connection* on page 3-139
- *Access permissions for PFTv1.0 SinglePower implementations* on page 3-140
- *Access permissions for PFTv1.0 with multiple power implementations* on page 3-143
- *Access permissions for PFTv1.1 SinglePower implementations* on page 3-146
- *Access permissions for PFTv1.1 with multiple power implementations* on page 3-149

3.18.1 Access types

The PTM access permission descriptions refer to the following types of access:

Debugger accesses

These are accesses from an external debug device. ARM recommends that these use an ARM Debug Interface v5, see the *ARM Debug Interface v5 Architecture Specification*, that provides a memory-mapped interface to the ETM registers.

An external debug device can also access a PTM through a Direct JTAG interface, or a similar interface.

Processor accesses through a memory-mapped interface

These are accesses from a device in the system, such as a processor, that use the memory-mapped interface to the PTM registers, see *Memory-mapped access* on page 3-68.

If an implementation includes a memory-mapped processor interface to the PTM registers then it must also implement the software lock, controlled by the ETMLAR.

Processor accesses through the coprocessor interface

These are accesses that originate from an on-chip device, such as a processor, using the coprocessor interface to the PTM, see *Coprocessor access* on page 3-66.

A PTM can distinguish between memory-mapped debugger accesses and memory-mapped processor accesses.

3.18.2 Meanings of terms and abbreviations used in this section

The following terms and abbreviations are used in the tables that summarize the access permissions:

Error	Slave-generated error response. Writes are Ignored and reads return an UNKNOWN value. For a memory-mapped access, an error is returned through the memory system. For a coprocessor access, an Undefined Instruction exception is taken.
NPoss	Not possible. Accessing the trace registers while the processor is powered down is not possible if a single power domain is implemented. The response is system dependent and IMPLEMENTATION DEFINED.
OK	The read or write access succeeds. Writes to RO locations are ignored. Reads from RAZ/WO locations return zero.
Unp	The access has UNPREDICTABLE results. Reads return an UNKNOWN value.
WI	Writes Ignored. Reads return the register value.
RAZ	Reads-As-Zero.
Unk	UNKNOWN
SBZP	Should-Be-Zero-or-Preserved.
ImpDef	IMPLEMENTATION DEFINED.
CS Lock	CoreSight Lock. Indicated by the ETMLSR. See *Lock Status Register, ETMLSR* on page 3-119. This is one of the Management registers.
ETM_PD	ETM Power Down status. Indicated by the ETM Power Down bit, bit [0] of the PTM. See *Main Control Register, ETMCR* on page 3-75.

3.18.3 Restrictions on accesses using a Direct JTAG connection

If an implementation includes a Direct JTAG connection to the PTM then, when using that connection:

- it is not possible to access the PTM management registers
- it is not possible to access the ETMLAR, and the state of the CoreSight lock does not affect register accesses
- it is not possible to access the OS Save and Restore mechanism
- if the core power domain is powered down it is not possible to access the PTM trace registers.

Note

An ARM Debug Interface v5 can include a JTAG-like external interface. Such an interface can provide memory-mapped access to the PTM registers. This section does not apply to such an implementation. For more information see the *ARM Debug Interface v5 Architecture Specification*.

3.18.4 Effect of DBGSWENABLE on register access

The ARM Debug Interface version 5 defines a signal, **DBGSWENABLE**, that can be used to disable access to some of the processor registers. If this interface is implemented, from PFTv1.1, then **DBGSWENABLE** has the following effects on accesses to the PTM registers:

- **DBGSWENABLE** has no effect on external debugger accesses.
- If **DBGSWENABLE** is LOW, then memory-mapped accesses to all PTM registers return Error, otherwise accesses are unaffected.
- **DBGSWENABLE** has no effect on coprocessor accesses to PTM registers.

To ensure that on-chip software can save and restore PTM registers, ARM recommends that **DBGSWENABLE** is held HIGH.

3.19 Access permissions for PFTv1.0 SinglePower implementations

This section describes register access permissions for a PTM that follows version 1.0 of the PFT protocol, where the processor and PTM are implemented in the same power domain.

3.19.1 PTM state definitions, PFTv1.0 SinglePower implementations

The following list shows the definitions of PTM states for SinglePower implementations in PFTv1.0. These states determine the behavior of accesses to the registers listed in the tables in this section.

No Power This behavior applies if the PTM is powered down. Also, for memory-mapped accesses, this state applies when **DBGSWENABLE** is LOW.

Non-Privileged

This behavior applies to coprocessor accesses when all of the following apply:

- the PTM is not in the No Power state
- the processor is operating in a Non-Privileged mode
- accesses to the trace unit are disabled using the CPACR, NSACR, or HCPTR.

If the PTM is in a state which is not covered by one of these definitions then the general access permissions apply as defined in the Otherwise column in each table.

3.19.2 Debugger accesses, PFTv1.0 SinglePower implementations

Table 3-83 shows the behavior of debugger accesses in a SinglePower implementation in PFTv1.0. See *PTM state definitions, PFTv1.0 SinglePower implementations* for the meanings of the column headings.

Table 3-83 Debugger accesses

Register	PTM state	
	No Power	Otherwise
Trace registers	Error	OK[a]
ETMLSR	Error	OK/RAZ
ETMLAR	Error	WI
ETMPDSR	Error	OK
ETMOSLSR	Error	OK
ETMOSLAR	Error	OK
ETMOSSRR	Error	Unp
ETMDEVID, ETMAUTHSTATUS	Error	OK
Other Management	Error	OK
Reserved Trace	Error	UNK/SBZP
Reserved Management	Error	UNK/SBZP

a. When ETM_PD is 1, register writes to all Trace registers except certain bits of the ETMCR might be ignored

3.19.3 Memory-mapped accesses, PFTv1.0 SinglePower implementations

Table 3-84 shows the behavior of memory-mapped accesses in a SinglePower implementation in PFTv1.0. See *PTM state definitions, PFTv1.0 SinglePower implementations* on page 3-140 for the meanings of the column headings.

Table 3-84 Memory-mapped accesses

Register	PTM state	
	No Power	Otherwise
Trace Registers	Error	OK[a, b]
ETMLSR	Error	OK
ETMLAR[c]	Error	OK
ETMPDSR	Error	OK
ETMOSLSR	Error	OK
ETMOSLAR	Error	OK[b]
ETMOSSRR	Error	UNP
ETMDEVID, ETMAUTHSTATUS	Error	OK
Other Management	Error	OK[b]
Reserved Trace	Error	UNK/SBZP
Reserved Management	Error	UNK/SBZP

a. When ETM_PD is 1, register writes to all Trace registers except certain bits of the ETMCR might be ignored

b. When the CS Lock is set, these registers are WI.

c. ETMLAR is not visible to Debugger accesses, so writes are ignored.

3.19.4 Coprocessor accesses, PFTv1.0 SinglePower implementations

Table 3-85 shows coprocessor access permissions for SinglePower implementations in PFTv1.0. See *PTM state definitions, PFTv1.0 SinglePower implementations* on page 3-140 for the meanings of the column headings.

Table 3-85 Coprocessor accesses

Register	PTM state		
	No Power	Non-Privileged	Otherwise
Trace registers	NPoss	Error	OK[a]
ETMLSR	NPoss	Error	OK/RAZ
ETMLAR[b]	NPoss	Error	WI
ETMPDSR	NPoss	Error	OK
ETMOSLSR	NPoss	Error	OK
ETMOSLAR	NPoss	Error	OK

Table 3-85 Coprocessor accesses (continued)

Register	PTM state		
	No Power	Non-Privileged	Otherwise
ETMOSSRR	NPoss	Error	UNP
ETMDEVID, ETMAUTHSTATUS	NPoss	Error	OK
Other Management	NPoss	Error	OK
Reserved Trace	NPoss	Error	UNK/SBZP
Reserved Management	NPoss	Error	UNK/SBZP

a. When ETM_PD is 1, register writes to all Trace registers except certain bits of the ETMCR might be ignored

b. ETMLAR is not visible to Debugger accesses, so writes are ignored

3.20 Access permissions for PFTv1.0 with multiple power implementations

This section describes register access permissions for a PTM that follows version 1.0 of the PFT protocol, where the processor and PTM are implemented in different power domains.

3.20.1 PTM state definitions, PFTv1.0 with multiple power implementations

The following list shows the definitions of PTM states for multiple power implementations in PFTv1.0. These states determine the behavior of accesses to the registers listed in the tables in this section.

No Debug Power

This behavior applies if the PTM is powered down. Also, for memory-mapped accesses, this state applies when **DBGSWENABLE** is LOW.

No Core Power

This behavior applies when all the following apply:

- the PTM is not in the No Debug Power state
- the core domain is powered down.

Sticky State Set

This behavior applies when all the following apply:

- the PTM is not in the No Debug Power state
- the PTM is not in the No Core Power state
- the Sticky State is set to 1.

OS Lock set This behavior applies when all the following apply:

- the PTM is not in the No Debug Power state
- the PTM is not in the No Core Power state
- the PTM is not in the Sticky State Set state
- the OS Lock is set to 1.

Non-Privileged

This behavior applies to coprocessor accesses when all the following apply:

- the PTM is not in the No Debug Power state
- the PTM is not in the No Core Power state
- the processor is operating in a Non-Privileged mode
- accesses to the trace unit are disabled using the CPACR, NSACR, or HCPTR.

This state takes precedence over the Sticky State Set or OS Lock Set states.

If the PTM is in a state which is not covered by one of these definitions then the general access permissions apply as defined in the Otherwise column in each table.

3.20.2 Debugger accesses, PFTv1.0 with multiple power implementations

Table 3-86 shows debugger access permissions for full tracing implementations in PFTv1.0. See *PTM state definitions, PFTv1.0 with multiple power implementations* for the meanings of the column headings.

Table 3-86 Debugger accesses

Register	PTM state				
	No Debug Power	No Core Power	Sticky State Set	OS Lock Set	Otherwise
Trace registers	Error	Error	Error	Error	OK[a]
ETMLSR	Error	OK/RAZ	OK/RAZ	OK/RAZ	OK/RAZ
ETMLAR[b]	Error	WI	WI	WI	WI
ETMPDSR	Error	OK	OK	OK	OK

Table 3-86 Debugger accesses (continued)

Register	PTM state				
	No Debug Power	No Core Power	Sticky State Set	OS Lock Set	Otherwise
ETMOSLSR	Error	OK	OK	OK	OK
ETMOSLAR	Error	UNP	OK	OK	OK
ETMOSSRR	Error	UNP	UNP	OK	UNP
ETMDEVID, ETMAUTHSTATUS	Error	OK	OK	OK	OK
Other Management	Error	OK	OK	OK	OK
Reserved Trace	Error	Error	Error	Error	UNK/SBZP
Reserved Management	Error	UNK/SBZP	UNK/SBZP	UNK/SBZP	UNK/SBZP

a. When ETM_PD is 1, register writes to all Trace registers except certain bits of the ETMCR might be ignored.

b. ETMLAR is not visible to Debugger accesses, so writes are ignored.

3.20.3 Memory-mapped accesses, PFTv1.0 with multiple power implementations

Table 3-87 shows memory-mapped access permissions for full tracing implementations in PFTv1.0. See *PTM state definitions, PFTv1.0 with multiple power implementations* on page 3-143 for the meanings of the column headings.

Table 3-87 Memory-mapped accesses

Register	PTM state				
	No Debug Power	No Core Power	Sticky State set	OS Lock Set	Otherwise
Trace registers	Error	Error	Error	Error	OK[a, c]
ETMLSR	Error	OK	OK	OK	OK
ETMLAR	Error	OK	OK	OK	OK
ETMPDSR	Error	OK[b]	OK[b]	OK[b]	OK[b]
ETMOSLSR	Error	OK	OK	OK	OK
ETMOSLAR	Error	UNP	OK[c]	OK[c]	OK[c]
ETMOSSRR	Error	UNP	UNP	OK[c]	UNP
ETMDEVID, ETMAUTHSTATUS	Error	OK[c]	OK[c]	OK[c]	OK[c]
Other Management	Error	OK[c]	OK[c]	OK[c]	OK[c]
Reserved Trace	Error	Error	Error	Error	UNK/SBZP
Reserved Management	Error	UNK/SBZP	UNK/SBZP	UNK/SBZP	UNK/SBZP

a. When ETM_PD is 1, register writes to all Trace registers except certain bits of the ETMCR might be ignored.

b. When the CS Lock is set, reads from the ETMPDSR do not clear the Sticky State.

c. When the CS Lock is set, these registers are WI.

3.20.4 Coprocessor accesses, PFTv1.0 with multiple power implementations

Note

Coprocessor access to these registers is not possible when the core domain is powered down.

Table 3-88 shows coprocessor access permissions for full tracing implementations in PFTv1.0. See *PTM state definitions, PFTv1.0 with multiple power implementations* on page 3-143 for the meanings of the column headings.

Table 3-88 Coprocessor accesses

Register	PTM state					
	No Debug Power	No Core Power	Non- Privileged	Sticky State Set	OS Lock Set	Otherwise
Trace registers	Error	NPoss	Error	Error	Error	OK[a]
ETMLSR	Error	NPoss	Error	OK/RAZ	OK/RAZ	OK/RAZ
ETMLAR[b]	Error	NPoss	Error	WI	WI	WI
ETMPDSR	Error	NPoss	Error	OK	OK	OK
ETMOSLSR	Error	NPoss	Error	OK	OK	OK
ETMOSLAR	Error	NPoss	Error	OK	OK	OK
ETMOSSRR	Error	NPoss	Error	UNP	OK	UNP
ETMDEVID, ETMAUTHSTATUS	Error	NPoss	Error	OK	OK	OK
Other Management	Error	NPoss	Error	OK	OK	OK
Reserved Trace	Error	NPoss	Error	UNP	UNP	UNK/SBZP
Reserved Management	Error	NPoss	Error	UNK/SBZP	UNK/SB ZP	UNK/SBZP

a. When ETM_PD is 1, register writes to all Trace registers except certain bits of the ETMCR might be ignored.

b. ETMLAR is not visible to coprocessor accesses, so writes are ignored.

3.21 Access permissions for PFTv1.1 SinglePower implementations

This section describes register access permissions for a PTM that follows version 31.1 of the PFT protocol, where the processor and PTM are implemented in the same power domain.

3.21.1 PTM state definitions, PFTv1.1 SinglePower implementations

The following list shows the definitions of PTM states for SinglePower implementations in PFTv1.1. These states determine the behavior of accesses to the registers listed in the tables in this section.

No Power This behavior applies if the PTM is powered down. Also, for memory-mapped accesses, this state applies when **DBGSWENABLE** is LOW.

Non-Privileged

This behavior applies to coprocessor accesses when all of the following apply:

- the PTM is not in the No Power state
- the processor is operating in a Non-Privileged mode
- accesses to the trace unit are disabled using the CPACR, NSACR, or HCPTR.

If the PTM is in a state which is not covered by one of these definitions then the general access permissions apply as defined in the Otherwise column in each table.

3.21.2 Debugger accesses, PFTv1.1 SinglePower implementations

Table 3-89 shows the behavior of debugger accesses in a SinglePower implementation in PFTv1.1. See *PTM state definitions, PFTv1.1 SinglePower implementations* for the meanings of the column headings.

Table 3-89 Debugger accesses

Register	PTM state	
	No Power	Otherwise
Trace registers	Error	OK[a]
ETMLSR	Error	OK/RAZ
ETMLAR[b]	Error	UNP
ETMPDSR	Error	OK
ETMOSLSR	Error	OK
ETMOSLAR	Error	OK
ETMOSSRR	Error	unp
ETMDEVID, ETMAUTHSTATUS	Error	OK
ETMITCTRL	Error	OK
Other Management	Error	OK
Reserved Trace	Error	UNK/SBZP
Reserved Management	Error	UNK/SBZP

a. When ETM_PD is 1, register writes to all Trace registers except certain bits of the ETMCR might be ignored

b. ETMLAR is not visible to Debugger accesses, so accesses are UNPREDICTABLE.

3.21.3 Memory-mapped accesses, PFTv1.1 SinglePower implementations

Table 3-90 shows the behavior of memory-mapped accesses in a SinglePower implementation in PFTv1.1. See *PTM state definitions, PFTv1.1 SinglePower implementations* on page 3-146 for the meanings of the column headings.

Table 3-90 Memory-mapped accesses

Register	PTM state	
	No Power	Otherwise
Trace Registers	Error	OK[a, b]
ETMLSR	Error	OK
ETMLAR	Error	OK
ETMPDSR	Error	OK
ETMOSLSR	Error	OK
ETMOSLAR	Error	OK[b]
ETMOSSRR	Error	UNP
ETMDEVID, ETMAUTHSTATUS	Error	OK
ETMITCTRL	Error	OK[b]
Other Management	Error	OK[b]
Reserved Trace	Error	UNK/SBZP
Reserved Management	Error	UNK/SBZP

a. When ETM_PD is 1, register writes to all Trace registers except certain bits of the ETMCR might be ignored

b. When the CS Lock is set, these registers are WI.

3.21.4 Coprocessor accesses, PFTv1.1 SinglePower implementations

Table 3-91 shows coprocessor access permissions for SinglePower implementations in PFTv1.1. See *PTM state definitions, PFTv1.1 SinglePower implementations* on page 3-146 for the meanings of the column headings.

Table 3-91 Coprocessor accesses

Register	PTM state		
	No Power	**Non-Privileged**	**Otherwise**
Trace registers	NPoss	Error	OK[a]
ETMLSR	NPoss	Error	UNP
ETMLAR	NPoss	Error	UNP
ETMPDSR	NPoss	Error	UNP
ETMOSLSR	NPoss	Error	OK
ETMOSLAR	NPoss	Error	OK
ETMOSSRR	NPoss	Error	UNP
ETMDEVID, ETMAUTHSTATUS	NPoss	Error	OK
ETMITCTRL	NPoss	Error	UNP
Other Management	NPoss	Error	UNP
Reserved Trace	NPoss	Error	UNK/SBZP
Reserved Management	NPoss	Error	UNP

a. When ETM_PD is 1, register writes to all Trace registers except certain bits of the ETMCR might be ignored

3.22 Access permissions for PFTv1.1 with multiple power implementations

This section describes register access permissions for a PTM that follows version 1.1 of the PFT protocol, where the processor and PTM are implemented in different power domains.

Note

From PFTv1.1, for all multiple power implementations, bit [1] of ETMSR must be polled before saving or restoring state. See *Status Register, ETMSR* on page 3-81.

3.22.1 PTM state definitions, PFTv1.1 with multiple power implementations

The following list shows the definitions of PTM states for multiple power implementations, from PFTv1.1. These states determine the behavior of accesses to the registers listed in the tables in this section.

No Debug Power This behavior applies when the debug domain is powered down. Also, for memory-mapped accesses, this behavior applies when **DBGSWENABLE** is LOW.

No Core Power The behavior applies when all of the following apply:

- the core power domain is powered down.
- for debugger and memory-mapped accesses, the PTM is not in the No Debug Power state

OS Lock set The behavior applies when all of the following apply:

- the PTM is not in the No Debug Power state
- the PTM is not in the No Core Power state
- the OS Lock is set to 1.

Non-Privileged This behavior applies to coprocessor accesses when all of the following apply:

- the PTM is not in the No Debug Power state
- the PTM is not in the No Core Power state
- the processor is operation in a Non-Privileged mode
- accesses to the trace unit are disabled using the CPACR, NSACR, or HCPTR.

This state takes precedence over the OS Lock Set state.

If the PTM is in a state which is not covered by one of these definitions then the general access permissions apply as defined in the Otherwise column in each table.

3.22.2 Debugger accesses, PFTv1.1 with multiple power implementations

Table 3-92 shows debugger access permissions for multiple power implementations in PFTv1.1. See *PTM state definitions, PFTv1.1 with multiple power implementations* on page 3-149 for the meanings of the column headings.

Table 3-92 Debugger accesses

Register	PTM state			
	No Debug Power	No Core Power	OS Lock set	Otherwise
Trace registers	Error	Error	Error	OK[a]
ETMLSR	Error	OK/RAZ	OK/RAZ	OK/RAZ
ETMLAR[b]	Error	UNP	UNP	UNP
ETMPDSR	Error	OK	OK	OK
ETMOSLSR	Error	OK	OK	OK
ETMOSLAR	Error	Error	OK	OK
ETMOSSRR	Error	UNP	UNP	UNP
ETMDEVID, ETMAUTHSTATUS	Error	OK	OK	OK
ETMITCTRL	Error	OK	OK	OK
Other Management	Error	OK	OK	OK
Reserved Trace	Error	Error	Error	UNK/SBZP
Reserved Management	Error	UNK/SBZP	UNK/SBZP	UNK/SBZP

a. When ETM_PD is 1, register writes to all Trace registers except certain bits of the ETMCR might be ignored.

b. ETMLAR is not visible to debugger accesses, so accesses are UNPREDICTABLE.

3.22.3 Memory-mapped accesses, PFTv1.1 with multiple power implementations

Table 3-93 shows the behavior of memory-mapped PTM register accesses in a PFTv1.1 implementation that has separate debug and core power domains. See *PTM state definitions, PFTv1.1 with multiple power implementations* on page 3-149 for the meanings of the column headings.

Table 3-93 Memory-mapped accesses

Register	PTM state			
	No Debug Power	**No Core Power**	**OS Lock Set**	**Otherwise**
Trace registers	Error	Error	OK[d], [a]	OK[b], [d]
ETMLSR	Error	OK	OK	OK
ETMLAR	Error	OK	OK	OK
ETMPDSR	Error	OK[c]	OK[c]	OK[c]
ETMOSLSR	Error	OK	OK	OK
ETMOSLAR	Error	Error	OK[d]	OK[d]
ETMOSSRR	Error	UNP	UNP	UNP
ETMDEVID, ETMAUTHSTATUS	Error	OK[d]	OK[d]	OK[d]
ETMITCTRL	Error	IMPDEF	IMPDEF	OK[d]
Other Management	Error	OK[d]	OK[d]	OK[d]
Reserved Trace	Error	Error	UNK/SBZP	UNK/SBZP
Reserved Management	Error	UNK/SBZP	UNK/SBZP	UNK/SBZP

a. When the OS Lock is set, Trace registers must always be writeable regardless of the value of ETM_PD.

b. When ETM_PD is 1, register writes to all Trace registers except certain bits of the ETMCR might be ignored.

c. When the CS Lock is set, reads from the ETMPDSR do not clear the Sticky State.

d. When the CS Lock is set, these registers are WI.

3.22.4 Coprocessor accesses, PFTv1.1 with multiple power implementations

Note

Coprocessor access to these registers is not possible when the core domain is powered down.

Table 3-94 shows coprocessor access permissions for multiple power implementations in PFTv1.1. See *PTM state definitions, PFTv1.1 with multiple power implementations* on page 3-149 for the meanings of the column headings.

Table 3-94 Coprocessor accesses

Register	PTM state			
	No Core Power	Non- Privileged	OS Lock Set	Otherwise[a]
Trace registers	NPoss	Error	OK[b]	OK[c]
ETMLSR	NPoss	Error	UNP	UNP
ETMLAR[d]	NPoss	Error	UNP	UNP
ETMPDSR	NPoss	Error	UNP	UNP
ETMOSLSR	NPoss	Error	OK	OK
ETMOSLAR	NPoss	Error	OK	OK
ETMOSSRR	NPoss	Error	UNP	UNP
ETMDEVID, ETMAUTHSTATUS	NPoss	Error	OK	OK
ETMITCTRL	NPoss	Error	UNP	UNP
Other Management	NPoss	Error	UNP	UNP
Reserved Trace	NPoss	Error	UNK/SBZP	UNK/SBZP
Reserved Management	NPoss	Error	UNP	UNP

a. These settings also apply to the No Debug Power state. This allows the ETM state to be saved and restored, and the ETM to be configured, when parts of the ETM are powered down.

b. When the OS Lock is set, Trace registers must always be writeable regardless of the value of ETM_PD.

c. When ETM_PD is 1, register writes to all Trace registers except certain bits of the ETMCR might be ignored.

d. ETMLAR is not visible to coprocessor accesses, so writes are ignored.

3.23 Programming the PTM to trace all execution

If you want the PTM to trace all permitted processor execution, program the PTM registers as follows while the programming bit is set to 1:

- In the TraceEnable Control Register:
  - set bit [24] to 1
  - set all other bits to 0.

  See *TraceEnable Control Register, ETMTECR1* on page 3-84 for more information.
- Program the TraceEnable Event Register to `0x6F`, TRUE. See *TraceEnable Event Register, ETMTEEVR* on page 3-83.
- Program the Trigger Event Register with the trigger event. See *Trigger Event Register, ETMTRIGGER* on page 3-80.
- Program the CoreSight Trace ID Register with a valid trace source ID. See *CoreSight Trace ID Register, ETMTRACEIDR* on page 3-108.
- If you want the PTM to generate timestamps:
  - Set bit [28] of the Main Control Register to 1, to enable timestamping. See *Main Control Register, ETMCR* on page 3-75.
  - Program the Timestamp Event Register with a valid event. See *Timestamp Event Register, ETMTSEVR* on page 3-107.

Chapter 4
Program Flow Trace Protocol

This chapter describes the types of trace information that are output from the PTM trace port. It contains the following sections:

4.1 About the Program Flow Trace protocol

A PTM generates a program flow *trace stream*. The PFT protocol describes the formatting of this trace stream. It is constructed of multiple packets of one or more bytes of data. Each packet consists of a single header byte followed by zero or more payload bytes. In general, the number of payload bytes in a packet is variable.

The trace data is output over the trace output interface, for export to an off-chip Trace Port Analyzer or capture in an on-chip buffer. Typically, this interface is based on the *AMBA Trace Bus* (ATB) standard, for connection to a CoreSight trace capture device.

Note

Any instruction that requires tracing is presented to the PTM only when the processor commits it for execution.

4.2 PFT atoms

The PFT architecture uses *atoms* to indicate waypoint instruction execution. In addition, if you have enabled cycle-accurate tracing, atoms can be used to indicate cycle boundaries.

The atoms that can be generated by a PTM are:

E atom	Indicates that a branch instruction passed its condition code check.
N atom	Indicates that a branch instruction failed its condition code check.
W atom	When cycle-accurate tracing is enabled, indicates a cycle.

