ARM® Cortex®-M23 Processor User Guide

Revision: r1p0

Reference Material

Confidential

ARM®

ARM DUI 0963B (ID111716)

ARM Cortex-M23 Processor User Guide
Reference Material

Release Information

The following changes have been made to this book.

Change history

Date	Issue	Confidentiality	Change
29 July 2016	A	Confidential	First release for r1p0
18 November 2016	B	Confidential	Second release for r1p0

Proprietary Notice

ARM Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

Confidentiality Status

This document is Confidential. This document may only be used and distributed in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Product Status

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

Web Address

`http://www.arm.com`

Contents
Cortex-M23 User Guide Reference Material

Preface

This preface introduces the *Reference Material for a Cortex-M23 processor User Guide*. It contains the following sections:

About the reference material

This document provides reference material that you, as an ARM partner, can configure and include in a *User Guide* for an ARM Cortex-M23 processor. Typically:

- Each chapter in this reference material might correspond to a section in the User Guide.
- Each top-level section in this reference material might correspond to a chapter in the User Guide.

However, you can organize this material in any way, subject to the conditions of the license agreement under which ARM supplied the material and subject to the incorporation of the following ARM copyright notice to any section or chapter taken from the User Guide Reference material ‘Copyright © 2016 ARM’.

Note

ARM suggests User Guide as a suitable title for a book based on this material, but partners can choose a different name for their document.

Figures in the reference material are in `.svg` format, and ARM supplies a source and an object file for each figure. The filename of each graphic matches the title of the corresponding figure.

Cortex-M23 processor options

You must customize the information in this document to produce your own user documentation, as the following sections describe:

- *Cortex-M23 processor implementation options*.
- *Microcontroller system-level configuration options* on page x.

In addition, you might want to retitle your customized document, and replace references to Cortex-M23 to refer to your product name.

Conventions used in the reference material on page xi describes how this document indicates material that is optional or that you must configure.

Cortex-M23 processor implementation options

The following table shows the Cortex-M23 processor implementation options.

Effects of the Cortex-M23 processor implementation options

Option	Description, and affected documentation
Inclusion of MPU	You can implement the Cortex-M23 processor with or without a *Memory Protection Unit* (MPU). The number of MPU regions is configurable to 0, 4, 8, 12, or 16. This affects references to the MPU or MPU registers in: • *About the Cortex-M23 processor and core peripherals* on page 1-3. • *Memory regions, types, and attributes* on page 2-12. • *Behavior of memory accesses* on page 2-15, after Table 2-10 on page 2-15. • *Exception types* on page 2-22, in the description of *MemManage*. • *Fault handling* on page 2-32. • Footnote [a] to Table 2-10 on page 2-15. • Table 4-1 on page 4-2. Include either: — The row for `0xE000ED90-0xE000ED93`, MPU Type Register, reads as zero. — The row for `0xE000ED90-0xE000EDB8`, Memory Protection Unit. Also, include or omit *Security Attribution and Memory Protection* on page 4-29.
Number of interrupts	You must decide how many interrupts your Cortex-M23 processor implementation supports, in the range 0-239. This affects: • The maximum value of ISR_NUMBER in Table 2-6 on page 2-7. • Entries in the last row of Table 2-13 on page 2-23, particularly if you implement no interrupts. • The maximum interrupt number, and associated information where appropriate, in: — *Exception handlers* on page 2-24. — Figure 2-1 on page 2-25. — *Nested Vectored Interrupt Controller* on page 4-3. • The number of implemented Nested Vectored Interrupt Controller (NVIC) registers in: — *NVIC register summary* on page 4-3. — The appropriate register descriptions in sections *Interrupt Set-enable Registers* on page 4-5to *Interrupt Priority Registers* on page 4-8. • *Vector Table Offset Register* on page 4-15. See the configuration information in the section for guidance on the required configuration.

Effects of the Cortex-M23 processor implementation options (continued)

Option	Description, and affected documentation
Inclusion of the WIC	You can implement the Cortex-M23 processor with or without a *Wakeup Interrupt Controller* (WIC). This affects references to the WIC in *About the Cortex-M23 processor and core peripherals* on page 1-3. Also, include or omit: • The description of deep sleep mode and the selection of the sleep mode in the introduction to the section *Power management* on page 2-34. • The section *Wakeup Interrupt Controller* on page 2-35.
Sleep mode power-saving	You must decide the power-saving options implemented in the sleep modes. This means you must add a description of the implemented sleep modes in the section *Power management* on page 2-34, and reference this information from the introduction to this section. Sleep mode power saving might also affect SysTick behavior, and you might have to revise the description in *SysTick usage hints and tips* on page 4-28.
Endianness	The memory system can be either little-endian or big-endian. This affects: • Descriptions of endianness in: — *Data types* on page 2-11. — The introductory paragraph in *Memory endianness* on page 2-17. Include either *Byte-invariant big-endian format* on page 2-18 or *Little-endian format* on page 2-18. but not both. • The endianness is defined by your implementation and depends on the external signal, **CFGBIGEND**. This means you must configure the endianness bit in Table 4-15 on page 4-16.
Memory features	Some features of the memory system are implementation-specific. This affects details of vendor-specific memory in *Memory model* on page 2-12, including: • *Cortex-M23 processor implementation* on page 1-3. • Information in Table 2-10 on page 2-15.
SysTick timer	You can choose to implement 0, 1, or 2 SysTick timers. **Note** If you implement only one SysTick timer, with the ARMv8-M Security Extension, then the SysTick timer has a specific behavior. The SysTick timer and its SYST_CALIB register are implementation-defined. This means you must configure the description of the SysTick timer and its SYST_CALIB register according to your implementation, or omit the description if you believe it is not useful to your customers. This affects: • *System timer, SysTick* on page 4-25. • The entry for SYST_CALIB in Table 4-23 on page 4-25. • *SysTick Calibration Value Register* on page 4-27.
Inclusion of ARMv8-M Security Extension	You can implement the Cortex-M23 processor with or without the ARMv8-M Security Extension, and some instructions can be omitted when the Security Extension is not implemented. This affects: • MPU, include or omit *Security Attribution and Memory Protection* on page 4-29. • `BXNS, BLXNS, and SG` in *BXNS and BLXNS* on page 3-51and *SG* on page 3-63.

Microcontroller system-level configuration options

The Cortex-M23 processor includes various system-level implementation options that are determined by the silicon vendor. In general, ARM cannot document these options. ARM recommends that the following information is added to the manual.

