Application Note

Armv8.1-M Performance Monitoring User Guide

Version 1.186
Document ID: ARM051-799564642-251
Non-Confidential-Published

Version 1.186
Document ID: ARM051-799564642-251

Release Information

Document History

Issue	Date	Confidentiality	Change
A	**20/07/2020**	**Non-Confidential**	**First Release**

Contents

1 About this document

1.1. References

Reference	Document number	Author(s)	Title
[1]	ARM DDI 0553B	Arm	Arm®v8-M Architecture Reference Manual
[2]	ARM DDI 0487F.a	Arm	Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile
[3]	ARM DDI 0568A.b	Arm	ARM Architecture Reference Manual Supplement ARMv8, for the ARMv8-R AArch32 architecture profile
[4]	ARM IHI 0022G	Arm	AMBA® AXI and ACE Protocol specification
[5]	ARM IHI 0033B	Arm	Arm® AMBA® 5 AHB Protocol specification
[6]	N/A	Arm	Whitepaper: Introduction to the Armv8.1-M Architecture
[7]	101051	Arm	Arm® Cortex-M55 Processor Technical Reference Manual (TRM)
[8]	101052	Arm	Arm® Cortex-M55 Processor Integration and Implementation Manual (IIM)
[9]		Arm	Arm® Cortex-M55 Processor Release Notes
[10]	ARM DDI 0314H	Arm	CoreSight™ Components Technical Reference Manual

1.2. Terms and Abbreviations

This document uses the following terms and abbreviations.

Term	Meaning
AHB	Advanced High-performance Bus
API	Application Programming Interface
AXI	Master Advanced eXtensible Interface
CTI	Cross Trigger Interface
CMSIS	Cortex Microcontroller Software Interface Standard
CPU	Central Processing Unit
DAP	Debug Access Port
DFP	Device Family Pack
DS	(Arm) Development Studio
DSP	Digital Signal Processing
DTCM	Data TCM interface
DWT	Data Watchpoint and Trace Unit
ECC	Error Correcting Code
EDA	Embedded Design Automation
ETM	Embedded Trace Macrocell
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit

IIM	Integration and Implementation Manual
ITCM	Instruction TCM interface
ITM	Instrumentation Trace Macrocell
LOB	Low Overhead Branch
M-AXI	Master AXI interface
MCU	Microcontroller Unit
MDK	(Keil) Microcontroller Development Kit
MPS	Cortex-M Prototyping System
MVE	M-Profile Vector Extension
NPU	Neural Processing Unit
NVIC	Nested Vectored Interrupt Controller
OS	Operating System
P-AHB	Peripheral AHB interface
PE	Processing Element
PPB	Private Peripheral Bus
PMU	Performance Monitoring Unit
RTL	Register Transfer Level
S-AHB	Slave AHB interface
SoC	System-on-Chip
SWV	Serial Wire Viewer
TCM	Tightly Coupled Memory
TRM	Technical Reference Manual

1.3. Scope

This document describes how to use the PMU as defined by the Armv8.1-M Architecture.

Conventions and Feedback

The following describes the typographical conventions and how to give feedback:

Convention	Meaning
`monospace`	denotes text that can be entered at the keyboard, such as commands, file and program names, and source code.
`monospace`	denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.
`monospace italic`	denotes arguments to commands and functions where the argument is to be replaced by a specific value.
`monospace bold`	denotes language keywords when used outside example code.
italic	highlights important notes, introduces special terminology, denotes internal cross-references, and citations.

bold	highlights interface elements, such as menu names. Also used for emphasis in descriptive lists, where appropriate, and for Arm® processor signal names.

Feedback on this product

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

Your name and company.

The serial number of the product.

Details of the release you are using.

Details of the platform you are using, such as the hardware platform, operating system type and version.

A small standalone sample of code that reproduces the problem.

A clear explanation of what you expected to happen, and what actually happened.

The commands you used, including any command-line options.

Sample output illustrating the problem.

The version string of the tools, including the version number and build numbers.

Feedback on documentation

If you have comments on the documentation, e-mail errata@arm.com. Give:

The title.

The number, [Document ID Value], [Issue].

If viewing online, the topic names to which your comments apply.

If viewing a PDF version of a document, the page numbers to which your comments apply.

A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Arm periodically provides updates and corrections to its documentation on the Arm Information Center, together with knowledge articles and *Frequently Asked Questions* (FAQs).

Other information

Arm Developer, https://developer.arm.com

Arm Documentation, https://developer.arm.com/docs

Arm Support and Maintenance, http://www.arm.com/support/services/support-maintenance.php

Arm Glossary, https://developer.arm.com/support/arm-glossary

2 Introduction

2.1. Profiling Overview

There are several reasons why a user might want to profile their application. These reasons include:

- Identifying critical sections of code and bottlenecks
- Profile-guided optimization
- Call graph / call tree information
- Code coverage analysis
- Instrumentation trace capture
- Counting specific events
- Measuring exact cycle timing
- Benchmarking
- Verification, validation, and testing
- Checking how well different blocks of IP, such as caches, are being utilized
- Measuring latency in the system

The M-profile architecture provides different features to help users carry out such tasks, including the *Performance Monitoring Unit* (PMU), introduced in the Mainline variant of the Armv8.1-M Architecture. This application note demonstrates a variety of use cases for the PMU, as described in section 7 *Using the PMU in your application* and section 8 *PMU Profiling Example*.

2.2. Profiling Armv8-M systems

Prior to the Armv8.1-M PMU, the M-profile architecture provided the following profiling features:

- *Data Watchpoint and Trace* (DWT) profiling
- *Instrumentation Trace Macrocell* (ITM) profiling
- *Embedded Trace Macrocell* (ETM) profiling
- SysTick timer

2.2.1 DWT profiling

You can configure any implementation of the Armv8.0-M Mainline architecture, for example, the Cortex-M33 processor (or an implementation of the Armv7-M architecture, for example, the Cortex-M7 processor), with a basic set of profiling counters. Provided that both the DWT and the ITM are configured in such an implementation, the DWT will provide a Cycle Count Register and five performance profiling counters (in addition to its watchpoint functionality):

- Cycle Count Register (DWT_CYCCNT) [1]
- CPI Count Register (DWT_CPICNT)[2]
- Exception Overhead Count Register (DWT_EXCCNT)
- Sleep Count Register (DWT_SLEEPCNT)

[1] PMU_CYCCNT is an alias of DWT_CYCCNT.

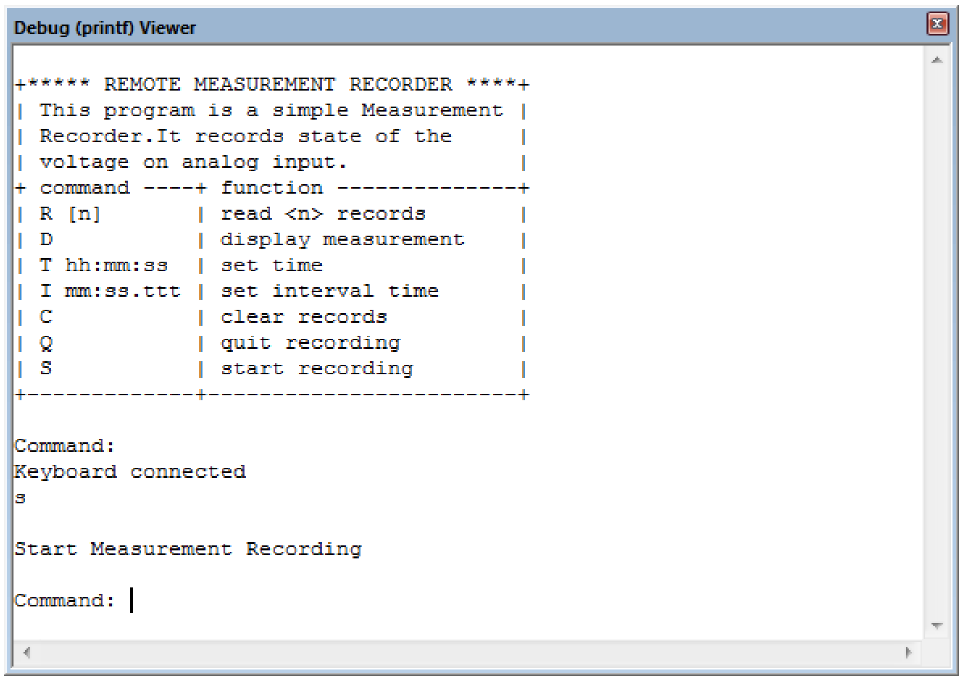
[2] Counts additional cycles required to execute multicycle instructions and instruction fetch stalls.

- LSU Count Register (DWT_LSUCNT)[3]
- Folded Instruction Count Register (DWT_FOLDCNT)

Debug tools and the application itself can use these counters for profiling.

2.2.2 ITM profiling

The ITM provides a mechanism for the application to carry out instrumentation trace or 'printf debugging'. Software can write directly to ITM stimulus registers to generate trace packets. These packets can also include timestamp information.



```
+***** REMOTE MEASUREMENT RECORDER ****+
| This program is a simple Measurement |
| Recorder.It records state of the     |
| voltage on analog input.             |
+ command ----+ function --------------+
| R [n]       | read <n> records       |
| D           | display measurement    |
| T hh:mm:ss  | set time               |
| I mm:ss.ttt | set interval time      |
| C           | clear records          |
| Q           | quit recording         |
| S           | start recording        |
+-------------+------------------------+

Command:
Keyboard connected
s

Start Measurement Recording

Command:
```

Figure 2-1 - Keil MDK Debug (printf) Viewer

The ITM requires a debugger to be connected to the system to retrieve and decode the information contained in the trace packets. As described in the *CoreSight Components Technical Reference Manual*, the ITM and *Serial Wire Output* (SWO) can be used to form a *Serial Wire Viewer* (SWV). Debug tools are capable of displaying instrumentation trace data packets transmitted via a SWV, like in Figure 2-1.

Although this type of instrumentation profiling or trace capture is not very intrusive compared with halting the system, it does require some additional code to be added to the program, which could affect timing and measurements.

2.2.3 ETM profiling

An Armv8.0-M implementation with the Main Extension (or an Armv7-M implementation) can optionally include the ETM. The ETM provides either:

- Instruction trace only.
- Instruction and data trace.

[3] Increments on the additional cycles required to execute all load or store instructions.

Similar to the ITM, the ETM generates trace packets with timestamp information for different operations that have executed on the M-profile target system. Debug tools that support the ETM can retrieve, decode, and display the execution history of the application. When debug information is present in the application, a debug tool can deduce further profiling information such as thread or function execution time statistics, and code coverage. The Keil MDK Trace Data window is an example of such a debug tool, which is described in more detail on the Keil website in the *µVision User's Guide*:

`http://www.keil.com/support/man/docs/uv4/uv4_db_dbg_tracedata.htm`

2.2.4 SysTick timer

The System Timer Extension can be implemented in any M-profile implementation. The SysTick timer can be used in different ways. A common use case is for SysTick to act as the heartbeat for an operating system so it can carry out tasks like regularly switching threads. SysTick can also be used as a generic timer. Another SysTick use case is profiling code; its regular tick can be based on the processor core's clock frequency or an external reference clock, and its decrementing counter that can be read by software.

2.3. The PMU Profiling Feature introduced in the Armv8.1-M Architecture

In addition to existing M-profile profiling features described in section 2.2 *Profiling Armv8-M systems*, Armv8.1-M provides a Performance Monitoring Extension that permits Armv8.1-M implementations with the Main Extension, like the Cortex-M55 processor, to be configured with a *Performance Monitoring Unit* (PMU). The Armv8.1-M PMU provides a rich set of profiling resources and behaves in a similar way to the PMU feature available in other architecture profiles such as Armv8-A.

As described in the Whitepaper, *Introduction to the Armv8.1-M Architecture*, the Armv8.1-M architecture is an enhancement to the original Armv8-M architecture and brings many additional features, including an *M-Profile Vector Extension* (MVE) for signal processing and machine learning applications, also known as Helium. New features like MVE, the *Low Overhead Branch* (LOB) Extension, and half-precision floating-point instructions, can significantly improve the performance of applications running on an Armv8.1-M implementation such as the Cortex-M55 processor.

An Armv8.1-M PMU includes counters for counting cycles and a wide range of other events while an application is running on the target platform. The PMU counters can be read by software at runtime or when the processor is being debugged. These counters are a useful and convenient resource for measuring the performance of an M-profile system, including the M-profile features added in Armv8.1-M.

Note
An implementation that does not include the Main Extension, does not support the Performance Monitoring Extension.

2.3.1 Cycle counter and event counters

The PMU provides two types of counters:

- A 32-bit Cycle Counter Register that is hard-wired to count CPU cycles. This register is an alias of the Cycle Counter Register in the DWT (DWT_CYCCNT) and is always present in an Armv8.1-M implementation, like Cortex-M55, configured with the PMU.
- Up to 31 Event Counters. A minimum number of two Event Counters is permitted for an implementation that includes the PMU. The Cortex-M55 processor includes eight Event Counters.

2.3.2 Counting cycles

There are two approaches for counting cycles with the PMU:

- Use the dedicated 32-bit Cycle Counter Register (CYCCNT).
- Use a 16-bit Event Counter to count an event related to cycles. One such event is CPU_CYCLES, which like CYCCNT, increments every CPU cycle. There are also other more specific events relating to cycle counting. For example, there are different events that can be used to count certain stalls in the processor pipeline. Additionally, there is an event that counts bus cycles. It is also possible to form a 32-bit cycle counter by chaining 16-bit counters together (see section 7.6 *Chaining Event Counters to create a 32-bit Counter*).

2.3.3 Counting events

Any 16-bit Event Counter can be used to count various general-purpose events. These events are described in more detail in section 4 *Armv8.1-M PMU Programmers' Model*.

2.4 Performance Monitoring in other Arm Systems

Performance monitoring and profiling features are also supported in other Arm systems. This section provides a brief description of these systems and their performance monitoring features, and how they might interact with an M-profile system.

2.4.1 Performance monitoring in A-profile and R-profile systems

Performance monitoring was originally added as a new feature to the Arm architecture in Armv7-A/R under the Performance Monitors Extension. The most recent *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition* provides a specification for PMUv1 and PMUv2.

The *Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile* describes the latest version of the PMU specification, PMUv3, and further information about the PMU relevant to Armv8-R architecture can be found in the *ARM Architecture Reference Manual Supplement ARMv8, for the ARMv8-R AArch32 architecture profile*.

Some of the Armv8-M Performance Monitoring Extension is based on the A-Profile and R-Profile PMU specifications, but the Armv8-M Performance Monitoring Extension specification is a standalone specification that does not belong to the PMUv3 specification.

M-profile implementations might be used in a larger SoC that includes A-profile or R-profile implementations. In such implementations, separate programming APIs will need to be used for programming the respective PMUs.