Normally, the PTM does not output atoms immediately, but assembles them into atom packets. When the PTM has assembled an atom packet it outputs it into the trace stream:

- when tracing is not cycle-accurate, atoms are assembled into single-byte packets, and a packet outputs between one and five atoms
- when tracing is cycle-accurate, an atom packet is 1-5 bytes, and outputs a single E or N atom, together with zero or more W atoms.

For details of the atom packets see *Atom packet* on page 4-164.

In normal operation, the PTM assembles as many atoms as possible into an atom packet, and then outputs that packet. This minimizes the amount of trace data generated. However, some situations require the PTM to output immediately all of the atoms that it has assembled.

In addition, some trace packets imply a PFT atom, and therefore no atom corresponding to the relevant instruction appears in the trace stream. For more information about trace packets see *Summary of PFT packets* on page 4-158 and *PFT packet formats* on page 4-161.

`ISB` instructions can be conditional when used in an IT block. From PFTv1.1, the E/N atom used to trace `ISB`s does not indicate whether the `ISB` passed or failed the condition code check. Instead, the E/N atom is used to indicate whether the `ISB` really executed.This applies to all forms of the `ISB` instruction:

- On an unconditional `ISB`, an E atom is traced.
- On a conditional `ISB` which is executed regardless of the condition code, an E atom is traced.
- On a conditional `ISB` which fails the condition code and is not executed, an N atom is traced.

When branch broadcasting is enabled, using bit [8] of the ETMCR, the following trace is generated:

- On an unconditional `ISB`, a branch packet is traced. The branch packet implies an E atom.
- On a conditional `ISB` which is executed regardless of the condition code, a branch packet is traced. The branch packet implies an E atom.
- On a conditional `ISB` which fails the condition code and is not executed, an N atom is traced.

4.3 Summary of PFT packets

A PTM assembles PFT trace, including PFT atoms, into packets. It then outputs these packets as a stream of trace data. This section summarizes the different PFT packets and their meanings. *PFT packet formats* on page 4-161 describes each packet format in detail.

The PFT architecture defines the following packets:

A-sync An A-sync (address synchronization) packet identifies a packet boundary. The first byte of data after the end of an A-sync packet is the header of a new packet.

For more information see *A-sync, alignment synchronization packet* on page 4-161.

I-sync An I-sync (instruction synchronization) packet specifies the processor state for the next instruction to be executed. It does not indicate that any instructions have been executed.

A debugger uses an I-sync packet to obtain instruction address synchronization and processor state information for the instruction at the specified address. It cannot start tracing until it has decoded an I-sync packet.

For more information see *I-sync, instruction synchronization packet* on page 4-162.

Atom Unless you have enabled cycle-accurate tracing, each atom packet is a single byte and is often called an atom header. In this case, the atom header contains one or more E or N atoms.

When cycle-accurate tracing is enabled, each atom packet contains a single E or N atom, and the cycle count value corresponding to that atom.

For each atom, a debugger must decode instructions from the current instruction until it reaches a waypoint instruction. It knows that the processor has executed each of these instructions. When the debugger decodes the waypoint instruction it updates:

- the next instruction address to be the target address of the waypoint instruction
- the processor instruction set state to the state for the instruction at that address.

The debugger obtains the required information from one or more of:

- the program image
- the return stack, if enabled
- the status of the atom, that is, whether it is an E or an N atom.

For more information see *Atom packet* on page 4-164.

Branch without exception

A branch address packet without an exception implies a single E atom and contains the branch target address.

The debugger processes the atom as described for the atom packet. It then uses the branch target address from the branch address packet to update the next instruction address and the processor instruction set state.

For more information see *Branch address packet* on page 4-166.

Branch with exception

In a branch address packet with an exception, the address in the packet is the exception vector address. The packet does not indicate any execution of instructions.

The debugger must update:

- the next instruction address to be the exception vector address
- the processor instruction set state to the state for the instruction at that address.

For more information see *Branch address packet* on page 4-166.

Waypoint update

A waypoint update packet contains a waypoint update address, that indicates an instruction that the PTM must upgrade to a waypoint instruction.

When a debugger decodes a waypoint update packet it must decode instructions from the current instruction until it reaches an instruction address that is greater than the waypoint update address. It knows that the processor has executed all instructions up to and including any instruction at the waypoint update address. With correct PTM operation the executed instructions cannot include any waypoint instructions.

The PTM generates waypoint update packets to indicate:

- Where execution had reached before an exception.
- That more than 4096 bytes of instructions have been executed since the last waypoint. This use of the waypoint update packet can help a debugger manage memory, and detect errors.

For more information see *Waypoint update packet* on page 4-177.

Context ID A Context ID packet contains a new context ID. However, the trace stream does not indicate precisely the point where the Context ID Register value changed.

When the debugger decodes a Context ID packet it must update the current Context ID. It knows that every instruction decoded after this packet was executed with this new Context ID.

For more information see *Context ID packet* on page 4-179.

Trigger A trigger packet indicates when a trigger occurred. It is inserted precisely in the trace stream.

A debugger is not required to do anything when it decodes a trigger packet, but it might indicate where the trigger occurred.

A trigger packet is always a single byte. Therefore, it is also called a trigger header.

For more information see *Trigger packet* on page 4-179.

Ignore An ignore packet has no effect.

A debugger does not have to do anything when it decodes an ignore packet, but it might indicate that it received the packet.

An ignore packet is always a single byte. Therefore, it is also called an ignore header.

For more information see *Ignore packet* on page 4-184.

Exception return

A PTM for an ARMv7-M processor generates an exception return packet to indicate the return from an exception handler. The packet indicates that the most recent waypoint instruction was the exception return instruction.

An exception return packet is always a single byte. Therefore, it is also called an exception return header.

For more information see *Exception return packet* on page 4-183.

Note

For PFTv1.0. a PTM for an ARMv7-A or ARMv7-R processor never generates exception return packets.

Timestamp A timestamp packet contains the absolute value of the timestamp.

To trace program execution, a debugger does not have to do anything when it decodes a timestamp packet. However, normally a debugger outputs the new timestamp value.

For more information see *Timestamp packet* on page 4-181.

A timestamp packet is sometimes called a T-sync packet. See *Timestamp synchronization* on page 4-193.

VMID packet

From PFTv1.1, the VMID packet indicates the Virtual Machine ID of the processor.

For more information see *VMID packets* on page 4-181.

4.4 Cycle-accurate tracing

Cycle-accurate tracing is enabled by setting the CycleAccurate bit, bit [12], in the Main Control Register to 1. However, cycle-accurate tracing in the PFT architecture differs from cycle-accurate tracing in previous ARM trace architectures.

In the PFT architecture, the PTM generates cycle-accurate trace by including an explicit cycle count in each of the following trace packets:

- I-sync
- atom packet
- branch
- timestamp.

In the cycle-accurate versions of these packets, the cycle count always indicates the number of cycles since the last cycle count output by the PTM:

- Each branch address packet includes its own cycle count.
- An atom packet includes only one atom, and includes the cycle count for that atom.
- In PFTv1.0, a timestamp packet includes the precise cycle count for the timestamp. In PFTv1.1, the timestamp packet always gives the cycle count as zero. Because the timestamp value is that of the last traced waypoint, the number of cycles between the timestamp and the waypoint is always zero.

This means that a debugger can determine:

- the number of cycles between any two waypoint instructions
- the timestamp for any waypoint instruction, by comparing the cycle count of the waypoint instruction with the cycle count of a timestamp packet.

The mechanism for cycle-accurate tracing means that, when you enable cycle-accurate tracing:

- The format of the I-sync packet is modified to include a cycle count. See *I-sync, instruction synchronization packet* on page 4-162
- The formats of the branch address and timestamp packets are extended to include a cycle count. See:
  - *Branch address packet* on page 4-166
  - *Timestamp packet* on page 4-181.
- A different format is used for atom packets. See *Atom packet* on page 4-164.

From PFTv1.1, the cycle count value in an I-Sync packet which indicates a Trace Overflow or Debug Exit is UNKNOWN and must not be relied on.

4.5 PFT packet formats

Table 4-1 lists the PFT packet formats.

Table 4-1 Packet formats

Header:		Payload	Category[a]	Remarks
Description	Value			
A-sync	b00000000	At least 5 bytes	Sync.	Alignment synchronization, see *A-sync, alignment synchronization packet*.
I-sync	b00001000	5-14 bytes	Sync.	Instruction flow synchronization, see *I-sync, instruction synchronization packet* on page 4-162.
Atom	b1xxxxxx0	0-4 bytes	Instruction	See *Atom packet* on page 4-164.
Branch address	bCxxxxxx1	0-11[b] bytes	Instruction	C is 1 if another byte follows, and 0 otherwise. See *Branch address packet* on page 4-166.
Waypoint update	b01110010	1-5 bytes	Instruction	See *Waypoint update packet* on page 4-177.
Trigger	b00001100	None	Misc.	See *Trigger packet* on page 4-179.
Context ID	b01101110	1-4 bytes	Instruction	See *Context ID packet* on page 4-179.
VMID	b00111100	1 byte	Instruction	See *VMID packets* on page 4-181
Timestamp[c]	b01000x10	1-14 bytes	Sync.	See *Timestamp packet* on page 4-181.
Exception return	b01110110	None	Instruction	See *Exception return packet* on page 4-183.
Ignore	b01100110	None	Misc.	See *Ignore packet* on page 4-184.

a. Sync. = Synchronization, Misc. = Miscellaneous.
b. The header byte forms part of the address section of the branch address packet.
c. The timestamp header is also referred to as a T-sync header. See *Timestamp synchronization* on page 4-193.

4.5.1 A-sync, alignment synchronization packet

Periodically, the PTM outputs an A-sync packet. Whenever the PTM is enabled, the first packet output is an A-sync packet. In addition, an A-sync packet is output:

- Whenever the Programming bit is cleared to 0. The A-sync packet is output before any other trace.
- Periodically, based on periodic synchronization requests. See *Synchronization* on page 4-191.

The A-sync packet is a sequence of five or more A-sync headers, b00000000, followed by the binary value b10000000, as Figure 4-1 shows. This is equivalent to a string of 47 or more 0 bits followed by a 1.

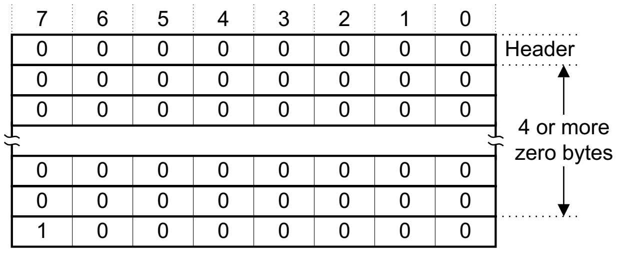


Figure 4-1 A-sync alignment synchronization packet

This bit sequence can occur only as the result of an A-sync packet. To synchronize, the decompressor must search for this sequence. Trace capture devices are usually byte-aligned, but this might not be the case for sub-byte ports. Therefore the decompressor must realign all data following the A-sync sequence if required.

After an A-sync packet, the next byte is always a header, that can be of any type.

4.5.2 I-sync, instruction synchronization packet

Periodically, the PTM outputs an I-sync packet. This packet consists of:

- The 1-byte I-sync header. This byte is always present.
- A 4-byte instruction address. These bytes are always present.
- The 1-byte Information byte. This byte is always present.
- Zero to five cycle count bytes. These bytes are present only when cycle-accurate tracing is enabled in the Main Control Register and the reason code is not b00. In any cycle count byte the C bit is set to 1 if another cycle count byte follows. This means that the C bit in the last cycle count byte is 0.
- Zero to four Context ID bytes. These bytes are present only when Context ID tracing is enabled in the Main Control Register. When Context ID tracing is enabled, 0, 1, 2, or 4 Context ID bytes are output, depending on the Context ID size selected by the ContextIDsize field of the Main Control Register. See *Main Control Register, ETMCR* on page 3-75.

Figure 4-2 shows the structure of the I-sync packet from PFTv1.1.

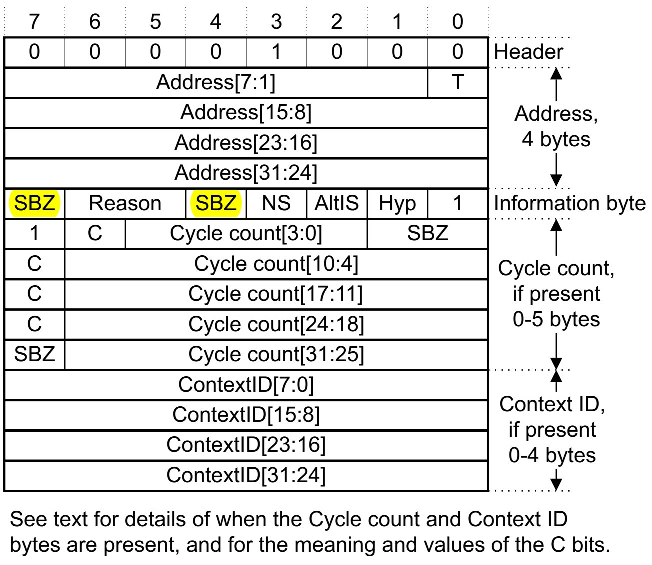


Figure 4-2 I-sync instruction synchronization packet, PFTv1.1

An I-sync packet comprises the following contiguous components:

I-sync header Indicates that this is an I-sync packet.

Address The address depends on why the packet was generated, as indicated by the Reason field. Normally, the address indicated for each reason code is:

Periodic	Most recent waypoint target address, or last nonperiodic I-sync address if there has not been a waypoint since the last nonperiodic I-sync packet.
Trace_on	Most recent waypoint target address or exception target address.
Overflow	Most recent waypoint target address or exception target address.
Debug	Address of the first instruction out of Debug state.

T Indicates the processor instruction set state when the instruction at the address specified by the Address field is executed. The possible values are:

0	ARM state.
1	Thumb state, or ThumbEE state. See the AltIS bit for more information.

Information byte

In this byte:

- Bits [6:5] indicate the reason why this I-sync packet was generated. The possible values are:

00, Periodic	Periodic I-sync.
01, Trace_on	Trace turned on normally.
10, Overflow	FIFO overflow.
11, Debug	Exit from Debug state.

  For more information see *Periodic and Nonperiodic I-sync packets* on page 4-164.
- Bit [3] is set to 1 if the processor is in Non-secure state at the address specified in the Address field.
- Bit [2], when the T bit is set to 1, indicates the processor instruction set state when the instruction at the address specified by the Address field is executed. The possible values are:

0	Thumb state.
1	ThumbEE state.

  When the T bit is set to 0, this bit is Reserved, SBZ.
- Bit [1]:
  — For PFTv1.0, this bit is Reserved, SBZ.
  — For PFTv1.1, in implementations that include the Virtualization extension, this bit is set to 1 if the processor is in Hyp mode.

Cycle count See *The I-sync cycle count field*.

Context ID The current Context ID. This is the Context ID of the instruction at the address specified by the Address field.

An I-sync packet does not indicate that the instruction at the address given in the packet was executed. You must look at the subsequent packets to determine whether that instruction was executed.

On return from Jazelle state, the Address field gives the address of the first instruction out of Jazelle state.

On return from a region where tracing is prohibited, the Address field gives the address of the first instruction that is not in the prohibited region.

For more information about instruction synchronization see *Instruction synchronization* on page 4-192.

The I-sync cycle count field

In the first I-sync packet after the PTM is enabled, the cycle count packet indicates the number of cycles since the Programming bit was cleared to 0.

The cycle count is always the incremental cycle count up to the last waypoint, not up to the time the I-sync packet was generated. After the first I-sync packet, the I-sync cycle count field indicates the number of cycles from the last cycle count to the time of the waypoint instruction indicated in the address field of the packet. Table 4-2 shows an example of the cycle count values output during tracing.

Table 4-2 Cycle count example with late trace turn-on

Cycle	Execution	Trace and count	Count
0	Waypoint A	Traced	-
10	Waypoint B	Traced, count=10 (10-0)	10 (10-0)
30	Waypoint C	Indicator that execution reached Waypoint C, count=20 (30-10). Trace turned off here.	20 (30-10)
50	Waypoint D	-	-
60	-	Trace turned on here. I-sync for Waypoint D.	20 (50-30)[a]
70	Waypoint E	Traced	20 (70-50)[b]
100	Waypoint F	Traced	30 (100-70)

a. The cycle count is the number of cycles between the last cycle count, cycle 30, and the last waypoint, Waypoint D. It does not indicate where tracing was turned on.

b. The cycle count is the number of cycles since the last cycle count. That is, the number of cycles between Waypoint D and Waypoint E.

From PFTv1.1, the cycle count value in an I-Sync packet which indicates a Trace Overflow or Debug Exit is UNKNOWN and must not be relied on.

Periodic and Nonperiodic I-sync packets

The reason code of a periodic I-sync packet is b00. No cycle count bytes are output in a periodic I-sync packet.

A nonperiodic I-sync packet is an I-sync packet with a reason code other than b00. If cycle-accurate mode is enabled, a nonperiodic I-sync packet always includes one or more cycle count bytes. The cycle count is compressed. See *Cycle count compression* on page 4-175. The final cycle count byte output cannot be zero.

When a nonperiodic I-sync packet is generated, its reason code is determined by applying the following rules, in order:

1. If Halting debug-mode is enabled and the I-sync packet indicates the first nonprohibited instruction since exit from Debug state, then the reason code is b11, exit from Debug state.
2. If the PTM has recovered from a FIFO overflow, the reason code is b10, FIFO overflow.
3. Otherwise, the reason code is b01, Trace turned on normally.

The priority of these rules means that the FIFO overflow reason code is lost if both of the following apply:

- Halting debug-mode is enabled and the overflow occurred immediately before entry to Debug state
- you trace the first instruction after exit from Debug state.

4.5.3 Atom packet

In the PFT architecture, the PTM generates E and N atoms only for waypoint instructions that can be determined statically:

- an E atom indicates that the waypoint caused a program flow change
- an N atom indicates that the waypoint did not cause a program flow change.

How the PTM assembles and outputs the atoms depends on whether you have enabled cycle-accurate tracing. See the following sections:

- *Atom packets when cycle-accurate tracing is not enabled* on page 4-165

- *Atom packets when cycle-accurate tracing is enabled.*

Atom packets when cycle-accurate tracing is not enabled

When you have not enabled cycle-accurate tracing, the PTM assembles E and N atoms into atom packets that are always one byte. In this case the atom packet is a packet header that also encodes between one and five atoms. Figure 4-3 shows the general format of this atom packet.

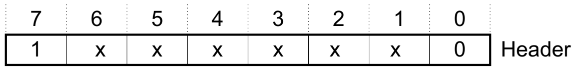


Figure 4-3 Atom packet, cycle-accurate tracing not enabled

Table 4-3 shows how this header is encoded for different numbers of atoms.

Table 4-3 Atom header encoding when cycle-accurate tracing is not enabled

Format	Encoding	Meaning[a]
Format 0	`100000x0`	Reserved
Format 1	`100001F0`	One atom, `F`
Format 2	`10001FF0`	Two atoms, `FF`
Format 3	`1001FFF0`	Three atoms, `FFF`
Format 4	`101FFFF0`	Four atoms, `FFFF`
Format 5	`11FFFFF0`	Five atoms, `FFFFF`

a. `F` is 0 for an E atom and 1 for an N atom. The least-significant `F` is the most recent waypoint.

Note

A non-exception branch address packet implies an E atom.

Atom packets when cycle-accurate tracing is enabled

When you have enabled cycle-accurate tracing, the PTM assembles every E or N atom with zero or more W atoms, and outputs a cycle-accurate atom packet, that is between one and five bytes long. Figure 4-4 shows the format of this packet.

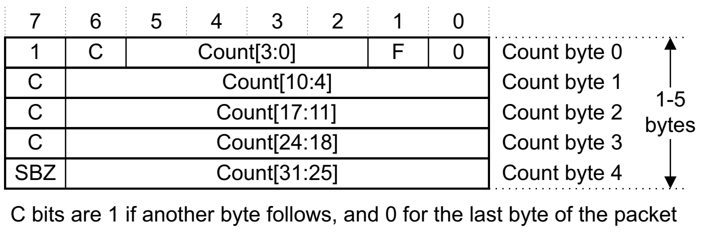

7 6 5 4 3 2 1 0
1 C Count[3:0] F 0 Count byte 0
C Count[10:4] Count byte 1
C Count[17:11] Count byte 2
C Count[24:18] Count byte 3
SBZ Count[31:25] Count byte 4
1-5 bytes
C bits are 1 if another byte follows, and 0 for the last byte of the packet

Figure 4-4 Cycle-accurate atom packet

The first byte of the packet, count byte 0, is also the packet header. It conforms to the general format of an atom packet header in Figure 4-3.

——— **Note** ———

Although the format of the atom packet header is the same regardless of whether cycle-accurate tracing is enabled, the meaning of bit [6] of the header is different. When cycle-accurate tracing is enabled, if bit [6] of the header is set to 1 then the packet has one or more additional bytes.

————————

The fields in the cycle-accurate atom packet are:

Count Cycle count. The number of cycles since the last cycle count output. The cycle count is compressed using leading zero compression. See *Cycle count compression* on page 4-175.

A W atom is generated for each clock cycle, and therefore the cycle count is identical to the number of W atoms issued since the last atom or branch address packet.

The possible range of cycle count values is the same as for a branch address packet. See *Branch address packet cycle count information in cycle-accurate mode* on page 4-169.

F The E or N atom for this atom packet:

F==0	E atom.
F==1	N atom.

4.5.4 Branch address packet

The PTM generates a branch address packet to indicate a change in the program flow. It outputs a branch address packet when any of the following occurs:

- The processor executes an indirect branch instruction that passes its condition code check. However, if use of the return stack is enabled, the PTM might generate an E atom for this instruction. See *Use of a return stack* on page 4-195.
- Branch broadcasting is enabled and the processor executes a direct branch instruction that passes its condition code check. See *Branch broadcasting* on page 4-185 for more information.
- An exception.
- The processor security state changes.

The branch address packet generated when an exception occurs is called an exception branch address packet, and is indicated by a nonzero Exception field. Branch address packets are sometimes called branch packets.

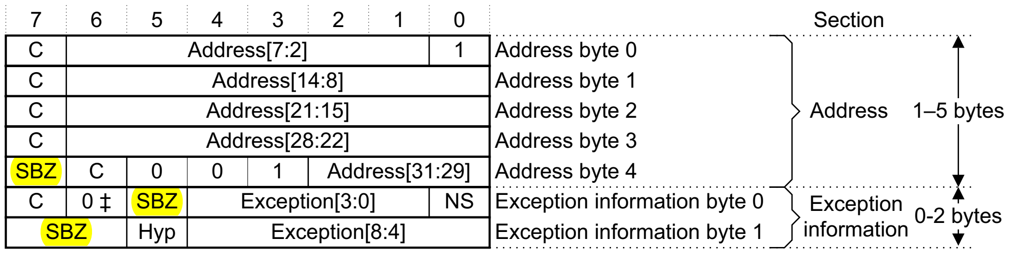
A non-exception branch address packet implies an E atom before the branch.

A branch address packet consists of 1-5 address bytes followed by 0-2 exception information bytes. The exact format of the packet depends on the instruction set state of the processor after the branch is taken:

- Figure 4-5 on page 4-167 shows the format for ARM state
- Figure 4-6 on page 4-167 shows the format for Thumb and ThumbEE states
- Figure 4-7 on page 4-167 shows the format for a branch into Jazelle state.

As Table 4-1 on page 4-161 shows, the first address byte of the packet is also recognized as the packet header.

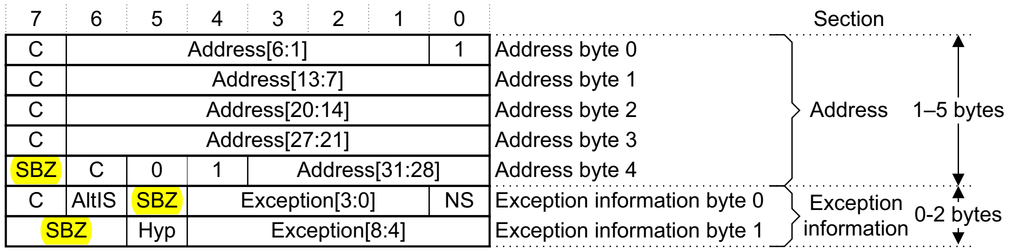
In cycle-accurate mode, the branch address packet is extended by 1-5 bytes of cycle count. See *Branch address packet cycle count information in cycle-accurate mode* on page 4-169.


7 6 5 4 3 2 1 0
Section
C | Address[7:2] | 1 | Address byte 0
C | Address[14:8] | Address byte 1
C | Address[21:15] | Address byte 2
C | Address[28:22] | Address byte 3
SBZ | C | 0 | 0 | 1 | Address[31:29] | Address byte 4
C | 0 ‡ | SBZ | Exception[3:0] | NS | Exception information byte 0
SBZ | Hyp | Exception[8:4] | Exception information byte 1
Address
1–5 bytes
Exception information
0-2 bytes

C bits are 1 if another byte follows in that section, and 0 for the last byte of a section

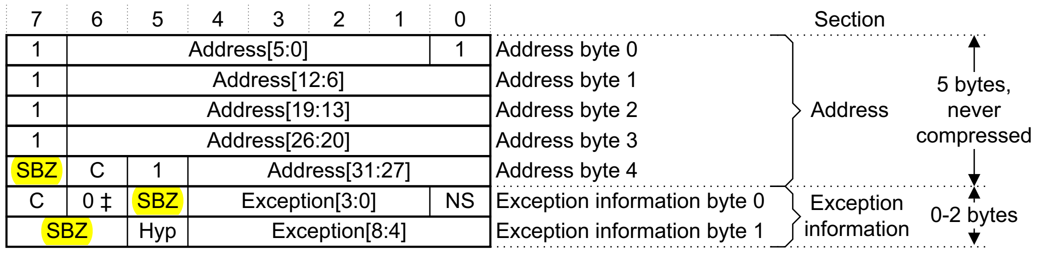
‡ AltIS bit: Always 0 for a branch address packet in ARM state

Figure 4-5 Full branch address packet with exception, ARM state


7 6 5 4 3 2 1 0
Section
C | Address[6:1] | 1 | Address byte 0
C | Address[13:7] | Address byte 1
C | Address[20:14] | Address byte 2
C | Address[27:21] | Address byte 3
SBZ | C | 0 | 1 | Address[31:28] | Address byte 4
C | AltIS | SBZ | Exception[3:0] | NS | Exception information byte 0
SBZ | Hyp | Exception[8:4] | Exception information byte 1
Address
1–5 bytes
Exception information
0-2 bytes

C bits are 1 if another byte follows in that section, and 0 for the last byte of a section

Figure 4-6 Full branch address packet with exception, Thumb state


7 6 5 4 3 2 1 0
Section
1 | Address[5:0] | 1 | Address byte 0
1 | Address[12:6] | Address byte 1
1 | Address[19:13] | Address byte 2
1 | Address[26:20] | Address byte 3
SBZ | C | 1 | Address[31:27] | Address byte 4
C | 0 ‡ | SBZ | Exception[3:0] | NS | Exception information byte 0
SBZ | Hyp | Exception[8:4] | Exception information byte 1
Address
5 bytes, never compressed
Exception information
0-2 bytes

C bits are 1 if another byte follows in that section, and 0 for the last byte of a section

‡ AltIS bit: Always 0 for a branch address packet in Jazelle state

Figure 4-7 Full branch address packet with exception, Jazelle state

The fields in the branch address packet are:

Address The target address, 1-5 bytes. The PTM compresses the address section of a branch address packet, by generating only the bytes required to output all the address bits that are changed, compared with the address in the last branch address packet or I-sync packet. However, the address section is not compressed when the instruction alignment changes, for example from ARM to Thumb. See *Address and cycle count compression in branch address packets* on page 4-169 for more information.