System-level configuration options

Option	Description
Interrupt assignment	Add details of the assignment of interrupts. You might add this in the section *Exception model* on page 2-22 possibly in Table 2-13 on page 2-23.
Interrupt type	Define the assigned interrupts as *level-sensitive* or *pulse*. You might include this with your interrupt assignment information. Configure *Level-sensitive and pulse interrupts* on page 4-9 to reference this information.
NMI assignment	Add details of the assignment of the *Non-Maskable Interrupt* (NMI). You might add this in the section *Exception model* on page 2-22 in Table 2-13 on page 2-23.
Event input	Add details of the assignment of the Event input, possibly in the section *External event input* on page 2-35. This can also affect references to the external event in: • *Entering sleep mode* on page 2-34. • *Wakeup from sleep mode* on page 2-35.
Event output	Executing an `SEV` instruction can generate a pulse on the **TXEV** output. However, use of this signal is implementation-defined. Add a description of the use of this signal if required. You might add this to the section *Power management* on page 2-34.
System reset request output	The description of the SYSRESETREQ bit, in *Application Interrupt and Reset Control Register* on page 4-16, indicates that there is a system reset request signal. Configure this description according to your implementation. You might add more information about the use of this signal in your implementation.
SysTick **STCALIB** input	Use of this signal is implementation-defined, so it is not described in the documentation. It affects: • The reset value of the CLKSOURCE bit of the SYST_CSR, see *SysTick Control and Status Register* on page 4-25. This might affect the reset value of the SYST_CSR in Table 4-23 on page 4-25. • SysTick register descriptions, in the section *System timer, SysTick* on page 4-25, and register reset values in Table 4-23 on page 4-25. The reset values and register descriptions are affected in both Secure and Non-secure states, and depend on the number of SysTicks implemented.

Conventions used in the reference material

Information that you must configure is highlighted as:

Configurable

Identifies text that describes a value or feature of the processor that you can configure at the implementation stage. Sometimes the reference material includes multiple sentences highlighted as configurable and you can choose the sentence that corresponds to your implementation.

In a few cases, configurable text includes an indication of the required configuration, enclosed in angle brackets, < >.

Optional Identifies text that describes an optional feature of the processor that you can choose whether to implement.

In the FrameMaker source files this highlighting is applied either:

- By assigning the value *Configurable* or *Optional* to the *Condition* attribute of a FrameMaker element.
- By wrapping the required text in the *Phrase* element, and then assigning the required value to the *Condition* attribute of that element.

Where features in a figure are colored to identify them as configurable or optional, you must update the source file to correspond to your implementation, and then regenerate the `.svg` object file by resaving the file as `.svg`.

All ARM documents include all the information in *Typographical conventions* on page 1-2 in the preface. However, some of these conventions are not used in this reference material, and these are marked as Optional.

Using this material

This book is organized into the following chapters:

Chapter 1 *Introduction, Reference Material*.

Use information from this chapter in the introductory material for your document. This chapter includes:

- A description of conventions used in the supplied material.
- An overview of the Cortex-M23 processor.

Chapter 2 *The Cortex-M23 Processor, Reference Material*.

Configure the information in this chapter to provide your description of the processor.

Chapter 3 *The Cortex-M23 Instruction Set, Reference Material*.

Configure the information in this chapter to provide your description of the instruction set supported by the processor.

Chapter 4 *Cortex-M23 Peripherals, Reference Material*.

Configure the information in this chapter to provide your description of the peripherals that are integrated with the processor.

Glossary

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

See *ARM Glossary*, http://infocenter.arm.com/help/topic/com.arm.doc.aeg0014-/index.html.

Feedback

ARM welcomes feedback on this product and its documentation.

Feedback on this product

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

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to `errata@arm.com`. Give:

- The title.
- The number, ARM DUI 0963B.
- The page numbers to which your comments apply.
- A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Note

ARM tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Chapter 1
Introduction, Reference Material

The following sections are the reference material for the introduction to a Cortex-M23 User Guide:

- *About this document* on page 1-2. An ARM document includes this information in the document preface.
- *About the Cortex-M23 processor and core peripherals* on page 1-3. An ARM document includes this information in chapter 1 of the document.

1.1 About this document

This document provides the information required for application and system-level software development. It does not provide information on debug components, features, or operation.

This material is for microcontroller software and hardware engineers, including those who have no experience of ARM products.

1.1.1 Typographical conventions

The typographical conventions used in this document are:

italic	Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
bold	Highlights interface elements, such as menu names. Denotes signal names. Used for terms in descriptive lists, where appropriate.
`monospace`	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
`mono`space	Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`monospace italic`	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
`monospace bold`	Denotes language keywords when used outside example code.
< and >	Enclose replaceable terms for assembler syntax where they appear in code or code fragments. For example: `CMP Rn, <Rm|#imm>`

1.2 About the Cortex-M23 processor and core peripherals

The Cortex-M23 processor is an entry-level 32-bit ARM Cortex processor designed for a broad range of embedded applications. It offers significant benefits to developers, including:

- A simple architecture that is easy to learn and program.
- Ultra-low power, energy-efficient operation.
- Excellent code density.
- Deterministic, high-performance interrupt handling.
- Upward compatibility with Cortex-M processor family.
- Platform security robustness, with integrated Memory Protection.
- Extended security features with Security Extension for ARMv8-M.