The Arm Streamline Performance Analyzer tool can be programmed to generate a series of charts for visualizing information generated by a A-profile and R-profile PMU counters. These charts can help with various aspects of Cortex-A and Cortex-R system profiling such as checking the effectiveness of the caches and identifying how well the system bus is being utilized. This assistance helps programmers to profile their code quicker and to get the best performance out of the Arm processors they are working with.

2.4.2 Performance monitoring in Arm Neural Processing Units

Arm Neural Processing Units (NPUs) can be combined with one or more other Arm processors specifically to target accelerating *Machine Learning* (ML) code. For example, the Ethos-U55 NPU can be combined with the Cortex-M55 processor to accelerate ML code running within area-constrained embedded and IoT devices.

Ethos-U55 has its own self-contained PMU with a maximum of four 32-bit event counters and one 48-bit cycle counter. This is a separate IP block and has no interaction with an Armv8.1-M implementation's PMU.

There is a separate API for programming the Ethos-U55 PMU, which works in a similar way to the CMSIS-Core API for Armv8.1-M and Cortex-M55, however, the list of PMU events is very different.

Configuring the Ethos-U55 PMU registers can be achieved from software running on Armv8.1-M implementation, or from a host computer using Arm Mbed DAPLink or the Arm Streamline Performance Analyzer tool.

A typical use case for the Ethos-U55 PMU would be to measure the performance related to the AXI bus interface. For example, a user might clear the PMU counters immediately before a network operation begins, and then read them again after the execution has finished. The counters would be used to measure the performance and detected potential bottlenecks.

2.4.3 Performance monitoring in Arm Graphics and Multimedia Processors

Arm Mali GPUs implement a comprehensive range of performance counters for closely monitoring GPU activity whilst an application runs.

The Arm Streamline Performance Analyzer tool can be programmed to generate a series of charts for visualizing information generated by a GPU's performance counters. These charts can help with various aspects of GPU profiling such as identifying the cause of heavy rendering loads or workload inefficiencies. This assistance helps GPU programmers to profile their code quicker and to get the best performance out of the graphics processor they are working with.

3 Tools Support for the Armv8.1-M PMU

3.1 PMU Tools Support Overview

There are two ways to use the PMU as a developer:

- Programming the PMU in C code using an Arm C compiler.
- Accessing the PMU using an Arm debugger.

Arm also offers several different Armv8.1-M and Cortex-M55 platforms and simulation models that support the PMU.

3.2 Hardware Platform and Simulation Model Support for the PMU

As of August 2020, the following platforms and simulation models support the Armv8.1-M PMU.

Platform / Simulation Model	Notes / URL

Fast Models and Fixed Virtual Platforms (FVPs)	An FVP is a virtual development platform built with Arm Fast Models for software development without a physical board.
	An FVP can be used standalone from a command-line interface.
	Some FVPs are packaged as part of software development tools like Arm Development Studio and Keil MDK. These toolkits provide connection dialogs to allow the user to connect to the FVP through an IDE.
	The Fast Models tool provides an environment to design and create custom virtual platforms, like FVPs, for early software development.
	The Fast Models tool provides different types of ready-made M-profile Fast Models that support the PMU:
	• Armv8.1-M Architecture Fast Model available in Fast Models 11.6 and later
	• Cortex-M55 CPU Fast Model available in Fast Models 11.10 and later.
	`https://developer.arm.com/tools-and-software/simulation-models/fast-models`
	There is also a Corstone-300 Ecosystem FVP available, which includes support for the PMU:
	`https://developer.arm.com/tools-and-software/open-source-software/arm-platforms-software/arm-ecosystem-fvps`
	Note
	FVPs support a limited number of PMU events and are not cycle accurate.

Cycle Models	Cycle Models are 100% cycle accurate models of Arm IP for performance analysis and IP selection.
	A Cortex-M55 Cycle Model is available at the Arm IP Exchange website:
	`https://developer.arm.com/tools-and-software/simulation-models/cycle-models`
	`https://ipx.arm.com/models?type=Cortex-M55`

RTL Simulators from Arm EDA tool vendors	See the Release Note and Integration and Implementation information for your Armv8.1-M implementation for further information on RTL simulator support. The *Arm Cortex-M55 Release Note* and *Arm Cortex-M55*

	Processor Integration and Implementation Manual are confidential documents that are only available to licensees.
Arm MPS3 FPGA Prototyping Board	The Arm MPS3 platform provides a way to load pre-built Arm sub-system images into its FPGA. More information about the Arm MPS3 platform is available at: `https://developer.arm.com/tools-and-software/development-boards/fpga-prototyping-boards/mps3` An SSE-300 FPGA image is due to be released sometime in 2020. SSE-300 is a CoreLink subsystem that includes a Cortex-M55 processor More information about SSE-300 is available at: `https://developer.arm.com/ip-products/subsystem/corelink-subsystem/corelink-sse-300-subsystem`
Third party platforms	Arm works closely with its partners who license Arm technology. Arm partners who have licensed Armv8.1-M technology, such as Cortex-M55, typically develop their own platforms. Such platforms might or might not be publicly available.

Table 3-1 Hardware Platform and Simulation Support for the PMU

3.3 CMSIS Programming API for the PMU

The programmers' model and system address map of the Armv8-M architecture (described in section 4 *Armv8.1-M PMU Programmers' Model*) make it simple to program the PMU in software. To make things even easier for developers to work with the PMU, CMSIS-Core provides ready-to-use *PMU functions and macros* written in C and described in Table 3-2 CMSIS-Core PMU Function Prototypes.

CMSIS-Core is part of the Cortex Microcontroller Software Interface Standard (CMSIS) and provides a standardized API for different aspects of software development for the Cortex-M devices, including:

- Startup and initialization code templates.
- Processor core instruction intrinsics.
- Processor core peripheral *functions and macros*.
- Memory description files (scatter files / linker scripts).
- Device-specific system clock and peripheral macros and functions.

CMSIS-Core source code and documentation is available from the following CMSIS GitHub repository:

- `https://github.com/ARM-software/CMSIS_5`

The following CMSIS-Core C header files support the PMU:

- `https://github.com/ARM-software/CMSIS_5/blob/develop/CMSIS/Core/Include/pmu_armv8.h`
- `https://github.com/ARM-software/CMSIS_5/blob/develop/CMSIS/Core/Include/core_armv81mml.h`
- `https://github.com/ARM-software/CMSIS_5/blob/develop/CMSIS/Core/Include/core_cm55.h`

The following two macros must be set appropriately before using these headers to program the PMU:

- `__PMU_PRESENT`
- `__PMU_NUM_EVENTCNT`

These macros are defined by the CMSIS-Core device header file, which is normally provided by Arm microcontroller device vendors. The device header file is typically available as part of a CMSIS Device Family Pack (DFP) that includes other files, such as the startup and initialization code mentioned in the list above, enabling the user to develop a CMSIS-compliant embedded application. The DFP, which is essentially an archive file, is created by the device vendor.

Arm also acts as a device vendor by providing some device headers and DFPs targeted at its models and platforms described in section 3.2 *Hardware Platform and Simulation Model Support for the PMU*. Two generic Armv8.1-M/Cortex-M55 CMSIS-Core device headers can be found at:

- `https://github.com/ARM-software/CMSIS_5/blob/develop/Device/ARM/ARMv81MML/Include/ARMv81MML_DSP_DP_MVE_FP.h`
- `https://github.com/ARM-software/CMSIS_5/blob/develop/Device/ARM/ARMCM55/Include/ARMCM55.h`

DFPs are supported by different embedded development tools such as Keil MDK and Arm DS. These packs/archives can be downloaded from the following repository on the Arm website.

- `https://developer.arm.com/tools-and-software/embedded/cmsis/cmsis-packs`

Also, some IDEs provide an integrated 'Pack Installer' to make it even easier to download, install, and use, the DFPs and the CMSIS-Core files contained within.

Description	CMSIS-Core PMU function prototype

PMU enable and disable functions.	`__STATIC_INLINE void ARM_PMU_Enable(void);`
	`__STATIC_INLINE void ARM_PMU_Disable(void);`

PMU event type configuration function	`__STATIC_INLINE void ARM_PMU_Set_EVTYPER(uint32_t num, uint32_t type);`

PMU counter reset functions	`__STATIC_INLINE void ARM_PMU_CYCCNT_Reset(void);`
	`__STATIC_INLINE void ARM_PMU_EVCNTR_ALL_Reset(void);`

PMU counter enable and disable functions	`__STATIC_INLINE void ARM_PMU_CNTR_Enable(uint32_t mask);`
	`__STATIC_INLINE void ARM_PMU_CNTR_Disable(uint32_t mask);`

PMU functions for reading counters.	`__STATIC_INLINE uint32_t ARM_PMU_Get_CCNTR(void);`
	`__STATIC_INLINE uint32_t ARM_PMU_Get_EVCNTR(uint32_t num);`

PMU functions for checking and clearing the overflow status	`__STATIC_INLINE uint32_t ARM_PMU_Get_CNTR_OVS(void);`
	`__STATIC_INLINE void ARM_PMU_Set_CNTR_OVS(uint32_t mask);`

PMU functions for enabling and disabling an interrupt on overflow	`__STATIC_INLINE void ARM_PMU_Set_CNTR_IRQ_Enable(uint32_t mask);`
	`__STATIC_INLINE void ARM_PMU_Set_CNTR_IRQ_Disable(uint32_t mask);`

PMU function for manually incrementing a counter in software	`__STATIC_INLINE void ARM_PMU_CNTR_Increment(uint32_t mask);`

Table 3-2 CMSIS-Core PMU Function Prototypes

CMSIS-Core supports the following compilers:

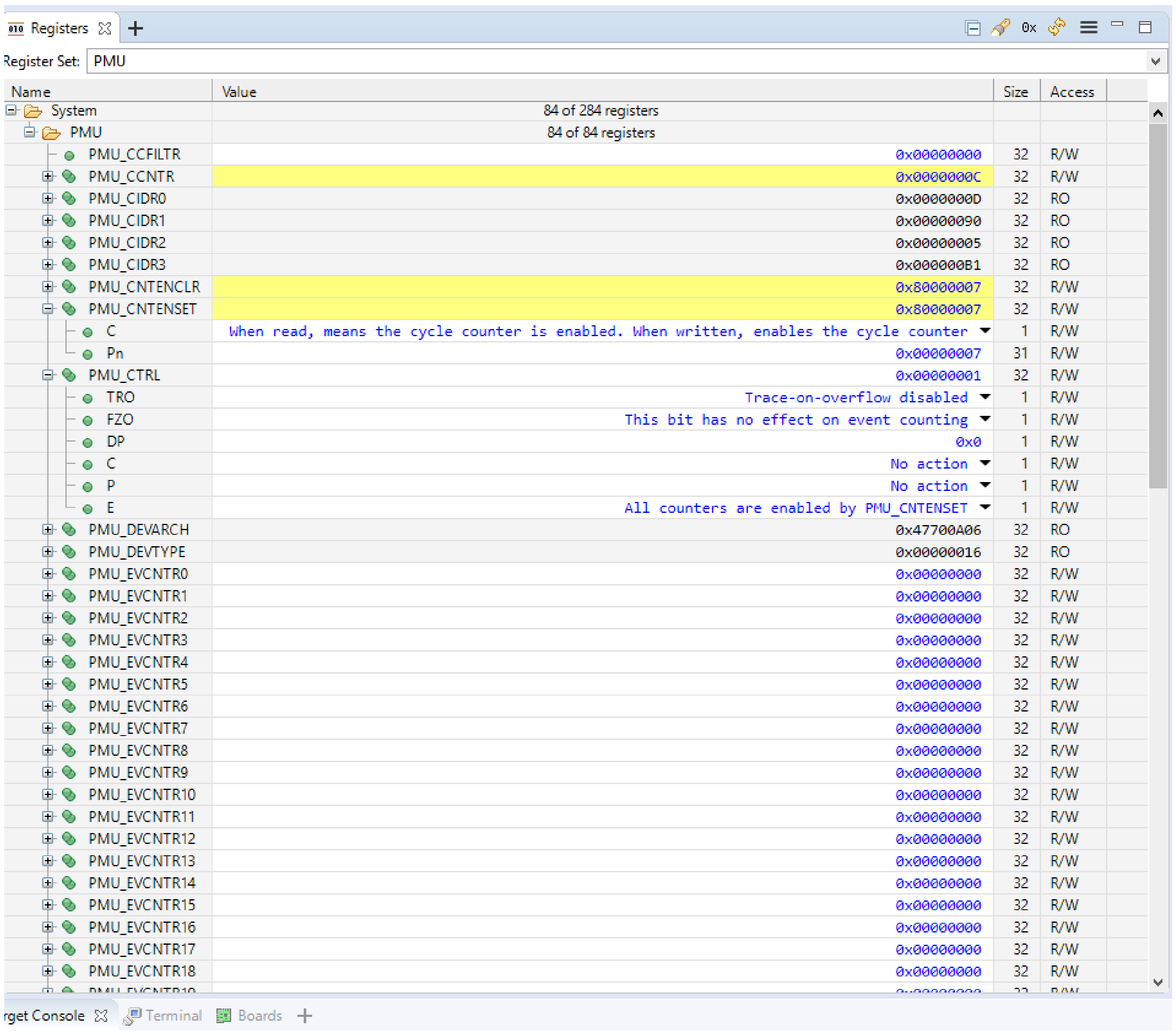
- Arm Compiler 6.
- GNU Arm Embedded Toolchain.
- IAR C/C++ Compiler.

Arm Compiler 6 is available as part of the following products:

- Arm Development Studio (Arm DS).
- Keil Microcontroller Development Kit (Keil MDK).

3.4 Debug Tool Support for the PMU

Software development toolkits such as Arm DS and Keil MDK from Arm, as well as third-party tool vendor Arm debug solutions, provide support for Armv8.1-M and Cortex-M55. These tools typically provide convenient ways to access the PMU registers through memory and register windows.