Exception information

0-2 bytes of additional information, that indicate:

- the exception that occurred, if any
- the NS value
- the AltIS value, if the branch address packet indicates a change to or from ThumbEE state.

The PTM generates Exception information byte 0 only if at least one of the following applies:

- The value of the AltIS bit has changed since the last branch address packet or I-sync packet. This applies if the processor changes instruction set state to or from ThumbEE state.

- The value of the NS bit has changed since the last branch address packet or I-sync packet. This applies if the processor changes security state.
- An exception caused the change in program flow.

The PTM generates Exception information byte 1 only if the exception causing the change in program flow has an exception number greater than 15 (greater than b1111).

From PFTv1.1, for implementations that support virtualization, bit [5] of information byte 1 is set to 1 when the processor operates in Hyp mode.

Exception[3:0]

If the branch address packet includes at least one exception information byte, Table 4-4 lists the meaning of the Exception[3:0] field.

Table 4-4 Values of Exception[3:0] for ARMv7-A and ARMv7-R processors

Exception[3:0]	Exception
b0000	None
b0001	Entered Debug state when Halting debug-mode is enabled
b0010	Secure Monitor Call (SMC)
b0011	PFTv1.0: Reserved
	From PFTv1.1: entry to Hyp mode
b0100	Asynchronous Data Abort
b0101	ThumbEE check failed
b011x	Reserved
b1000	Processor Reset
b1001	Undefined Instruction
b1010	Supervisor Call (SVC)
b1011	Prefetch Abort or software breakpoint
b1100	Synchronous Data Abort or software watchpoint
b1101	Generic exception
b1110	IRQ
b1111	FIQ

NS If the branch address packet includes at least one exception information byte, NS is set to 1 if the processor is in Non-secure state when the instruction at the address specified by the Address field is executed, and otherwise is set to 0.

AltIS If a branch address packet includes at least one exception information byte, the AltIS bit indicates the processor instruction set state when the instruction at the address specified by the Address field is executed. The possible values are:

0	Thumb state.
1	ThumbEE state.

In ARM and Jazelle state branch address packets this bit is always 0.

Hyp If a branch address packet includes two exception information bytes, the Hyp bit indicates if the processor is branching into Hyp mode. The possible values are:

0	Processor is not branching into Hyp mode.

1 Processor is branching into Hyp mode.

The Hyp field is only set to 1 from PFTv1.1 and when the Virtualization extensions are implemented.

The branch address packet does not include any exception information bytes if all of the following apply:

- no exception occurred
- the NS value is the same as the value in the most recent I-sync or branch address packet output
- the AltIS value is the same as the value in the most recent I-sync or branch address packet output
- from PFT v1.1, the Hyp value is the same as the value in the most recent I-sync or branch address packet output.

Note

- The PTM does not support tracing in Jazelle state, but it traces any branch into Jazelle state, using the Jazelle state format in Figure 4-7 on page 4-167.
- A non-exception branch address packet implies an E atom.

Branch address packet cycle count information in cycle-accurate mode

If you enable cycle-accurate tracing, by setting bit [12] of the Main Control Register to 1, each branch address packet is followed immediately with 1-5 bytes of cycle count information. Figure 4-8 shows the format of the cycle count bytes.

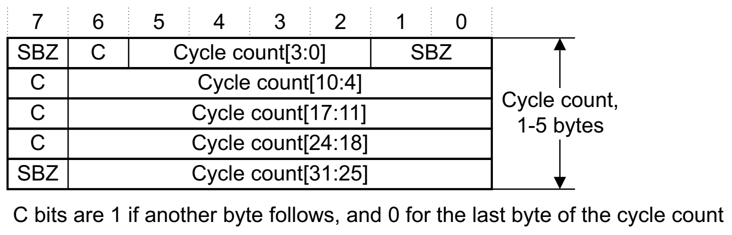


Figure 4-8 Branch address packet cycle count bytes, when cycle-accurate tracing is enabled

The PTM compresses the branch address packet cycle count information, by not generating any leading bytes that are zero. This compression is described as leading-zero compression. See *Address and cycle count compression in branch address packets* for more information.

A cycle count of all 1s, corresponding to the value `0xFFFFFFFF`, indicates a cycle count overflow. This means that the valid cycle count range is 0 to (2^{31}-2), `0x00000000` to `0xFFFFFFFE`.

Note

The format of the cycle count field permits a cycle count of 0, because a processor might execute two branches in the same cycle. In this case, if you have enabled cycle-accurate tracing, the branch address packet of the second branch includes a cycle count of 0.

Address and cycle count compression in branch address packets

When generating branch address packets, the PTM compresses the address and cycle count sections of the packet, generating the minimum possible number of bytes:

- for the cycle count, the PTM does not generate any leading bytes that would be zero
- for the address, the PTM does not generate any leading bytes that have not changed since the last branch address or I-sync packet.

However, the address section is not compressed when the instruction address alignment changes, for example from ARM to Thumb. This is because address byte 4 indicates the instruction address alignment. See Figure 4-5 on page 4-167, Figure 4-6 on page 4-167, and Figure 4-7 on page 4-167.

The following subsections give more information about the compression, with examples of the different lengths of the address and cycle count packet sections:

- *Address compression*
- *Cycle count compression* on page 4-175.

Address compression

When generating branch address packets, the PTM compresses the address, generating only the bytes required to output the address bits that have changed since the last branch address or I-sync packet. So, for example, if the most significant address bit that changes is bit [11], the PTM generates only two address bytes. Figure 4-9 shows this address compression example, for a Thumb state packet that does not require any exception information bytes.

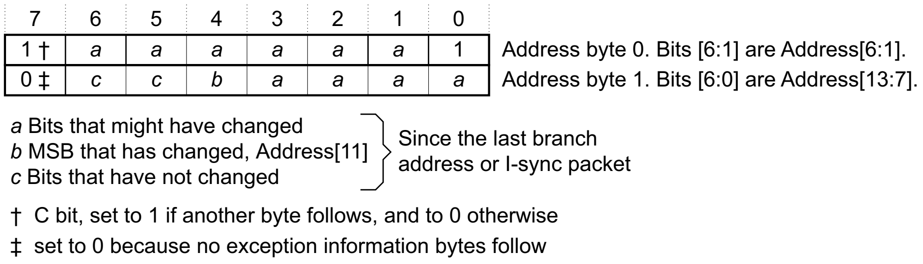

7 6 5 4 3 2 1 0
1 † a a a a a a 1
Address byte 0. Bits [6:1] are Address[6:1].
0 ‡ c c b a a a a
Address byte 1. Bits [6:0] are Address[13:7].
a Bits that might have changed
b MSB that has changed, Address[11]
c Bits that have not changed
Since the last branch address or I-sync packet
† C bit, set to 1 if another byte follows, and to 0 otherwise
‡ set to 0 because no exception information bytes follow

Figure 4-9 Address bytes when bit [11] is the most significant bit that changes, in Thumb state

Note

All of the generated address bits are always valid. So, in Figure 4-9, the bits marked c are output with the correct values even though their values have not changed since the last branch address or I-sync packet.

Table 4-5 shows how the number of address bytes generated depends on:

- The most significant bit (MSB) of the address that changes, compared with the address in the last branch address or I-sync packet.
- The instruction set state of the processor after the branch. This determines the format of the branch address packet. See Figure 4-5 on page 4-167 to Figure 4-7 on page 4-167.
- In some cases, whether the PTM must include an information byte in the packet.

Table 4-5 Number of address bytes generated for branch address packets

MSB of address that changes[a], for given state		Information byte required?	Number of address bytes generated
ARM state	Thumb state		
7 to 2	6 to 1	No	1
7 to 2	6 to 1	Yes	2
13 to 8	12 to 7	-	2
20 to 14	19 to 13	-	3
27 to 21	26 to 20	-	4
31 to 28	31 to 27	-	5

a. Most significant bit (MSB) of address that changes, compared with the address in the last branch address or I-sync packet.

Note

- A PTM traces entry into Jazelle state, but does not trace instruction execution in Jazelle state, and therefore there is no requirement for address compression in Jazelle state.
- The entry into Jazelle state is always traced with five address bytes, because of the change in instruction address alignment.

In all branch address packets, regardless of whether the PTM compresses the address:

- in all address bytes, bit [7] is 0 if this is the last address byte in the packet, and is 1 if another address byte follows
- in address bytes 1-4, if bit [7] is zero, bit [6] indicates whether at least one exception information byte follows the address byte.

Note

When the PTM must generate one or more exception information bytes it must generate at least two address bytes.

Table 4-6 shows how to interpret bit [7] of address byte 0, and Table 4-7 shows how to interpret bits [7:6] of address bytes 1-4. These tables apply regardless of whether the PTM has performed address compression.

Table 4-6 Interpretation of bit [7] in address byte 0

Address byte 0, bit [7]	Interpretation
1	The address continues in the next byte of the packet
0	This is the only address byte in the packet, and there are no exception information bytes[a]

a. If you have enabled cycle-accurate tracing at least one cycle count byte follows this byte.

Table 4-7 Interpretation of bits [7:6] in the address bytes 1-4

Address bytes 1-4		Interpretation
Bit [7]	Bit [6]	
1	-	The address continues in the next byte of the packet. Bit [6] of this byte is part of the address field.
0	1	This is the final address byte of this packet. There is at least one exception information byte after this byte.
0	0	This is the final address byte of this packet, and there are no exception information bytes.[a]

a. If you have enabled cycle-accurate tracing at least one cycle count byte follows this byte.

The following subsections show all the possible branch address packet formats with address compression:

- *Branches to Thumb or ThumbEE state, without exception information byte* on page 4-172
- *Branches to Thumb or ThumbEE state, with exception information byte* on page 4-172
- *Branches to ARM state, without exception information byte* on page 4-174
- *Branches to ARM state, with exception information byte* on page 4-174.

Branches to Thumb or ThumbEE state, without exception information byte

The following figures show all possible packet formats for branches to Thumb or ThumbEE state that do not require an exception information byte. If there is a change of instruction address alignment that does not require an exception information byte then the PTM generates the five-bye packet in Figure 4-14. This happens when there is a change from ARM or Jazelle state to Thumb state.

Note

The figure titles refer only to Thumb state, but each figure applies to branches to both Thumb state and ThumbEE state.

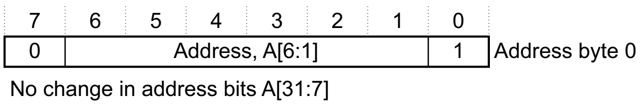

7 6 5 4 3 2 1 0
0 | Address, A[6:1] | 1 | Address byte 0
No change in address bits A[31:7]

Figure 4-10 Branch to Thumb state with change in A[6:1], no exception information byte

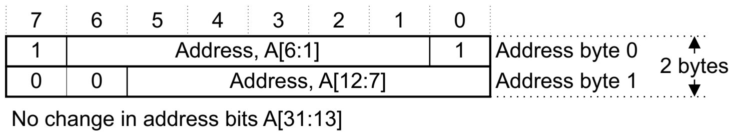

7 6 5 4 3 2 1 0
1 | Address, A[6:1] | 1 | Address byte 0
0 | 0 | Address, A[12:7] | Address byte 1
2 bytes
No change in address bits A[31:13]

Figure 4-11 Branch to Thumb state with change in A[12:7], no exception information byte

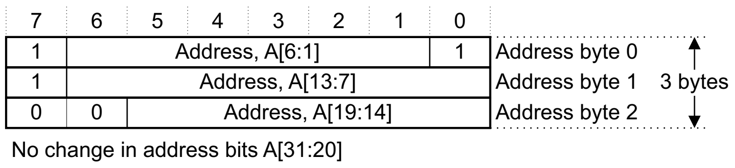

7 6 5 4 3 2 1 0
1 | Address, A[6:1] | 1 | Address byte 0
1 | Address, A[13:7] | Address byte 1
0 | 0 | Address, A[19:14] | Address byte 2
3 bytes
No change in address bits A[31:20]

Figure 4-12 Branch to Thumb state with change in A[19:13], no exception information byte

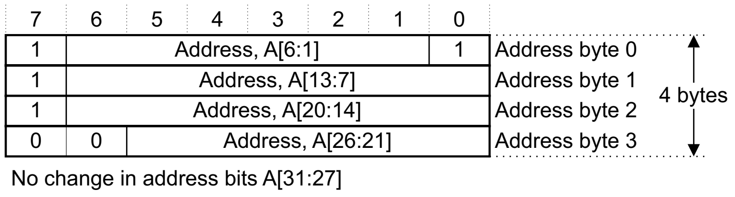

7 6 5 4 3 2 1 0
1 | Address, A[6:1] | 1 | Address byte 0
1 | Address, A[13:7] | Address byte 1
1 | Address, A[20:14] | Address byte 2
0 | 0 | Address, A[26:21] | Address byte 3
4 bytes
No change in address bits A[31:27]

Figure 4-13 Branch to Thumb state with change in A[26:20], no exception information byte

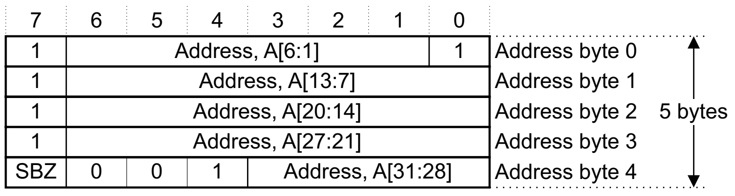

7 6 5 4 3 2 1 0
1 | Address, A[6:1] | 1 | Address byte 0
1 | Address, A[13:7] | Address byte 1
1 | Address, A[20:14] | Address byte 2
1 | Address, A[27:21] | Address byte 3
SBZ | 0 | 0 | 1 | Address, A[31:28] | Address byte 4
5 bytes

Figure 4-14 Branch to Thumb state with change in A[31:27], no exception information byte

Branches to Thumb or ThumbEE state, with exception information byte

The following figures show all possible packet formats for branches to Thumb or ThumbEE state that require an exception information byte. If a packet requires an exception information byte the PTM must generate at least two address bytes.

Note

- The figure titles refer only to Thumb state, but each figure applies to branches to both Thumb state and ThumbEE state.
- Most of the figures in this subsection show only a single exception information byte. However, the PFT architecture permits two exception information bytes. Figure 4-16 shows this format.

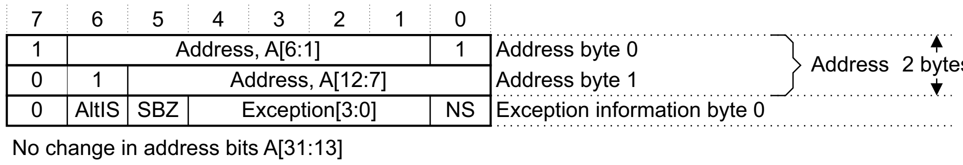

7 6 5 4 3 2 1 0
1 | Address, A[6:1] | 1 — Address byte 0
0 | 1 | Address, A[12:7] — Address byte 1
0 | AltIS | SBZ | Exception[3:0] | NS — Exception information byte 0
Address 2 bytes
No change in address bits A[31:13]

Figure 4-15 Branch to Thumb state with change in A[12:1], with exception information byte

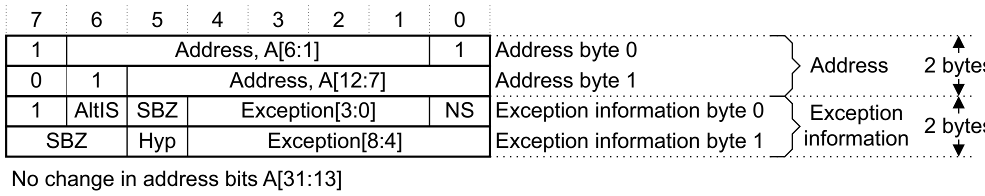

7 6 5 4 3 2 1 0
1 | Address, A[6:1] | 1 — Address byte 0
0 | 1 | Address, A[12:7] — Address byte 1
1 | AltIS | SBZ | Exception[3:0] | NS — Exception information byte 0
SBZ | Hyp | Exception[8:4] — Exception information byte 1
Address 2 bytes
Exception information 2 bytes
No change in address bits A[31:13]

Figure 4-16 Branch to Thumb state with change in A[12:1], with two exception information bytes

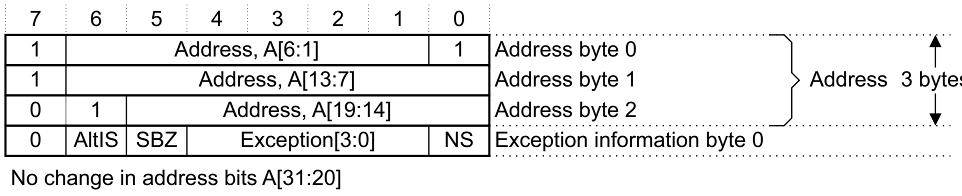

7 6 5 4 3 2 1 0
1 | Address, A[6:1] | 1 — Address byte 0
1 | Address, A[13:7] — Address byte 1
0 | 1 | Address, A[19:14] — Address byte 2
0 | AltIS | SBZ | Exception[3:0] | NS — Exception information byte 0
Address 3 bytes
No change in address bits A[31:20]

Figure 4-17 Branch to Thumb state with change in A[19:13], with exception information byte

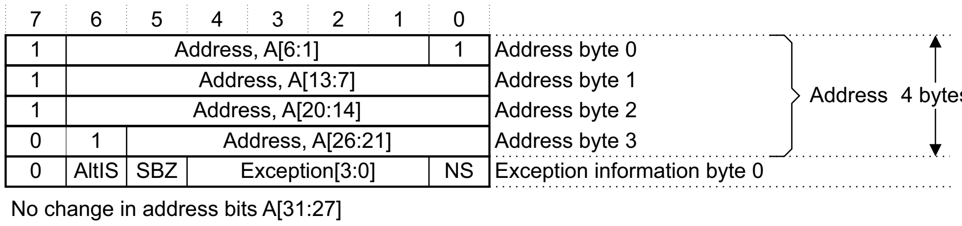

7 6 5 4 3 2 1 0
1 | Address, A[6:1] | 1 — Address byte 0
1 | Address, A[13:7] — Address byte 1
1 | Address, A[20:14] — Address byte 2
0 | 1 | Address, A[26:21] — Address byte 3
0 | AltIS | SBZ | Exception[3:0] | NS — Exception information byte 0
Address 4 bytes
No change in address bits A[31:27]

Figure 4-18 Branch to Thumb state with change in A[26:20], with exception information byte

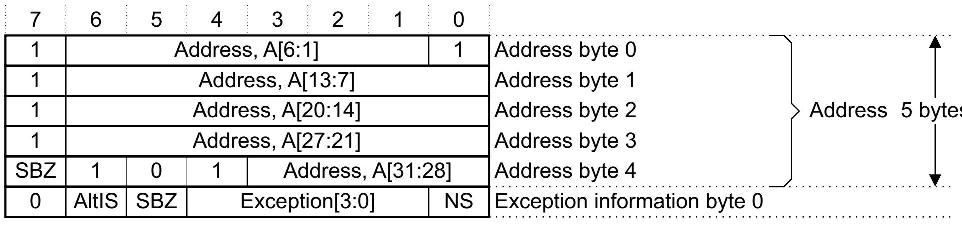

7 6 5 4 3 2 1 0
1 | Address, A[6:1] | 1 — Address byte 0
1 | Address, A[13:7] — Address byte 1
1 | Address, A[20:14] — Address byte 2
1 | Address, A[27:21] — Address byte 3
SBZ | 1 | 0 | 1 | Address, A[31:28] — Address byte 4
0 | AltIS | SBZ | Exception[3:0] | NS — Exception information byte 0
Address 5 bytes

Figure 4-19 Branch to Thumb state with change in A[31:27], with exception information byte

Branches to ARM state, without exception information byte

The following figures show all possible packet formats for branches to ARM state that do not require an exception information byte. If there is a change of instruction address alignment that does not require an exception information byte then the PTM generates the five-bye packet in Figure 4-24. This happens when there is a change from Thumb or Jazelle state to ARM state.

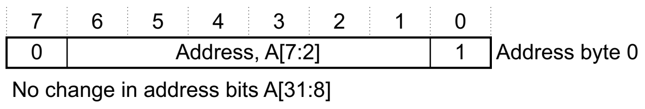

7 6 5 4 3 2 1 0
0 | Address, A[7:2] | 1 | Address byte 0
No change in address bits A[31:8]

Figure 4-20 Branch to ARM state with change in A[7:2], no exception information byte

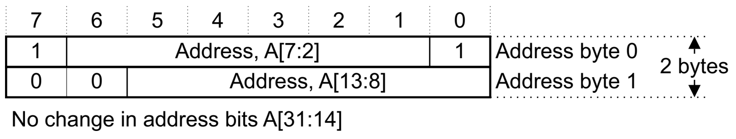

7 6 5 4 3 2 1 0
1 | Address, A[7:2] | 1 | Address byte 0
0 | 0 | Address, A[13:8] | Address byte 1
2 bytes
No change in address bits A[31:14]

Figure 4-21 Branch to ARM state with change in A[13:8], no exception information byte

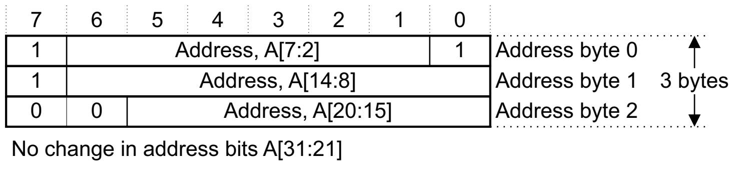

7 6 5 4 3 2 1 0
1 | Address, A[7:2] | 1 | Address byte 0
1 | Address, A[14:8] | Address byte 1
0 | 0 | Address, A[20:15] | Address byte 2
3 bytes
No change in address bits A[31:21]

Figure 4-22 Branch to ARM state with change in A[20:14], no exception information byte

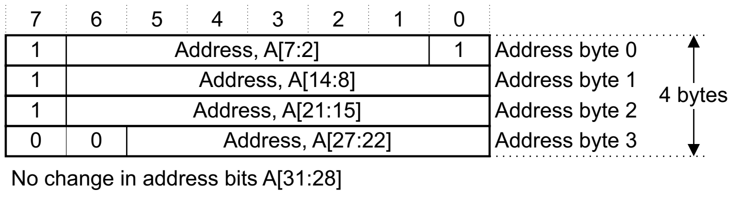

7 6 5 4 3 2 1 0
1 | Address, A[7:2] | 1 | Address byte 0
1 | Address, A[14:8] | Address byte 1
1 | Address, A[21:15] | Address byte 2
0 | 0 | Address, A[27:22] | Address byte 3
4 bytes
No change in address bits A[31:28]

Figure 4-23 Branch to ARM state with change in A[27:21], no exception information byte

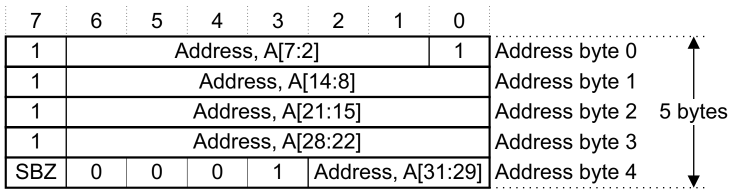

7 6 5 4 3 2 1 0
1 | Address, A[7:2] | 1 | Address byte 0
1 | Address, A[14:8] | Address byte 1
1 | Address, A[21:15] | Address byte 2
1 | Address, A[28:22] | Address byte 3
SBZ | 0 | 0 | 0 | 1 | Address, A[31:29] | Address byte 4
5 bytes

Figure 4-24 Branch to ARM state with change in A[31:28], no exception information byte

Branches to ARM state, with exception information byte

Figure 4-25 on page 4-175 to Figure 4-28 on page 4-175 show all possible packet formats for branches to ARM state that require an exception information byte. If a packet requires an exception information byte the PTM must generate at least two address bytes.

——— **Note** ———

The figures in this subsection show only a single exception information byte. However, the PFT architecture permits two exception information bytes, and this format is shown, for a branch to Thumb state, in Figure 4-16 on page 4-173.

———————————

7	6	5	4	3	2	1	0		
1	Address, A[7:2]						1	Address byte 0	Address 2 bytes
0	1	Address, A[13:8]						Address byte 1	
0	0 ‡	SBZ	Exception[3:0]				NS	Exception information byte 0	

‡ AltIS bit: Always 0 for a branch address packet in ARM state.
No change in address bits A[31:14]. There might be no change in A[13:8].

Figure 4-25 Branch to ARM state with change in A[13:2], with exception information byte

7	6	5	4	3	2	1	0		
1	Address, A[7:2]						1	Address byte 0	Address 3 bytes
1	Address, A[14:8]							Address byte 1	
0	1	Address, A[20:15]						Address byte 2	
0	0 ‡	SBZ	Exception[3:0]				NS	Exception information byte 0	

‡ AltIS bit: Always 0 for a branch address packet in ARM state.
No change in address bits A[31:21].