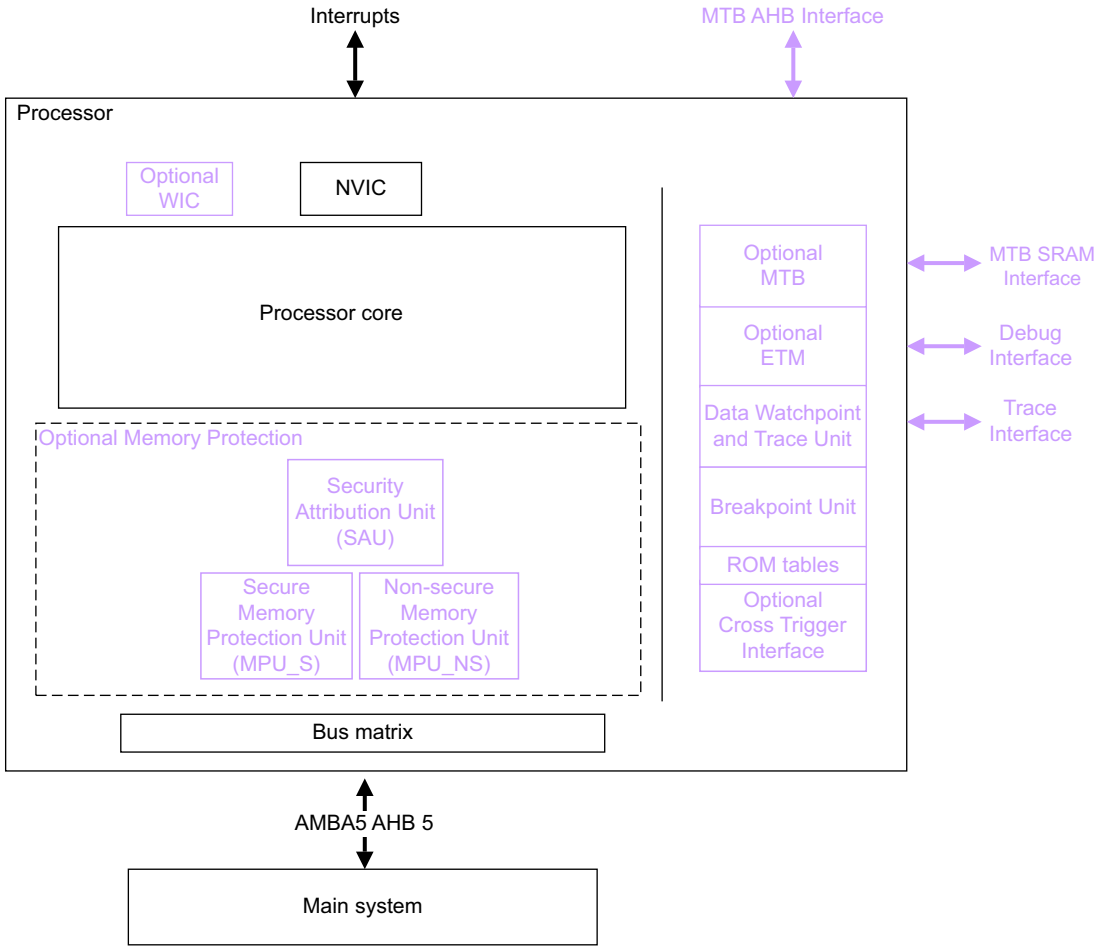


Figure 1-1 Cortex-M23 processor implementation

The Cortex-M23 processor is built on a highly area and power optimized 32-bit processor core, with a 2-stage pipeline von Neumann architecture. The processor delivers high energy efficiency through a small but powerful instruction set and extensively optimized design, providing high-end processing hardware including a single-cycle multiplier and a 17-cycle divider.

For each security state, the Cortex-M23 processor implements the baseline profile of the ARMv8-M architecture, which is based on the 32-bit Thumb® instruction set and includes Thumb-2 technology. This provides the exceptional performance expected of a modern 32-bit architecture, with a higher code density than other 8-bit and 16-bit microcontrollers.

The Cortex-M23 processor closely integrates a configurable *Nested Vectored Interrupt Controller* (NVIC), to deliver industry-leading interrupt performance. The NVIC:

- Includes a *Non-Maskable Interrupt* (NMI).
- Provides a zero jitter interrupt option.
- Provides four programmable priority levels, and additional levels for NMI and Hardfault.

The tight integration of the processor core and NVIC provides fast execution of *Interrupt Service Routines* (ISRs), significantly reducing the interrupt latency. This is achieved through the hardware stacking of registers, and the ability to abandon load-multiple and store-multiple operations. Interrupt handlers do not require any assembler wrapper code, removing any code overhead from the ISRs. Tail-chaining optimization also significantly reduces the overhead when switching from one ISR to another.

To optimize low-power designs, the NVIC integrates with the sleep modes which include a deep sleep function that enables the entire device to be deeply powered down.

1.2.1 Cortex-M23 processor features summary

- Thumb® instruction set with Thumb-2 Technology.
- High code density with 32-bit performance.
- Unprivileged and Privileged access.
- Tools and binaries upwards compatible with Cortex-M processor family.
- Integrated ultra low-power sleep modes.
- Efficient code execution enabling slower processor clock or increased sleep time.
- Single-cycle 32-bit hardware multiplier and fast 17-cycle hardware divider.
- Zero jitter interrupt handling.
- *Security Attribution Unit* (SAU) for security management.
- *Memory Protection Unit* (MPU) for safety-critical applications.
- Low latency, high-speed peripheral I/O port.
- A Vector Table Offset Register, which is banked between Secure and Non-secure state.
- Extensive debug capabilities.

1.2.2 System-level interface

The Cortex-M23 processor implements a complete hardware debug solution. This provides high system visibility of the processor and memory through either a traditional JTAG port or a 2-pin *Serial Wire Debug* (SWD) port that is ideal for microcontrollers and other small package devices. The MCU vendor determines the debug feature configuration, therefore debug features can differ across different devices and families.

The optional CoreSight technology components, *Embedded Trace Macrocell* (ETM), and *Micro Trace Buffer* (MTB), deliver unrivalled instruction trace capture in an area far smaller than traditional trace units, enabling many low-cost MCUs to implement full instruction trace for the first time.

The breakpoint unit provides up to four hardware breakpoint comparators that debuggers can use.

The data watchpoint unit provides up to four data watchpoint comparators that debuggers can use.

1.2.3 Security Extension

The Security Extension to the ARMv8-M baseline adds security and code and data protection features. The Security Extension introduces a new security state to the existing thread and handler modes. A Cortex-M23 processor with the Security Extension has two security states, Secure and Non-secure.

With the Security Extension implemented, the following happens:

- The Cortex-M23 processor always resets into Secure state.
- Some registers are banked between security states. There are two separate instances of the same register, one in Secure state and the other in Non-secure state.
- The Secure state can access Non-secure versions of banked registers through the Non-secure alias.
- Some exceptions are banked between security states, some other exceptions are configurable.
- Some faults are banked between security states.
- Secure memory can only be accessed from Secure state.

1.2.4 Cortex-M23 processor core peripherals

The Cortex-M23 core peripherals are:

NVIC The NVIC is an embedded interrupt controller that supports low latency interrupt processing.

System Control Space

The *System Control Space* (SCS) is the programmers model interface to the processor. It provides system implementation information and system control, including configuration, control, and reporting of system exceptions.

System Timer

The System Timer, SysTick, is a 24-bit count-down timer. Use this as a *Real Time Operating System* (RTOS) tick timer or as a simple counter.

<Use the following statement to suit your implementation that supports the ARMv8-M Security Extension:

Either one configurable SysTick is implemented, or two SysTicks banked between security states are implemented.>

<Use the following statement to suit your implementation that does not support the ARMv8-M Security Extension:

Either no SysTicks are implemented, or one SysTick is implemented.>

Security Attribution Unit

The SAU determines the security of an address.

Memory Protection Unit

The MPU improves system reliability by defining the memory attributes for different memory regions. It provides up to eight different regions, and an optional predefined background region.

<Remove or edit the following to suit your implementation:

There are two MPUs, one for Secure state and one for Non-secure state. Each MPU can define memory access permissions and attributes independently.

I/O port The I/O port provides single-cycle loads and stores to tightly-coupled peripherals.

1.2.5 ARMv8-M enablement

Although the following documents are not specific to this product, they do contain information that might enable you in developing your Cortex-M23 processor.