Register Set: PMU

Name	Value	Size	Access
System	84 of 284 registers		
PMU	84 of 84 registers		
PMU_CCFILTR	0x00000000	32	R/W
PMU_CCNTR	0x0000000C	32	R/W
PMU_CIDR0	0x0000000D	32	RO
PMU_CIDR1	0x00000090	32	RO
PMU_CIDR2	0x00000005	32	RO
PMU_CIDR3	0x000000B1	32	RO
PMU_CNTENCLR	0x80000007	32	R/W
PMU_CNTENSET	0x80000007	32	R/W
C	When read, means the cycle counter is enabled. When written, enables the cycle counter	1	R/W
Pn	0x00000007	31	R/W
PMU_CTRL	0x00000001	32	R/W
TRO	Trace-on-overflow disabled	1	R/W
FZO	This bit has no effect on event counting	1	R/W
DP	0x0	1	R/W
C	No action	1	R/W
P	No action	1	R/W
E	All counters are enabled by PMU_CNTENSET	1	R/W
PMU_DEVARCH	0x47700A06	32	RO
PMU_DEVTYPE	0x00000016	32	RO
PMU_EVCNTR0	0x00000000	32	R/W
PMU_EVCNTR1	0x00000000	32	R/W
PMU_EVCNTR2	0x00000000	32	R/W
PMU_EVCNTR3	0x00000000	32	R/W
PMU_EVCNTR4	0x00000000	32	R/W
PMU_EVCNTR5	0x00000000	32	R/W
PMU_EVCNTR6	0x00000000	32	R/W
PMU_EVCNTR7	0x00000000	32	R/W
PMU_EVCNTR8	0x00000000	32	R/W
PMU_EVCNTR9	0x00000000	32	R/W
PMU_EVCNTR10	0x00000000	32	R/W
PMU_EVCNTR11	0x00000000	32	R/W
PMU_EVCNTR12	0x00000000	32	R/W
PMU_EVCNTR13	0x00000000	32	R/W
PMU_EVCNTR14	0x00000000	32	R/W
PMU_EVCNTR15	0x00000000	32	R/W
PMU_EVCNTR16	0x00000000	32	R/W
PMU_EVCNTR17	0x00000000	32	R/W
PMU_EVCNTR18	0x00000000	32	R/W

Figure 3-1 Arm Development Studio Registers View

Also, the PMU can issue an event counter trace packet each time the lower 8 bits of a counter overflows. This only occurs when a counter increments naturally and not when it is written to directly by software or using a debugger. Additionally, the PMU can serve as an event source for the *Cross Trigger Interface* (CTI), which might be useful for debugging, tracing, and profiling, systems with multiple processors.

4 Armv8.1-M PMU Programmers' Model

4.1 Armv8.1-M PMU Registers Overview

Like other Armv8-M peripheral, system, and debug registers, the PMU registers are memory-mapped to the *Private Peripheral Bus* (PPB) address space. The PMU registers are located in a 4KB debug component block within the System space of the PPB.

The *Registers index* section of the Armv8-M architecture provides a complete list of registers that can be implemented in an Armv8-M implementation. This section shows that the block of system memory for the PMU registers begins at address 0xE0003000. The PMU registers and their associated addresses are listed with links to more detailed register descriptions.

Address	Register	Description
0xE0003000	PMU_EVCNTRn	Performance Monitoring Unit Event Counter Register
0xE000307C	PMU_CCNTR	Performance Monitoring Unit Cycle Counter Register
0xE0003400	PMU_EVTYPERn	Performance Monitoring Unit Event Type and Filter Register
0xE000347C	PMU_CCFILTR	Performance Monitoring Unit Cycle Counter Filter Register
0xE0003C00	PMU_CNTENSET	Performance Monitoring Unit Count Enable Set Register
0xE0003C20	PMU_CNTENCLR	Performance Monitoring Unit Count Enable Clear Register
0xE0003C40	PMU_INTENSET	Performance Monitoring Unit Interrupt Enable Set Register
0xE0003C60	PMU_INTENCLR	Performance Monitoring Unit Interrupt Enable Clear Register
0xE0003C80	PMU_OVSCLR	Performance Monitoring Unit Overflow Flag Status Clear Register
0xE0003CA0	PMU_SWINC	Performance Monitoring Unit Software Increment Register
0xE0003CC0	PMU_OVSSET	Performance Monitoring Unit Overflow Flag Status Set Register
0xE0003E00	PMU_TYPE	Performance Monitoring Unit Type Register
0xE0003E04	PMU_CTRL	Performance Monitoring Unit Control Register
0xE0003FB8	PMU_AUTHSTATUS	Performance Monitoring Unit Authentication Status Register
0xE0003FBC	PMU_DEVARCH	Performance Monitoring Unit Device Architecture Register
0xE0003FCC	PMU_DEVTYPE	Performance Monitoring Unit Device Type Register
0xE0003FD0	PMU_PIDR4	Performance Monitoring Unit Peripheral Identification Register 4
0xE0003FE0	PMU_PIDR0	Performance Monitoring Unit Peripheral Identification Register 0
0xE0003FE4	PMU_PIDR1	Performance Monitoring Unit Peripheral Identification Register 1
0xE0003FE8	PMU_PIDR2	Performance Monitoring Unit Peripheral Identification Register 2
0xE0003FEC	PMU_PIDR3	Performance Monitoring Unit Peripheral Identification Register 3

0xE0003FF0	PMU_CIDR0	Performance Monitoring Unit Component Identification Register 0
0xE0003FF4	PMU_CIDR1	Performance Monitoring Unit Component Identification Register 1
0xE0003FF8	PMU_CIDR2	Performance Monitoring Unit Component Identification Register 2
0xE0003FFC	PMU_CIDR3	Performance Monitoring Unit Component Identification Register 3

Table 4-1 PMU Registers and Address Mappings

The PMU Type Register, PMU_TYPE, helps software identify information about a device's PMU configuration. For example, PMU_TYPE can be read to find out the number of counters available in the PMU.

Another important register is the PMU Control Register, PMU_CTRL. PMU_CTRL can be used to enable/disable the PMU and reset the PMU counters.

A PMU Event Type Register, PMU_EVTYPERn, can be programmed by software to determine which event a specific counter is monitoring.

The PMU Count Enable Set Register, PMU_CNTENSET, and PMU Count Enable Clear Register, PMU_CNTENCLR, can be used to enable and disable individual event counters.

A PMU Event Counter Registers, PMU_EVCNTRn, can be read by software to determine the current count of an event associated with that counter. There is also the PMU Cycle Count Register, PMU_CYCCNT, which is dedicated to counting cycles. These registers can be reset to zero and can also be written to so that they have a starting value.

Some of the other PMU registers are covered throughout this document such as the Performance Monitoring Unit Software Increment Register, PMU_SWINC, as well as registers related to overflow and interrupt generation.

The PMU counters count upwards. When a PMU counter overflows the action taken depends on how the PMU is configured. For example, it can be configured to generate an interrupt upon an overflow. These use cases are described in more detail in section 7 *Using the PMU in your application*.

Before using the PMU, software needs to ensure that trace is enabled via the Debug Exception Monitor Control Register, DEMCR.

Figure 4-1 shows a typical usage flow for configuring the PMU. Section 7 provides code examples on how to use the PMU in your application.

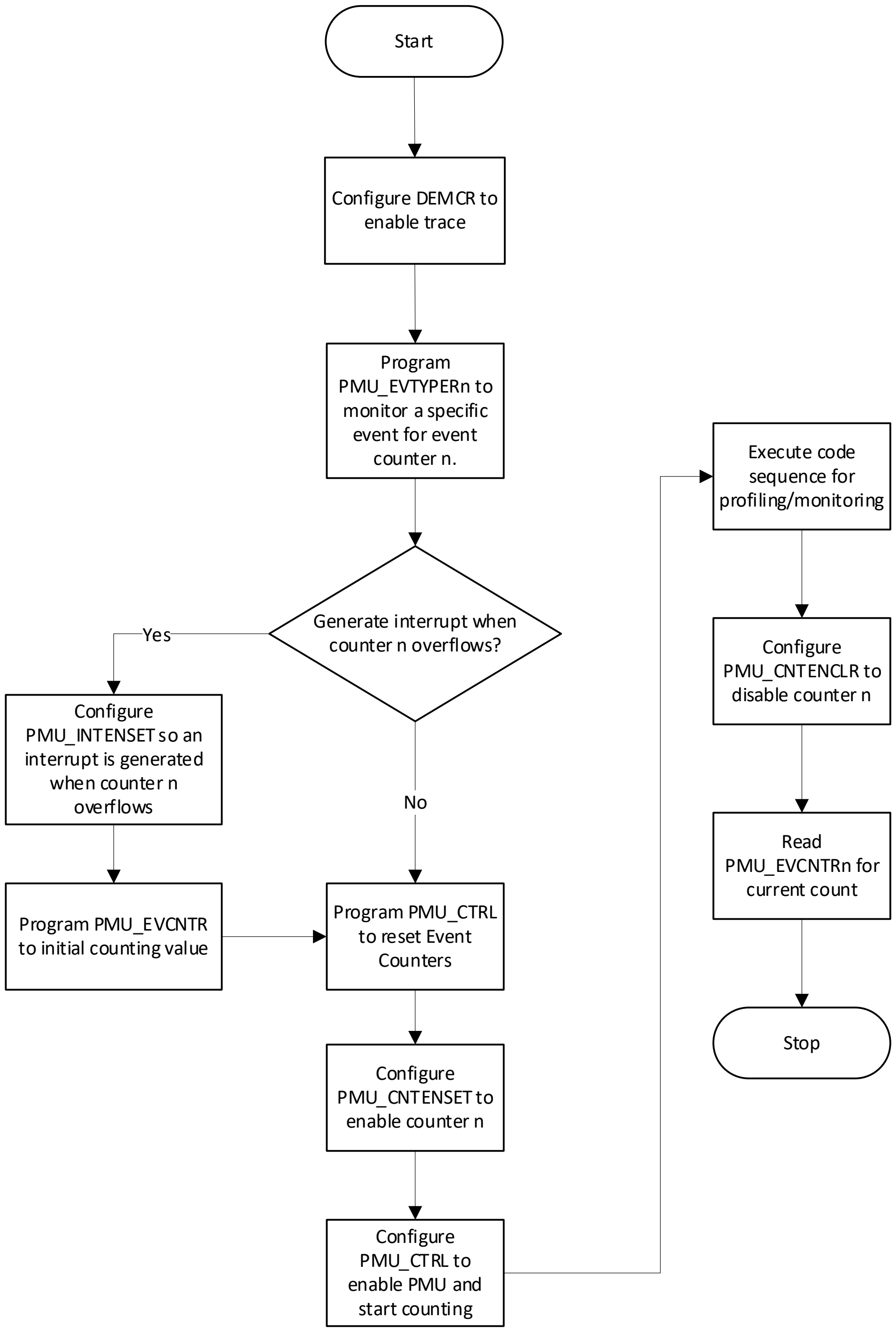

Start
Configure DEMCR to enable trace
Program PMU_EVTYPERn to monitor a specific event for event counter n.
Generate interrupt when counter n overflows?
Yes
No
Configure PMU_INTENSET so an interrupt is generated when counter n overflows
Program PMU_EVCNTR to initial counting value
Program PMU_CTRL to reset Event Counters
Configure PMU_CNTENSET to enable counter n
Configure PMU_CTRL to enable PMU and start counting
Execute code sequence for profiling/monitoring
Configure PMU_CNTENCLR to disable counter n
Read PMU_EVCNTRn for current count
Stop

Figure 4-1 PMU Configuration Example

4.2 Armv8.1-M PMU Events Overview

This section aims to provide any additional usage information about certain events, including information specific to Cortex-M55.

Furthermore, the CMSIS-Core header file `pmu_armv8.h` and relevant `core_<cpu>.h` header contain macros corresponding to each supported event with some brief descriptions. These macros can be used by software, as shown in section 7 *Using the PMU in your application* and section 8 *PMU Profiling Example*.

Currently, Armv8.1-M supports up to 131 different PMU events.

The section titled *List of supported architectural and microarchitectural events* in the *Armv8-M Architecture Reference Manual* (DI0553B.k) provides a full list and descriptions of the supported events that can be counted.

Note

The number of supported events may change in future revisions of the architecture.

4.1.1 Architectural vs microarchitectural events

Information statement IFHHC from the *Armv8-M Architecture Reference Manual* states that events fall into two categories:

- Architectural events, which are the same across all implementations
- Microarchitectural events, which might vary across different implementations.

For example, the architectural event, EXC_TAKEN (Exception taken), can be used to count each time any implementation, such as Cortex-M55, takes an exception. The behavior of this event would work the same on any other Armv8.1-M implementation.

Good examples of microarchitectural events are any events relating to caches or branch prediction, since these features may vary across different implementations. For example, Cortex-M55 does not support branch prediction, but it is possible for another Armv8.1-M implementation to support it. Also, even if features like these are supported in a particular implementation, they might not be configured/enabled in the RTL or by software. Additionally, even, if such features are implemented, they may be configured differently. For example, there are different RTL configuration options for the sizes of the caches in Cortex-M55.

There are 11 required architectural and microarchitectural events.

There are also some architectural and microarchitectural events that are not supported by Cortex-M55 (see section 4.3.3 *Unsupported architectural and microarchitectural events*).

4.1.2 Event types

The 131 architectural and microarchitectural PMU events can be broadly put into the following event type categories:

- Instruction execution.
- Instruction speculation.
- Operation execution.
- Operation speculation.
- MVE instruction execution.
- MVE instruction speculation.
- External memory accesses.
- *Tightly Coupled Memory* (TCM) accesses.
- Cache behavior.

- Branch prediction.
- Exceptions.
- Security state transitions.
- Pipeline stalls.
- Chaining counters.
- CPU Cycles.
- Debug and trace events.
- Software increment.
- Memory errors.

4.3 Cortex-M55 Events

4.3.1 Architectural and microarchitectural events supported by Cortex-M55

Cortex-M55 supports 82 out of the 131 architectural and microarchitectural events. A full list of events can be found in Table 4-2.

4.3.2 Implementation-defined events

The architecture allows an implementation to include extra events. The Cortex-M55 processor supports an additional 18 implementation-specific PMU events related to the following:

- Error Correcting Code (ECC) in the TCM or Cache memories - events beginning with ECC_.
- No Write-Allocate mode - event prefixed with NWA (NWAMODE_ENTER and NWAMODE).
- S-AHB accesses - SAHB_ACCESS.
- P-AHB accesses - PAHB_ACCESS.
- M-AXI accesses - AXI_WRITE_ACCESS and AXI_READ_ACCESS.
- Data cache prefetching - PF_LINEFILL, PF_CANCEL and PF_DROP_LINEFILL.
- Internal watchdog - DOSTIMEOUT_DOUBLE and DOSTIMEOUT_TRIPLE.

4.3.3 Unsupported architectural and microarchitectural events

Some of the architectural and microarchitectural events are not supported by Cortex-M55. Cortex-M55 implements only a subset of the µArch PMU events which reflect the simple inline, non-speculative nature of the processor pipeline. Events that are not supported fall into the following categories:

- Branch prediction events.
- Level 2 cache events.
- Level 3 cache events.
- Instruction speculation events.
- Operation speculation events.
- Level 1 data cache allocate.
- Operation execution.
- All pipeline stall ‘slot’ events.
- Branch Future.

4.4 Event Usage Notes

4.4.1 Architectural, microarchitectural and implementation-defined event usage table

This section focuses on all the architectural, microarchitectural and Cortex-M55-specifc events. Table 4-2 below provides a brief description for all supported event categories along with some additional usage notes, including Cortex-M55-specific details, where applicable. The table is intended to be used as a supplement to the information about events that already exists in the *Armv8-M Architecture Reference Manual*.