Figure 4-26 Branch to ARM state with change in A[20:14], with exception information byte

1	Address, A[7:2]						1	Address byte 0	Address 4 bytes
1	Address, A[14:8]							Address byte 1	
1	Address, A[21:15]							Address byte 2	
0	1	Address, A[27:22]						Address byte 3	
0	0 ‡	SBZ	Exception[3:0]				NS	Exception information byte 0	

‡ AltIS bit: Always 0 for a branch address packet in ARM state.
No change in address bits A[31:28]

Figure 4-27 Branch to ARM state with change in A[27:21], with exception information byte

7	6	5	4	3	2	1	0		
1	Address, A[7:2]						1	Address byte 0	Address 5 bytes
1	Address, A[14:8]							Address byte 1	
1	Address, A[21:15]							Address byte 2	
1	Address, A[28:22]							Address byte 3	
SBZ	1	0	0	1	Address, A[31:29]			Address byte 4	
0	0 ‡	SBZ	Exception[3:0]				NS	Exception information byte 0	

‡ AltIS bit: Always 0 for a branch address packet in ARM state.

Figure 4-28 Branch to ARM state with change in A[31:28], with exception information byte

Cycle count compression

The PTM generates the minimum number of cycle count bytes required to hold the cycle count value. For example, if the *most significant* (MS) nonzero bit of the cycle count is bit [13], the PTM generates only three cycle count bytes, as Figure 4-29 shows.

7	6	5	4	3	2	1	0	
SBZ*	1 †	b3	b2	b1	b0	SBZ		Cycle count byte 0
1 †	b10	b9	b8	b7	b6	b5	b4	Cycle count byte 1
0 †	0	0	0	0	1	b12	b11	Cycle count byte 2

b0-b12: Bits [12:0] of the cycle count
Bit [13] of the cycle count is the MSB, given in bit [2] of Cycle count byte 2
† C bits, set to 1 if another byte follows, and to 0 otherwise
* SBZ in address packets, 1 in I-sync packets

Figure 4-29 Cycle count bytes when bit [13] is the MS nonzero bit

Table 4-8 shows how, when cycle-accurate tracing is enabled, the number of cycle count bytes generated depends on the MS nonzero bit of the cycle count.

Table 4-8 Number cycle count bytes generated for branch address packets

MS nonzero bit of cycle count	Number of cycle count bytes generated
3 to 0	1
10 to 4	2
17 to 11	3
24 to 18	4
31 to 25	5

The following diagrams show each of these cases, for the cycle count bytes in an address packet:

- Figure 4-30 shows the single cycle count byte generated when the MS nonzero bit is in the range Count[3:0]
- Figure 4-31 shows the two cycle count bytes generated when the MS nonzero bit is in the range Count[10:4]
- Figure 4-32 shows the three cycle count bytes generated when the MS nonzero bit is in the range Count[17:11]
- Figure 4-33 on page 4-177 shows the four cycle count bytes generated when the MS nonzero bit is in the range Count[24:18]
- Figure 4-34 on page 4-177 shows the five cycle count bytes generated when the MS nonzero bit is in the range Count[31:25].

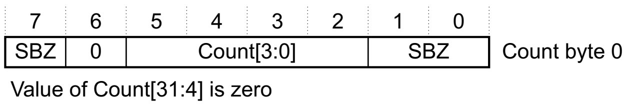


Figure 4-30 Cycle count byte when MS nonzero bit is in Count[3:0]

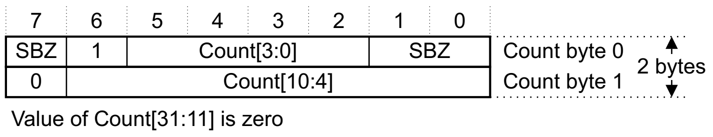


Figure 4-31 Cycle count bytes when MS nonzero bit is in Count[10:4]

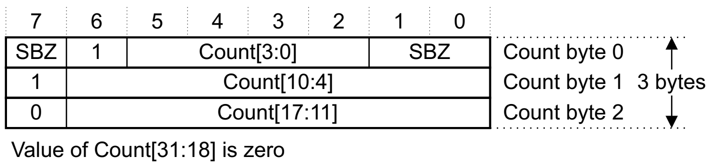


Figure 4-32 Cycle count bytes when MS nonzero bit is in Count[17:11]

7	6	5	4	3	2	1	0		
SBZ	1	Count[3:0]				SBZ		Count byte 0	4 bytes
1	Count[10:4]							Count byte 1	
1	Count[17:11]							Count byte 2	
0	Count[24:18]							Count byte 3	

Value of Count[31:25] is zero

Figure 4-33 Cycle count bytes when MS nonzero bit is in Count[24:18]

7	6	5	4	3	2	1	0		
SBZ	1	Count[3:0]				SBZ		Count byte 0	5 bytes
1	Count[10:4]							Count byte 1	
1	Count[17:11]							Count byte 2	
1	Count[24:18]							Count byte 3	
SBZ	Count[31:25]							Count byte 4	

Figure 4-34 Cycle count bytes when MS nonzero bit is in Count[31:25]

4.5.5 Waypoint update packet

An exception causes a branch in program execution that cannot be determined statically from the program image. When this occurs and at least one nonwaypoint instruction has executed since the last waypoint, then the most recent nonwaypoint instruction is upgraded to a waypoint instruction. The PTM generates a waypoint update packet to trace this upgraded instruction, and a debugger can reconstruct the full program flow from this packet.

The PTM also generates a waypoint update packet if a waypoint occurs that is more than 4096 bytes from the target address of the previous waypoint. See *Large blocks of instructions* on page 4-190.

A waypoint update packet outputs an address that indicates the point program execution reached before the exception. The packet indicates that the processor has executed all the instructions from the last waypoint, up to and including the address given in the packet. This address is related to the R14 value the processor stores before taking the exception. A waypoint update packet consists of:

- The 1-byte waypoint update header. This byte is always present.
- One to five bytes of waypoint address.
- If required, a single waypoint information byte. This byte is present only if there has been a change to or from ThumbEE state since the last branch or I-sync packet. Such a change requires the PTM to output the new value of the AltIS bit.

The exact format of the waypoint packet depends on the instruction set state of the processor when it executes the execution of the upgraded instruction:

- Figure 4-35 shows the format for ARM state
- Figure 4-36 on page 4-178 shows the format for Thumb and ThumbEE states.

7	6	5	4	3	2	1	0			
0	1	1	1	0	0	1	0	Header		
C	Address, A[7:2]						1	Address byte 0	Waypoint address	1–5 bytes
C	Address, A[14:8]							Address byte 1		
C	Address, A[21:15]							Address byte 2		
C	Address, A[28:22]							Address byte 3		
SBZ	C	0	0	1	Address, A[31:29]			Address byte 4		
SBZ	‡	SBZ						Waypoint information byte, if required		

‡ AltIS bit: Always 0 for a waypoint update packet in ARM state.
C bits are 1 if another byte follows, and 0 for the last byte of the packet.

Figure 4-35 Waypoint update packet, ARM state

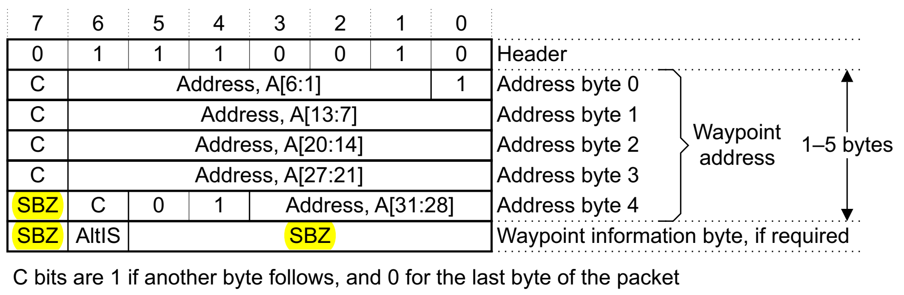


Figure 4-36 Waypoint update packet, Thumb state

The PTM compresses the waypoint address in the same way that it compresses the address in a branch address packet. See *Address and cycle count compression in branch address packets* on page 4-169.

The waypoint address indicates the last instruction that was executed. It does not indicate whether that instruction passed its condition code test. A waypoint update packet indicates that the program executed from the last waypoint instruction up to and including the address output in the packet. It does not indicate the execution of any instructions beyond the output address.

The address output in a waypoint update packet must be both:

- greater than or equal to the address of the last successfully executed instruction
- less than the address of the next instruction, in program execution order.

Applying this requirement, Table 4-9 shows the permitted waypoint address outputs.

Table 4-9 Permitted waypoint address outputs in waypoint update packets

Instruction type[a]	Instruction address	Waypoint address[b]	Permitted, with explanation of forbidden cases
ARM	`0x1000`	<`0x1000`	No. This is less than the address of the instruction.
	`0x1000`	`0x1000`	Yes.
	`0x1000`	`0x1001`	No. Cannot be output, because the ARM state waypoint update packet does not output address bits A[1:0].
	`0x1000`	`0x1002`	
	`0x1000`	`0x1003`	
	`0x1000`	≥`0x1004`	No. This is not less than the next instruction address, `0x1004`.
Thumb, 16-bit instruction	`0x1000`	<`0x1000`	No. This is less than the address of the instruction.
	`0x1000`	`0x1000`	Yes.
	`0x1000`	`0x1001`	No. Cannot be output, because the Thumb state waypoint update packet does not output address bit A[0].
	`0x1000`	≥`0x1002`	No. This is not less than the next instruction address, `0x1002`.

Table 4-9 Permitted waypoint address outputs in waypoint update packets (continued)

Instruction type[a]	Instruction address	Waypoint address[b]	Permitted, with explanation of forbidden cases
Thumb, 32-bit instruction	`0x1000`	<`0x1000`	No. This is less than the address of the instruction.
	`0x1000`	`0x1000`	Yes.
	`0x1000`	`0x1001`	No. Cannot be output, because the Thumb state waypoint update packet does not output address bit A[0].
	`0x1000`	`0x1002`	Yes.
	`0x1000`	`0x1003`	No. Cannot be output, because the Thumb state waypoint update packet does not output address bit A[0].
	`0x1000`	≥`0x1004`	No. This is not less than the next instruction address, `0x1004`.

a. Instruction set state, and the instruction length for Thumb instructions. The Thumb entries also apply to the ThumbEE instruction set state.

b. The address output in the waypoint update packet.

When a PTM upgrades a 32-bit Thumb instruction, it is IMPLEMENTATION SPECIFIC whether the address is output in the waypoint update packet.

Note

On an imprecise abort, the waypoint address might not indicate the exact address where execution stopped. This is because the ARM architecture does not define this point precisely.

For more information about how a PTM handles each exception, see Chapter 5 *Tracing Exceptions*.

4.5.6 Trigger packet

The PTM generates a trigger packet when the trigger occurs. The trigger packet is always one byte, and consists of the trigger header, as Figure 4-37 shows.

7	6	5	4	3	2	1	0	
0	0	0	0	1	1	0	0	Header

Figure 4-37 Trigger packet

When a trigger occurs between two waypoints or at the same time as a second waypoint, the trigger must be inserted no earlier than the first waypoint, and no later than an IMPLEMENTATION DEFINED number of waypoints after the first waypoint. This IMPLEMENTATION DEFINED number of waypoints is generally limited to the number of waypoints the processor can process simultaneously.

Note

If you have enabled cycle-accurate tracing, the PTM generates an atom packet for each atom. See *Atom packet* on page 4-164. Therefore, when a trigger occurs between two waypoints the PTM must output the trigger packet between the two atom packets.

4.5.7 Context ID packet

When the Context ID changes, the PTM outputs a Context ID packet to give the new value. The packet consists of a one-byte Context ID packet header, followed by 1, 2 or 4 bytes that give the new Context ID value, as Figure 4-38 on page 4-180 shows.

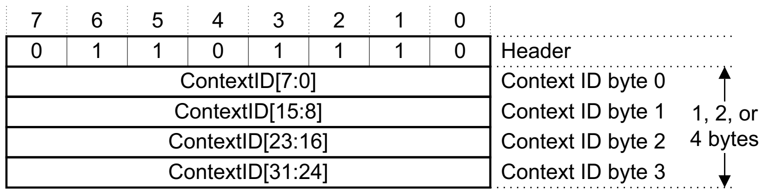


Figure 4-38 Context ID packet

The Context ID value is not compressed. The PTM always outputs the number of bytes specified in the ContextIDsize field of the Main Control Register. See *Main Control Register, ETMCR* on page 3-75.

The PTM outputs the Context ID packet immediately after tracing the first waypoint after the processor Context ID changes. When interpreting the trace output, the new Context ID value can be applied only to instructions after that waypoint. Table 4-10 shows an example of tracing a Context ID change.

Table 4-10 Tracing a change of Context ID

Execution	Trace	Comments
`B` to `0x1000`	E atom	-
`NOP`	-	-
`MCR` to Context ID Register	-	-
`ISB`	E atom	Atom packet generated to output assembled atoms. Trace implies code up to the `ISB` instruction was executed with the old Context ID.
	Context ID packet	Context ID changed.
`NOP`	-	-
`NOP`	-	-
`LDR PC`, with value of `0x2000`	Branch address packet, target address `0x2000`	Trace implies code up to and including the `LDR PC` instruction is executed with the new Context ID.

The PTM must generate an atom packet to output any assembled atoms before it generates the Context ID packet.

The Context ID might change on a security level change. In this case, the PTM outputs a Context ID packet immediately after the branch address packet that traces the security level change.

Writes to the Context ID register in the processor are not required to take effect until the next ISB, exception, or exception return. Therefore, a change in the Context ID might not be reported until it actually changes in the processor.

Note

- A PTM might only output a Context ID packet when it detects an actual change in the Context ID. For example, if an `MCR` writes to the Context ID Register but does not change the Context ID then the PTM might not output a Context ID packet.
- If there are multiple Context ID changes between two waypoints, the PTM only considers the last change. If this last change restores the Context ID to the value it had at the first waypoint then the PTM might not output any Context ID packet. However this situation is unlikely, because software normally issues an `ISB` instruction after changing the Context ID, and the `ISB` is a waypoint instruction.

4.5.8 VMID packets

From PFTv1.1, in implementations that support the Virtualization Extensions, the VMID packet indicates the Virtual Machine ID of the trace source. This packet consists of a single header byte with a single payload byte.

Figure 4-39 shows the format of a VMID packet.

7	6	5	4	3	2	1	0	
0	0	1	1	1	1	0	0	Header
VMID[7:0]								Payload

Figure 4-39 VMID packet

The VMID packet is output each time the Virtual Machine ID changes. This packet must be output before the first instruction that was executed using the changed VMID is traced. At the latest this must occur immediately after the next ISB or exception entry or return after the instruction the updated the VMID.

If multiple VMID changes occur between two waypoints, only the last change is guaranteed to be considered for tracing. If the VMID has not changed since the previously traced value, the new VMID might not be traced.

When an I-Sync packet occurs, a VMID packet must be output before the next Atom, Branch, or Waypoint Update packet. This ensures that trace analysis tools have the complete context before decompressing any instructions.

The VMID is not traced on entry to a prohibited region.

When the processor is reset, the Virtual Machine ID is reset to zero. A VMID packet might not be traced in this scenario. However, this can be detected because an exception is traced indicating the reset exception.

4.5.9 Timestamp packet

Timestamping enables correlation between multiple trace streams, and is provided by timestamp packets. See *Timestamping* on page 4-198. The timestamp packet consists of:

- The 1-byte timestamp update header. This byte is always present.
- One to nine bytes of timestamp value. This section is always present.
- If you have enabled cycle-accurate tracing, one to five bytes of cycle count.

In PFTv1.0 timestamp values can have a maximum size of 48 bits. From PFTv1.1, timestamp values can have a maximum size of either 48 or 64 bits, as specified by bit [29] of the ETMCCER. See *Configuration Code Extension Register, ETMCCER* on page 3-103.

Figure 4-40 shows the format of the 48-bit timestamp packet.

7	6	5	4	3	2	1	0	
0	1	0	0	0	R	1	0	Header
C	Timestamp[6:0]							Value byte 0
C	Timestamp[13:7]							Value byte 1
C	Timestamp[20:14]							Value byte 2
C	Timestamp[27:21]							Value byte 3
C	Timestamp[34:28]							Value byte 4
C	Timestamp[41:35]							Value byte 5
0	SBZ	Timestamp[47:42]						Value byte 6

Figure 4-40 48-bit timestamp packet

Figure 4-41 on page 4-182 shows the format of the 48-bit timestamp packet in cycle-accurate mode.

7	6	5	4	3	2	1	0	
0	1	0	0	0	R	1	0	Header
C	Timestamp[6:0]							Value byte 0
C	Timestamp[13:7]							Value byte 1
C	Timestamp[20:14]							Value byte 2
C	Timestamp[27:21]							Value byte 3
C	Timestamp[34:28]							Value byte 4
C	Timestamp[41:35]							Value byte 5
0	SBZ	Timestamp[47:42]						Value byte 6
SBZ	C	Count[3:0]				SBZ		Cycle count byte 0
C	Count[10:4]							Cycle count byte 1
C	Count[17:11]							Cycle count byte 2
C	Count[24:18]							Cycle count byte 3
SBZ	Count[31:25]							Cycle count byte 4

Figure 4-41 48-bit timestamp packet in cycle-accurate mode

Figure 4-42 shows the format of the 64-bit timestamp packet.

7	6	5	4	3	2	1	0	
0	1	0	0	0	R	1	0	Header
C	Timestamp[6:0]							Value byte 0
C	Timestamp[13:7]							Value byte 1
C	Timestamp[20:14]							Value byte 2
C	Timestamp[27:21]							Value byte 3
C	Timestamp[34:28]							Value byte 4
C	Timestamp[41:35]							Value byte 5
C	Timestamp[48:42]							Value byte 6
C	Timestamp[55:49]							Value byte 7
Timestamp[63:56]								Value byte 8

Figure 4-42 64-bit timestamp packet

Figure 4-43 shows the format of the 64-bit timestamp packet in cycle-accurate mode.

7	6	5	4	3	2	1	0	
0	1	0	0	0	R	1	0	Header
C	Timestamp[6:0]							Value byte 0
C	Timestamp[13:7]							Value byte 1
C	Timestamp[20:14]							Value byte 2
C	Timestamp[27:21]							Value byte 3
C	Timestamp[34:28]							Value byte 4
C	Timestamp[41:35]							Value byte 5
C	Timestamp[48:42]							Value byte 6
C	Timestamp[55:49]							Value byte 7
Timestamp[63:56]								Value byte 8
SBZ	C	Count[3:0]				SBZ		Cycle count byte 0
C	Count[10:4]							Cycle count byte 1
C	Count[17:11]							Cycle count byte 2
C	Count[24:18]							Cycle count byte 3
SBZ	Count[31:25]							Cycle count byte 4

Figure 4-43 64-bit timestamp packet in cycle-accurate mode

The fields in the timestamp packet are:

R The R bit in the timestamp packet header is set to 1 if the clock frequency of the processor has changed since the last timestamp packet, and is 0 otherwise. The PFT protocol does not give a precise indication of when the clock speed changes.

Timestamp	The timestamp header is always followed by at least one byte of timestamp. The timestamp value is compressed, so that the PTM generates only enough bytes of timestamp to output the most significant bit that changes. This is a similar compression mechanism to that used for address values, where the value of the bits that are not output have not changed since the last time they were output. A value of zero indicates that the timestamp is unknown. This might also indicate that the implementation does not fully support timestamping.
C	The C bit indicates if another byte of timestamp information follows this byte. If the C bit is 1 then there is another byte of timestamp information in the packet. If the C bit is 0, then this is the last byte of information in the timestamp packet.

If cycle-accurate tracing is enabled then the timestamp section of the packet is followed by at least one byte of cycle count. In PFTv1.0, the cycle count section gives the number of cycles since the last cycle count. In PFTv1.1, the cycle count is always zero, indicating zero cycles have passed between the last cycle count and the timestamp.

The cycle count value is compressed using leading zero compression. See *Cycle count compression* on page 4-175.

Encoding of the timestamp value

In PFTv1.0 the timestamp value is encoded as a Gray-coded number. From PFTv1.1, the timestamp value can be encoded as either a Gray-coded number or a natural binary number. Bit [28] of the ETMCCER specifies the encoding of the timestamp value in the timestamp packet. See *Configuration Code Extension Register, ETMCCER* on page 3-103

If the PTM outputs the timestamp values as a Gray-coded number, the number is calculated from the natural binary number using the following equation, where Gray[n] is the nth bit of the resultant Gray code, and binary[n] is the nth bit of the binary number:

Gray[n] = binary[n] XOR binary[n+1]

The debugger generates the binary number from the traced Gray code using the following equation:

binary[n] = XOR(Gray[N:n]).

In this equation, N+1 is the size in bits of the timestamp.

4.5.10 Exception return packet

A PTM for an ARMv7 processor generates an exception return packet to indicate that the processor is returning from an exception handler. ARMv7-M code performs an exception return by moving a special value to the PC. A debugger cannot determine this return from the program image. However, any instruction that modifies the PC is a waypoint instruction, and therefore an exception return packet in the trace stream indicates that the most recent waypoint instruction was the exception return instruction.

Table 4-11 shows the instructions that generate an exception return.

Table 4-11 Exception return instructions

Description	Example Mnemonic	ARM Profile
Load multiple with the PC and CPSR	`LDM` (exception return)	ARMv-7A, ARMv7-R, ARMv6 and earlier
Return from Exception	`RFE`	
Data processing instruction that modifies the PC and has the S bit set	`MOVS PC, LR PC, SUBS PC, LR`	
Exception return[a]	`ERET`	

Table 4-11 Exception return instructions (continued)

Description	Example Mnemonic	ARM Profile
POP or LDM that loads into the PC	`LDM, POP`	ARMv7-M[b]
Load to the PC	`LDR PC`	
Branch and exchange with any register	`BX Rn`	

a. Only if the Virtualization Extensions are implemented. See *Virtualization* on page 2-29.

b. In ARMv7-M, these instructions are only considered to be exception return instructions if they transfer one of the special values into the PC.

The exception return packet is always one byte, and consists of the exception return header. See Figure 4-44.

7	6	5	4	3	2	1	0	
0	1	1	1	0	1	1	0	Header

Figure 4-44 Exception return packet

This packet is only generated for ARMv7-A, ARMv7-R, and ARMv6 processors when PFTv1.1 is implemented. It is not generated for ARMv7-A, ARMv7-R, and ARMv6 processors if PFTv1.0 is implemented.

4.5.11 Ignore packet

The ignore packet has no effect. A PTM might insert an ignore packet in unused bytes of the trace port if it must output trace and there is insufficient trace to fill the entire port. The ignore packet is always one byte, and consists of the ignore header. See Figure 4-45.

7	6	5	4	3	2	1	0	
0	1	1	0	0	1	1	0	Header

Figure 4-45 Ignore packet

An implementation that uses an *AMBA Trace Bus* (ATB) to output trace data might not generate any ignore packets.

4.6 Branch broadcasting

You can enable branch broadcasting by setting bit [8] of the Main Control Register to 1. See *Main Control Register, ETMCR* on page 3-75. When this bit is set to 1, the PTM outputs the destination address for all branch instructions, including direct branch instructions. This is useful for tracing parts of a program where you do not have a code image, for example for tracing an area of self-modifying code.

When you enable branch broadcast mode, the PTM:

- traces any branch instruction that passes its condition code test by outputting a branch address packet
- traces any branch instruction that fails its condition code test by generating an N atom
- never generates any E atoms in atom packets.

If you enable branch broadcasting you must not enable use of a return stack, otherwise behavior is UNPREDICTABLE. See *Use of a return stack* on page 4-195 for information about this option.

Note

To use the return stack, the decompressor requires a code image, to identify whether the current waypoint is a branch, and if so, what type of branch it is. Therefore use of the return stack is not appropriate in situations where branch broadcasting is useful.

4.7 Prohibited regions

Tracing of regions of code can be prohibited. Entry to a prohibited region can occur for the following reasons:

- changing from Secure User mode to a Secure privileged mode while **SPNIDEN** is LOW
- changing from Non-secure state to Secure state while **SPNIDEN** is LOW
- deasserting **SPNIDEN**, from HIGH to LOW, when in a Secure privileged mode
- deasserting **NIDEN** and **DBGEN** from HIGH to LOW.

When the processor is executing code from a prohibited region, the PTM must not output any information about program execution. *Behavior of the PTM when the processor is in a prohibited region* on page 4-187 describes how the PTM behaves when the processor is in a prohibited region.

Normally, if tracing is enabled when the processor enters a prohibited region:

- If necessary, the PTM generates an atom packet to output any atoms it has assembled.
- The PTM generates a branch address packet to address 0, indicating the secure security state and ARM instruction set. The Hyp mode bit is always zero.

  From PFTv1.1, the instruction set indicated is IMPLEMENTATION DEFINED but fixed. Tools must ignore the instruction set traced and must not rely on it.
- If an exception caused the entry to the prohibited region, the branch address packet identifies the exception type. An exception is the most usual reason for an entry to a prohibited region.

  If the branch requires a waypoint update, the PTM generates this.
- If the Context ID changes the PTM does not trace this change.
- If the VMID changes the PTM does not trace this change.
- As soon as the PTM has output the branch address packet, and the waypoint update packet if it is required, tracing stops.

In the unusual case where the deassertion of one or more of **NIDEN**, **DBGEN** or **SPNIDEN** causes entry to the prohibited region, tracing stops and if necessary the PTM generates an atom packet to output any atoms it has assembled. The PTM might not generate a branch address packet or any waypoint update packet to indicate entry to the prohibited region.

Note

When a change of one or more of **NIDEN**, **DBGEN** or **SPNIDEN** causes entry to or exit from a prohibited region, the exact time of that entry or exit is IMPLEMENTATION SPECIFIC, and might not be until a waypoint occurs.

When the processor leaves the prohibited region, if tracing is enabled tracing restarts. The PTM treats the first instruction executed as the target of a waypoint, and generates an I-sync packet that outputs this address. Typically, this address is the target of a waypoint, because processing normally leaves a prohibited region by executing an indirect branch instruction.