- ARMv8-M Processor Debug.
- ACLE Extensions for ARMv8-M.
- Fault Handling and Detection.
- ARMv8-M Exception Handling.
- Memory Protection Unit for ARMv8-M based platforms.
- ARM®v8-M Architecture Reference Manual.
- TrustZone® technology for ARMv8-M Architecture.
- Introduction to the ARMv8-M Architecture.

Chapter 2
The Cortex-M23 Processor, Reference Material

The following sections are the reference material for the Cortex-M23 processor description in a User Guide:

2.1 Programmers model

This section describes the programmers model. In addition to the individual core register descriptions, it contains information about the processor modes, privilege levels for software execution, security states, and stacks.

2.1.1 Processor modes and privilege levels for software execution

The processor *modes* are:

Thread mode Executes application software. The processor enters Thread mode on Reset, or as a result of an exception return.

Handler mode Handles exceptions. The processor returns to Thread mode when it has finished all exception processing.

The *privilege levels* for software execution are:

Unprivileged The software:

- Has limited access to system registers using the `MSR` and `MRS` instructions, and cannot use the `CPS` instruction to mask interrupts.
- Cannot access the system timer, NVIC, or system control block.
- Might have restricted access to memory or peripherals.

Unprivileged software executes at the unprivileged level.

Privileged Software can use all the instructions and has access to all resources. *Privileged software* executes at the privileged level.

In Thread mode, the CONTROL register controls whether software execution is privileged or unprivileged, see *CONTROL register* on page 2-9. In Handler mode, software execution is always privileged.

Only privileged software can write to the CONTROL register to change the privilege level for software execution in Thread mode. Unprivileged software can use the `SVC` instruction to make a *Supervisor Call* to transfer control to privileged software.

2.1.2 Security states

The programmers model includes the following security states:

Secure state The processor always resets into Secure state.

Non-secure state The programmers model includes only the Non-secure state.

Registers in the System Control Space are banked across Secure and Non-secure states, with the Non-secure register view available at an aliased address to Secure state.

Each security state includes a set of independent operating modes and supports both privileged and unprivileged user access.

2.1.3 Stacks

The processor uses a full descending stack. This means the Stack Pointer indicates the last stacked item on the stack memory. When the processor pushes a new item onto the stack, it decrements the Stack Pointer and then writes the item to the new memory location. The processor implements two stacks per security state, the *main stack* and the *process stack*, with independent copies of the Stack Pointer, see *Stack Pointer* on page 2-4.

In Thread mode, the CONTROL register controls whether the processor uses the main stack or the process stack, see *CONTROL register* on page 2-9. In Handler mode, the processor always uses the main stack. The options for processor operations are:

Table 2-1 Summary of processor mode, execution privilege level, and stack use options

Processor mode	Used to execute	Privilege level for software execution	Stack used
Thread	Applications	Privileged or unprivileged[a]	Main stack or process stack[a]
Handler	Exception handlers	Always privileged	Main stack

a. See *CONTROL register* on page 2-9.

2.1.4 Core registers

The processor core registers are:

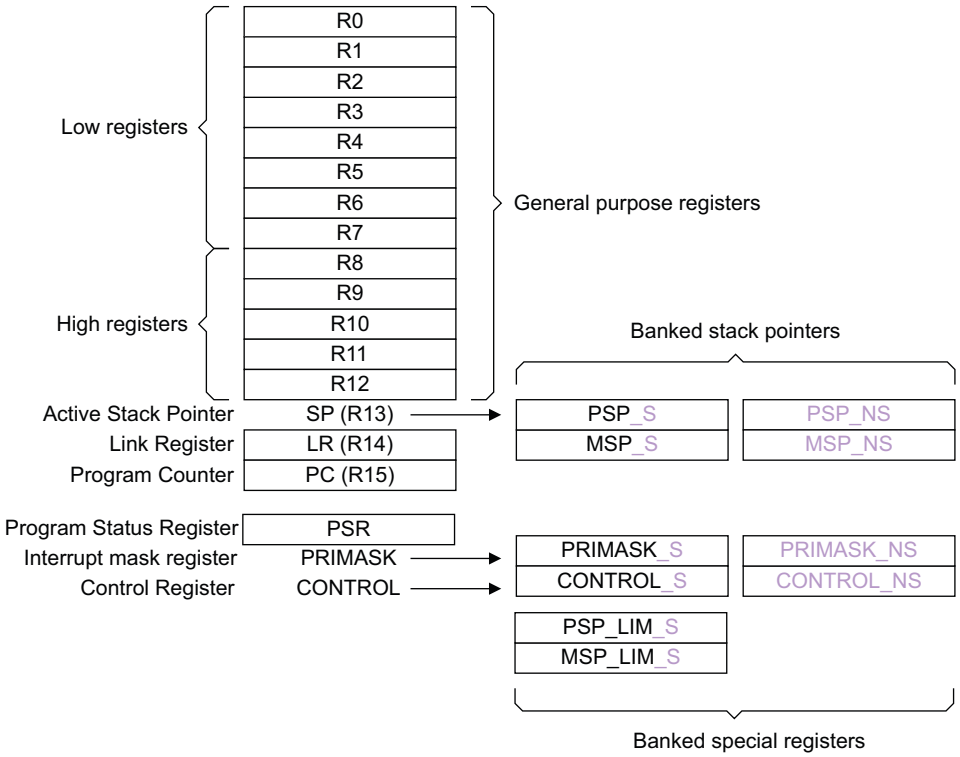


Table 2-2 Core register set summary

Name	Type[a]	Reset value	Description
R0-R12	RW	Unknown	*General-purpose registers* on page 2-4.
MSP_S	RW	See description	*Stack Pointer* on page 2-4.
MSP_NS			

Table 2-2 Core register set summary (continued)

Name	Type[a]	Reset value	Description
PSP_S	RW	Unknown	
PSP_NS			
LR	RW	Unknown	*Link Register* on page 2-5
PC	RW	See description	*Program Counter* on page 2-5.
PSR[b]	RW	Unknown[c]	*Program Status Register* on page 2-5.
APSR	RW	Unknown	*Application Program Status Register* on page 2-6.
IPSR	RO	0x00000000	*Interrupt Program Status Register* on page 2-7.
EPSR	RO	Unknown[c]	*Execution Program Status Register* on page 2-7.
PRIMASK_S	RW	0x00000000	*Priority Mask Register* on page 2-9.
PRIMASK_NS			
CONTROL_S	RW	0x00000000	*CONTROL register* on page 2-9.
CONTROL_NS			

a. Describes the access type during program execution in Thread mode and Handler mode. Debug access can differ.

b. PSR includes APSR, IPSR, and EPSR.

c. Bit[24] is the T-bit and is loaded from bit[0] of the reset vector.

General-purpose registers

R0-R12 are 32-bit general-purpose registers for data operations.

Stack Pointer

The *Stack Pointer* (SP) is register R13.