Key for Table 4-2:

- Type: A = Arch, AR = Arch Required, I = IMPDEF, M = µArch, MR = µArch Required
- Bits: This column refers to the Cortex-M55 PMU Event Bus bits (see section 4.4.7 *Event bus bits*).

No.	Type	Name and Description	Usage Notes	Bits
0x0000	AR	SW_INCR Instruction architecturally executed, condition code check pass, software increment	See section 7.5 *Manually incrementing a Counter in Software*.	0
0x0001	M	L1I_CACHE_REFILL Attributable Level 1 instruction cache refill	See section 4.4.2 *Level 1 cache events*.	1
0x0003	MR	L1D_CACHE_REFILL Attributable Level 1 data cache refill	See section 4.4.2 *Level 1 cache events*.	2
0x0004	MR	L1D_CACHE Attributable Level 1 data cache access	See section 4.4.2 *Level 1 cache events*.	3
0x0006	A	LD_RETIRED Instruction architecturally executed, condition code check pass, load	The architecture states: Whether the preload instructions PLD, PLDW, and PLI, count as memory-reading instructions is IMPLEMENTATION DEFINED On Cortex-M55: - PLD and PLDW are both fully supported on Cortex-M55. (PLDW requests a line-fill for a cache miss in a Write-allocate region). - PLI is not operationally supported and acts like a NOP. PLI has no effect on LD_RETIRED.	4
0x0007	A	ST_RETIRED Instruction architecturally executed, condition code check pass, store		5

0x0008	AR	INST_RETIRED Instruction architecturally executed		6
0x0009	A	EXC_TAKEN Exception taken		7
0x000A	A	EXC_RETURN Instruction architecturally executed, condition code check pass, exception return		8
0x000C	A	PC_WRITE_RETIRED Instruction architecturally executed, condition code check pass, software change of the PC	The architecture states: It is IMPLEMENTATION DEFINED whether the counter increments for any or all of: • BKPT instructions. • An exception generated because an instruction is UNDEFINED. • The exception-generating instructions, SVC, and UDF. It is IMPLEMENTATION DEFINED whether an ISB is counted as a software change of the PC. On Cortex-M55: PMU <x>_RETIRED events only count operations which complete so they don't include any cases where an instruction is interrupted by an exception or debug event (including BKPT). However, SVC is included in this event as it functionally behaves like software-controlled branch. ISB is also counted as a PC write event.	9
0x000D	A	BR_IMMED_RETIRED Instruction architecturally executed, immediate branch	The architecture states: If an ISB is counted as a software change of the PC instruction, then it is IMPLEMENTATION DEFINED whether an ISB is counted as an immediate branch instruction. On Cortex-M55: This event only counts true immediate branches i.e. B #imm, CB{N}Z #imm	10
0x000E	A	BR_RETURN_RETIRED Instruction architecturally executed, condition code check pass, procedure return		11
0x000F	A	UNALIGNED_LDST_RETIRED Instruction architecturally executed, condition code check pass, unaligned load or store		12

0x0010	A	BR_MIS_PRED Mispredicted or not predicted branch speculatively executed	The architecture states: If branches are decoded before the branch predictor, so that the branch prediction logic dynamically predicts only some branches, for example conditional and indirect branches, then it is IMPLEMENTATION DEFINED whether other branches are counted as predictable branches. On Cortex-M55 branch prediction isn't supported and therefore this event is not supported.	
0x0011	MR	CPU_CYCLES Cycle	The Cortex-M55 processor has a four-stage pipeline. Once the pipeline is full the Cortex-M55 is capable of executing at least one instruction per cycle. Some 16-bit instruction pairs can be dual-issued to further improve how many instructions per cycle the Cortex-M55 processor is capable of.	14
0x0012	M	BR_PRED Predictable branch speculatively executed	Unsupported on Cortex-M55.	
0x0013	M	MEM_ACCESS Data memory access		16
0x0014	M	L1I_CACHE Attributable Level 1 instruction cache access	See section 4.4.2 *Level 1 cache events*.	17
0x0015	M	L1D_CACHE_WB Attributable Level 1 data cache write-back	See section 4.4.2 *Level 1 cache events*.	18
0x0016	M	L2D_CACHE Attributable Level 2 data cache access	Unsupported on Cortex-M55.	
0x0017	M	L2D_CACHE_REFILL Attributable Level 2 data cache refill	Unsupported on Cortex-M55.	
0x0018	M	L2D_CACHE_WB Attributable Level 2 data cache write-back	Unsupported on Cortex-M55.	
0x0019	M	BUS_ACCESS Attributable Bus access	The architecture states: Whether bus accesses include operations that do use the bus but that do not explicitly transfer data is IMPLEMENTATION DEFINED. The maximum increment in any given cycle is	19

			IMPLEMENTATION DEFINED. This event is triggered by any beat on M-AXI, P-AHB and EPPB interfaces. These beats do not necessarily map onto architectural load/store/instruction fetches, which are not used, i.e. cache line entries, speculative instruction fetches, etc.	
0x001A	M	MEMORY_ERROR Local memory error	See also sections 4.4.2 *Level 1 cache events* and 4.4.3 *TCM events*, and section 4.3.2 *Implementation-defined events*.	20
0x001B	M	INST_SPEC Operation speculatively executed	Unsupported on Cortex-M55.	
0x001D	M	BUS_CYCLES Bus cycle		22
0x001E	A	CHAIN For an odd numbered counter, increment when an overflow occurs on the preceding even-numbered counter on the same PE	See section 7.6 *Chaining Event Counters to create a 32-bit Counter*.	23
0x001F	M	L1D_CACHE_ALLOCATE Attributable Level 1 data cache allocation without refill	See section 4.4.2 *Level 1 cache events*.	
0x0020	M	L2D_CACHE_ALLOCATE Attributable Level 2 data cache allocation without refill	Unsupported on Cortex-M55.	
0x0021	AR	BR_RETIRED Instruction architecturally executed, branch	The architecture states: It is IMPLEMENTATION DEFINED whether this includes each of: – Unconditional direct branch instructions. – Exception-generating instructions. – Exception return instructions. – Context synchronization instructions. This event follows the same rules as the PC_WRITE_RETIRED event (see above), apart from conditional branch instructions where BR_RETIRED is counted whether the instruction executes or not, i.e. for B<c> #imm will result in a BR_RETIRED event but will only result in a PC_WRITE_RETIRED event if the conditional code <c> passes.	25
0x0022	MR	BR_MIS_PRED_RETIRED	Cortex-M55 has single cycle branch latency, without a requirement for branch prediction.	26

		Instruction architecturally executed, mispredicted branch	Therefore, on Cortex-M55, this event counts all retired not-taken branches.	
0x0023	MR	STALL_FRONTEND No operation issued because of the frontend	The architecture states: The division between frontend and backend is IMPLEMENTATION DEFINED. On Cortex-M55: If there are no instructions available from the fetch stage of the processor pipeline (into the main decode/execution stages), the processor considers the front-end of the processor pipeline as being stalled. A high number in STALL_FRONTEND might mean: - The instruction cache is too small, or the program access pattern does not cache well. - Instruction access latency is high. When running code that is accessed via the AXI interface, latency on the AXI bus can occur due to cache misses, which can in turn add stall cycles.	27
0x0024	MR	STALL_BACKEND No operation issued because of the backend	The architecture states: The division between frontend and backend is IMPLEMENTATION DEFINED. On Cortex-M55: If there is an instruction available from the fetch stage of the pipeline but it cannot be accepted by the decode stage of the processor pipeline, the processor considers the back-end of the processor pipeline as being stalled. This is likely caused by memory access wait states, but could also be caused by multi-cycle operations in the coprocessor interface. When running code that is accessed via the AXI interface, latency on the AXI bus can occur due to cache misses, which can in turn add stall cycles.	28
0x0027	M	L2I_CACHE Attributable Level 2 instruction cache access	Unsupported on Cortex-M55. Cortex-M55 cannot be configured with a Level 2 cache.	
0x0028	M	L2I_CACHE_REFILL Attributable Level 2 instruction cache refill	Unsupported on Cortex-M55. Cortex-M55 cannot be configured with a Level 2 cache.	
0x0029	M	L3D_CACHE_ALLOCATE Attributable Level 3 data cache allocation without refill	Unsupported on Cortex-M55. Cortex-M55 cannot be configured with a Level 3 cache.	

0x002A	M	L3D_CACHE_REFILL Attributable Level 3 data cache refill	Unsupported on Cortex-M55. Cortex-M55 cannot be configured with a Level 3 cache.	
0x002B	M	L3D_CACHE Attributable Level 3 data cache access	Unsupported on Cortex-M55. Cortex-M55 cannot be configured with a Level 3 cache.	
0x002C	M	L3D_CACHE_WB Attributable Level 3 data cache access write-back	Unsupported on Cortex-M55. Cortex-M55 cannot be configured with a Level 3 cache.	
0x0036	M	LL_CACHE_RD Last level data cache read	The last level cache on Cortex-M55 is the level 1 cache. See section 4.3.2.	29
0x0037	M	LL_CACHE_MISS_RD Last level data cache read miss	The last level cache on Cortex-M55 is the level 1 cache. See section 4.4.2 *Level 1 cache events*.	30
0x0039	M	L1D_CACHE_MISS_RD Level 1 data cache read miss	See section 4.4.2 *Level 1 cache events*.	31
0x003A	M	OP_COMPLETE Operation retired	Unsupported on Cortex-M55.	
0x003B	M	OP_SPEC Operation speculated	Unsupported on Cortex-M55.	
0x003C	M	STALL No operation sent for execution	This general case stall event counts when there is no instruction executing this cycle. This could be due to STALL_FRONTEND or STALL_BACKEND, or simply a register/memory hazard.	34
0x003D	M	STALL_OP_BACKEND No operation sent for execution on a slot because of the backend	Unsupported on Cortex-M55.	
0x003E	M	STALL_OP_FRONTEND No operation sent for execution on a slot because of the frontend	Unsupported on Cortex-M55.	
0x003F	M	STALL_OP No operation sent for execution on a slot	Unsupported on Cortex-M55.	

0x0040	M	L1D_CACHE_RD Level 1 data cache read		38
0x0100	M	LE_RETIRED Loop end instruction architecturally executed, entry registered in the LO_BRANCH_INFO cache		39
0x0101	M	LE_SPEC Loop end instruction speculatively executed entry registered in LO_BRANCH_INFO cache	Unsupported on Cortex-M55.	
0x0104	M	BF_RETIRED Branch future instruction architecturally executed, condition code check pass, and registers an entry in the LO_BRANCH_INFO cache	Unsupported on Cortex-M55.	
0x0105	M	BF_SPEC Branch future instruction speculatively executed, condition code check and registers an entry in the LO_BRANCH_INFO cache	Unsupported on Cortex-M55.	
0x0108	M	LE_CANCEL LO_BRANCH_INFO cache containing a valid loop entry cleared while not in the last iteration of the loop		43
0x0109	M	BF_CANCEL LO_BRANCH_INFO cache containing a valid BF entry cleared and associated branch not taken	Unsupported on Cortex-M55.	
0x0114	A	SE_CALL_S Call to secure function, resulting in Security state change		45
0x0115	A	SE_CALL_NS Call to non-secure function, resulting in Security state change		46
0x0118	A	DWT_CMPMATCH0 DWT comparator 0 match	See section 7.7.8 *Triggering an overflow after the core has executed code 'N' times*.	47
0x0119	A	DWT_CMPMATCH1 DWT comparator 1 match	See section 7.7.8 *Triggering an overflow after the core has executed code 'N' times*.	48

0x011A	A	DWT_CMPMATCH2 DWT comparator 2 match	See section 7.7.8 *Triggering an overflow after the core has executed code 'N' times*.	49
0x011B	A	DWT_CMPMATCH3 DWT comparator 3 match	See section 7.7.8 *Triggering an overflow after the core has executed code 'N' times*.	50
0x0200	AR	MVE_INST_RETIRED MVE instruction architecturally executed	See section 4.4.4 *EPU events*.	51
0x0201	M	MVE_INST_SPEC MVE instruction speculatively executed	This event is unsupported on Cortex-M55.	
0x0204	A	MVE_FP_RETIRED MVE floating-point instruction architecturally executed	See section 4.4.4 *EPU events*.	53
0x0205	M	MVE_FP_SPEC MVE floating-point instruction speculatively executed	This event is unsupported on Cortex-M55.	
0x0208	A	MVE_FP_HP_RETIRED MVE half-precision floating-point instruction architecturally executed	See section 4.4.4 *EPU events*.	55
0x0209	M	MVE_FP_HP_SPEC MVE half-precision floating-point instruction speculatively executed	This event is unsupported on Cortex-M55.	
0x020C	A	MVE_FP_SP_RETIRED MVE single-precision floating-point instruction architecturally executed	See section 4.4.4 *EPU events*.	57
0x020D	M	MVE_FP_SP_SPEC MVE single-precision floating-point instruction speculatively executed	This event is unsupported on Cortex-M55.	
0x0214	A	MVE_FP_MAC_RETIRED MVE floating-point multiply or multiply-accumulate instruction architecturally executed	See section 4.4.4 *EPU events*.	59
0x0215	M	MVE_FP_MAC_SPEC	This event is unsupported on Cortex-M55.	

		MVE floating-point multiply or multiply-accumulate instruction speculatively executed		
0x0224	A	MVE_INT_RETIRED MVE integer instruction architecturally executed	See section 4.4.4 *EPU events*.	61
0x0225	M	MVE_INT_SPEC MVE integer instruction speculatively executed	This event is unsupported on Cortex-M55.	
0x0228	A	MVE_INT_MAC_RETIRED MVE integer multiply or multiply-accumulate instruction architecturally executed	See section 4.4.4 *EPU events*.	63
0x0229	M	MVE_INT_MAC_SPEC MVE integer multiply or multiply-accumulate instruction speculatively executed	This event is unsupported on Cortex-M55.	
0x0238	AR	MVE_LDST_RETIRED MVE load or store instruction architecturally executed	See section 4.4.4 *EPU events*.	65
0x0239	M	MVE_LDST_SPEC MVE load or store instruction speculatively executed	This event is unsupported on Cortex-M55.	
0x023C	A	MVE_LD_RETIRED MVE load instruction architecturally executed	See section 4.4.4 *EPU events*.	67
0x023D	M	MVE_LD_SPEC MVE load instruction speculatively executed	This event is unsupported on Cortex-M55.	
0x0240	A	MVE_ST_RETIRED MVE store instruction architecturally executed	See section 4.4.4 *EPU events*.	69
0x0241	M	MVE_ST_SPEC MVE store instruction speculatively executed	This event is unsupported on Cortex-M55.	
0x0244	A	MVE_LDST_CONTIG_RETIRED MVE contiguous load or store instruction architecturally executed	See section 4.4.4 *EPU events*.	71
0x0245	M	MVE_LDST_CONTIG_SPEC	This event is unsupported on Cortex-M55.	