4.7.1 Behavior of the PTM when the processor is in a prohibited region

When the processor is executing code from a prohibited region:

- No instructions in the prohibited region are traced.
- The security state of the processor is considered to be Secure. The Non-secure resource is LOW.
- If cycle-accurate mode is enabled, the cycle counter continues to count. When tracing restarts, cycles spent in the prohibited region are included in the cycle count.
- Instruction address comparators do not match on any instruction in the prohibited region.
- Context ID comparators are disabled during execution from the prohibited region.
- Instrumentation instructions executed when in a prohibited region have no effect. Entry to a prohibited region has no effect on the current state of any of the Instrumentation resources.
- Other resources, such as the counters, sequencers, external outputs and trigger, behave as normal. These can be disabled using the trace prohibited resource, b110 1110. See *The PTM event resources* on page 3-49.

4.7.2 Non-invasive debug disabled

Some systems allow non-invasive debug to be disabled. Sometimes a signal called *Non Invasive Debug ENable* (**NIDEN**), is used to disable or enable PTM functionality. Systems do not have to support the Security Extensions to implement this functionality.

When non-invasive debug is disabled, the PTM behaves as if the processor has entered a prohibited region.

For more information, see *Behavior of the PTM when the processor is in a prohibited region*. The following additional restrictions apply:

- Whether the branch packet is output is IMPLEMENTATION SPECIFIC.
- All trace in the PTM FIFO must be output.
- Trigger generation is disabled.
- The trace prohibited resource, if supported, is HIGH.
- The security state of the processor is considered Secure, so the Non-secure resource, if supported, is LOW.
- External inputs and extended external inputs must be ignored.
- Other resources such as counters, the sequencer, and external outputs, stop operating and are held in their current state. Some PTM implementations might drive the external outputs LOW.
- It is IMPLEMENTATION DEFINED if the cycle counter continues to count.

As defined in the *CoreSight Architecture Specification*, the effect of the timing of disabling non-invasive debug is imprecise. Therefore, tracing might continue after non-invasive debug is disabled, and might take time to re-enable when non-invasive debug is re-enabled.

When non-invasive debug is disabled, the ETMAUTHSTATUS register represents this. For more information, see *Authentication Status Register, ETMAUTHSTATUS* on page 3-119.

When non-invasive debug is disabled, the PTM programmers' model behaves normally.ARMv7 processors must implement the **NIDEN** functionality, and PTMs that are connected to ARMv7 processors must implement this functionality.

Normally, **NIDEN** is used in conjunction with a signal that enables invasive debug, **DBGEN**. Non-invasive debug is disabled only if both **NIDEN** and **DBGEN** are LOW. In a PTM, typically these signals are ORed together and the result is used to determine whether non-invasive debug is enabled.

4.8 Trace FIFO overflow

A single cycle on the processor can generate more trace than the PTM can output in this time. Therefore, the PTM includes a FIFO to buffer trace data. This FIFO can overflow if a large volume of trace is generated. The PFT protocol handles FIFO overflow as follows:

- if the trace FIFO is full, all tracing is stopped
- tracing restarts only when the FIFO has emptied
- the PTM indicates the loss of trace data by outputting an I-sync packet with reason code b10.

While tracing is stopped because of FIFO overflow, all PTM resources continue to operate. This means that any trigger condition, such as a trigger generated by the address comparators, can be detected and output after the FIFO has emptied and tracing restarted.

Any source that can request a periodic synchronization of the PTM can force the PTM to behave as if the FIFO had overflowed. See *Forced overflow* on page 4-192.

4.9 Wait for Interrupt and Wait for Event

The `WFI` and `WFE` instructions are not waypoint instructions, and entering a Wait for Interrupt (WFI) or Wait for Event (WFE) state is not traced as a waypoint.

However, when the processor commits a `WFI` or `WFE` instruction for execution:

- The PTM outputs any trace it is generating. If necessary, it generates an atom packet to output any atoms it has assembled.
- If the processor supports one or both of clock stopping and complete power removal, the PTM must drain its FIFO and signal to the processor or power control unit when the FIFO is empty. This means that the processor or power control unit can maintain the clock, or delay power removal, until the FIFO is empty, ensuring no loss of trace data.

Entering a WFI or WFE state does not stop tracing. However, normally these states stop the processor, meaning no more trace is generated.

If tracing is off when the processor enters a WFI or WFE state, trace cannot be enabled until the processor has left the WFI or WFE state.

If PTM power is removed, execution from the previous waypoint to the `WFI` or `WFE` instruction is not traced.

4.10 Large blocks of instructions

When a trace decompressor processes a branch address packet or atom packet, it must decode instructions from the last waypoint target until it encounters another waypoint. In exceptional situations, such as if the program has branched into an uninitialized memory region or the trace data stream is corrupted, the process of decoding to the next instruction could take a very long time. To prevent this, the PFT architecture implements a mechanism that enables a decompressor to bound the amount of instruction data it processes after identifying a waypoint.

If a PTM reaches a waypoint that is more than 4096 bytes of instruction data from the previous waypoint target then it outputs a waypoint update packet. The address in this packet is (new waypoint address-*n*), where *n* is 2 or 4. After outputting this waypoint update packet the PTM generates the atom or branch address packet for the new waypoint.

If there has not been a waypoint since the most recent I-sync packet, the waypoint update packet is output if the new waypoint is more than 4096 byes from the address in the I-sync packet.

If an exception means that the PTM must output a waypoint update packet before it processes the new waypoint, then no additional waypoint update packet is generated.

For example, if a program branches into a region of uninitialized memory where all of the data is zero, the processor interprets this data as a series of `NOP` instructions, and might execute a very large number of these instructions before it encounters the next waypoint. If this happens, when the processor encounters the next waypoint the PTM outputs a waypoint update packet, giving the address of the instruction before the waypoint. The trace decompressor recognizes that this address is a long way in the future, and might determine that the address is in uninitialized memory, and therefore that the program has gone wrong. Alternatively, the decompressor can process the code between the two waypoints. Knowing that this is a large block of code it can break the processing down into smaller blocks.

In addition, if the trace decompressor does not receive a waypoint update packet immediately before an atom packet or branch address packet, it knows it does not have to process more that 4096 consecutive bytes of instruction data beyond the previous waypoint.

If program execution rolls over the top of memory, where the instruction at `0xFFFFFFFC` or `0xFFFFFFFE` is followed by the instruction at `0x00000000` or `0x00000002`, this causes UNPREDICTABLE processor operation. In this situation, it is IMPLEMENTATION SPECIFIC whether the waypoint update packet is output if more than 4096 consecutive instruction bytes have been executed.

4.11 Synchronization

Trace systems often store trace data in a circular buffer, meaning that the initial output from a PTM might be overwritten before the trace is analyzed. In addition, a trace decompression tool might start decompressing at an arbitrary point in the output trace stream. Therefore, trace decompression tools must determine:

- byte boundary alignment
- packet boundary alignment
- the complete instruction address, current instruction set and security state, and Context ID
- the complete timestamp.

This is done using the following synchronization packets:

A-sync	For alignment synchronization.
I-sync	For instruction synchronization.
T-sync	For timestamp synchronization. T-sync packets are a form of the timestamp packet. See *Timestamp synchronization* on page 4-193.

The following sections describe the different PFT synchronization operations:

- *Periodic synchronization*
- *Alignment synchronization* on page 4-192
- *Instruction synchronization* on page 4-192
- *Timestamp synchronization* on page 4-193.

4.11.1 Periodic synchronization

When the PTM receives a periodic synchronization request it generates all three synchronization packets. A periodic synchronization request is generated when any of the following occurs:

- The Programming bit is cleared in the Main Control Register. See *Main Control Register, ETMCR* on page 3-75.
- The OS Lock is cleared, by writing to the ETMOSLAR. See *OS Lock Access Register, ETMOSLAR* on page 3-110.
- The Periodic synchronization counter in the PTM wraps round. See *Synchronization Frequency Register, ETMSYNCFR* on page 3-101.
- The PTM receives an external request, for example from a CoreSight system.

When the PTM receives a periodic synchronization request it generates the synchronization packets in the following order:

1. A-sync.
2. I-sync.
3. T-sync.

When the Programming bit is cleared or the OS Lock is cleared, the PTM must generate the A-sync and I-sync packets before generating any other trace:

1. The A-sync packet must be generated as the first packet.
2. This is followed by a nonperiodic I-sync packet. This is generated when trace is turned on, unless a trigger occurs before trace is turned on.
3. The PTM must generate a T-sync packet, but this might not be output immediately after the I-sync packet.

The PTM must generate the three synchronization packets in close succession, but they do not have to immediately follow each other. This means the PTM can delay the generation of any synchronization packet if the FIFO does not have enough space for the packet.

If the PTM receives a periodic synchronization request while trace is disabled because the **TraceEnable** signal is LOW, it delays synchronization until trace is re-enabled.

Forced overflow

If the PTM is servicing a periodic synchronization request, and receives a second periodic synchronization request from the same source, it immediately stops generating trace and drains the trace FIFO. This is described as a forced overflow.

When trace restarts, the PTM must generate the full periodic synchronization sequence. In this sequence, the I-sync packet indicates the overflow condition.

4.11.2 Alignment synchronization

A-sync packets provide alignment synchronization. The PTM generates an A-sync packet when it is initialized, and then generates additional A-sync packets periodically. A trace decompressor uses the A-sync packets to achieve:

- Byte alignment. This ensures the decompressor can determine the start of each byte of trace.
- Packet boundary synchronization. This ensures the decompressor does not attempt to start decompression in the middle of a multi-byte packet.

The PFT protocol guarantees that the first byte (8 bits) after the last bit of an A-sync packet is the first byte of a new packet. Therefore, detecting an A-sync packet achieves byte alignment and packet boundary synchronization.

For details of the A-sync packet see *A-sync, alignment synchronization packet* on page 4-161.

4.11.3 Instruction synchronization

I-sync packets provide instruction synchronization. A trace decompressor uses an I-sync packet to synchronize the instruction address, instruction set state, security state, and Context ID. The PTM generates both nonperiodic and periodic I-sync packets. See *Nonperiodic I-sync* and *Periodic I-sync* on page 4-193.

Nonperiodic and periodic I-sync packets have the same format. See *I-sync, instruction synchronization packet* on page 4-162.

Nonperiodic I-sync

The PTM generates a nonperiodic I-sync packet every time tracing turns on. Tracing turns on for the following reasons:

- The processor leaves Debug state.
- TraceEnable becomes active. See *TraceEnable* on page 3-34
- The PTM recovers from overflow. See *Trace FIFO overflow* on page 4-188
- The processor leaves a prohibited region. See *Prohibited regions* on page 4-186
- The processor leaves Jazelle state. See *Jazelle state* on page 4-201.

The I-sync packet includes a reason code that indicates why tracing has started or re-started. See *I-sync, instruction synchronization packet* on page 4-162 for more information.

In addition, an I-sync packet contains:

- An instruction address. This is the most recently known instruction address:
  - for normal trace turn on and FIFO overflow recovery, this is the address of the most recent waypoint target or exception target
  - for debug exit, this is the address of the first instruction to be executed, and is related to the PC value while the processor was in Debug state.
- The instruction set state, and security state, when that instruction is executed.
- The Context ID of the instruction at the given address, if you have enabled Context ID tracing.

- The cycle count, if you have enabled cycle-accurate tracing. This is the number of cycles from the last cycle count output up to the most recent waypoint.

Periodic I-sync

Periodic I-sync packets have the Periodic reason code. They include the address, instruction set state and security state of the target of the most recent waypoint.

If you have enabled Context ID tracing the packet includes the Context ID.

Periodic I-sync packets never include a cycle count.

The PFT architecture requires the generation of periodic I-sync packets so that a decompressor can start operating from a point some way through the generated trace stream.

A trace decompressor can use the contents of a periodic I-sync packet to confirm the address of the current instruction, and other information about the processor state when that instruction is executed. Therefore, they provide a limited form of error checking.

A VMID packet must also be output before the next branch or atom packet after an I-sync packet.

4.11.4 Timestamp synchronization

Usually, a timestamp packet compresses the timestamp value. See *Timestamp packet* on page 4-181. A T-sync packet is a timestamp packet that outputs all of the timestamp.

4.12 Tracing security state changes

The following subsections describe the tracing of security state changes:

- *Changing from Non-secure to Secure state*
- *Changing from Secure to Non-secure state*.

Note

See the *ARM Architecture Reference Manual* for the ARM recommendations for changing between Secure and Non-secure security states. To avoid security holes it is very important that you follow these recommendations. This section includes descriptions of the tracing of security state change methods that ARM recommends that you do not use.

4.12.1 Changing from Non-secure to Secure state

The processor changes from Non-secure to Secure state only because of an exception. The PTM traces this in the usual way. It outputs a waypoint update packet, if required, and then outputs an exception branch address packet. If you have enabled Context ID tracing then the PTM generates a Context ID packet, if required, immediately after generating the exception branch address packet.

4.12.2 Changing from Secure to Non-secure state

The processor can change from Secure to Non-secure state for the following reasons:

- An exception return. ARM recommends this as the normal way of returning to Non-secure state.
- Execution of an `MCR` instruction to write to bit [0] of the SCR, to change the security state.
- Execution of an `MSR` instruction when in Monitor mode, to change the security state.

When the state change is because of an exception return, the PTM generates a branch address packet that indicates the security state change. If you have enabled Context ID tracing the PTM generates a Context ID packet, if required, immediately after the branch address packet.

ARM recommends that you do not use the other mechanisms to change security state, and if one of these mechanisms causes the state change, the ARM architecture does not guarantee the exact point of the security state change. The state change might not take effect until the next time the processor executes an ISB or exception return instruction or takes an exception and therefore the PTM does not trace the state change immediately. This means that the security state change is not marked in the trace output when it occurs, but only at the next branch address packet generated by the PTM. After a security state update, the PTM always generates a branch address packet for a taken direct branch, and therefore the state change is reported at the first of:

- the next taken branch, whether the branch is direct or indirect
- the next exception.

In the unusual case that the next waypoint after the security state change is a not-taken branch, the PTM generates an N atom, and does not report the state change until the next taken branch or exception. If the next waypoint after the security state change is an exception that returns the processor to Secure state the PTM does not indicate the entry to Non-secure state in the trace output stream.

4.13 Use of a return stack

The PFT architecture includes a *return stack* that you can use to reduce the number of branch address packets generated by the PTM. Branch address packets are a high proportion of the data in the trace stream, and using the return stack significantly reduces the amount of trace output.

You enable use of the return stack by setting the Return stack enable bit of the Main Control Register to 1. See *Main Control Register, ETMCR* on page 3-75. On a PTM reset, this bit is set to 0, disabling the use of the return stack.

The PTM can use the return stack for tracing branch with link instructions. Table 4-12 lists these instructions.

Table 4-12 Branch with link instructions

Instruction		**Instruction set**		
Mnemonic	**Description**	**ARM**	**Thumb, 16-bit**	**Thumb, 32-bit**
`BL`	Branch and link	Yes	-	-
`BL <immed>`	Branch and link	-	-	Yes
`BLX <reg>`	Branch with link and exchange	Yes	Yes	-
`BLX <immed>`	Branch immediate with link and exchange	Yes	-	Yes
`HBL`, `HBLP`	Handler branch with link	-	Yes	-

When a branch with link instruction is executed, the PTM puts the return address of the branch instruction onto the return stack. This address is the address stored in the LR for the link. The return stack entry includes the instruction set and security states that correspond to this return address. If the stack is full, the oldest entry is discarded to make room for the new entry.

When an indirect branch instruction is executed and passes its condition code check, if the return stack is not empty the target of the branch is compared with the most recent entry on the stack. If the address, instruction set state, and security state all match then:

- The PTM does not generate a branch address packet for the instruction. Instead, it generates the E atom that is normally implied by a branch address packet.
- The entry is removed from the stack.

If the return stack is empty, or the indirect branch instruction does not match the most recent stack entry, then the PTM traces the indirect branch normally, by generating a branch address packet, and does not change the contents of the return stack.

If the most recent entry on the return stack matches, but the waypoint causes a change in security state, then the PTM always generates a branch address packet for the waypoint. The entry is not removed from the return stack.

Any periodic or nonperiodic I-sync packet clears the contents of the return stack.

Table 4-13 summarizes how the PTM traces branch instructions when you have enabled use of the return stack.

Table 4-13 PTM branch tracing when using the return stack

Branch type	Condition code check	Return stack		Trace generated
		Matches[a]	Operation	
Indirect	Pass	Yes	Remove most recent entry	E atom
		No	No change	Branch address packet
	Fail	X	No change	N atom
Direct	Pass	X	No change	E atom
	Fail	X	No change	N atom

a. Does the top entry on the stack match the address, instruction set state, and security state of the branch target address? If the return stack is empty the processing is the same as for a failed match.

When an exception occurs, the return stack has no effect. On an exception, the PTM always outputs a branch address packet and leaves the return stack unchanged.

The ordering of operations on the return stack is important. When tracing an indirect branch instruction, the PTM compares the branch address with the most recent entry on the stack before performing any push operation associated with the instruction. For example, if at address `0x1000` there is an indirect `BLX` to `0x2000`, the PTM operates as follows:

1. Compare the destination address of the `BLX` instruction, `0x2000`, with the most recent address on the return stack. If the address, instruction set state, and security state all match then remove the most recent entry from the stack.
2. Regardless of the outcome of step 1, push the return address of the `BLX` instruction, `0x1004`, onto the return stack.

If the security state of an entry that the PTM might push onto the return stack is different from the last security state it output in the trace stream, then the PTM:

- does not push the entry onto the return stack
- clears the contents of the return stack, so there are no valid entries on the stack.

From PFTv1.1, the return stack must not match on:

- exceptions being taken
- exception returns
- security state changes
- Hyp mode changes.

This ensures an explicit branch packet is traced for each of these scenarios.

Note

- This cannot occur in code that uses only the methods recommended by ARM to change the security state. See the *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition*.
- A trace decompression tool does not have to know that the PTM has cleared the return stack.

The size of the return stack is IMPLEMENTATION DEFINED, from 0-15 entries. There is no mechanism for detecting the size of this stack, and a trace decompressor does not require this information. The Return stack implemented bit in the Configuration Code Extension Register is set to 1 if the return stack is implemented. See *Configuration Code Extension Register, ETMCCER* on page 3-103.

Note

- Depending on the application code, using the return stack reduces the amount of trace generated by 25-40%.
- There are no packet formats or other PFT protocol elements associated with using the return stack. The effect of using the return stack is to replace some branch address packets with E atoms.
- Use of the return stack is a prediction technique. The PTM predicts the target of a branch and if the prediction is:
  - — correct, it outputs an E atom
  - — incorrect, it outputs a branch address packet.

4.14 Timestamping

You enable timestamping by setting the Timestamp enable bit in the Main Control Register to 1. See *Main Control Register, ETMCR* on page 3-75. On power up, or after a reset, this bit is set to 0, disabling timestamping.

Timestamp packet on page 4-181 describes the timestamp packet. The T-sync packet is a special case of the timestamp packet, that always outputs all seven bytes of the timestamp.

If you have enabled timestamping, the following events request the generation of a timestamp packet:

- Periodic synchronization. See *Periodic synchronization* on page 4-191.
- If you have enabled cycle-accurate tracing, the output of a cycle count that indicates overflow.
- A change in the clock frequency of the processor.
- The processor executes an `ISB` instruction, and the instruction is traced.
- Bit [25] of the Configuration Code Extension Register is set to 1, the processor executes a `DMB` or `DSB` instruction, and the instruction is traced.
- The PTM receives a request to flush the trace FIFO. See *Trace flushing* on page 4-199.
- The timestamp event goes active. See *Timestamp Event Register, ETMTSEVR* on page 3-107.
- An exception is traced, from PFTv1.1.
- An instruction which causes a return from an exception is traced, from PFTv1.1. See *Exception return packet* on page 4-183.

When it receives a timestamp request, the PTM generates a timestamp packet only if at least one of the following applies:

- Tracing is currently active.
- The request is the first request since tracing was last disabled.

  In this case, the timestamp packet can be compressed, even if it is a response to a full synchronization request. However, if the timestamp is requested while tracing is turned off, full periodic synchronization is required when trace turns on again.

The PTM might not generate the timestamp packet immediately it receives the request. It can delay generating the packet to a point that is convenient to the PTM.

In PFTv1.0, the timestamp indicates the time the packet is generated, not the time the request was made. This means that a timestamp does not indicate the time when the requesting event occurred. A timestamp is only a time indicator inserted into the trace stream near the event that requested a timestamp.

In PFTv1.1, the timestamp indicates the time of the last traced waypoint.

If the PTM receives multiple timestamp requests close together, it might not generate a timestamp packet for each request. However:

- when it receives a periodic synchronization request it must generate all of the synchronization packets, including the full T-sync packet
- when the clock frequency of the processor changes it must generate a timestamp packet that indicates this change.

If you have enabled timestamping, timestamps are generated based on the value of the timestamp event. If the timestamp event is TRUE for many consecutive cycles, the PTM might generate many timestamp packets in close succession. Therefore, ARM recommends that, when programming the Timestamp Event Register, you use event resources that are active for only a single cycle at a time.

4.15 Trace flushing

There are some situations when the PTM must flush its trace FIFO, and output any atoms it has assembled. Most of these requirements are described elsewhere in this specification, and are included here for reference. The following sections summarize the requirements:

- *CoreSight or other ATB flush request*
- *Setting the Programming bit or the OS Lock*
- *WFI or WFE request*.

4.15.1 CoreSight or other ATB flush request

The PTM might output its trace data over an *Advanced Trace Bus* (ATB). This data is received by a trace sink, such as a CoreSight *Trace Port Interface Unit* (TPIU) in a CoreSight system.

A trace sink can send a flush request to the PTM. When the PTM receives such a request it must drain all trace currently in the FIFO, and output any trace it is constructing. It must output any atoms it has assembled in an atom packet.

The PTM acknowledges the trace flush only after it has output all of the flushed data.

If you have enabled timestamping, the PTM must output a timestamp when it has output all of the required data. Normally, it does this before acknowledging the trace flush. However, if conditions exist that prevent it outputting the timestamp then it does not delay the flush acknowledge excessively. This means that the PFT architecture does not guarantee that the timestamp is issued before the flush is acknowledged. However, after an ATB flush request the PTM must issue a timestamp, even if it does so after acknowledging the flush.

The PTM does not stop trace generation while servicing this flush request.

4.15.2 Setting the Programming bit or the OS Lock

If you set the Programming bit in the Main Control Register to 1, or set the OS Lock, then the PTM immediately stops all trace generation, drains all trace from its FIFO, and outputs any atoms it has assembled.

4.15.3 WFI or WFE request

When the PTM receives a WFI or WFE request it must drain all trace from its FIFO, and output any atoms it has assembled. In these cases the PTM does not stop trace generation, but normally does not generate any more trace. See *Wait for Interrupt and Wait for Event* on page 4-189.

When the PTM has output all of the required data it signals that it is ready to enter the WFI or WFE state.

4.16 Tracing Thumb instructions

This section describes particular issues about how a PTM traces some Thumb instructions. It contains the following subsections:

- *32-bit Thumb instructions*
- *Thumb CBZ and CBNZ instructions*.

4.16.1 32-bit Thumb instructions

How a PTM traces 32-bit Thumb waypoint instructions depends on the setting of bit [18], Support for 32-bit Thumb instructions, of the ID Register. See *ID Register, ETMIDR* on page 3-101:

- If bit [18] is set to 1, each 32-bit Thumb waypoint instruction is traced as a single instruction. Address comparators only match on the address of the lower halfword of the instruction.
- If bit [18] is set to 0, each 32-bit Thumb waypoint instruction is traced as two instructions. Exceptions can occur between the two instructions. An address comparator can match on the address of either halfword of the instruction. The waypoint instruction address is the address of the upper halfword of the instruction.

4.16.2 Thumb `CBZ` and `CBNZ` instructions

If a `CBZ` or `CBNZ` instruction does not branch then it is traced as a waypoint instruction that failed its condition code test. `CBZ` and `CBNZ` are direct branches and do not require a branch address packet output when the condition matches, unless the BranchBroadcast bit, bit [8], of the Main Control Register is set to 1. See *Main Control Register, ETMCR* on page 3-75.

Note

Although the `CBZ` or `CBNZ` instruction makes the required comparison, NE or E, with zero, the instruction does not update the CPSR flags.

4.17 Jazelle state

Note

In this section, *non-Jazelle state* means any processor instruction set state other than Jazelle state.

When the processor is in Jazelle state it executes Java bytecodes. The PFT architecture does not support the tracing of Java bytecode execution.

If a branch into Jazelle state occurs while tracing is enabled:

- The PTM traces the branch into Jazelle state, indicating the address of the first instruction in Jazelle state.
- Tracing stops. No more trace is generated, except for triggers and timestamps, until the processor leaves Jazelle state.

While the processor is in Jazelle state, no address comparators match, but all other PTM resources behave as normal, including Context ID and VMID comparators. Instrumentation resources remain in their current state.

If TraceEnable becomes active while the processor is in Jazelle state this change is ignored and trace does not turn on. When the processor leaves Jazelle state, trace turns on if TraceEnable remains active.

When the processor leaves Jazelle state, if tracing is enabled the PTM generates an I-sync packet that indicates the address of the first instruction in non-Jazelle state. This packet is similar to normal trace turn on. If you have enabled cycle-accurate tracing then the packet includes the number of cycles between the last cycle count output and this first instruction in non-Jazelle state.

This I-sync packet is generated regardless of whether the instruction in non-Jazelle state is actually executed. For example, if an exception occurs on this instruction, the I-sync packet is generated and the exception is traced normally.

4.18 Debug state

When the processor enters Debug state, instruction execution stops. This means that tracing also stops and the FIFO continues to drain until empty.

Instructions executed in Debug state are ignored by the PTM.

The operation of all the PTM resources is unaffected by Debug state and the resources continue to operate as normal.

Instrumentation resources maintain their current state while in Debug state.

Context ID and VMID comparators continue to operate in Debug state. If the Context ID or VMID is changed while in Debug state, the Context ID or VMID comparators might not observe the change until the processor leaves Debug state and one of the following has occurred:

- an ISB instruction is executed
- an exception entry occurs
- an exception return instruction is executed.

When the processor exits Debug state, tracing restarts if tracing is enabled. The reason code that is generated indicates that the processor has exited from Debug state.