There are four stacks and four Stack Pointer registers banked between Secure and Non-secure state.

Table 2-3 Stack Pointer register

Stack		Stack Pointer register	Stack Pointer Limit register
Secure	Main	MSP_S	MSPLIM
	Process	PSP_S	PSPLIM
Non-secure	Main	MSP_NS	-
	Process	PSP_NS	-

In Thread mode, bit[1], CONTROL.SPSEL, of the CONTROL register indicates the Stack Pointer to use:

- 0 = *Main Stack Pointer* (MSP). This is the reset value.
- 1 = *Process Stack Pointer* (PSP).

Link Register

The *Link Register* (LR) is register R14. It stores the return information for subroutines, function calls, and exceptions. On reset, the LR value is Unknown.

Program Counter

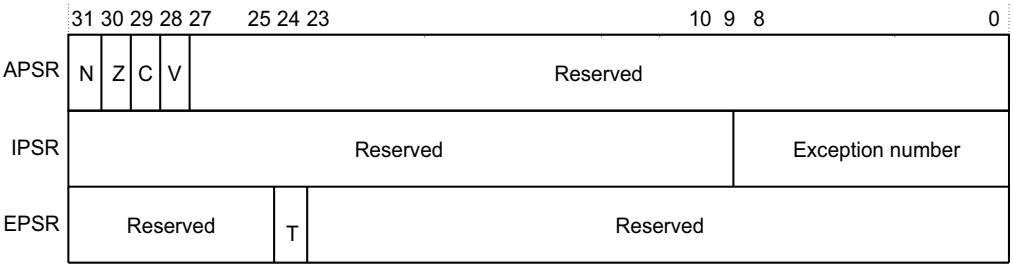
The *Program Counter* (PC) is register R15. It contains the current program address. On reset, the processor loads the PC with the value of the reset vector, which is at address `0x00000004`. Bit[0] of the value is loaded into the EPSR T-bit at reset and must be 1.

Program Status Register

The *Program Status Register* (PSR) combines:

- *Application Program Status Register* (APSR).
- *Interrupt Program Status Register* (IPSR).
- *Execution Program Status Register* (EPSR).

These registers are allocated as mutually exclusive bit fields within the 32-bit PSR. The PSR bit assignments are:



Access these registers individually or as a combination of any two or all three registers, using the register name as an argument to the `MSR` or `MRS` instructions. For example:

- Read all the registers using `PSR` with the `MRS` instruction.
- Write to the APSR N, Z, C, and V bits using `APSR` with the `MSR` instruction.

The PSR combinations and attributes are:

Table 2-4 PSR register combinations

Register	Type	Combination
PSR	RW[a, b]	APSR, EPSR, and IPSR.
IEPSR	RO	EPSR and IPSR.
IAPSR	RW[a]	APSR and IPSR.
EAPSR	RW[b]	APSR and EPSR.

a. The processor ignores writes to the IPSR bits.
b. Reads of the EPSR bits return zero, and the processor ignores writes to these bits.

See the instruction descriptions *MRS* on page 3-59 and *MSR* on page 3-60 for more information about how to access the Program Status Registers.

Application Program Status Register

The APSR contains the current state of the condition flags, from previous instruction executions. See the register summary in Table 2-2 on page 2-3 for its attributes. The bit assignments are:

Table 2-5 APSR bit assignments

Bits	Name	Function
[31]	N	Negative flag.
[30]	Z	Zero flag.
[29]	C	Carry or borrow flag.
[28]	V	Overflow flag.
[27:0]	-	Reserved.

See *The condition flags* on page 3-13 for more information about the APSR negative, zero, carry or borrow, and overflow flags.

Interrupt Program Status Register

The IPSR contains the exception number of the current ISR. See the register summary in Table 2-2 on page 2-3 for its attributes. The bit assignments are:

Table 2-6 IPSR bit assignments

Bits	Name	Function
[31:6]	-	Reserved. <See the configurable information after this table for information about the configuration of this field and the Exception number field that follows.>
[5:0]	Exception number	This is the number of the current exception: 0 = Thread mode. 1 = Reserved. This exception number is used when Secure code calls a Non-secure function and Secure code was executing in handler mode. 2 = NMI. 3 = HardFault. 4-10 = Reserved. 11 = SVCall. 12, 13 = Reserved. 14 = PendSV. 15 = SysTick | Reserved. 16 = IRQ0. . . 255 = IRQ239. See *Exception types* on page 2-22 for more information.

<Configure the next statement to give the information required for your implementation, the statement reminds you of how to determine the alignment requirement.> The last bit of the Exception number bit field depends on the number of interrupts implemented.

0-47 interrupts = [5:0].

48-111 interrupts = [6:0].

112-239 interrupts = [7:0].

Execution Program Status Register

The EPSR contains the Thumb state bit.

See the register summary in Table 2-2 on page 2-3 for the EPSR attributes. The bit assignments are:

Table 2-7 EPSR bit assignments

Bits	Name	Function
[31:25]	-	Reserved.
[24]	T	Thumb state bit.
[23:0]	-	Reserved.

Attempts by application software to read the EPSR directly using the `MRS` instruction always return zero. Attempts to write the EPSR using the `MSR` instruction are ignored. The following can clear the T bit to 0:

- Instructions `BLX`, `BX` and, `POP{PC}`.
- Restoration from the stacked xPSR value on an exception return.
- Bit[0] of the vector value on an exception entry.

Attempting to execute instructions when the T bit is 0 results in a HardFault or Lockup. See *Lockup* on page 2-33 for more information.

Interruptible-restartable instructions

The interruptible-restartable instructions are `LDM` and `STM`, `PUSH`, `POP`, `SDIV`, `UDIV`, and `MULS` <if 32-cycle multiplier is used>. When an interrupt occurs during the execution of one of these instructions, the processor abandons execution of the instruction. After servicing the interrupt, the processor restarts execution of the instruction from the beginning.

Exception mask register

The exception mask register disables the handling of exceptions by the processor. Disable exceptions where they might impact on timing critical tasks or code sequences requiring atomicity.

To disable or re-enable exceptions, use the `MSR` and `MRS` instructions, or the `CPS` instruction, to change the value of PRIMASK. See *MRS* on page 3-59, *MSR* on page 3-60, and *CPS* on page 3-55 for more information.

This register is banked between security states.

Priority Mask Register

The PRIMASK register prevents activation of all exceptions with configurable priority. See the register summary in Table 2-2 on page 2-3 for its attributes. The bit assignments are:

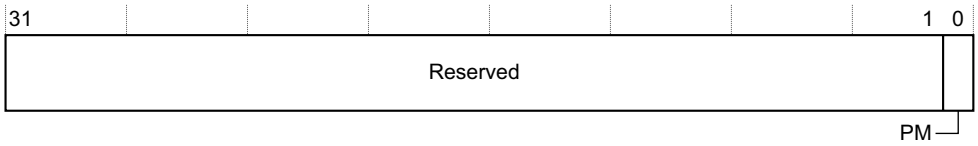


Table 2-8 PRIMASK register bit assignments

Bits	Name	Function
[31:1]	-	Reserved.
[0]	PM	Prioritizable interrupt mask: 0 = No effect. 1 = Prevents the activation of all exceptions with configurable priority.