		MVE contiguous load or store instruction speculatively executed		
0x0248	A	MVE_LD_CONTIG_RETIRED MVE contiguous load instruction architecturally executed	See section 4.4.4 *EPU events*.	73
0x0249	M	MVE_LD_CONTIG_SPEC MVE contiguous load instruction speculatively executed	This event is unsupported on Cortex-M55.	
0x024C	A	MVE_ST_CONTIG_RETIRED MVE contiguous store instruction architecturally executed	See section 4.4.4 *EPU events*.	75
0x024D	M	MVE_ST_CONTIG_SPEC MVE contiguous store instruction speculatively executed	This event is unsupported on Cortex-M55.	
0x0250	A	MVE_LDST_NONCONTIG_RETIRED MVE non-contiguous load or store instruction architecturally executed	See section 4.4.4 *EPU events*.	77
0x0251	M	MVE_LDST_NONCONTIG_SPEC MVE non-contiguous load or store instruction speculatively executed	This event is unsupported on Cortex-M55.	
0x0254	A	MVE_LD_NONCONTIG_RETIRED MVE non-contiguous load instruction architecturally executed	See section 4.4.4 *EPU events*.	79
0x0255	M	MVE_LD_NONCONTIG_SPEC MVE non-contiguous load instruction speculatively executed	This event is unsupported on Cortex-M55.	
0x0258	A	MVE_ST_NONCONTIG_RETIRED MVE non-contiguous store instruction architecturally executed	See section 4.4.4 *EPU events*.	81
0x0259	M	MVE_ST_NONCONTIG_SPEC MVE non-contiguous store instruction speculatively executed	This event is unsupported on Cortex-M55.	
0x025C	A	MVE_LDST_MULTI_RETIRED	See section 4.4.4 *EPU events*.	83

		MVE memory instruction targeting multiple registers architecturally executed		
0x025D	M	MVE_LDST_MULTI_SPEC MVE memory instruction targeting multiple registers speculatively executed	This event is unsupported on Cortex-M55.	
0x0260	A	MVE_LD_MULTI_RETIRED MVE memory load instruction targeting multiple registers architecturally executed	See section 4.4.4 *EPU events*.	85
0x0261	M	MVE_LD_MULTI_SPEC MVE memory load instruction targeting multiple registers speculatively executed	This event is unsupported on Cortex-M55.	
0x0264	A	MVE_ST_MULTI_RETIRED MVE memory store instruction targeting multiple registers architecturally executed	See section 4.4.4 *EPU events*.	87
0x0265	M	MVE_ST_MULTI_SPEC MVE memory store instruction targeting multiple registers speculatively executed	This event is unsupported on Cortex-M55.	
0x028C	A	MVE_LDST_UNALIGNED_RETIRED MVE unaligned memory load or store instruction architecturally executed	See section 4.4.4 *EPU events*.	89
0x028D	M	MVE_LDST_UNALIGNED_SPEC MVE unaligned memory load or store instruction speculatively executed	This event is unsupported on Cortex-M55.	
0x0290	A	MVE_LD_UNALIGNED_RETIRED MVE unaligned memory load instruction architecturally executed	See section 4.4.4 *EPU events*.	91
0x0291	M	MVE_LD_UNALIGNED_SPEC MVE unaligned memory load instruction speculatively executed	This event is unsupported on Cortex-M55.	
0x0294	A	MVE_ST_UNALIGNED_RETIRED MVE unaligned store instruction architecturally executed	See section 4.4.4 *EPU events*.	93

0x0295	M	MVE_ST_UNALIGNED_SPEC MVE unaligned store instruction speculatively executed	This event is unsupported on Cortex-M55.	
0x0298	A	MVE_LDST_UNALIGNED_NONCONTIG_RETIRED MVE unaligned noncontiguous load or store instruction architecturally executed	See section 4.4.4 *EPU events*.	95
0x0299	M	MVE_LDST_UNALIGNED_NONCONTIG_SPEC MVE unaligned non-contiguous load or store instruction speculatively executed	This event is unsupported on Cortex-M55.	
0x02A0	A	MVE_VREDUCE_RETIRED MVE vector reduction instruction architecturally executed	See section 4.4.4 *EPU events*.	97
0x02A1	M	MVE_VREDUCE_SPEC MVE vector reduction instruction speculatively executed	This event is unsupported on Cortex-M55.	
0x02A4	A	MVE_VREDUCE_FP_RETIRED MVE floating-point vector reduction instruction architecturally executed	See section 4.4.4 *EPU events*.	99
0x02A5	M	MVE_VREDUCE_FP_SPEC MVE floating-point vector reduction instruction speculatively executed	This event is unsupported on Cortex-M55.	
0x02A8	A	MVE_VREDUCE_INT_RETIRED MVE integer vector reduction instruction architecturally executed	See section 4.4.4 *EPU events*.	101
0x02A9	M	MVE_VREDUCE_INT_SPEC MVE integer vector reduction instruction speculatively executed	This event is unsupported on Cortex-M55.	
0x02B8	M	MVE_PRED Cycles where one or more predicated beats architecturally executed	See section 4.4.4 *EPU events*. The architecture states: The ratio (BEATS_PER_TICK) / (4 * then offers an approximate insight into the proportion of MVE instructions affected by predication, where BEATS_PER_TICK is an IMPLEMENTATION DEFINED average number of beats, including from	102

			distinct overlapping instructions, executed per Architecture tick. On Cortex-M55 MVE is implemented in a ‘two beats per tick’ configuration. This event counts a cycle where any MVE operation tick executes with any corresponding bits of the predicate flags set to 0 and the instruction is VPT compatible or supports tail predication.	
0x02CC	M	MVE_STALL Stall cycles caused by an MVE instruction	See section 4.4.4 *EPU events*.	103
0x02CD	M	MVE_STALL_RESOURCE Stall cycles caused by an MVE instruction because of resource conflicts	See section 4.4.4 *EPU events*.	104
0x02CE	M	MVE_STALL_RESOURCE_MEM Stall cycles caused by an MVE instruction because of memory resource conflicts	See section 4.4.4 *EPU events*.	105
0x02CF	M	MVE_STALL_RESOURCE_FP Stall cycles caused by an MVE instruction because of floating-point resource conflicts	See section 4.4.4 *EPU events*.	106
0x02D0	M	MVE_STALL_RESOURCE_INT Stall cycles caused by an MVE instruction because of integer resource conflicts	See section 4.4.4 *EPU events*.	107
0x02D3	M	MVE_STALL_BREAK Stall cycles caused by an MVE chain break	See section 4.4.4 *EPU events*.	108
0x02D4	M	MVE_STALL_DEPENDENCY Stall cycles caused by MVE register dependency	See section 4.4.4 *EPU events*.	109
0x4007	M	ITCM_ACCESS Instruction TCM access	See section 4.4.3 *TCM events*.	110
0x4008	M	DTCM_ACCESS Data TCM access	See section 4.4.3 *TCM events*.	111
0x4010	M	TRCEXTOUT0 ETM external output 0		112

0x4011	M	TRCEXTOUT1 ETM external output 1		113
0x4012	M	TRCEXTOUT2 ETM external output 2		114
0x4013	M	TRCEXTOUT3 ETM external output 3		115
0x4018	M	CTI_TRIGOUT4 Cross-trigger Interface output trigger 4		116
0x4019	M	CTI_TRIGOUT5 Cross-trigger Interface output trigger 5		117
0x401A	M	CTI_TRIGOUT6 Cross-trigger Interface output trigger 6		118
0x401B	M	CTI_TRIGOUT7 Cross-trigger Interface output trigger 7		119
0xC000	I	ECC_ERR Any ECC error	See section 4.4.5 *ECC events*. See also sections 4.4.2 *Level 1 cache events* and 4.4.3 *TCM events*.	120
0xC001	I	ECC_ERR_FATAL One or more multi-bit ECC errors detected	See section 4.4.5 *ECC events*. See also sections 4.4.2 *Level 1 cache events* and 4.4.3 *TCM events*.	121
0xC010	I	ECC_ERR_DCACHE One or more ECC errors in the data cache	See section 4.4.5 *ECC events*. See also section 4.4.2 *Level 1 cache events*.	122
0xC011	I	ECC_ERR_ICACHE One or more ECC errors in the instruction cache	See section 4.4.5 *ECC events*. See also section 4.4.2 *Level 1 cache events*.	123
0xC012	I	ECC_ERR_FATAL_DCACHE One or more multi-bit ECC errors in the data cache	See section 4.4.5 *ECC events*. See also section 4.4.2 *Level 1 cache events*.	124
0xC013	I	ECC_ERR_FATAL_ICACHE One or more multi-bit ECC errors in the instruction cache	See section 4.4.5 *ECC events*. See also section 4.4.2 *Level 1 cache events*.	125

0xC020	I	ECC_ERR_DTCM One or more ECC errors in the DTCM	See section 4.4.5 *ECC events*. See also section 4.4.3 *TCM events*.	126
0xC021	I	ECC_ERR_ITCM One or more ECC errors in the ITCM	See section 4.4.5 *ECC events*. See also section 4.4.3 *TCM events*.	127
0xC022	I	ECC_ERR_FATAL_DTCM One or more multi-bit ECC errors in the DTCM	See section 4.4.5 *ECC events*. See also section 4.4.3 *TCM events*.	128
0xC023	I	ECC_ERR_FATAL_ITCM One or more multi-bit ECC errors in the ITCM	See section 4.4.5 *ECC events*. See also section 4.4.3 *TCM events*.	129
0xC100	I	PF_LINEFILL The prefetcher starts a linefill	See section 4.4.2 *Level 1 cache events*.	130
0xC101	I	PF_CANCEL The prefetcher stops prefetching	See section 4.4.2 *Level 1 cache events*.	131
0xC102	I	PF_DROP_LINEFILL A linefill triggered by the prefetcher has been dropped because of lack of buffering	See section 4.4.2 *Level 1 cache events*.	132
0xC200	I	NWAMODE_ENTER No-write allocate mode entry	See section 4.4.2 *Level 1 cache events*.	133
0xC201	I	NWAMODE Write-Allocate store is not allocated into the data cache due to no-write-allocate mode	See section 4.4.2 *Level 1 cache events*.	134
0xC300	I	SAHB_ACCESS Read or write access on the S-AHB interface to the TCM		135
0xC301	I	PAHB_ACCESS Read or write access to the P-AHB write interface		136
0xC302	I	AXI_WRITE_ACCESS Any beat access to M-AXI write interface		137
0xC303	I	AXI_READ_ACCESS Any beat access to M-AXI read interface		138

0xC400	I	DOSTIMEOUT_DOUBLE Denial of Service timeout has fired twice and caused buffers to drain to allow forward progress	See section 4.4.6 *Denial-of-service events*.	140
0xC401	I	DOSTIMEOUT_TRIPLE Denial of Service timeout has fired three times and blocked the LSU to force forward progress	See section 4.4.6 *Denial-of-service events*.	141

Table 4-2 PMU Event Usage

4.4.2 Level 1 cache events

The Cortex-M55 can be configured to include level 1 instruction and data caches:

- Level 1 instruction and data caches can be configured for different sizes.
- The level 1 instruction cache can be enabled using the CMSIS-Core function `SCB_EnableICache()`.
- The level 1 data cache can be enabled using the CMSIS-Core function `SCB_EnableDCache()`.
- The Cortex-M55 implementation-defined events, PF_LINEFILL, PF_CANCEL and PF_DROP_LINEFILL, refer to the level 1 data cache prefetcher.

The Cortex-M55 also has a bit in its Auxiliary Control Register (ACTLR) to disable write allocation. Disabling write allocation is generally worse for performance but can improve performance in some situations where allocating on writes is undesirable, such as executing the C standard library `memset()` or whilst initializing memory before the `main()` program begins.

4.4.3 TCM events

The Cortex-M55 can be configured to include instruction and data *Tightly Coupled Memories* (TCMs):

- TCMs are implementation defined features of Cortex-M55 and are not described by the Armv8-M architecture.
- The instruction and data TCMs can be configured for different sizes.
- The instructions TCM can be configured to be enabled or disabled out-of-reset.
- The TCMs can also be enabled or disabled by software by writing to the Cortex-M55's ITCM Control Register and DTCM Control Registers.

4.4.4 EPU events

The Extension Processing Unit (EPU) performs:

- Scalar floating-point operations.
- MVE operations.

The EPU is disabled at reset. Software can typically enable the EPU using code in the CMSIS-Core device header: `system_<device>.c` file in the `SystemInit()` function. See section 3.3 *CMSIS Programming API for the PMU* for further information about CMSIS.

4.4.5 ECC events

A Cortex-M55 can be optionally configured to enable *Error Correcting Code* (ECC) out of reset to allow the processor to check for memory errors in its level 1 memories (caches and TCMs).

4.4.6 Denial-of-service events

The Denial-of-service events (DOSTIMEOUT_DOUBLE and DOSTIMEOUT_TRIPLE) record cases when the internal watchdog (affectionally known as the 'watchcat') times-out due to a stream of requests taking up too much resource - usually a case where a continual stream of load/store requests can potentially block an unrelated event from completing. These events can indicate software sequences which could be inefficient for performance and power as they overconsume the bandwidth of the processor memory system.

4.4.7 Event bus bits

All events are exported to the external output signal EVENTBUS as a single cycle pulse allowing system level analysis of processor performance.

4.4.8 Other metrics derived from events

It is possible to generate other metrics once the user has counts for certain events. For example:

Instructions per cycle (IPC)

$$\mathrm{IPC} = \frac{\mathrm{INST\_RETIRED}}{\mathrm{CPU\_CYCLES}}$$

The inverse, cycles per instructions, or CPI, is also commonly used.

MIPS (retired)

$$\mathrm{MIPS_{retired}} = \frac{\mathrm{INST\_RETIRED}}{t_{\mathrm{elapsed}} \times 10^6}$$

Where t_{elapsed} is the elapsed time, in seconds.

5 Configuring the PMU in an Armv8.1-M implementation or model

5.1 Cortex-M55 RTL Configuration

There is no specific RTL configuration parameter for the PMU in Cortex-M55. Provided that some level of debug is selected, the PMU will be configured in a Cortex-M55 implementation. See the *Arm Cortex-M55 Processor Integration and Implementation Manual* - for further information. The *Arm Cortex-M55 Processor Integration and Implementation Manual is a confidential document that is only available to licensees.*

5.2 Cortex-M55 Cycle Model Configuration

The Cortex-M55 Cycle Model is 100% cycle accurate and based on the Cortex-M55 RTL. Therefore, the same configuration information described in section 5.1 *Cortex-M55 RTL Configuration* applies to the Cortex-M55 Cycle Model.