If an overflow has occurred on entry into Debug state, the debug tools can detect this by reading the Status Register. For more information see *Status Register, ETMSR* on page 3-81.

For more information about Debug state see:

- the Debug section of the *ARM Architecture Reference Manual*
- the data sheet or *Technical Reference Manual* for the appropriate processor.

Chapter 5
Tracing Exceptions

This chapter describes how a PTM traces exceptions. It contains the following sections:

5.1 About exception tracing in the PFT architecture

Upgrading a nonwaypoint instruction on an exception on page 2-27 introduced PFT requirements for tracing exceptions. However, these requirements for tracing exceptions are very different to the requirements of other ARM trace architectures, and therefore this chapter gives a detailed description of this topic:

- *The different exception cases* on page 5-205 describes how exception tracing depends on where the exception occurred, and considers the cases:
  - — an exception occurs after a nonwaypoint instruction
  - — an exception occurs immediately after a waypoint instruction
  - — an exception occurs immediately after another exception
  - — an exception occurs immediately after you turn tracing on.
- *Tracing the different exception types* on page 5-211 describes how a PTM traces each exception type. The exception types are:
  - — a processor reset
  - — an Undefined Instruction exception
  - — a Supervisor Call exception, on an `SVC` instruction
  - — a Secure Monitor Call exception, on an `SMC` instruction
  - — a Prefetch Abort exception
  - — a synchronous Data Abort exception
  - — an asynchronous Data Abort exception
  - — an FIQ exception, including a nonmaskable FIQ (NMFI)
  - — an IRQ exception
  - — Debug state entry
  - — a ThumbEE check that goes to a handler
  - — an exception caused by a `BKPT` instruction, a hardware breakpoint or watchpoint, or similar.
- *Waypoint update addresses* on page 5-217 describes the addresses used in waypoint update packets, when these are required.

5.2 The different exception cases

This section describes how the trace a PTM generates on an exception depends on where that exception occurs, relative to the waypoint instructions and to other exceptions. The following subsections describe the different possibilities:

- *Exception occurs after a nonwaypoint instruction*
- *Exception occurs immediately after a waypoint instruction*
- *Exception occurs immediately after another exception* on page 5-206
- *Exception occurs immediately after trace turn-on* on page 5-210.

5.2.1 Exception occurs after a nonwaypoint instruction

If an exception occurs after the processor has successfully executed at least one nonwaypoint instruction, then the PTM:

- upgrades the last executed instruction to a waypoint instruction, and traces it with a waypoint update packet, see *Waypoint update packet* on page 4-177
- traces the exception with an exception branch address packet, see *Branch address packet* on page 4-166.

Table 5-1 shows an example of how the PTM traces an exception in this case.

Table 5-1 Tracing an exception occurring after execution of a nonwaypoint instruction

Address	Instruction	Trace, if any, with explanation
`0x0F00`	`BEQ 0x1004`	This waypoint is executed. The PTM traces it by generating an E atom.
`0x1004`	`ADD`	-
`0x1008`	`ADD`	`ADD` is not a waypoint instruction so the PTM does not trace anything when the `ADD` instruction is executed.
-	IRQ occurs	The PTM upgrades the `ADD` at `0x1008` to a waypoint instruction and generates a waypoint update packet with the address of `0x1008`[a]. It then issues an exception branch address packet, indicating an IRQ, with a destination address of `0x0018`, the IRQ vector address.
`0x0018`	`LDR PC`, loads `0x3000`	Branch address packet with destination `0x3000`. This implies an E atom. This is a normal indirect branch.
`0x3000`	(IRQ handler)	Traced normally. Depending on the instruction stream and TraceEnable configuration, the PTM might generate some packets.
-		
-		
-		

a. If the PTM is holding any assembled atoms it outputs these in an atom header before generating the waypoint update packet. This means that the trace output stream has any outstanding atoms, followed by the waypoint update packet, followed by the exception branch address packet.

5.2.2 Exception occurs immediately after a waypoint instruction

If an exception occurs immediately after the processor has successfully executed a waypoint instruction, then the PTM traces the exception by generating only an exception branch address packet.

Table 5-2 shows an example of tracing an exception that occurs immediately after a waypoint instruction. The PTM traces the exception in this way because nothing is executed between the `BEQ` at `0x1004` and the instruction at the IRQ vectors. No address comparators match for the instruction at `0x1004` until the processor returns from the exception handler to the instruction at `0x1004`.

Table 5-2 Tracing an exception immediately after execution of a waypoint instruction

Address	Instruction	Trace, if any, with explanation
`0x0F00`	`BEQ 0x1004`	This waypoint is executed. The PTM traces it by generating an E atom.
-	IRQ occurs	PTM issues an exception branch address packet, indicating an IRQ, with a destination address of `0x0018`, the IRQ vector address[a]. The last instruction executed is the waypoint instruction at `0x0F00`.
`0x0018`	`LDR PC`, loads `0x3000`	Branch address packet with destination `0x3000`. This implies an E atom. This is a normal indirect branch.
`0x3000`	(IRQ handler)	Traced normally. Depending on the instruction stream and TraceEnable configuration, the PTM might generate some packets.
-		
-		
-		

a. If the PTM is holding any assembled atoms it outputs these in an atom header before generating the exception branch address packet. This means it always outputs the E atom for the waypoint instruction that was just executed before outputting the exception branch address packet.

5.2.3 Exception occurs immediately after another exception

If two exceptions occur back-to-back, without the processor executing any instructions between them, then the PTM traces each exception with an exception branch address packet, without any waypoint update packet between them. In the trace stream, an exception branch address packet occurring immediately after another exception branch address packet indicates that one exception occurred immediately after the other.

For example, consider the case where an FIQ interrupts an IRQ:

- When the IRQ occurs, the processor updates R14_irq with the exception return address and switches to IRQ mode.
- The FIQ occurs before the processor has fetched the instruction from the IRQ vector. Therefore, the processor updates R14_fiq with the IRQ vector address and switches to FIQ mode.
- The processor fetches the instruction at the FIQ vector and branches to the FIQ handler.

The FIQ handler returns to the IRQ handler as if it was a normal branch. Therefore, for execution up to the entry to the FIQ handler, the PTM must generate a trace stream that indicates that both exceptions occurred, without any instructions being executed at the IRQ vector address. Table 5-3 shows how this is traced.

Table 5-3 Tracing back-to-back exceptions

Address	Instruction	Trace, if any, with explanation
`0x1000`	`MOV`	Nonwaypoint instruction. Not traced when processor executes the instruction.
-	IRQ occurs	The PTM upgrades the `MOV` at `0x1000` to a waypoint instruction and generates a waypoint update packet with address `0x1000`[a]. It then issues an exception branch address packet, indicating an IRQ, with a destination address of `0x0018`, the IRQ vector address.
-	FIQ occurs	No instruction executed at `0x0018`. To trace the FIQ, the PTM issues an exception branch address packet, indicating an FIQ, with a destination address of `0x001C`, the FIQ vector address. The last instruction executed is the upgraded waypoint instruction at `0x1000`.

Table 5-3 Tracing back-to-back exceptions (continued)

Address	Instruction	Trace, if any, with explanation
`0x001C`	`LDR PC`, loads `0x2000`	Traced as a normal indirect branch, with a branch address packet with a destination address of `0x2000`.
`0x2000`	(FIQ handler)	Traced normally. Depending on the instruction stream and TraceEnable configuration, the PTM might generate some packets.
-		
-		
-		
`0x2100`	`SUBS PC, R14, #4`	Return from FIQ handler, returns to IRQ vector. PTM traces the return with a non-exception branch address packet with a destination address of `0x0018`.
`0x3000`	(IRQ handler)	Traced normally. Depending on the instruction stream and TraceEnable configuration, the PTM might generate some packets.
-		
-		
-		
`0x3100`	`SUBS PC, R14, #4`	Return from IRQ handler, returns to instruction after the last executed instruction. PTM traces the return with a non-exception branch address packet with a destination address of `0x1004`.
`0x1004`	`MOV`	-

a. If the PTM is holding any assembled atoms it outputs these atoms in an atom header before generating the waypoint update packet. This means that the trace output stream has any outstanding atoms, followed by the waypoint update packet, followed by the exception branch address packet.

When the PTM detects that two exceptions have occurred back-to-back:

- no single address comparators match
- all address range comparators retain their previous value.

Note

A processor microarchitecture might not permit two exceptions to behave in the way shown in Table 5-3 on page 5-206.

Exceptions occurring close together but not back-to-back

For comparison with the tracing of back-to-back exceptions, this subsection gives examples of the PTM trace generated when the FIQ occurs after the processor has executed the instruction at the IRQ vector.

In the first example, the processor executes a branch at the IRQ vector address before the FIQ occurs. Table 5-4 shows this case.

Table 5-4 Normal tracing of an FIQ after executing a branch at the IRQ vector address

Address	Instruction	Trace, if any, with explanation
`0x1000`	`MOV`	Nonwaypoint instruction. Not traced when processor executes the instruction.
-	IRQ occurs	The PTM upgrades the `MOV` at `0x1000` to a waypoint instruction and generates a waypoint update packet with address `0x1000`[a]. It then issues an exception branch address packet, indicating an IRQ, with a destination address of `0x0018`, the IRQ vector address.
`0x0018`	`LDR PC`, loads `0x3000`	Processor executes the instruction at the IRQ vector address. Traced as a normal indirect branch, with a branch address packet with a destination address of `0x3000`.
-	FIQ occurs	Last instruction, at `0x0018`, was a waypoint instruction. Therefore the PTM traces the FIQ by issuing an exception branch address packet, indicating an FIQ, with a destination address of `0x001C`, the FIQ vector address.
`0x001C`	`LDR PC`, loads `0x2000`	The instruction at the FIQ vector address, traced as a normal indirect branch, with a branch address packet with a destination address of `0x2000`.
`0x2000`	(FIQ handler)	Traced normally. Depending on the instruction stream and TraceEnable configuration, the PTM might generate some packets.
-		
-		
-		
`0x2100`	`SUBS PC, R14, #4`	Return from FIQ handler, returns to the destination of the `LDR PC` at the IRQ vector address, `0x3000`. PTM traces the return with a non-exception branch address packet with a destination address of `0x3000`.
`0x3000`	(IRQ handler)	Traced normally. Depending on the instruction stream and TraceEnable configuration, the PTM might generate some packets.
-		
-		
-		
`0x3100`	`SUBS PC, R14, #4`	Return from IRQ handler, returns to instruction after the last executed instruction. PTM traces the return with a non-exception branch address packet with a destination address of `0x1004`.
`0x1004`	`MOV`	-

a. If the PTM is holding any assembled atoms it outputs these in an atom header before generating the waypoint update packet. This means that the trace output stream has any outstanding atoms, followed by the waypoint update packet, followed by the exception branch address packet.

The second example shows the case where the instruction executed at the IRQ vector address is not a waypoint instruction. Table 5-5 shows this case, with the processor executing a `NOP` at the IRQ vector address before the FIQ occurs.

Table 5-5 Normal tracing of an FIQ after executing a `NOP` at the IRQ vector address

Address	Instruction	Trace, if any, with explanation
`0x1000`	`MOV`	Nonwaypoint instruction. Not traced when processor executes the instruction.
-	IRQ occurs	The PTM upgrades the `MOV` at `0x1000` to a waypoint instruction and generates a waypoint update packet with the address `0x1000`[a]. It then issues an exception branch address packet, indicating an IRQ, with a destination address of `0x0018`, the IRQ vector address.
`0x0018`	`NOP`, 16-bit Thumb instruction	Processor executes the instruction at the IRQ vector address. This is not a waypoint instruction so the PTM does not generate any trace.
-	FIQ occurs	The PTM upgrades the `NOP` at `0x0018` to a waypoint instruction and generates a waypoint update packet. It then issues an exception branch address packet, indicating an FIQ, with a destination address of `0x001C`, the FIQ vector address.
`0x001C`	`LDR PC`, loads `0x2000`	The instruction at the FIQ vector address, traced as a normal indirect branch, with a branch address packet with a destination address of `0x2000`.
`0x2000`	(FIQ handler)	Traced normally. Depending on the instruction stream and TraceEnable configuration, the PTM might generate some packets.
-		
-		
-		
`0x2100`	`SUBS PC, R14, #4`	Return from FIQ handler, returns to the instruction after the `NOP` at the IRQ vector address, therefore returns to `0x001A`. PTM traces the return with a non-exception branch address packet with a destination address of `0x001A`.
`0x0001A`	(IRQ handler)	Traced normally. Depending on the instruction stream and TraceEnable configuration, the PTM might generate some packets.
-		
-		
-		
`0x0100`	`SUBS PC, R14, #4`	Return from IRQ handler, returns to instruction after the last executed instruction. PTM traces the return with a non-exception branch address packet with a destination address of `0x1004`.
`0x1004`	`MOV`	-

a. If the PTM is holding any assembled atoms it outputs these in an atom header before generating the waypoint update packet. This means that the trace output stream has any outstanding atoms, followed by the waypoint update packet, followed by the exception branch address packet.

Turning trace on between two back-to-back exceptions

Considering the case when an IRQ and an FIQ occur back-to-back, if you turn tracing on between the IRQ and the FIQ then the PTM generates an I-sync packet, with the address of the IRQ vector, followed by an exception branch address packet with a destination address of the FIQ vector address. It does not generate any waypoint update packet.

5.2.4 Exception occurs immediately after trace turn-on

Whenever the PTM turns on tracing, it generates an I-sync packet. The address of this packet is the target address of the last waypoint. The following subsections describe the PTM trace generated when an exception occurs before the processor reaches a waypoint instruction:

- *Exception occurs before execution of instruction at the I-sync target address*
- *Exception occurs after execution of instruction at the I-sync target address*.

Exception occurs before execution of instruction at the I-sync target address

If an exception occurs before the processor has executed any instructions after the last waypoint target address, the PTM outputs an exception branch address packet immediately after the I-sync packet that indicates trace turn-on. This means the trace output is:

1. The I-sync packet indicating trace turn-on. The address is the target of the last waypoint, for example `0x1000`.
2. An exception branch address packet, indicating the exception type and vector address.

This trace indicates that the exception occurred before the processor executed any instructions at the target address of the I-sync packet, `0x1000` in the example.

Exception occurs after execution of instruction at the I-sync target address

If an exception occurs after the processor has executed one or more instructions after the last waypoint, but before it reaches a new waypoint instruction, the PTM upgrades the last instruction executed to a waypoint instruction, and then traces the exception branch. This means the trace output is:

1. The I-sync packet indicating trace turn-on. The address is the target of the last waypoint, for example `0x1000`.
2. A waypoint update packet indicating the last instruction executed, for example `0x1050`.
3. An exception branch address packet, indicating the exception type and vector address.

5.3 Tracing the different exception types

This section describes how the PTM traces each type of exception. In some cases, it traces a number of exceptions in the same way. Table 5-6 lists all the exceptions, and shows where the corresponding PTM trace is described.

Table 5-6 Descriptions of how a PTM traces different exceptions

Exception	Description, see
Processor reset	*Processor reset exception*
Undefined Instruction	*Undefined Instruction exception* on page 5-212
SVC	*SVC (Supervisor Call) or SMC (Secure Monitor Call) exception* on page 5-212
SMC	
Prefetch Abort	*Prefetch Abort exception* on page 5-212
Synchronous Data Abort	*Synchronous Data Abort exception* on page 5-212
Asynchronous Data Abort	*Asynchronous Data Abort, FIQ or IRQ exception* on page 5-214
FIQ[a]	
IRQ	
Debug state entry	*Debug state entry, when Halting debug-mode is enabled* on page 5-214
Hypervisor mode entry	*Entry to Hyp mode* on page 5-214
ThumbEE check handler	*ThumbEE check that goes to a handler, including the CHKA instruction* on page 5-215
Jazelle exception	*Jazelle exception that goes to an ARM or Thumb state handler* on page 5-216
Secure to Non-secure state change	*Secure to Non-secure state change* on page 5-216
`BKPT` instruction	*Other exceptions* on page 5-216
Hardware breakpoint or watchpoint	
Other	

a. Including NMFI (Nonmaskable fast interrupt).

Before tracing any exception, if the PTM has assembled any atoms it outputs them in an atom header. In the trace stream, the PTM outputs this atom header before it traces the exception. However, if the first stage of tracing the exception is to trace a waypoint instruction, and that waypoint instruction is traced with an atom, the PTM might generate a single trace packet containing any assembled atoms and the atom from the waypoint instruction, see *Prefetch Abort exception* on page 5-212, *Synchronous Data Abort exception* on page 5-212, *Asynchronous Data Abort, FIQ or IRQ exception* on page 5-214, *Debug state entry, when Halting debug-mode is enabled* on page 5-214.

5.3.1 Processor reset exception

The PTM traces a processor reset as an indirect branch, by generating an exception branch address packet. It never generates a waypoint update packet, and therefore the trace does not indicate the position in the program where the reset occurs. The exception branch address packet indicates the reset exception.

The PTM treats the last waypoint in the trace stream as the last instruction successfully executed.

5.3.2 Undefined Instruction exception

The PTM upgrades the Undefined Instruction to a waypoint instruction and outputs the following trace:

1. A waypoint update packet, indicating the address of the Undefined Instruction that the processor committed for execution.
2. An exception branch address packet that identifies the Undefined Instruction exception and gives the address of the exception vector.

An instruction might cause an Undefined Instruction exception only if it passes its condition code check. If such an instruction fails its condition code check, the PTM does not upgrade it to a waypoint instruction.

5.3.3 SVC (Supervisor Call) or SMC (Secure Monitor Call) exception

If an `SVC` or `SMC` instruction passes its condition code check, the PTM upgrades the instruction to a waypoint instruction and outputs the following trace:

1. A waypoint update packet, indicating the address of the executed `SVC` or `SMC` instruction.
2. An exception branch address packet that identifies the SVC or SMC exception and gives the address of the exception vector.

If an `SVC` or `SMC` instruction fails its condition code check, the PTM does not upgrade it to a waypoint instruction.

5.3.4 Prefetch Abort exception

If the Prefetch Abort exception occurs when the processor branches to the destination of a branch instruction, or when it commits for execution the instruction immediately after any other waypoint, the PTM:

1. Traces the waypoint normally, by generating one of:
   - An atom, that it outputs in an atom packet. This packet might include any atoms assembled before the exception occurred.
   - A branch address packet.
2. Generates an exception branch address packet that identifies the Prefetch Abort exception and gives the address of the exception vector.

If the Prefetch Abort exception occurs on an instruction that follows an executed nonwaypoint instruction, the PTM:

1. Upgrades the last instruction executed to a waypoint instruction, and generates a waypoint update packet that indicates the address of that instruction.
2. Generates an exception branch address packet that identifies the Prefetch Abort exception and gives the address of the exception vector.

   The address that aborts is the address of the instruction that immediately follows the upgraded waypoint instruction.

5.3.5 Synchronous Data Abort exception

If the synchronous Data Abort exception occurs on an instruction that immediately follows a waypoint instruction, the PTM:

1. Traces the waypoint instruction normally, by generating one of:
   - An atom, that it outputs in an atom packet. This packet might include any atoms assembled before the exception occurred.
   - A branch address packet.
2. Generates an exception branch address packet that identifies the synchronous Data Abort exception and gives the address of the exception vector.

In this case the instruction that caused the Data Abort exception is at the target address of the last waypoint.

Table 5-7 shows an example of this trace.

Table 5-7 Tracing a synchronous Data Abort exception after a waypoint instruction

Address	Instruction	Trace, if any, with explanation
`0x0F00`	`LDR PC`, loads `0x1004`	This waypoint is executed. The PTM traces it by generating a branch address packet, with target address `0x1004`.
`0x1004`	`LDR R0`, Data aborts	Exception branch address packet, indicating a synchronous Data Abort exception, with a destination address of `0x0010`, the Data Abort vector address.
`0x0010`	`LDR PC`, loads `0x3400`	Branch address packet with destination `0x3400`. This implies an E atom. This is a normal indirect branch.
`0x3400`	(Data Abort exception handler)	Traced normally. Depending on the instruction stream and TraceEnable configuration, the PTM might generate some packets.
-		
-		
-		

If the synchronous Data Abort exception occurs on an instruction that follows an executed nonwaypoint instruction then the PTM:

1. Upgrades the instruction immediately before the instruction that caused the Data Abort exception to a waypoint instruction, and generates a waypoint update packet that indicates the address of that instruction.
2. Generates an exception branch address packet that identifies the synchronous Data Abort exception and gives the address of the exception vector.

In this case the instruction that caused the Data Abort exception is the instruction that immediately follows the upgraded waypoint instruction.

Table 5-8 shows an example of this trace.

Table 5-8 Tracing a synchronous Data Abort exception after a nonwaypoint instruction

Address	Instruction	Trace, if any, with explanation.
`0x0F00`	`LDR PC`, loads `0x1000`	This waypoint is executed. The PTM traces it by generating a branch address packet, with target address `0x1000`.
`0x1000`	`MOV`	Nonwaypoint instruction. Not traced when processor executes the instruction.
`0x1004`	`LDR R0`, Data aborts	The PTM upgrades the `MOV` at `0x1000` to a waypoint instruction and generates a waypoint update packet with the address `0x1000`[a]. It then issues an exception branch address packet, indicating an synchronous Data Abort exception, with a destination address of `0x0010`, the Data Abort vector address.
`0x0010`	`LDR PC`, loads `0x3400`	Branch address packet with destination `0x3400`. This implies an E atom. This is a normal indirect branch.
`0x3400`	(Data Abort exception handler)	Traced normally. Depending on the instruction stream and TraceEnable configuration, the PTM might generate some packets.
-		
-		
-		

a. If the PTM is holding any assembled atoms it outputs these in an atom header before generating the waypoint update packet. This means that the trace output stream has any outstanding atoms, followed by the waypoint update packet, followed by the exception branch address packet.

5.3.6 Asynchronous Data Abort, FIQ or IRQ exception

Note

FIQ exceptions include NMFI (Nonmaskable Fast Interrupt) exceptions.

If the exception occurs immediately after execution of a waypoint instruction, then the PTM:

1. Traces the waypoint instruction normally, by generating one of:
    - An atom, that it outputs in an atom packet. This packet might include any atoms assembled before the exception occurred.
    - A branch address packet.
2. Generates an exception branch address packet that identifies the exception and gives the address of the exception vector.

If the exception occurs when the processor has executed at least one nonwaypoint instruction since the last waypoint then the PTM:

1. Upgrades the last instruction executed to a waypoint instruction, and generates a waypoint update packet that indicates the address of that instruction.
2. Generates an exception branch address packet that identifies the exception and gives the address of the exception vector.

5.3.7 Entry to Hyp mode

An exception can be taken into Hyp mode for various reasons:

- An exception occurs in Hyp mode and is taken in Hyp mode.

  The exception branch packet indicates the exception taken. The branch address is the address of exception vector.
- An interrupt is trapped to Hyp mode.

  The exception branch packet indicates the interrupt type which is taken. The branch address is the address of interrupt vector. The Hyp mode bit is set in the branch packet.
- Any other exception occurs in another mode which causes entry to Hyp mode.

  These exceptions are always taken at [vector base address]+0x14. The exception branch packet uses a previously Reserved encoding, Exception[3:0] == b0011, to indicate Hyp mode entry. The Hyp mode bit is also set in the branch packet.
- An instruction is trapped into Hyp mode.

  These exceptions are always taken at [vector base address]+`0x14`. The exception branch packet uses a previously Reserved encoding, Exception[3:0] == b0011, to indicate Hyp mode entry. The Hyp mode bit is also set in the branch packet.

5.3.8 Debug state entry, when Halting debug-mode is enabled

If Debug state entry occurs immediately after execution of a waypoint instruction, then the PTM:

1. Traces the waypoint instruction normally, by generating one of:
    - An atom, that it outputs in an atom packet. This packet might include any atoms assembled before the exception occurred.
    - A branch address packet.
2. Generates an exception branch address packet that identifies the Debug state entry exception and with a target address of zero.

If Debug state entry occurs when the processor has executed at least one nonwaypoint instruction since the last waypoint then the PTM:

1. Upgrades the last instruction executed to a waypoint instruction, and generates a waypoint update packet that indicates the address of that instruction.
2. Generates an exception branch address packet that identifies the Debug state entry exception and with a target address of zero.

When entering debug state, a branch packet might be generated which indicating a halting debug exception. The following information in the branch packet is not valid and must be ignored:

- Instruction set, including the AltIS bit
- Security state, through the NS bit
- Hypervisor mode, through the Hyp bit.

New values are always output on exit from debug state.

If the Context-ID or VMID changes on entry to debug state, it is IMPLEMENTATION SPECIFIC whether these are output. The current values are always output on exit from debug state.

If the processor also entered a prohibited region when entering debug state, the rules for entering a prohibited region take precedence over these rules.

Tracing exit from Debug state

On exit from Debug state, the PTM treats the instruction address at which the processor restarts as the target of a waypoint. If tracing is enabled at this point, it generates a nonperiodic I-sync packet that includes the address of this instruction. The reason code in the I-sync packet indicates exit from Debug state. If the first instruction executed on exit from Debug state is a waypoint instruction the PTM then traces this instruction normally, using an atom or a branch address packet.

If the processor takes an exception before executing the first instruction on return from Debug state, the PTM generates:

1. An I-sync packet with a reason code of exit from Debug state, that indicates the address of the first instruction intended for execution out of Debug state.
2. An exception branch address packet that identifies the exception and gives the address of the exception vector.

When decompressing the trace stream, an exception branch address packet that comes immediately after a debug return I-sync packet shows that, on return from Debug state, the processor took the exception before executing any instructions.

If, on exit from Debug state, the processor executes at least one nonwaypoint instruction before taking an exception then the PTM promotes the last instruction executed to a waypoint instruction, and generates a waypoint update packet before tracing the exception.

5.3.9 ThumbEE check that goes to a handler, including the `CHKA` instruction

If the instruction at the target of a waypoint fails a ThumbEE check, then the PTM generates an exception branch address packet, indicating the ThumbEE exception type and the address of the handler. The instruction that failed the ThumbEE check is the target of the original waypoint.

Otherwise, if an instruction fails its ThumbEE check:

1. If the instruction before the instruction fails its ThumbEE check is not a waypoint, the PTM generates a waypoint update packet indicating the address of that instruction.
2. The PTM generates an exception branch address packet indicating the ThumbEE exception type and the address of the handler.