PRIMASK_S masks all configurable interrupts.

If PRIS=0, PRIMASK_NS masks all configurable interrupts.

If PRIS=1, PRIMASK_NS masks all non-configurable interrupts and Secure configurable interrupts if their priority is 128 and 192.

CONTROL register

The CONTROL register controls the stack used, and the privilege level for software execution, when the processor is in Thread mode.

This register is banked between security states on a bit by bit basis.

See the register summary in Table 2-2 on page 2-3 for its attributes. The bit assignments are:

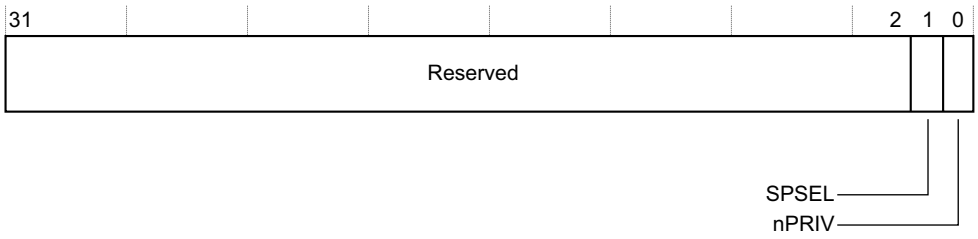


Table 2-9 CONTROL register bit assignments

Bits	Name	Function
[31:2]	-	Reserved.
[1]	SPSEL	Defines the current stack: 0 = MSP is the current Stack Pointer. 1 = PSP is the current Stack Pointer. In Handler mode this bit is ignored, the processor always uses the MSP.
[0]	nPRIV	Defines the Thread mode privilege level: 0 = Privileged. 1 = Unprivileged.

The SPSEL bit can be written at any time, but in Handler mode MSP is always used, regardless of the value of SPSEL.

In an OS environment, ARM recommends that threads running in Thread mode use the process stack and the kernel and exception handlers use the main stack.

By default, Thread mode uses the MSP. To switch the Stack Pointer used in Thread mode to the PSP, use the `MSR` instruction to set the active Stack Pointer bit to 1, see *MRS* on page 3-59.

Note

When changing the Stack Pointer, software must use an `ISB` instruction immediately after the `MSR` instruction. This ensures that instructions after the `ISB` execute using the new Stack Pointer. See *ISB* on page 3-58.

2.1.5 Exceptions and interrupts

The Cortex-M23 processor supports interrupts and system exceptions. The processor and the NVIC prioritize and handle all exceptions. An interrupt or exception changes the normal flow of software control. The processor uses Handler mode to handle all exceptions except for reset. See *Exception entry* on page 2-27 and *Exception return* on page 2-29 for more information.

The NVIC registers control interrupt handling. See *Nested Vectored Interrupt Controller* on page 4-3 for more information.

2.1.6 Data types

The processor:

- Supports the following data types:
  - 32-bit words.
  - 16-bit halfwords.
  - 8-bit bytes.
- Manages all data memory accesses as little-endian or big-endian. See *Memory regions, types, and attributes* on page 2-12 for more information.

2.1.7 The Cortex Microcontroller Software Interface Standard

ARM provides the *Cortex Microcontroller Software Interface Standard* (CMSIS) for programming microcontrollers. The CMSIS is an integrated part of the device driver library. For a Cortex-M23 microcontroller system, CMSIS defines:

- A common way to:
  - Access peripheral registers.
  - Define exception vectors.
- The names of:
  - The registers of the core peripherals.
  - The core exception vectors.
- A device-independent interface for RTOS kernels.

The CMSIS includes address definitions and data structures for the core peripherals in the Cortex-M23 processor.

The CMSIS simplifies software development by enabling the reuse of template code, and the combination of CMSIS-compliant software components from various middleware vendors. Software vendors can expand the CMSIS to include their peripheral definitions and access functions for those peripherals.

This document includes the register names defined by the CMSIS, and gives short descriptions of the CMSIS functions that address the processor core and the core peripherals.

Note

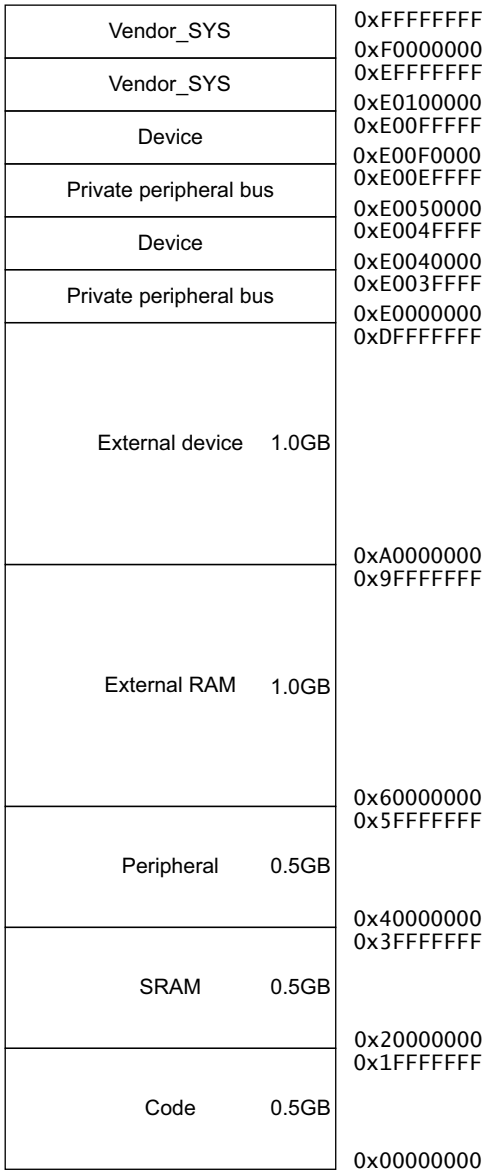
This document uses the register short names defined by the CMSIS. In a few cases, these differ from the architectural short names that might be used in other documents.

Related information:

- *Power management programming hints* on page 2-36.
- *CMSIS functions* on page 3-5.
- *Accessing the Cortex-M23 NVIC registers using CMSIS* on page 4-4.
- *NVIC programming hints* on page 4-10.