5.3 Cortex-M55 FVP and Armv8.1-M Architecture Envelope Model Configuration

When working with a Fast Model Fixed Virtual Platform that supports the PMU, there are a couple of configuration parameters that need to be set to ensure that the PMU is present:

Model	Parameter	Default Value	Description
`Armv8.1-M AEM`	`has_pmu`	`0`	Set to `1` to ensure PMU is present.
	`num_pmu_counters`	`0x1F` (31)	Set according to the number of desired counters in the AEM
`Cortex-M55 FVP`	`has_pmu`	`0`	Set to `1` to ensure PMU is present.
	`num_pmu_counters`	`0x1F` (31)	Set according to the number of desired counters in the Cortex-M55 FVP. Note that the default value is not valid for a Cortex-M55, so this value should be changed to either `8` or `0` to match the RTL configuration options.

Table 5-1 Cortex-M55 FVP and Armv8.1-M AEM PMU parameters

For example, when using cpu0 on the Cortex-M55 FVP, start the fast model with the following parameters to ensure the PMU is present with eight counters:

```
FVP_MPS2_Cortex-M55.exe -c cpu0.has_pmu=1 -c cpu0.num_pmu_counters=8
```

6 PMU accessibility and restrictions

6.1 Accessing the PMU Registers

The PMU registers are only accessible to privileged code. Any unprivileged accesses generate a fault.

Arm recommends using CMSIS-Core PMU support code, which is written in C, to access the PMU registers. If accessing the PMU registers in assembly language, please note that these registers are word accessible only. Halfword and byte accesses are UNPREDICTABLE.

6.2 Debug

The PMU counters do not increment when:

- The Processing Element (PE) is in Debug state.
- The PE is in Secure state and secure non-invasive debug is disabled.
- The PE is in Non-secure state and non-invasive debug is disabled.

The *Armv8-M Architecture Reference Manual* also lists some restrictions when the PMU is used at the same time as the Armv8.0-M DWT Performance Monitors.

Also see section 6.3 *Security*.

6.3 Security

The PMU registers are not banked between Security states. This means that both secure privileged and non-secure privileged code can access the PMU registers.

The PMU registers are accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

The PMU_CTRL.DP bit is an alias of the DWT_CTRL.CYCDISS bit, which is set to zero on a Cold reset. When PMU_CTRL.DP is zero, the PMU cycle counter increments regardless of the Security state of the PE. Therefore, to ensure that the cycle counter does not count in Secure state, set PMU_CTRL.DP (or DWT_CTRL.CYCDISS) to 1: This can be achieved with the following CMSIS-Core compliant code:

```
PMU->PMU_Ctrl |= PMU_CTRL_CYCCNT_DISABLE_Msk;
```

Also see section 6.2 *Debug*.

6.4 Low Power State

The PMU counters do not increment when:

- The PE is in low-power state the counters retain their previous value.

6.5 CPU Lockup

It is UNKNOWN whether the counters increment in lockup state.

7 Using the PMU in your application

7.1 Checking whether your Device has a PMU

There are a few ways to find out whether the PMU is present in your device, and if so, how many PMU counters are available:

- Reading your device's documentation
  It is usually possible to find out whether the device you are working with includes a feature like the PMU by reading the device's documentation.

- Reading the CMSIS-Core device header
  It should also be possible to find out whether the PMU is present by checking whether the CMSIS-Core macro `__PMU_PRESENT` is set or not. The `__PMU_NUM_EVENTCNT` macro tells you how many event counters are implemented on the device.

- Reading the PMU Type Register
  If you are writing generic software for any Armv8.1-M-based device, or just want to be certain that the PMU is present or not, you can either write some code or use a debugger to check the N field in the PMU Type Register. If this field returns a non-zero number, this means that the PMU is implemented and shows how many counters are available for software to use.

  Software can simply use the CMSIS-Core API for the PMU (in `pmu_armv8.h`) to find out the number of available counters, for example:

  ```
  uint32_t num_event_counters;
  num_event_counters = ((PMU->PMU_Type) & PMU_TYPE_NUM_CNTS_Msk);
  ```

7.2 Enabling and disabling the PMU

On a Warm reset the PMU is disabled. Before software can begin enabling Event Counters, the PMU must be enabled. The following CMSIS-Core function call can be used to enable the PMU:

```
/* Enable the PMU */
ARM_PMU_Enable();
```

The PMU can be disabled again with the following CMSIS-Core function call:

```
/* Disable the PMU */
ARM_PMU_Disable();
```

The user also needs to ensure that trace is enabled inside the processor in order to make use of the PMU. The following CMSIS-Core code can be used as a global enable for the DWT, PMU, and ITM features:

```
/* Enable Trace */
CoreDebug->DEMCR |= CoreDebug_DEMCR_TRCENA_Msk;
```

7.3 Using the 32-bit Cycle Counter

Using the Cycle Counter Register is very simple with the CMSIS-Core API. Below is an example of how to use the Cycle Counter Register:

```
/* Initialize variable for reading cycle count */
uint32_t cycle_count = 0;

/* Reset PMU Cycle Counter */
ARM_PMU_CYCCNT_Reset();

/* Enable PMU Cycle Counter */
ARM_PMU_CNTR_Enable(PMU_CNTENSET_CCNTR_ENABLE_Msk);

/* Add code you want to measure here */

/* Disable PMU Cycle Counter */
ARM_PMU_CNTR_Disable(PMU_CNTENSET_CCNTR_ENABLE_Msk);

/* Read PMU Cycle Counter */
cycle_count = ARM_PMU_Get_CCNTR();
```

A user might also find it useful to keep an incremental count, so that each time the code being measured is run, the program keeps track of the overall combined count. For example:

```
/* Get incremental cycle count */
cycle_count = cycle_count + ARM_PMU_Get_CCNTR();
```

The PMU Cycle Counter is set to an unknown value on a Warm reset. Therefore, the Cycle Counter should be reset before it is used for the first time. Whether the cycle counter needs resetting more than once will depend on the application. For example, reset the cycle counter to measure the performance of a new segment of code.

Note: Some project development environments make use of the Cycle Counter Register for debug features. For example, Keil MDK uses the Cycle Counter register for its Event Recorder and the States register:

http://www.keil.com/support/man/docs/uv4/uv4_db_dbg_evr.htm
http://www.keil.com/support/man/docs/uv4/uv4_db_dbg_cpuregs.htm

Therefore, using the CMSIS PMU API to modify the Cycle Counter Register may affect the usability of such debug features.

7.4 Using 16-bit Event Counters

Using the Event Counter Registers is very simple with the CMSIS-Core API. Below is an example of how to use configure, enable and use two Event Counter registers. The two events that are counted in this example are the number of instructions retired and level 1 data cache misses.

```
/* Initialize variables for counting instructions retired and L1 D-Cache misses */
uint32_t instructions_retired_count = 0;
uint32_t l1_dcache_miss_count = 0;

/*
   Configure Event Counter Register 0 to count instructions retired
   Configure Event Counter Register 1 to count L1 D-Cache misses
*/
ARM_PMU_Set_EVTYPER(0, ARM_PMU_INST_RETIRED);
ARM_PMU_Set_EVTYPER(1, ARM_PMU_L1D_CACHE_MISS_RD);

/* Reset PMU Event Counters */
ARM_PMU_EVCNTR_ALL_Reset();

/* Start incrementing Event Counter Registers 0 & 1 */
ARM_PMU_CNTR_Enable(PMU_CNTENSET_CNT0_ENABLE_Msk|PMU_CNTENSET_CNT1_ENABLE_Msk);
```

```
/* Add code you want to measure here */

/* Stop incrementing Event Counter Registers 0 & 1 */
ARM_PMU_CNTR_Disable(PMU_CNTENSET_CNT0_ENABLE_Msk|PMU_CNTENSET_CNT1_ENABLE_Msk);

/* Get number of instructions retired and number of L1 D-Cache misses (on read) */
instructions_retired_count = ARM_PMU_Get_EVCNTR(0);
l1_dcache_miss_count = ARM_PMU_Get_EVCNTR(1);
```

Use cases will obviously vary. This example simply reads Event Counters 0 and 1 into the variables `instructions_retired_count` and `l1_dcache_miss_count`. These variables should provide one-time-only counts relating to whatever code is added between enabling and disabling the counters. There are a wide range of ways the user can go about analyzing these values. For example, the values could be printed to a display or saved to a file on a storage device. The user could also add the variables to a Watch Window in a debugger and configure the debugger to break program execution when they reach a certain data value range.

A user might also find it useful to keep an incremental count, so that each time the code being measured is run, the program keeps track of the overall counts combined. For example:

```
/* Get incremental count of number of instructions retired */
instructions_retired_count = instructions_retired_count + ARM_PMU_Get_EVCNTR(0);
```

The PMU Event Counters are set to an unknown value on a Warm reset. Therefore, the user should reset the Event Counters before using them for the first time. Whether the counters need resetting more than once, or disabling, will depend on the application. For example, it might be desirable to reset the counters when a new thread becomes active. When reading multiple counter values, slightly more accurate results might be observed by disabling the counters before reading their current value.

7.5 Manually incrementing a Counter in Software

One of the PMU events, SW_INCR (event number 0x0), works differently to the other event counters.

The description from the *Armv8-M Architecture Reference Manual* for SW_INCR says:

"The counter increments on writes to the PMU_SWINC register."

Configuring and enabling an event counter so that it can be incremented by software can be achieved by using similar code as in section 7.4 *Using 16-bit Event Counters*. The `ARM_PMU_CNTR_Increment()` function can then be used to write to the relevant bit in the Software Increment Register to increment the counter, before it's read by software again sometime later. For example:

```
/* Initialize variable for reading software increment counter */
uint32_t sw_increment = 0;

/* Configure Event Counter Register 2 so it can be incremented by software */
ARM_PMU_Set_EVTYPER(2, ARM_PMU_SW_INCR);

/* Reset PMU Event Counters */
ARM_PMU_EVCNTR_ALL_Reset();

/* Enable Event Counter Register 2 */
ARM_PMU_CNTR_Enable(PMU_CNTENSET_CNT2_ENABLE_Msk);

/* Increment Event Counter Register 2 in software */
ARM_PMU_CNTR_Increment(PMU_SWINC_CNT2_Msk);

/* Read Event Counter Register 2 */
sw_increment = ARM_PMU_Get_EVCNTR(2);
```

The *Armv8-M Architecture Reference Manual* states:

"If the PE performs two Architecturally executed writes to the PMU_SWINC register without an intervening Context synchronization event, then the counter is incremented twice."

What this means is that a Context synchronization event, e.g., Instruction Synchronization Barrier (ISB), is not required between two writes to the Software Increment Register to guarantee that the related Event Counter increments twice.

```
/* Increment Event Counter Register 2 twice */
ARM_PMU_CNTR_Increment(PMU_SWINC_CNT2_Msk);
// __ISB(); is not required
ARM_PMU_CNTR_Increment(PMU_SWINC_CNT2_Msk);
```

7.6 Chaining Event Counters to create a 32-bit Counter

The Event Counter registers have a 16-bit Counter field. This might be suitably wide enough for counting some events but might not be wide enough for counting others that could potentially overflow one or more times.

To make it less likely that you need to handle a counter overflow, it is possible to chain an odd-numbered counter with a preceding even-numbered counter to form a 32-bit counter. For example, software could chain together Event Counter 7 with Event Counter 6 to form a 32-bit counter. This also means that the system can be configured by software to have a mixture of 16-bit and 32-bit counters.

The example below shows how you can create another 32-bit cycle counter from two Event Counter registers

```
/*
   Initialize variables for:
   - lower 16 bits of cycle count
   - upper 16 bits of cycle count
   - cycle count (concatenated)
*/
uint32_t cycle_count_lower = 0;
uint32_t cycle_count_upper = 0;
uint32_t cycle_count_combined = 0;

/*
   Configure Event Counter Register 6 to count CPU Cycles
   Configure Event Counter Register 7 to chain together with Event Counter Register 6
*/
ARM_PMU_Set_EVTYPER(6, ARM_PMU_CPU_CYCLES);
ARM_PMU_Set_EVTYPER(7, ARM_PMU_CHAIN);

/* Reset PMU Event Counters */
ARM_PMU_EVCNTR_ALL_Reset();

/* Enable Event Counter Registers 6 & 7 */
ARM_PMU_CNTR_Enable(PMU_CNTENSET_CNT6_ENABLE_Msk);
ARM_PMU_CNTR_Enable(PMU_CNTENSET_CNT7_ENABLE_Msk);

/* Add code you want to measure here */

/* Read Event Counter Registers 6 & 7 */
cycle_count_lower = ARM_PMU_Get_EVCNTR(6);
cycle_count_high = ARM_PMU_Get_EVCNTR(7);

/* Concatenate Event Counter Registers 6 & 7 */
```

```
cycle_count_combined = (cycle_count_high << 16) | PMU_EVCNTR_CNT_Msk & cycle_count_lower);
```

If your device implements the DSP Extension you might notice that a compiler translates the above logical OR operation into a single `PKHBT` instruction.