The instruction that failed the check is the instruction immediately after the last waypoint instruction or promoted nonwaypoint instruction.

`CHKA` is not a waypoint instruction, but if a `CHKA` instruction causes a branch to a handler then the PTM treats this branch like an exception and:

1. If the instruction before the branching `CHKA` instruction is not a waypoint, generates a waypoint update packet indicating the address of that instruction.
2. Generates an exception branch address packet indicating a ThumbEE exception.

The PTM does not generate any trace for a `CHKA` instruction that does not cause a branch to a handler.

5.3.10 Jazelle exception that goes to an ARM or Thumb state handler

Trace is disabled during Jazelle execution, and therefore the PFT architecture does not support tracing of Jazelle exceptions. However, if a Jazelle exception is trapped to ARM or Thumb state, and TraceEnable is active, the PTM restarts tracing with an I-sync packet that specifies the handler address. The PTM does not report the Jazelle exception.

5.3.11 Secure to Non-secure state change

You can change the processor state from Secure to Non-secure in many ways, and it is not always possible to determine the state change from the instruction opcode alone. See *Tracing security state changes* on page 4-194 for more information.

5.3.12 Other exceptions

Other exceptions include `BKPT` instruction exceptions, hardware breakpoints, and watchpoints.

If the exception occurs immediately after execution of a waypoint instruction, then the PTM:

1. Traces the waypoint instruction normally, by generating one of:
   - An atom, that it outputs in an atom packet. This packet might include any atoms assembled before the exception occurred.
   - A branch address packet.
2. Generates an exception branch address packet that identifies the exception and gives the address of the exception vector.

If the exception occurs when the processor has executed at least one nonwaypoint instruction since the last waypoint then the PTM:

1. Upgrades the last instruction executed to a waypoint instruction, and generates a waypoint update packet that indicates the address of that instruction.
2. Generates an exception branch address packet that identifies the exception and gives the address of the exception vector.

In both cases, see Table 4-4 on page 4-168 for the encoding used, in the exception branch address packet, to indicate the exception type.

In the specific cases of a `BKPT` instruction that causes entry to Debug state or a Prefetch Abort exception, if the PTM generates a waypoint update packet address that packet indicates the address of the instruction before the `BKPT` instruction. This is similar to how a PTM normally traces a Prefetch Abort exception, and is appropriate because the normal use of a `BKPT` instruction is that the debugger replaces the `BKPT` instruction with a real instruction and then restarts execution at the address of the `BKPT` instruction. If a `BKPT` instruction causes:

- entry to a software monitor, then the PTM reports a Prefetch Abort exception
- entry to Debug state, then the PTM reports a Debug Entry exception.

5.4 Waypoint update addresses

When the PTM must upgrade an instruction to a waypoint instruction, the address of the upgraded waypoint instruction depends on both the processor state when the exception occurred, and which exception occurred:

- Table 5-9 shows these addresses if the exception occurs when the processor is in ARM state
- Table 5-10 shows these addresses if the exception occurs when the processor is in Thumb state.

In these tables, LR is the address stored in the Link Register, LR (R14), for the exception.

Table 5-9 Waypoint update instruction addresses for exceptions in ARM state

Exception	Base LR	LR	Upgraded instruction address
Reset	-	-	-
Undefined Instruction	Instruction address[a]	(Base LR)+4	Base LR
SVC	Instruction address[a]	(Base LR)+4	Base LR
SMC	Instruction address[a]	(Base LR)+4	Base LR
HVC	Instruction address[a]	(Base LR)+4	Base LR
Data Abort	Address of aborted instruction	(Base LR)+8	(Base LR)-4
Prefetch Abort	Address of aborting instruction	(Base LR)+4	(Base LR)-4
IRQ	Address of next instruction	(Base LR)+4	(Base LR)-4
FIQ	Address of next instruction	(Base LR)+4	(Base LR)-4

a. The address of the undefined instruction, or of the `SVC`, `SMC`, or `HVC` instruction.

Table 5-10 Waypoint update instruction addresses for exceptions in Thumb state

Exception	Base LR	LR	Upgraded instruction address
Reset	-	-	-
Undefined Instruction	Instruction address[a]	(Base LR)+2	Base LR
SVC	Instruction address[a]	(Base LR)+2	Base LR
SMC	Instruction address[a]	(Base LR)+4	Base LR
HVC	Instruction address[a]	(Base LR)+4	Base LR
Data Abort	Address of aborted instruction	(Base LR)+8	(Base LR)-2 or (Base LR)-4[b]
Prefetch Abort	Address of aborting instruction	(Base LR)+4	(Base LR)-2 or (Base LR)-4
IRQ	Address of next instruction	(Base LR)+4	(Base LR)-2 or (Base LR)-4
FIQ	Address of next instruction	(Base LR)+4	(Base LR)-2 or (Base LR)-4
ThumbEE check	-	PC+4	(LR-6) or (LR-8)

a. The address of the undefined instruction, or of the `SVC`, `SMC`, or `HVC` instruction.

b. If the instruction referred to in the Base LR column is a 16-bit Thumb instruction then the address is (Base LR)-2. See the text for more information about the value for 32-bit Thumb instructions.

In the following cases, it is IMPLEMENTATION SPECIFIC whether the upgraded waypoint instruction address is (Base LR)-2 or (Base LR)-4:

- a Data Abort exception immediately after executing a 32-bit Thumb instruction
- a Prefetch Abort exception immediately after executing a 32-bit Thumb instruction
- an IRQ or FIQ exception after executing a 32-bit Thumb instruction.

However, a decompressor does not have to determine the option that is implemented. For more information see:

- Table 4-9 on page 4-178, in the section *Waypoint update packet* on page 4-177
- Table 3-2 on page 3-40, in the section *General behavior of address comparators* on page 3-39.

Note

When an asynchronous abort occurs, the ARM architecture does not guarantee that execution of the program being traced completed up to the last waypoint address output before the exception branch address packet.

Appendix A
PTM Quick Reference Information

This appendix contains quick-reference information for some key aspects of the PTM. It contains the following sections:

A.1 PTM event resources

This section contains quick-reference information about configuring the PTM event resources. It contains the following sections:

- *Resource identification and event encoding*
- *Resource control registers* on page A-222.

A.1.1 Resource identification and event encoding

A PTM event is a Boolean combination of PTM resources. An event is encoded in a 17-bit Event Register as Figure A-1 shows.

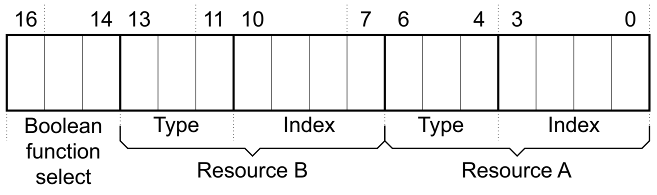


Figure A-1 Writing to an Event Register

Table A-1 lists the encodings used for resources in Event Registers.

Table A-1 Resource identification encoding

Resource type[a]	Index values[a]	Description of resource type
b000	0-15	Single address comparator.
b001	0-7	Address range comparison. Uses pairs of address comparators.
	8-11	Instrumentation resource 1-4. Software-controlled resources, see *Instrumentation resources* on page 3-58.
b010	0-7	EmbeddedICE module watchpoint comparators.
b100	0-3	Counter at zero.
b101	0-2	Sequencer in states 1-3.
	3-7	Reserved.
	8-10	Context ID comparator 1-3.
	11	VMID comparator.[b]
	12-14	Reserved.
	15	Trace start/stop resource.
b110	0-3	External inputs 1-4.
	4-7	Reserved.
	8-11	Extended external input selectors 1-4.
	12	Reserved.
	13	Processor is in Non-secure state.
	14	Trace prohibited by processor.
	15	Hard-wired resource, always TRUE.

a. The Resource type is bits [6:4] of the 7-bit resource identifier, and the Index value is bits [3:0] of the identifier. Sometimes, the combined 7-bit resource identifier is called the Resource number.

b. From PFTv1.1, and only in implementations that support Virtualization.

Table A-2 lists the encodings for the Boolean operations to be applied to the event resources.

Table A-2 Boolean function encoding for events

Encoding	Function
b000	A
b001	NOT(A)
b010	A AND B
b011	NOT(A) AND B
b100	NOT(A) AND NOT(B)
b101	A OR B
b110	NOT(A) OR B
b111	NOT(A) OR NOT(B)

Note

To permanently enable or disable an event, specify external input 16, using either function A or NOT (A).

Table A-3 lists the locations of the 17-bit Event Registers.

Table A-3 Locations of PTM event registers

Register number	Offset[a]	Register
`0x002`	`0x008`	Trigger event, ETMTRIGGER
`0x008`	`0x020`	TraceEnable event, ETMTEEVR
`0x054`	`0x150`	Counter enable event for counter 1, ETMCNTENR1
`0x055`	`0x154`	Counter enable event for counter 2, ETMCNTENR2
`0x056`	`0x158`	Counter enable event for counter 3, ETMCNTENR3
`0x057`	`0x15C`	Counter enable event for counter 4, ETMCNTENR4
`0x058`	`0x160`	Counter reload event for counter 1, ETMCNTRLDEVR1
`0x059`	`0x164`	Counter reload event for counter 2, ETMCNTRLDEVR2
`0x05A`	`0x168`	Counter reload event for counter 3, ETMCNTRLDEVR3
`0x05B`	`0x16C`	Counter reload event for counter 4, ETMCNTRLDEVR4
`0x060`	`0x180`	Event for sequencer transition from state 1 to state 2, ETMSQ12EVR
`0x061`	`0x184`	Event for sequencer transition from state 2 to state 1, ETMSQ21EVR
`0x062`	`0x188`	Event for sequencer transition from state 2 to state 3, ETMSQ23EVR
`0x063`	`0x18C`	Event for sequencer transition from state 3 to state 1, ETMSQ31EVR
`0x064`	`0x190`	Event for sequencer transition from state 3 to state 2, ETMSQ32EVR
`0x065`	`0x194`	Event for sequencer transition from state 1 to state 3, ETMSQ13EVR
`0x068`	`0x1A0`	Event for external output 1, ETMEXTOUTEVR1

Table A-3 Locations of PTM event registers (continued)

Register number	Offset[a]	Register
`0x069`	`0x1A4`	Event for external output 2, ETMEXTOUTEVR2
`0x06A`	`0x1A8`	Event for external output 3, ETMEXTOUTEVR3
`0x06B`	`0x1AC`	Event for external output 4, ETMEXTOUTEVR4
`0x07E`	`0x1F8`	Timestamp event, ETMTSEVR

a. When accessed in a memory-mapped scheme. The register offset is always (4 x (Register number)).

A.1.2 Resource control registers

This section contains register tables for PTM resource control. These are Table A-4 to Table A-18 on page A-225.

Table A-4 Trace Start/Stop Resource Control Register, `0x006`

Bits	Description
[31:16]	When a bit is set to 1, it selects a single address comparator 16 to 1 as stop addresses. For example, bit [16] set to 1 selects single address comparator 1.
[15:0]	When a bit is set to 1, it selects a single address comparator 16 to 1 as start addresses. For example, bit [0] set to 1 selects single address comparator 1.

Table A-5 TraceEnable Control Register, `0x009`

Bits	Description
[25]	Trace start/stop enable: **0** Tracing is unaffected by the trace start/stop logic. **1** Tracing is controlled by trace on and off addresses.
[24]	Include/exclude control: **0** Include. The specified resources indicate the regions in the tracing can occur. When outside this region tracing is prevented. **1** Exclude. The resources specified in bits [27:0] indicate regions to be excluded from the trace. When outside an exclude region, tracing can occur.
[23:8]	Reserved.
[7:0]	When a bit is set to 1, it selects an address range comparator 8 -1 for include/exclude control. For example, bit [0] set to 1 selects address range comparator 1.

Table A-6 FIFOFULL Level Register, `0x00B`

Bits	Access	Description
[7:0]	Read/Write	The number of bytes left in the FIFO, below which the **FIFOFULL** signal is asserted.

Table A-7 Address Comparator Value Registers, `0x010-0x01F`

Bits	Description
[31:0]	Address value

Table A-8 Address Comparator Access Type Registers, `0x020`-`0x02F`, PFTv1.0

Bits	Description	
[11:10]	Secure mode control:	
	b00	Security level ignored.
	b01	Match only if in Non-secure state.
	b10	Match only if in Secure state.
	b11	Reserved.
[9:8]	Context ID comparator control:	
	b00	Ignore Context ID comparators.
	b01	Address comparator matches only if Context ID comparator value 1 matches.
	b10	Address comparator matches only if Context ID comparator value 2 matches.
	b11	Address comparator matches only if Context ID comparator value 3 matches.
[2:0]	Access type, Read only:	
	b001	Instruction execute.

Table A-9 Address Comparator Access Type Registers, `0x020`-`0x02F`, PFTv1.1

Bits	Description	
[15]	VMID control:	
	b0	Ignore VMID
	b1	Match only if VMID matches value of ETMVMIDCVR.
[14]	Hyp mode control:	
	b0	Ignore Hyp mode
	b1	Match only if processor is operating in Hyp mode.
[13:10]	Mode control	
	For all implementations:	
	b0000	Match all modes.
	b0001	Match all Non-secure modes.
	b0100	Match all modes except Secure User.
	b0101	Match all modes except Secure Privileged.
	Implementations with Security Extensions:	
	b0010	Match all secure modes.
	b0011	No match.
	b0110	Match Secure Privileged mode only.
	b0111	Match Secure User mode only.
	b1000	Match all modes except Non-secure User.
	b1001	Match Non-secure Privileged mode only.
	b1100	Match all Privileged modes.
	b1101	Match Secure User and Non-secure Privileged modes only.
	b1010	Match all modes except Non-secure Privileged.
	b1011	Match Non-secure User mode only.
	b1111	Match all User modes.

Table A-9 Address Comparator Access Type Registers, `0x020`-`0x02F`, PFTv1.1 (continued)

Bits	Description
[9:8]	Context ID comparator control: **b00** Ignore Context ID comparators. **b01** Address comparator matches only if Context ID comparator value 1 matches. **b10** Address comparator matches only if Context ID comparator value 2 matches. **b11** Address comparator matches only if Context ID comparator value 3 matches.
[2:0]	Access type, Read only: **b001** Instruction execute.

Table A-10 Counter Reload Value Registers, `0x050`-`0x053`

Bits	Description
[15:0]	Counter reload value

Table A-11 Counter Value Registers, `0x05C`-`0x05F`

Bits	Description
[15:0]	Current counter value

Table A-12 Current Sequencer State Register, `0x067`

Bits	Description
[1:0]	Possible values are: **b00** State 1. **b01** State 2. **b10** State 3.

Table A-13 Locations of the Context ID Comparator Value Registers

Context ID Comparator value	Location
1	`0x6C`
2	`0x6D`
3	`0x6E`

Table A-14 Context ID Comparator Value Registers, `0x06C`-`0x06E`

Bits	Description
[31:0]	Context ID value

Table A-15 Context ID Comparator Mask Register, `0x06F`

Bits	Description
[31:0]	Context ID mask value

Table A-16 VMID Comparator Value Register, `0x090`, from PFTv1.1

Bits	Description
[7:0]	Virtual Machine ID.

Table A-17 Synchronization Frequency Register, `0x078`

Bits	Description
[11:0]	Synchronization frequency. Default value is 1024.

Table A-18 Extended External Input Selection Register, `0x07B`

Bits	Description
[31:24]	Fourth extended external input selector
[23:16]	Third extended external input selector
[15:8]	Second extended external input selector
[7:0]	First extended external input selector

A.2 Summary of IMPLEMENTATION DEFINED PTM features

This section lists the PTM features that are IMPLEMENTATION DEFINED. It also indicates how you can check the actual implementation of each feature.

Table A-19 lists the PTM features where it is IMPLEMENTATION DEFINED either:

- the number of times the feature is implemented
- the size of the feature.

With all of these features the minimum permitted value is 0, indicating that the feature is not supported in the PTM implementation.

Table A-19 PTM features with IMPLEMENTATION DEFINED number of instances or size

Feature	Permitted values	Value given by
Address comparators	0-8 pairs	Bits [3:0] of the Configuration Code Register.[a]
EmbeddedICE watchpoint comparators	0-8	Bits [19:16] of the Configuration Code Extension Register.[b]
Context ID comparators	0-3	Bits [25:24] of the Configuration Code Register.[a]
VMID comparators	0, 1	Bit [26] of the Configuration Code Extension Register.[b]
Counters	0-4	Bits [15:13] of the Configuration Code Register.[a]
Sequencer	0, 1	Bit [16] of the Configuration Code Register.[a]
External inputs	0-4	Bits [19:17] of the Configuration Code Register.[a]
External outputs	0-4	Bits [22:20] of the Configuration Code Register.[a]
Extended external input bus width	0-255	Bits [10:3] of the Configuration Code Extension Register.[b]
Extended external input selectors	0-4	Bits [2:0] of the Configuration Code Extension Register.[b]
Instrumentation resources	0-4	Bits [15:13] of the Configuration Code Extension Register.[b]

a. See *Configuration Code Register, ETMCCR* on page 3-78.
b. See *Configuration Code Extension Register, ETMCCER* on page 3-103.

Table A-20 lists the features that are optional in a PTM implementation. This means that it is IMPLEMENTATION DEFINED whether each of these features is supported.

Table A-20 Optional features in a PTM

Implementation of	Check for support by
FIFOFULL control	Reading bit [23] of the Configuration Code Register.[a] See also *Checking support for implementation defined features* on page 3-78.
Trace Start/Stop block	Reading bit [26] of the Configuration Code Register.[a]
Trace all branches	Testing whether you can set bit [8] of the Main Control Register to 1.[b]
Cycle-accurate trace	Writing 1 to bit [12] of the Main Control Register, see *Checking support for implementation defined features* on page 3-78.
EmbeddedICE behavior control	Reading bit [21] of the Configuration Code Extension Register.[c]
EmbeddedICE inputs to Trace Start/Stop block	Reading bit [20] of the Configuration Code Extension Register.[c]

Table A-20 Optional features in a PTM (continued)

Implementation of	Check for support by
OS Lock mechanism	Reading bit [0] of the OS Lock Status Register, see *OS Lock Status Register, ETMOSLSR* on page 3-110.
Return stack	Reading bit [23] of the Configuration Code Extension Register.[c]
Timestamping	Reading bit [22] of the Configuration Code Extension Register.[c]
Secure non-invasive debug	Reading bits [3:2] of the Authentication Status Register, see *Authentication Status Register, ETMAUTHSTATUS* on page 3-119.

a. See *Configuration Code Register, ETMCCR* on page 3-78.

b. See *Main Control Register, ETMCR* on page 3-75.

c. See *Configuration Code Extension Register, ETMCCER* on page 3-103.

Appendix B
Trace Decompressor Operation

This appendix summarizes the decompression of PTM trace output. It contains the following sections:

B.1 About PTM trace decompression

This appendix summarizes the basic operation of a decompressor that is processing the output from a PTM. It does not describe all possible features of the PTM. In particular, it does not describe the cycle-accurate trace option.

The inputs to the decompressor are:

- the trace output from the PTM, as a stream of bytes
- library calls to decode the opcode of an instruction at a given address.

The output from the decompressor is a list that contains:

- instruction objects
- event objects.

PFT output objects on page B-231 describes the instruction and event objects.

B.2 PFT trace state and objects

To reconstruct the program flow from the PTM trace stream, the debugger must maintain some PFT states. *PFT state information* defines the information it must hold for each of these states.

The output from the debugger consists of PFT objects, and *PFT output objects* defines these objects and the information they contain.

B.2.1 PFT state information

The debugger must maintain three states that relate to the execution of the program it is tracing. Two of these states consist of multiple information. The PFT states, and the information they hold, are:

LastState This is the state corresponding to the most recent explicit state information output in the trace stream. The state information consists of:

- the instruction set state of the processor
- the security state of the processor, if it implements the Security Extensions
- the Context ID
- the VMID, if the processor implements Virtualization Extensions
- an instruction address.

In the information for **LastState** and **CurrentState**, the Context ID and the VMID can have a value of *undefined*. They have this value after a reset, and when the debugger first starts to decode the trace stream.

CurrentState This is the state corresponding to the next instruction to be executed. It holds the same items of information as **LastState**.

B.2.2 PFT output objects

A PFT debugger outputs a list of objects. The possible objects are:

Instruction object

This contains the following information:

- the address of the instruction
- the instruction set state of the processor when it executed the instruction
- if the processor supports the Security Extensions, the security state of the processor when it executed the instruction
- the Context ID and the VMID when the instruction was executed
- the result of the condition code check for the instruction, pass, fail, or unknown
- information about the next instruction to be executed:
  - — the address of the instruction
  - — the instruction set state of the processor when executing that instruction
  - — if the processor supports the Security Extensions, the security state of the processor when executing that instruction.

Event object This contains information that identifies the event as one of:

- A trigger.
- An exception. In this case, the event object identifies the exception type.
- Trace turn on. In this case, the event object identifies the reason for trace turn on.
- An exception return.

B.3 PFT trace decompression flow

This section:

- describes the overall flow of trace decompression, see *Overall PFT trace decompression flow*
- gives details of the operations in the decompression flow, see *Details of PFT trace decompression operations* on page B-233.

B.3.1 Overall PFT trace decompression flow

Note

This description of the trace decompression flow refers to executing various operations. *Details of PFT trace decompression operations* on page B-233 describes each of these operations.

The operation of a PFT trace decompressor is:

1. Search for an A-sync packet, and use it to achieve packet boundary alignment.
2. Identify packets, and discard any packet that is not an I-sync packet.
3. The I-sync packet gives instruction synchronization. Process this packet:
    a. Output a trace turn on event object with the reason code from the I-sync packet.
    b. Store the address, instruction set state, security state and Context ID from the I-sync packet in **LastState**.
    c. If the reason code in the I-sync packet is periodic, and **CurrentState** is defined, check that **CurrentState** and **LastState** are identical. If these states are different, the decompressor must report an error.
    d. Copy **LastState** into **CurrentState**.
4. Decode next packet.
5. Process packet as follows:
    a. If the packet is a trigger packet, output a trigger event object and return to Stage 4.
    b. If the packet is an ignore packet, return to Stage 4.
    c. If the packet is an A-sync packet, return to Stage 4.
    d. If the packet is a branch address packet without an exception, execute `branch_no_excp()`, return to Stage 4.
    e. If the packet is a branch address packet with an exception, execute `branch_with_excp()`, return to Stage 4.
    f. If the packet is an atom packet, execute `analyze_atomheader()`, return to Stage 4.
    g. If the packet is a Context ID packet, execute `analyze_cid()`, return to Stage 4.
    h. If the packet is a VMID packet, execute `analyze_vmid()`, return to Stage 4.
    i. If the packet is a waypoint update packet, execute `analyze_waypoint_update()`, return to Stage 4.
    j. If the packet is an exception return packet, execute `analyze_eret()`, return to Stage 4.
    k. If the packet is an I-sync packet, go to Stage 3a.
    l. If no packet is received, stop decompression.

Figure B-1 on page B-233 shows this operation, but does not show how the decompressor processes the I-sync packet.

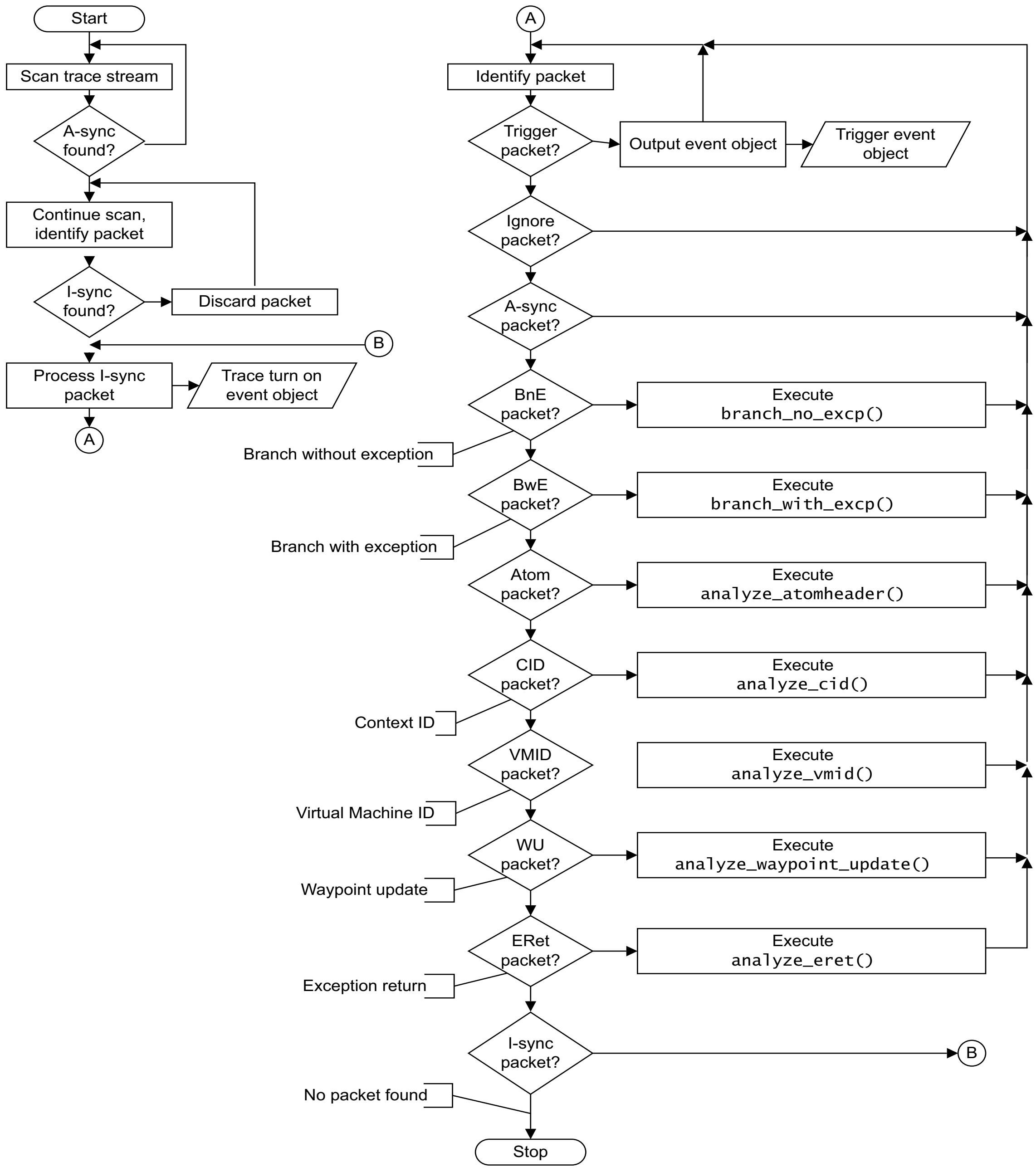


Figure B-1 Trace decompression operation

B.3.2 Details of PFT trace decompression operations

The following subsections describe the PFT trace decompression operations:

- *branch_no_excp()* on page B-234
- *branch_with_excp()* on page B-234
- *analyze_atomheader()* on page B-234
- *analyze_cid()* on page B-235
- *analyze_vmid()* on page B-235
- *analyze_waypoint_update()* on page B-235
- *analyze_eret()* on page B-235
- *decode_instr()* on page B-235.