2.2 Memory model

This section describes the processor memory map and the behavior of memory accesses. The processor has a fixed memory map that provides up to 4GB of addressable memory. The memory map is:



The processor reserves regions of the *Private Peripheral Bus* (PPB) address range for core peripheral registers, see *About the Cortex-M23 processor and core peripherals* on page 1-3.

2.2.1 Memory regions, types, and attributes

The memory map and the programming of the MPU splits into regions. Each region has a defined memory type, and some regions have additional memory attributes. The memory type and attributes determine the behavior of accesses to the region.

The memory types are:

Normal The processor can re-order transactions for efficiency, or perform speculative reads.

Device The processor preserves transaction order relative to other transactions to Device or Device-GRE memory.

The additional memory attributes include:

Shareable For a shareable memory region, the memory system might provide data synchronization between bus masters in a system with multiple bus masters, for example, a processor with a DMA controller.

If multiple bus masters can access a Non-shareable memory region, software must ensure data coherency between the bus masters.

<This description is required only if the device is likely to be used in systems where memory is shared between multiple processors.>

***eXecute Never* (XN)**

Means that the processor prevents instruction accesses. A HardFault exception is generated on executing an instruction fetched from an XN region of memory.

2.2.2 Device memory

Device memory must be used for memory regions that cover peripheral control registers. Some of the optimizations that are permitted for Normal memory, such as access merging or repeating, can be unsafe for a peripheral register.

The Device memory type has several attributes:

G or nG Gathering or non-Gathering. Multiple accesses to a device can be merged into a single transaction except for operations with memory ordering semantics, for example, memory barrier instructions, load acquire/store release.

R or nR Reordering.

E or nE Early Write Acknowledge.

Only four combinations of these attributes are valid:

- Device-nGnRnE.
- Device-nGnRE.
- Device-nGRE.
- Device-GRE.

Note

- Device-nGnRnE is equivalent to ARMv7-M Strongly Ordered memory type and Device-nGnRE is equivalent to ARMv7-M Device memory.
- Device-nGRE and Device-GRE are new to ARMv8-M.

Typically, peripheral control registers must be either Device-nGnRE or Device-nGnRnE to prevent reordering of the transactions in the programming sequences.

Device-nGRE and Device-GRE memory types can be useful for peripherals where results are not affected by memory access sequence and ordering. For example, bitmap or display buffers in display interface.

If the bus interface of such a peripheral can only accept certain transfer sizes, the peripheral must be set to Device memory with non-Gathering attribute.

Note

- For most simple processor designs, reordering, and gathering (merging of transactions) do not occur even if the memory attribute configuration allows it to do so.
- Device memory is shareable, and must not be cached.

2.2.3 Secure memory system and memory partitioning

The 4GB memory space is partitioned into Secure and Non-secure memory regions.

Secure (S)

Secure addresses are used for memories and peripherals that are only accessible by Secure software or Secure masters.

Secure transactions are those that originate from masters operating as, or deemed to be, Secure when targeting a Secure address.

Non-secure Callable (NSC)

NSC is a special type of Secure location. This type of memory is the only type which an ARMv8-M processor permits to hold an `SG` instruction that enables software to transition from Non-secure to Secure state.

The inclusion of NSC memory locations removes the need for Secure software creators to allow for the accidental inclusion of `SG` instructions, or data sharing encoding values, in normal Secure memory by restricting the functionality of the SG instruction to NSC memory only.

Non-secure (NS)

Non-secure addresses are used for memory and peripherals accessible by all software running on the device.

Non-secure transactions are those that originate from masters operating as, or deemed to be, Non-secure or from Secure masters accessing a Non-secure address. Non-secure transactions are only permitted to access Non-secure addresses, and the system must ensure that Non-secure transactions are denied access to Secure addresses.

Note

Secure software that accesses memory regions marked as Non-secure in the SAU or *Implementation Defined Attribution Unit (*IDAU) is marked as Non-secure on the AHB bus.

2.2.4 Behavior of memory accesses

The behavior of accesses to each region in the memory map is:

Table 2-10 Memory access behavior

Address range	Memory region	Memory type[a]	XN[a]	Description
0x00000000-0x1FFFFFFF	Code	Normal	-	Executable region for program code. You can also put data here.
0x20000000-0x3FFFFFFF	SRAM	Normal	-	Executable region for data. You can also put code here.
0x40000000-0x5FFFFFFF	Peripheral	Device	XN	External device memory.
0x60000000-0x9FFFFFFF	RAM	Normal	-	Executable region for data.
0xA0000000-0xDFFFFFFF	External device	Device	XN	External device memory.
0xE0000000-0xE003FFFF	Private Peripheral Bus	-	XN	This region includes the SCS, NVIC, MPU, and SAU registers. Only word accesses can be used in this region.
0xE0040000-0xE004FFFF	Device	Device	XN	This region is for debug components and can include the MTB, ETM, CTI, and TPIU configuration registers or none. <Stating which ones are outside the scoop of this document, Vendor to determine>
0xE0050000-0xE00EFFFF	Private Peripheral Bus	-	XN	Reserved.
0xE00F0000-0xE00FFFFF	Device	Device	XN	This region includes the Cortex-M23 MCU ROM when implemented.
0xE0100000-0xEFFFFFFF	Vendor_SYS	-	XN	Vendor specific.
0xF0000000-0xFFFFFFFF	Vendor_SYS	Device	XN	Vendor specific.

a. See *Memory regions, types, and attributes* on page 2-12 for more information.

The Code, SRAM, and external RAM regions can hold programs.

The MPU can override the default memory access behavior described in this section. For more information, see *Security Attribution and Memory Protection* on page 4-29.

Additional memory access constraints for caches and shared memory

When a system includes caches or shared memory, some memory regions have additional access constraints, and some regions are subdivided, as Table 2-11 shows:

Table 2-11 Memory region shareability and cache policies

Address range	Memory region	Memory type[a]	Shareability[a]	Cache policy[b]
0x00000000- 0x1FFFFFFF	Code	Normal	-	WT
0x20000000- 0x3FFFFFFF	SRAM	Normal	-	WBWA
0x40000000- 0x5FFFFFFF	Peripheral	Device	-	-
0x60000000- 0x7FFFFFFF	RAM	Normal	-	WBWA
0x80000000- 0x9FFFFFFF				WT
0xA0000000- 0xBFFFFFFF	External device	Device	Shareable	-
0xC0000000- 0xDFFFFFFF			Shareable	
0xE0000000- 0xE003FFFF	Private Peripheral Bus	Device	Shareable	-
0xE0040000- 0xE004FFFF	Device	Device	-	-
0xE0050000- 0xE00EFFFF	Private Peripheral Bus	-	-	Device
0xE00F0000- 0xE00FFFFF	Device	Device	-	Device
0xE0100000- 0xEFFFFFFF	Vendor_SYS	-	-	Device
0xF0000000- 0xFFFFFFFF	Vendor_SYS	Device	-	Device

a. See *Memory regions, types, and attributes* on page 2-12 for more information.
b. WT = Write through, no write allocate. WBWA = Write back, write allocate.