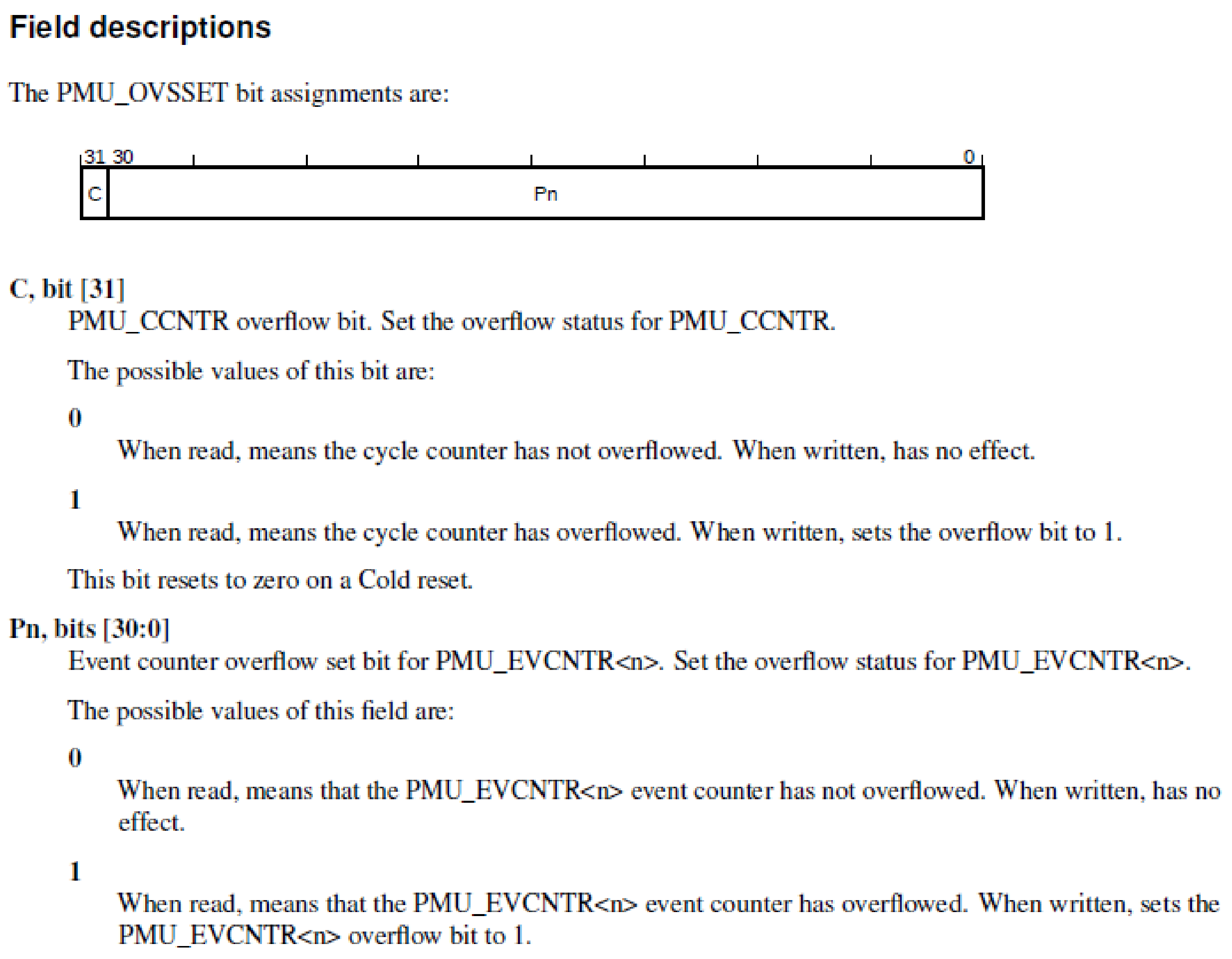
Note that there is a known issue in CMSIS v5.70 with `PMU_EVCNTR_CNT_Msk`. It should be set to `0xFFFFUL`, but instead it is incorrectly set to `16UL`. Use one of the following workarounds to avoid this issue:

- Edit your copy of `pmu_armv8.h` and correct the macro.
- Avoid using the macro and instead use the constant `0xFFFFUL` in your code.

This will issue will be fixed in future versions of CMSIS.

7.7 Handling Counter Overflow

It is possible for any of the counters (32-bit Cycle Counter, 16-bit Event Counters, or chained 16-bit Event Counters) to overflow. If a counter has overflowed this is indicated in the PMU Overflow Flag Status Set Register (PMU_OVSSET). The *Armv8-M Architecture Reference Manual* provides the following description of PMU_OVSSET:



Field descriptions

The PMU_OVSSET bit assignments are:

C, bit [31]

PMU_CCNTR overflow bit. Set the overflow status for PMU_CCNTR.

The possible values of this bit are:

0

When read, means the cycle counter has not overflowed. When written, has no effect.

1

When read, means the cycle counter has overflowed. When written, sets the overflow bit to 1.

This bit resets to zero on a Cold reset.

Pn, bits [30:0]

Event counter overflow set bit for PMU_EVCNTR<n>. Set the overflow status for PMU_EVCNTR<n>.

The possible values of this field are:

0

When read, means that the PMU_EVCNTR<n> event counter has not overflowed. When written, has no effect.

1

When read, means that the PMU_EVCNTR<n> event counter has overflowed. When written, sets the PMU_EVCNTR<n> overflow bit to 1.

Figure 7-1 ***Armv8-M Architecture Reference Manual*** **Snapshot of PMU Overflow Status Set Register Bit Descriptions**

7.7.1 Checking whether a Counter has overflowed

The following CMSIS-Core PMU code can be used to read PMU_OVSSET:

```
/* Initialize overflow status variable */
uint32_t pmu_overflow_status = 0;

/* Read PMU Overflow Set Register */
pmu_overflow_status = ARM_PMU_Get_CNTR_OVS();
```

A user might only be interested in whether a particular counter has overflowed. For example, to find out whether Event Counter 3 has overflowed a user could mask all other overflow bits except bit 3 (the bit that corresponds to Event Counter 3):

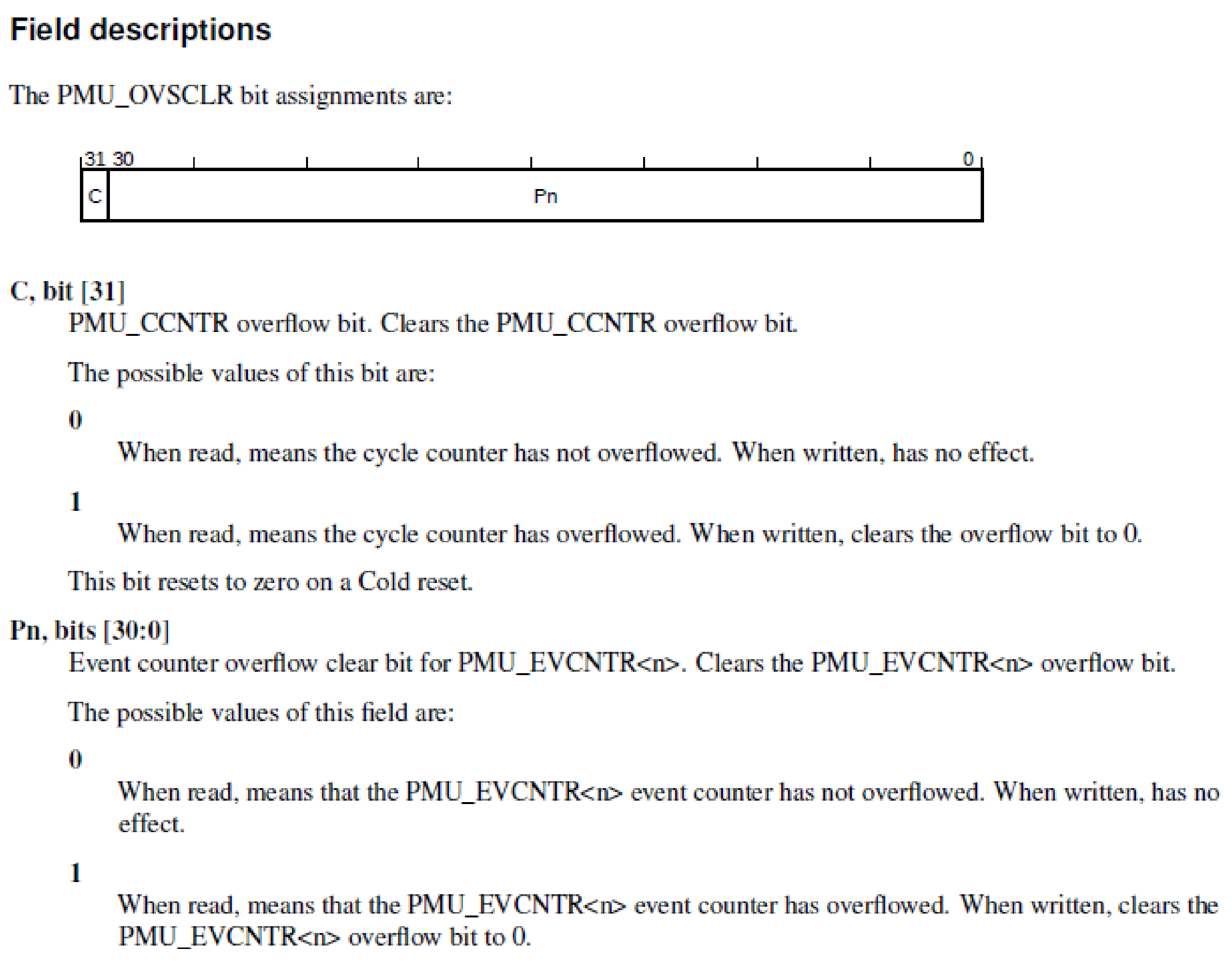
```
/* Initialize Event Counter 3 overflow status variable */
uint32_t pmu_overflow_status_evcntr_3 = 0;

/* Clear value read from PMU_OVSSET, except bit 3 */
pmu_overflow_status_evcntr_3 = PMU_OVSSET_CNT3_STATUS_Msk & pmu_overflow_status;

/* Check if bit 3 was set */
if(pmu_overflow_status_evcntr_3)
{
  printf("PMU Event Register 3 has overflowed.");
}
else
{
  printf("PMU Event Register 3 has not overflowed.").
}
```

7.7.2 Clearing the overflow status

Since there is only a single overflow bit for each counter, software may want to clear the overflow status before the counter overflows a second time, otherwise it might not be possible to determine how many times a counter has overflowed. The *Armv8-M Architecture Reference Manual* provides the following description of PMU_OVSCLR:



Field descriptions

The PMU_OVSCLR bit assignments are:

31	30 … 0
C	Pn

C, bit [31]

PMU_CCNTR overflow bit. Clears the PMU_CCNTR overflow bit.

The possible values of this bit are:

0

When read, means the cycle counter has not overflowed. When written, has no effect.

1

When read, means the cycle counter has overflowed. When written, clears the overflow bit to 0.

This bit resets to zero on a Cold reset.

Pn, bits [30:0]

Event counter overflow clear bit for PMU_EVCNTR<n>. Clears the PMU_EVCNTR<n> overflow bit.

The possible values of this field are:

0

When read, means that the PMU_EVCNTR<n> event counter has not overflowed. When written, has no effect.

1

When read, means that the PMU_EVCNTR<n> event counter has overflowed. When written, clears the PMU_EVCNTR<n> overflow bit to 0.

Figure 7-2 ***Armv8-M Architecture Reference Manual*** **Snapshot of PMU Overflow Status Clear Register Bit Descriptions**

The following CMSIS-Core PMU code clears overflow status of Event Counter Register 3:

```
ARM_PMU_Set_CNTR_OVS(PMU_OVSCLR_CNT3_STATUS_Msk);
```

Depending on the application it also might be perfectly ok to clear all counter overflow status bits, rather than just one bit.

7.7.3 Generating an interrupt on a counter overflow

The user might want to know exactly when a counter has overflowed. The CMSIS-Core PMU support code can be used to enable and disable interrupt generation when a given counter overflows. For example, the following code ensures that an interrupt is generated when Event Counter 3 overflows:

```
ARM_PMU_Set_CNTR_IRQ_Enable(PMU_INTENSET_CNT3_ENABLE_Msk);
```

There's also a corresponding 'Disable counter overflow interrupt request' function named `ARM_PMU_Set_CNTR_IRQ_Disable()`.

Note: on a Cold reset, counter overflow interrupt requests are disabled.

The interrupt associated with a PMU counter overflow is the DebugMonitor exception. Unlike Halting debug, the DebugMonitor exception is traditionally used as a method of debugging without putting the core into Debug state. Instead the processor carries on running, which is useful for debugging systems with hard real-time requirements when it is not a viable option to halt the processor's clock. Handling PMU counter overflows is a new usage model for the DebugMonitor exception in M-profile systems. The following code enables Monitor debug:

```
/* Enable Monitor debug */

CoreDebug->DEMCR |= CoreDebug_DEMCR_MON_EN_Msk;
```

The System Handler Priority Register 3 (SHPR3) can be programmed by privileged software to program the priority of the DebugMonitor exception. Therefore, in order to ensure that the DebugMonitor exception is taken, it is important to provide it with an appropriate priority level.

Note: there are secure and non-secure versions of the DebugMonitor exception and SHPR3.

The user is responsible for writing the DebugMonitor exception handling routine. When the DebugMonitor exception is generated on a counter overflow, the associated handler code could use a variable to count how many times a counter has overflowed, for example:

```
static uint32_t overflow_count;

void DebugMon_Handler(void)
{
  overflow_count++;
  return;
}
```

The current counter value can be concatenated with the overflow count variable (similar to concatenating chained counters in section 7.6 *Chaining Event Counters to create a 32-bit Counter*) to form a larger sized counter. Let's say that event counter 3 is being used to count the number of retired instructions. To form a larger 48-bit counter, an application could execute something similar to the following code:

```
/* Create 64-bit variable for storing 48-bit counter value */
Uint64_t count_combined;

/* Read Event Counter Register 3 */
instructions_retired_count = ARM_PMU_Get_EVCNTR(3);


/* Concatenate current value for Event Counter 3 with its overflow Count */
count_combined = (unsigned long long(overflow_count << 16)) |
                  PMU_EVCNTR_CNT_Msk & instructions_retired_count);
```

7.7.4 Checking which counter overflowed

The above method works fine if the user is only working with one counter. However, the user might have enabled multiple counters, in which case they would also need to have an overflow counter for each counter they are working with. Therefore, when working with multiple counters that could overflow, the handler would need to carry out something similar to the following:

1. Check which counter overflowed.
2. Increment a 32-bit count variable associated with the counter that overflowed.
3. Clear the counter overflow status.

These steps are simple to carry out if only a single counter has overflowed. The following example shows one way of handling this scenario within the `DebugMon_Handler()` exception handling routine.

```
/* Create a word array for each counters' overflows count */
static uint32_t overflow_count[__PMU_NUM_EVENTCNT+1];

void DebugMon_Handler(void)
{
  /* Read PMU overflow status */
  uint32_t pmu_overflow_status = ARM_PMU_Get_CNTR_OVS();

  /* Clear overflow status */
  ARM_PMU_Set_CNTR_OVS(pmu_overflow_status);

  /* Count leading zeroes to find out which bit position was set */
  pmu_overflow_status = __CLZ(pmu_overflow_status);

  /* Calculate trailing zeroes: take away no. of leading zeroes from no. of reg bits */
  pmu_overflow_status = (32 - pmu_overflow_status)-1;

  /* Increment overflow count for counter that overflowed */
  overflow_count[pmu_overflow_status]++;

  return;
}
```

This example uses the __PMU_NUM_EVENTCNT macro to create an array with an element for each event counter, plus the cycle counter, that can be used to count how many times a counter has overflowed. A Cortex-M55 implementation that includes a PMU has eight event counters, plus one cycle counter, so such an array in a piece of Cortex-M55 software would be nine words deep.

One further issue to consider is that more than one counter can potentially overflow at the same time, and therefore, multiple overflow bits could be set. Although this might be an unlikely scenario, for accurate information on counter overflow the DebugMonitor handler would need to carry out steps 1-3 again, but this time loop through each bit of the overflow status to check which bits are set. For example:

```
/* Create a word array for each counters' overflows count */
static uint32_t overflow_count[__PMU_NUM_EVENTCNT+1];

void DebugMon_Handler(void)
{
  uint32_t temp;

  /* Read PMU overflow status */
  uint32_t pmu_overflow_status = ARM_PMU_Get_CNTR_OVS();

  /* Clear overflow status */
  ARM_PMU_Set_CNTR_OVS(pmu_overflow_status);

  while(pmu_overflow_status)
  {
    /* Count leading zeroes to find out the highest bit position set */
    temp = __CLZ(pmu_overflow_status);

    /* Calculate trailing zeroes: take away no. of leading zeroes from no. of reg bits */
    temp = (32 - temp)-1;

    /* Increment overflow count for counter that overflowed */
    overflow_count[temp]++;
    /* Clear highest overflow bit set */
    pmu_overflow_status &= ~(1UL << temp);
  }
  return;
}
```

Note: The PMU handler code itself could affect various counters. Therefore, it might be a good idea to temporarily disable the counter(s) that caused the interrupt at the beginning of the handler routine and enable the counter(s) again before returning to the main application.