`branch_no_excp()`

The operation of `branch_no_excp()` is:

1. Extract branch target details from the branch address packet and **LastState** and store into **CurrentState**.
2. Execute `decode_instr(CurrentState)`.
3. Output an instruction object including information from **CurrentState**.
4. Increment **CurrentState.address** by decoded instruction size.
5. If decoded instruction is not a waypoint, go back to stage 2.
6. If decoded instruction is a waypoint:
   a. Update **CurrentState** and **LastState** with branch target address and state.
   b. If return stack is enabled and decoded instruction is a branch with link, push return address, instruction set state and security state onto top of return stack.
7. Return.

`branch_with_excp()`

The operation of `branch_with_excp()` is:

1. Extract branch target details from the branch address packet and **LastState**, and exception type from the branch address packet.
2. Output an exception event object including the exception type.
3. Update **CurrentState** and **LastState** with branch target address and state.
4. Return.

`analyze_atomheader()`

The operation of `analyze_atomheader()` is:

1. For each atom:
   a. Execute `decode_instr(CurrentState)`.
   b. If decoded instruction is not a waypoint:
      - Output an instruction object including information from **CurrentState**. In this object, the result of the condition code check is unknown.
      - Increment **CurrentState.address** by decoded instruction size.
      - Go back to step 1.a.
   c. If decoded instruction is a waypoint:
      - Output an instruction object including information from **CurrentState** and the atom value.
      - If atom is an E atom:
        — Update **CurrentState** with decoded instruction target address and state
        — If return stack is enabled and decoded instruction is an indirect branch, pop address, instruction set state and security state off the top of the return stack and update **CurrentState** with these values.
        — If return stack is enabled and decoded instruction is a branch with link, push return address, instruction set state and security state onto the top of the return stack.
      - If atom is an N atom:
        — Increment **CurrentState.address** by decoded instruction size.
2. Return.

`analyze_cid()`

The operation of `analyze_cid()` is:

1. Update **LastState** and **CurrentState** with the contents of the Context ID packet.
2. Return.

`analyze_vmid()`

The operation of `analyze_vmid()` is:

1. Update **LastState** and **CurrentState** with the contents of the Virtual Machine ID packet.
2. Return.

`analyze_waypoint_update()`

The operation of `analyze_waypoint_update()` is:

1. Determine waypoint address from address part of packet and **LastState.address**.
2. Execute `decode_instr(CurrentState)`.
3. Output an instruction object including information from **CurrentState**. In this object, the result of the condition code check is unknown.
4. If (**CurrentState.address** + decoded instruction size) > (waypoint address):
   - Return.
5. If the decoded instruction is not a waypoint:
   a. Increment **CurrentState.address** by decoded instruction size.
   b. Go back to Step 2.
6. Return.

`analyze_eret()`

The operation of `analyze_eret()` is:

1. Output an exception return event object.
2. Return.

`decode_instr()`

This routine acquires the instruction opcode for the decompressed instruction address and instruction set. The opcode is then decoded according to the rules of the instruction set to obtain the following attributes:

- Is the instruction a waypoint instruction?
- If it is a waypoint instruction, is it:
  - — A direct branch instruction?
  - — An indirect branch instruction?
- If it is a waypoint instruction, is it a branch with link instruction?
- If it is a direct branch, what is the branch target address and instruction set state?
- What is the size, in bytes, of the instruction? The decompressor uses this value to calculate the next instruction address. It also uses it to calculate the target address of a branch instruction that failed its condition code test, and the return address of a branch with link instruction.

Appendix C
Software Issues for PFT

This appendix describes software issues relating to PFT. It contains the following sections:

C.1 About tracing dynamically-loaded code

When a debugger is debugging a system, it communicates mainly in terms of accesses to addresses in memory or virtual memory. It translates between these addresses and the locations in the code images loaded on the system. This means that the debugger can present a symbolic or source-level view of the code running on the system.

In a simple statically-linked and loaded system, a single image is run to describe the mapping of target addresses as image locations. To perform debugging, the debugger requires only the name of the code image. However, many systems, including operating systems such as Windows® CE, Linux®, or Symbian OS™, load part or all of their software dynamically. This can have several effects:

- the address at the an image is loaded might not be known until it is loaded
- at different times, different images might be loaded at the same address
- in a complex system, the debugger might not know what images are candidates to be loaded until they are loaded.

To debug systems like these, the debugger must be able to examine the target, to determine what images are loaded and from where they are loaded.

The problem is more complex when using trace, because trace data is historical information. In any embedded trace solution, the trace decompression software of the debugger requires an image of the code that was executed, otherwise it cannot decode the trace.

The PFT protocol conserves data bandwidth by generating only the minimum of trace information, and compresses the instruction addresses it outputs. This means that, given a compressed address issued by the trace port, the debugger software must be able to know what instructions are at and around that point. This enables the debugger to infer the target address of direct branches, for example `B` and `BL` instructions in code in ARM state. This is difficult with, for example, virtual memory and software paging, because the debugger is unlikely to know where the code is executed from.

To resolve this problem, PTM uses Context IDs. These require both software and hardware support. See:

- *Software support for Context ID* on page C-241
- *Hardware support for Context ID* on page C-242.

C.1.1 Simple overlay support

You can implement a system for supporting simple overlays, that does not require specific support in the debugger. This solution is based on the requirement that the memory space into which the overlays are loaded exists in multiple places in the memory map as Figure C-1 on page C-239 shows. That is, some of the unused address bits are *don't care* when determining the memory to be accessed.

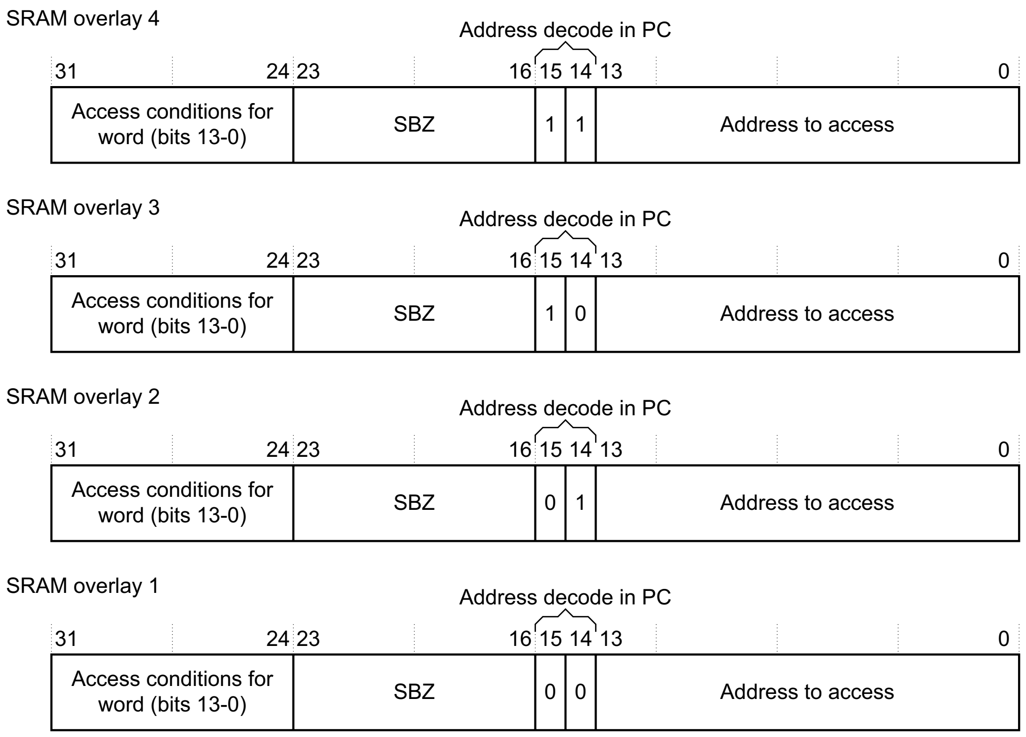


Figure C-1 SRAM overlay examples

For example, if you have 16KB of SRAM, bits [13:0] of the address determine the 32-bit word to access and bits [31:24] determine when to access that particular block. However, if bits [15:14] are in the address decoder, four copies of the memory block exist in the memory map. In other words the same word can be accessed using four different addresses, that is, when bits [15:14] of the address are b00, b01, b10, or b11. See Figure C-2 on page C-240.

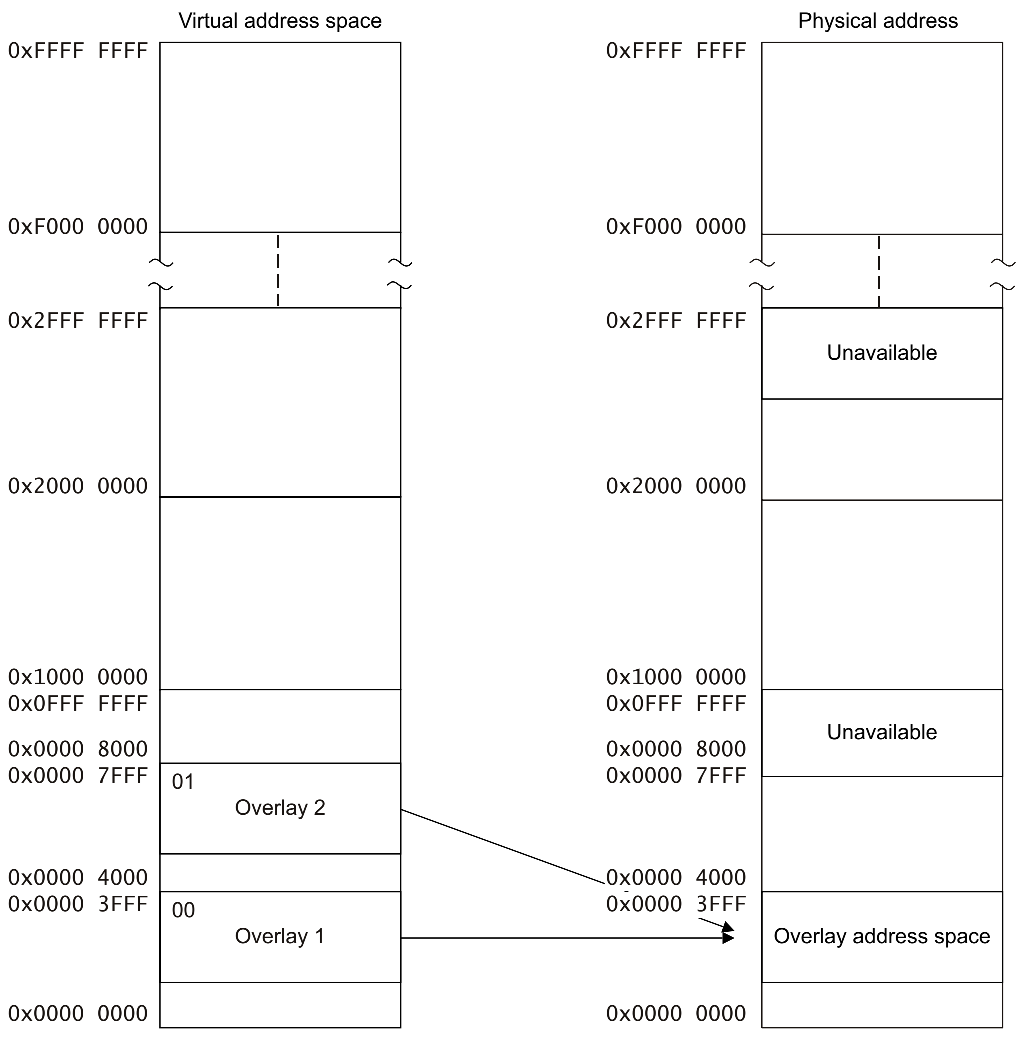


Figure C-2 Memory map and overlay physical address space

C.2 Software support for Context ID

When the operating system switches between binary images, or virtual memory spaces, it must update the value in the Context ID Register provided by CP15. The debugger must have access to a mapping file specifying the Context ID that correlates to each binary image. With this information, the debugger can then associate each binary image with the correct part of the trace.

C.3 Hardware support for Context ID

The PTM outputs a variable-length Context ID value whenever trace is enabled, and as part of the periodic synchronization packet. This enables the debugger to determine the current Context ID value. You can also filter out unwanted trace based on the current Context ID using programmable trigger resources.

To support tracing when only a partial binary image is available, the PFT protocol maintains synchronization even when the PTM branches into unknown code regions. When the code jumps back into a region for which the code image is available the debugger can restart decompression immediately.

Appendix D
Architecture Version Information

This appendix describes the major architecture changes for the PFT. It contains the following sections:

D.1 PFTv1.0 to PFTv1.1

The changes implemented between PFTv1.0 and PFTv1.1 are described in:

- *Programmers model*
- *Signal protocol* on page D-245.

D.1.1 Programmers model

Changes to the programmers model are:

- Added the option for a Reduced Function Counter. See *Reduced function counter, from PFTv1.1* on page 3-91.
- Added a field to the ETMCR to enable VMID tracing. See *Main Control Register, ETMCR* on page 3-75.
- Added fields to the ETMCCER to indicate if Virtualization is supported and if the Reduced Function Counter is supported. See *Configuration Code Extension Register, ETMCCER* on page 3-103.
- Added ETMVMIDCVR to enable comparisons against the current VMID. See *VMID Comparator Value Register, ETMVMIDCVR* on page 3-109.
- Added additional controls to ETMACTR to enable comparisons against the current VMID and mode/state. See *Address Comparator Access Type Registers, ETMACTRn* on page 3-87.
- Removed access to the following registers from coprocessor accesses:
  - — ETMLAR. See *Lock Access Register, ETMLAR* on page 3-118.
  - — ETMLSR. See *Lock Status Register, ETMLSR* on page 3-119.
  - — ETMDEVTYPE. See *Device Type Register, ETMDEVTYPE* on page 3-122.
  - — ETMPIDR0-7. See *Peripheral ID0 Register, ETMPIDR0* on page 3-124 to *Peripheral ID5 to Peripheral ID7 Registers, ETMPIDR5 to ETMPIDR7* on page 3-128.
  - — ETMCIDR0-3. See *Component ID0 Register, ETMCIDR0* on page 3-129 to *Component ID3 Register, ETMCIDR3* on page 3-131.
- The read value of event registers, such as ETMTRIGGER, is UNPREDICTABLE if an invalid resource is programmed. See *Read values of event registers* on page 3-53.
- Clarified the effect of DBGSWENABLE on PTM register accesses. See *Effect of DBGSWENABLE on register access* on page 3-139.
- Significant changes to power down support are introduced in PFTv1.1. See *Power-down support* on page 3-132 to *Access permissions for PFTv1.1 with multiple power implementations* on page 3-149.
  - — Added ETMPDCR to enable control over power and access to Trace registers. See *Device Power-Down Control Register, ETMPDCR* on page 3-113.
  - — Added the LK field to the ETMPDSR to indicate the current status of the OS Lock. See *Device Power-Down Status Register, ETMPDSR* on page 3-114.
  - — Changed the access permissions when the ETMPDSR.StickyState field is set. See *Device Power-Down Status Register, ETMPDSR* on page 3-114.
  - — If implemented, the OS Lock must be set from a PTM reset. See *Full Power Down Support from PFTv1.1* on page 3-135.
  - — Changed the access permissions so that the OS Lock only has an effect on external debugger accesses. See *PTM behavior when the OS Lock is set* on page 3-136.
  - — Changed the Claim registers to be Trace registers. See *PTM trace and PTM management registers* on page 3-66.
  - — Added bit [3] to the ETMOSLSR to indicate the type of OS Lock implemented. See *OS Lock Status Register, ETMOSLSR* on page 3-110.

D.1.2 Signal protocol

Changes to the signal protocol are:

- Added support for tracing of the Virtualization extensions:
  - Tracing of the current VMID.
  - Tracing whether the processor is executing in Hyp mode.
  - Tracing of entry to and exit from Hyp mode.
  - How Virtualization affects tracing on entry to a prohibited region.

  See *Branch address packet* on page 4-166, *VMID packets* on page 4-181, and *Instruction synchronization* on page 4-192.
- Removed the requirement for cycle accuracy to be maintained through a trace overflow. See *Cycle-accurate tracing* on page 4-160.
- Changed how a conditional `ISB` instruction might be traced. See *PFT atoms* on page 4-157.
- Tracing of exception return for ARMv7-A and ARMv7-R processors. See *Exception return packet* on page 4-183.
- Clarification of how entry to Debug state is traced. See *Debug state* on page 4-202.
- Changes to timestamping:
  - Additional rules for generation of timestamps.
  - Changes to cycle accuracy with timestamps.
  - Added support for timestamp values up to 64 bits.
  - Added support for natural binary encoded timestamps as well as Gray coded timestamps.

  See *Timestamp packet* on page 4-181.

Glossary

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

Abort	A mechanism that indicates to a processor that the value associated with a memory access is invalid. An abort can be caused by the external or internal memory system as a result of attempting to access invalid instruction or data memory. An abort is classified as either a Prefetch Abort or a Data Abort, and as an internal abort or an External Abort. *See also* Data Abort, External Abort and Prefetch Abort.
A-sync	*See* Alignment synchronization.
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.
Alignment synchronization (A-sync) packet	A sequence of bytes that enables the decompressor to byte-align the trace stream and determine the location of the next header.
ARM instruction	A word that specifies an operation for an ARM processor in ARM state to perform. ARM instructions are word-aligned. *See also* ARM state, Thumb instruction, ThumbEE instruction.
ARM state	An operating state of the processor, in the it executes 32-bit ARM instructions. *See also* ARM instruction, Jazelle RCT, Jazelle architecture, Thumb state, ThumbEE state.

Branch prediction

The process of predicting if conditional branches are to be taken or not in pipelined processors. Successfully predicting if branches are to be taken enables the processor to prefetch the instructions following a branch before the condition is fully resolved. Branch prediction can be done in software or by using custom hardware. Branch prediction techniques are categorized as static, in which the prediction decision is decided before run time, and dynamic, in which the prediction decision can change during program execution.

Breakpoint

A mechanism provided by debuggers to identify an instruction at the program execution is to be halted. Breakpoints are inserted the programmer to enable inspection of register contents, memory locations, and/or variable values at fixed points in the program execution to test that the program is operating correctly. Breakpoints are removed after the program is successfully tested.

See also Watchpoint.

Context

The environment that each process operates in for a multitasking operating system. In ARM processors, this is limited to mean the physical address range that it can access in memory and the associated memory access permissions.

See also Fast context switch.

Context ID

A 32-bit value accessed through CP15 register 13 that is used to identify and differentiate between different code streams.

CoreSight

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

Data Abort

An indication from a memory system to the processor of an attempt to access an illegal data memory location. An exception must be taken if the processor attempts to use the data that caused the abort.

See also Abort, External Abort, and Prefetch Abort.

Debugger

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

An application that monitors and controls the operation of a second application. Usually used to find errors in the application program flow.

EmbeddedICE logic

An on-chip logic block that provides debug support for ARM processors.

Embedded Trace Buffer (ETB)

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

ETB

See Embedded Trace Buffer.

Event

A boolean combination of PTM resources that is used by a PTM to control aspects of a trace.

Event resource

A configurable PTM resource such as an address comparator or a counter. Used to define *events*.

Exception

A fault or error event that is considered serious enough to require that program execution is interrupted. Examples include attempting to perform an invalid memory access, external interrupts, and undefined instructions. When an exception occurs, normal program flow is interrupted and execution is resumed at the corresponding exception vector. This contains the first instruction of the interrupt handler to deal with the exception.

Exception vectors

A number of fixed addresses in low memory, or in high memory if high vectors are configured, each of the contains the first instruction of the corresponding interrupt handler.

External Abort

An indication from an external memory system to a processor that the value associated with a memory access is invalid. An External Abort is caused by the external memory system as a result of attempting to access invalid memory.

See also Abort, Data Abort and Prefetch Abort.

I-sync *See* Instruction synchronization.

IMPLEMENTATION DEFINED

The behavior is not architecturally defined, but must be defined and documented by individual implementations.

IMPLEMENTATION SPECIFIC

The exact 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.

Imprecise Tracing

A filtering configuration where instruction or data tracing can start or finish earlier or later than expected. Most cases cause tracing to start or finish later than expected.

For example, if **TraceEnable** is configured to use a counter so that tracing begins after the fourth write to a location in memory, the instruction that caused the fourth write is not traced, although subsequent instructions are. This is because the use of a counter in the **TraceEnable** configuration always results in imprecise tracing.

See the descriptions of **TraceEnable** in Chapter 3 *Program Trace Macrocell Programmers Model*.

Instruction synchronization (I-sync)

Full output of the current instruction address and Context ID on the later trace is based.

Interrupt vector

See Exception vectors.

Jazelle architecture

The ARM Jazelle architecture extends the Thumb and ARM operating states by adding a Jazelle state to the processor. Instruction set support for entering and exiting Java applications, real-time interrupt handling, and debug support for applications that mix Java and ARM code is present.

When in Jazelle state, the processor fetches and decodes Java bytecodes and maintains the Jazelle operand stack.

See also ARM state, Jazelle RCT, Thumb state, ThumbEE state.

Jazelle RCT (Jazelle Runtime Compiler Target)

An extension to the ARM architecture targeting execution environments, such as Java or .NET Compact Framework. Jazelle RCT provides enhanced support for Ahead-Of-Time (AOT) and Just-In-Time (JIT) compilation. It extends the Thumb instruction set, and introduces a new processor state, ThumbEE.

See also ARM state, Jazelle architecture, Thumb state, ThumbEE state.

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.

Macrocell

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

Match Resources match for one or more cycles when the condition they have been programmed to check for occurs.

Packet A number of bytes of related data, consisting of a header byte and zero or more payload bytes.

Packet header

The first byte of a PTM *packet*. The header byte specifies the packet type and how to interpret the following bytes in the packet.

Prefetch Abort

An indication from a memory system to the processor that an instruction has been fetched from an illegal memory location. An exception must be taken if the processor attempts to execute the instruction. A Prefetch Abort can be caused by the external or internal memory system as a result of attempting to access invalid instruction memory.

See also Data Abort, External Abort and Abort.

Program Trace Macrocell (PTM)

A hardware macrocell that, when connected to a processor, outputs instruction trace information.

Prohibited region

A period of processor execution during the tracing is not permitted, for example because the processor is in Secure state.

PTM *See* Program Trace Macrocell.

RAZ *See* Read-As-Zero fields.

Read-As-Zero fields (RAZ)

Appear as zero when read.

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 zero and are Read-As-Zero.

Serial Wire Debug (SWD)

A debug implementation that uses a serial connection between the SoC and a debugger. This connection normally requires a bidirectional data signal and a separate clock signal, rather than the four to six signals required for a JTAG connection.

TAP *See* Test Access Port.

TCD *See* Trace Capture Device.

Test Access Port (TAP)

The collection of four mandatory connections and one optional connection 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**.

Thumb instruction

One or two halfwords that specify an operation for an ARM processor in Thumb state to perform. Thumb instructions must be halfword-aligned. In the original Thumb instruction set, all instructions are 16-bit. Thumb-2 technology, introduced in ARMv6T2, extends the original Thumb instruction set with many 32-bit instructions.

See also ARM instruction, Thumb state, ThumbEE instruction.

Thumb state

An operating state of the processor, in the it executes 16-bit and 32-bit Thumb instructions.

See also ARM state, Thumb instruction, ThumbEE state, Jazelle architecture.

ThumbEE instruction

One or two halfwords that specify an operation for an ARM processor in ThumbEE state to perform. ThumbEE instructions must be halfword-aligned.

ThumbEE is a variant of the Thumb instruction set that is designed as a target for dynamically generated code, that is, code compiled on the device either shortly before or during execution from a portable bytecode or other intermediate or native representation.

See also ARM instruction, Thumb instruction, ThumbEE state.

ThumbEE state

An operating state of the processor, in the it executes 16-bit and 32-bit ThumbEE instructions.

See also ARM state, Thumb state, ThumbEE instruction, Jazelle architecture, Jazelle RCT.

TPA *See* Trace Port Analyzer.

Trace Capture Device (TCD)

A generic term for Trace Port Analyzers, logic analyzers, and Embedded Trace Buffers.

Trace port

A port on a device, such as a processor or ASIC, that is 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.

UNDEFINED

Indicates an instruction that generates an Undefined Instruction exception.

UNKNOWN

An UNKNOWN value does not contain valid data, and can vary from moment to moment, instruction to instruction, and implementation to implementation. An UNKNOWN value must not be a security hole. UNKNOWN values must not be documented or promoted as having a defined value or effect.

UNPREDICTABLE

Means that the behavior of the PTM cannot be relied on. Such conditions have not been validated. When applied to the programming of an event resource, the only effect of the UNPREDICTABLE behavior is that the output of that event resource is UNKNOWN.

UNPREDICTABLE behavior can affect the behavior of the entire system, because the PTM can cause the processor to enter Debug state, and external outputs can be used for other purposes.

Virtual address

Is an address generated by an ARM processor. For processors that implement a *Protected Memory System Architecture* (PMSA), the virtual address is identical to the physical address.

Watchpoint

A watchpoint is a mechanism provided by debuggers to halt program execution when the data contained by a particular memory address is changed. Watchpoints are inserted by the programmer to enable inspection of register contents, memory locations, and variable values when memory is written, to test that the program is operating correctly. Watchpoints are removed after the program is successfully tested.

See also Breakpoint.