7.7.5 Stop counting events on a counter overflow

The Freeze-on-overflow bit in the PMU Control Register can be set as follows to stop the PMU counting events once any counter overflows:

```
PMU->CTRL |= PMU_CTRL_FRZ_ON_OV_Msk;
```

The user can check whether freeze-on-overflow support is available by reading the PMU Type Register. The following CMSIS-Core macros can be used by software `PMU_TYPE_FRZ_OV_SUPPORT_Pos` and `PMU_TYPE_FRZ_OV_SUPPORT_Pos`.

Note: setting freeze-on-overflow will cause the chaining of event counters to stop working, because the overflow of the odd-numbered counter freezes counting.

7.7.6 Halting the processor on a counter overflow

If the system supports debug, it is possible to halt the processor and put it into Debug state when a counter overflows. This can be achieved in software or with a debugger (via a DAP interface) by ensuring that the following fields are set in the Debug Halting Control and Status Register (DHCSR):

- C_PMOV (Halt on PMU overflow) field – bit [6].
- C_DEBUGEN - Debug enable control field, bit [0] - writes from software are ignored.

Note: when writing to the DHCSR, `0xA05F` must be written to the DEBUGKEY field - bits [31:16] – otherwise the write will be ignored.

Software can set the C_PMOV field using the following code:

```
DCB->DHCSR = 0xA05F0040;
```

Note: software cannot write to C_DEBUGEN and can only read this bit to see whether Halting debug has been enabled by a debugger.

A debugger must write `0xA05F041` to set both C_PMOV and C_DEBUGEN in the DHSCR, to halt the core when a PMU counter overflows.

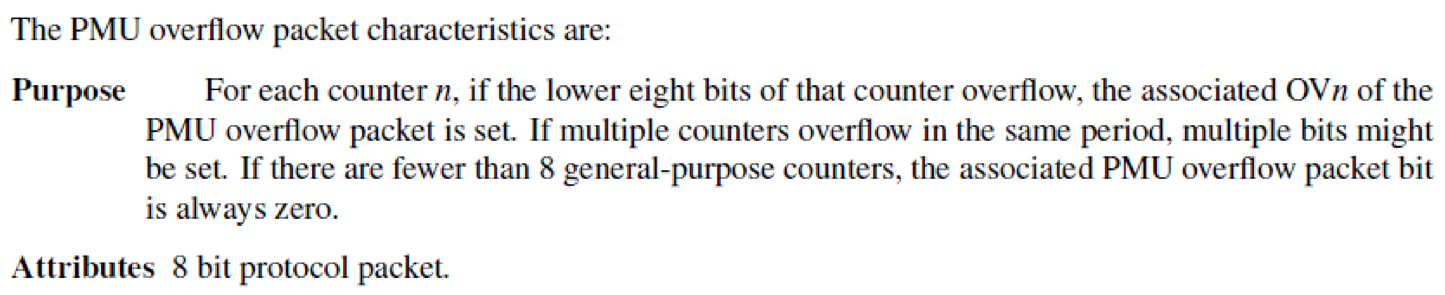
7.7.7 Emitting trace on a counter overflow

The system can emit trace packets whenever any of the first eight counters overflows an 8-bit value. This is achieved by enabling the trace-on-overflow bit in the PMU Control Register as follows:

```
PMU->CTRL |= PMU_CTRL_TRACE_ON_OV_Msk;
```

Note that the user can check whether trace-on-overflow support is available by reading the PMU Type Register. The following CMSIS-Core macros can be used by software `PMU_TYPE_TRACE_ON_OV_SUPPORT_Msk` and `PMU_TYPE_TRACE_ON_OV_SUPPORT_Msk`.

The *Armv8-M Architecture Reference Manual* describes the trace packet information as follows:



The PMU overflow packet characteristics are:

Purpose For each counter *n*, if the lower eight bits of that counter overflow, the associated OV*n* of the PMU overflow packet is set. If multiple counters overflow in the same period, multiple bits might be set. If there are fewer than 8 general-purpose counters, the associated PMU overflow packet bit is always zero.

Attributes 8 bit protocol packet.

Figure 7-3 ***Armv8-M Architecture Reference Manual*** **Snapshot of PMU Overflow Packet Description**

Trace packet bits are assigned as follows:

- Byte 0 bits [7:0] Packet Header
- Byte 1 bits [7:0] OVn

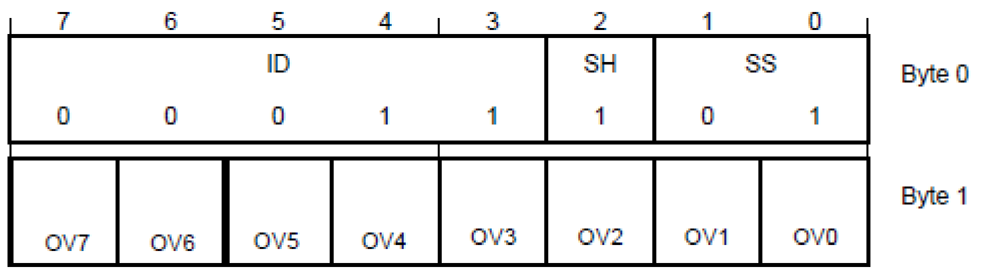


Figure 7-2 ***Armv8-M Architecture Reference Manual*** **Snapshot of PMU Overflow Packet**

7.7.8 Triggering an overflow after the core has executed code 'N' times

The PMU can also be used in conjunction with the DWT to halt the processor after a code sequence of interest, such as a loop, has executed a given number of iterations. The following sequence may be used to achieve this:

1. Decide how many times (N) you would like the loop (L) to execute before generating an overflow and initialize a 16-bit loop limit variable. N is set to 10 in the example below.
2. Configure DWT comparator <n> to match an Instruction Address, for example, a location in memory at the end of a loop.
3. Configure PMU event counter <m> to count on DWT_CMPMATCH<n>.
4. Set PMU event counter <m> to -N.
5. Decide how to handle the overflow and take appropriate action (see previous sections 7.7.x).
6. Enable event counter <m>.
7. Execute loop (L).

After the instruction (loop) being watched executes 'N' times, an overflow will occur.

This mechanism can also be used with chained event counters, as described in section 7.6 *Chaining Event Counters to create a 32-bit Counter* to trigger an overflow of the even numbered counter.

The following example code shows how to achieve this scenario where N is set to 10:

```
/* 1) Initialize 16-bit loop counter */
int16_t N = 10;

/* 2) Configure DWT Comparator 0 to watch for a PC value of 0x2660 */
DWT->FUNCTION0 = 0x402;     // Set DATAVSIZE bits [11:10] to 0b01
                            // MATCH bits [3:0] to 0b10

DWT->COMP0 = 0x00002700;    // Instruction Address, e.g., 0x2700 marks end of loop

/* 3) Configure PMU event counter 0 to count on DWT_CMPMATCH0 */
ARM_PMU_Set_EVTYPER(0, ARM_PMU_DWT_CMPMATCH0);

/* 4) Configure PMU Event Counter Register 0 to -N (0xFFF6) */
PMU->EVCNTR[0] = (uint16_t)-N)

/* 5) Generate an interrupt on a counter overflow */
ARM_PMU_Set_CNTR_IRQ_Enable(PMU_INTENSET_CNT0_ENABLE_Msk);   // See section 7.7.3
                                                             // for full details

/* 6) Enable Event Counter 0 */

ARM_PMU_CNTR_Enable(PMU_CNTENSET_CNT0_ENABLE_Msk);

/* 7) Execute code sequence of interest, e.g., loop L */
```

8 PMU Profiling Example

8.1 Traditional Loops vs Low Overhead Loops

The following example contains two simple string copy functions:

- A basic scalar example that uses a traditional counting down loop: `strcpy_scalar()`.
- A basic scalar example that uses a low-overhead-loop, `strcpy_scalar_lol()`.

The example is written in GNU assembly language syntax, which is supported by GCC and Arm Compiler 6.

```
/* strcpy.s */

    .section .text.strcpy.scalar, "ax"
    .type strcpy_scalar, %function
    .global strcpy_scalar

strcpy_scalar:
loopStart:
     LDRB R3, [R1], #1
     STRB R3, [R0], #1
     SUBS R2, R2, #1
     BNE  loopStart
     BX LR

    .section .text.strcpy.scalar_lol, "ax"
    .type strcpy_scalar_lol, %function
    .global strcpy_scalar_lol

strcpy_scalar_lol:
     PUSH {R0,LR}
     WLS  LR, R2, lolEnd        /* While Loop Start */
lolStart:
     LDRB R3, [R1], #1
     STRB R3, [R0], #1
     LE   LR, lolStart          /* Loop End */
lolEnd:
     POP {R0,PC}

     .end
```

Both functions have been exported using the `.global` and `.type` keywords so that code from other source files may reference them.

8.2 Profiling the Assembly Example

The following C program can be used to profile the two assembly routines in section 8.1 *Traditional Loops vs Low Overhead Loops*.

The program uses the CMSIS device header file by including `CMSIS_header_file` and `RTE_Components.h`. The CMSIS device header includes the Cortex-M55 processor core header file, which provides the user with access to the CMSIS PMU API. More information about using CMSIS in an application can be found online:

- https://arm-software.github.io/CMSIS_5/Core/html/using_pg.html
- https://community.arm.com/developer/tools-software/tools/b/tools-software-ides-blog/posts/using-cmsis-with-arm-compiler-6-without-an-ide

The `printf()` routines may need to be retargeted to the device that you're working with.

```
/* main.c */

#include <stdio.h>
#include "RTE_Components.h"     // include information about project configuration
#include CMSIS_device_header    // include <device>.h file

#define LENGTH 127

extern void strcpy_scalar(int8_t*, int8_t*, uint32_t);
extern void strcpy_scalar_lol(int8_t*, int8_t*, uint32_t);

__attribute__((noinline)) void init_arrays(void);

static int8_t a[LENGTH];
static int8_t b[LENGTH];

__attribute__((noinline)) void init_arrays(void)
{
  int i;

  for (i=1; i<(LENGTH+1); i++)
  {
    a[i-1] = (int8_t)i;
    b[i-1] = (int8_t)i;
  }
}

int main(void)
{
  /* Reset count variables for cycle count and retired Loop End instructions */
  uint32_t cycle_count = 0;
  uint32_t le_retired_count = 0;

  /* Initialize character array */
  init_arrays();

  /* Enable Trace */
  CoreDebug->DEMCR |= CoreDebug_DEMCR_TRCENA_Msk;

  /* Enable Low Overhead Loops */
  SCB->CCR |= SCB_CCR_LOB_Msk;

  /* Configure Event Counter Register 0 to count retired Loop End instructions */
  ARM_PMU_Set_EVTYPER(0, ARM_PMU_LE_RETIRED);

  /* Reset Cycle Counter and Event Counters */
  ARM_PMU_CYCCNT_Reset();
  ARM_PMU_EVCNTR_ALL_Reset();

  /* Enable Cycle Counter and Event Counter Register 0 */
  ARM_PMU_CNTR_Enable(PMU_CNTENSET_CCNTR_ENABLE_Msk|PMU_CNTENSET_CNT0_ENABLE_Msk);

  /* Enable the PMU */
  ARM_PMU_Enable();

  /* Call traditional scalar strcpy */
  strcpy_scalar(a, b, LENGTH);
  /* Disable Cycle Counter and Event Counter Register 0 */
  ARM_PMU_CNTR_Disable(PMU_CNTENCLR_CCNTR_ENABLE_Msk|PMU_CNTENSET_CNT0_ENABLE_Msk);

  /* Read Cycle Counter and Event Counter Register 0 */
  cycle_count = cycle_count + ARM_PMU_Get_CCNTR();
  le_retired_count = le_retired_count + ARM_PMU_Get_EVCNTR(0);

  /* Print results */
  printf("Cycles for strcpy_scalar = %d\n"
         "Loop End instructions retired = %d\n",cycle_count, le_retired_count);

  /* Reset Cycle Counter and Event Counters again */
  ARM_PMU_EVCNTR_ALL_Reset();
  ARM_PMU_CYCCNT_Reset();

  /* Enable Cycle Counter and Event Counter Register 0 again */
  ARM_PMU_CNTR_Enable(PMU_CNTENCLR_CCNTR_ENABLE_Msk|PMU_CNTENSET_CNT0_ENABLE_Msk);

  /* Call scalar strcpy with low overhead loop */
  strcpy_scalar_lol(a, b, LENGTH);

  /* Disable Cycle Counter and Event Counter Register 0 again */
  ARM_PMU_CNTR_Disable(PMU_CNTENCLR_CCNTR_ENABLE_Msk|PMU_CNTENSET_CNT0_ENABLE_Msk);

  /* Reset count variables for cycle count and retired Loop End instructions */
  cycle_count = 0;
  le_retired_count = 0;

  /* Read Cycle Counter and Event Counter Register 0 again */
  cycle_count = cycle_count + ARM_PMU_Get_CCNTR();
  le_retired_count = le_retired_count + ARM_PMU_Get_EVCNTR(0);

  printf("Cycles for strcpy_scalar_lol = %d\n"
         "Loop End instructions retired = %d\n",cycle_count, le_retired_count);

  return 0;
}
```

Running this code on Cortex-M55 RTL where instructions and data are stored in TCMs prints something similar to the following:

```
Cycles for strcpy_scalar = 658
Loop End instructions retired = 0
Cycles for strcpy_scalar_lol = 285
Loop End instructions retired = 1
```

Running the same code on a Cortex-M55 FVP prints something similar to the following:

```
Cycles for strcpy_scalar = 530
Loop End instructions retired = 0
Cycles for strcpy_scalar_lol = 269
Loop End instructions retired = 1
```

8.3 Summary of Results

The results show that low overhead loops can significantly improve the performance of an application and sometimes more than double the performance of small loop routines. This simple optimization only scratches the surface of the new and powerful Armv8.1-M instruction set enhancements.

The Fast Model results show that the FVP correctly counted the number of retired Loop End instructions, showing that it can provide a quick and convenient way to run functionally accurate simulations. The cycle count information is approximate and is more aligned with the total number of instructions retired, rather than the actual cycle count shown by an RTL simulator or hardware platform. A cycle accurate model would be an alternative solution where accuracy is required.