2.2.5 Software ordering of memory accesses

The order of instructions in the program flow does not always guarantee the order of the corresponding memory transactions. This is because:

- Memory or devices in the memory map might have different wait states.
- Some memory accesses associated with instruction fetches are speculative.

Device memory on page 2-13 describes the cases where the memory system guarantees the order of memory accesses. Otherwise, if the order of memory accesses is critical, software must include memory barrier instructions to force that ordering. The processor provides the following memory barrier instructions:

`DMB` The *Data Memory Barrier* (DMB) instruction ensures that outstanding memory transactions complete before subsequent memory transactions. See *DMB* on page 3-56.

`DSB` The *Data Synchronization Barrier* (DSB) instruction ensures that outstanding memory transactions complete before subsequent instructions execute. See *DSB* on page 3-57.

`ISB` The *Instruction Synchronization Barrier* (ISB) ensures that the effect of any context-changing operations is recognizable by subsequent instructions. See *ISB* on page 3-58.

`LDA, LDAB, LDAEX, LDAEXB, LDAEXH, LDAH` These instructions ensure that subsequent memory transactions are observed after the load.

`STL, STLB, STLEX, STLEXB, STLEXH, STLH` These instructions ensure that outstanding memory transactions complete before the store is observed.

The following are examples of using memory barrier instructions:

Vector table If the program changes an entry in the vector table, and then enables the corresponding exception, use a `DMB` instruction between the operations. This ensures that if the exception is taken immediately after being enabled, then the processor uses the new exception vector.

Self-modifying code

If a program contains self-modifying code, use an `ISB` instruction immediately after the code modification in the program. This ensures subsequent instruction execution uses the updated program.

Memory map switching

If the system contains a memory map switching mechanism, use a `DSB` instruction after switching the memory map. This ensures subsequent instruction execution uses the updated memory map.

MPU programming

Use a `DSB` followed by an `ISB` instruction or exception return to ensure that the new MPU configuration is used by subsequent instructions.

VTOR programming

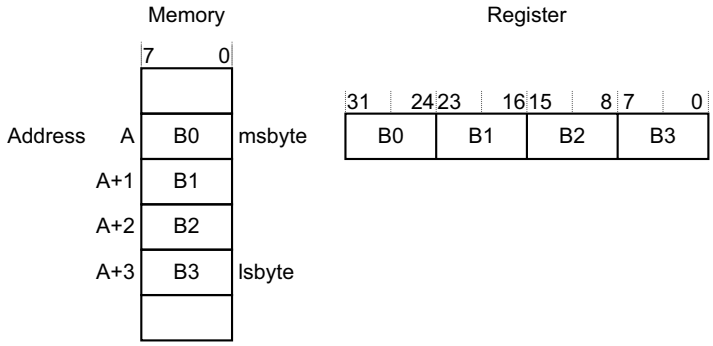
If the program updates the value of the VTOR, use a `DMB` instruction to ensure that the new vector table is used for subsequent exceptions.

2.2.6 Memory endianness

The processor views memory as a linear collection of bytes numbered in ascending order from zero. For example, bytes 0-3 hold the first stored word, and bytes 4-7 hold the second stored word. *Byte-invariant big-endian format* on page 2-18 or *Little-endian format* on page 2-18 describes how words of data are stored in memory.

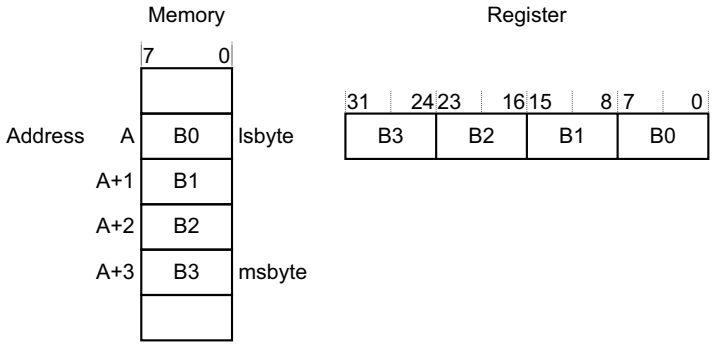
Byte-invariant big-endian format

In byte-invariant big-endian format, the processor stores the *most significant byte* (msbyte) of a word at the lowest-numbered byte, and the *least significant byte* (lsbyte) at the highest-numbered byte. For example:



Little-endian format

In little-endian format, the processor stores the *least significant byte* (lsbyte) of a word at the lowest-numbered byte, and the *most significant byte* (msbyte) at the highest-numbered byte. For example:



2.2.7 Synchronization primitives

The instruction set support for the Cortex-M23 processor includes pairs of *synchronization primitives*. These provide a non-blocking mechanism that a thread or process can use to obtain exclusive access to a memory location. Software can use them to perform a guaranteed read-modify-write memory update sequence, or for a semaphore mechanism.

A pair of synchronization primitives comprises:

A Load-Exclusive instruction

Used to read the value of a memory location, requesting exclusive access to that location.

A Store-Exclusive instruction

Used to attempt to write to the same memory location, returning a status bit to a register. If this bit is:

0 It indicates that the thread or process gained exclusive access to the memory, and the write succeeds,

1 It indicates that the thread or process did not gain exclusive access to the memory, and no write was performed.

The pairs of Load-Exclusive and Store-Exclusive instructions are:

- The word instructions:
  - LDAEX and STLEX.
  - LDREX and STREX.
- The halfword instructions:
  - LDAEXH and STLEXH.
  - LDREXH and STREXH.
- The byte instructions:
  - LDAEXB and STLEXB.
  - LDREXB and STREXB.

Software must use a Load-Exclusive instruction with the corresponding Store-Exclusive instruction.

To perform an exclusive read-modify-write of a memory location, software must:

1. Use a Load-Exclusive instruction to read the value of the location.
2. Modify the value, as required.
3. Use a Store-Exclusive instruction to attempt to write the new value back to the memory location.
4. Test the returned status bit. If this bit is:

   0 The read-modify-write completed successfully.

   1 No write was performed. This indicates that the value returned at step 1 might be out of date. The software must retry the entire read-modify-write sequence.

Software can use the synchronization primitives to implement a semaphore as follows:

1. Use a Load-Exclusive instruction to read from the semaphore address to check whether the semaphore is free.
2. If the semaphore is free, use a Store-Exclusive to write the claim value to the semaphore address.
3. If the returned status bit from step 2 indicates that the Store-Exclusive succeeded, then the software has claimed the semaphore. However, if the Store-Exclusive failed, another process might have claimed the semaphore after the software performed step 1.

The Cortex-M23 processor includes an exclusive access monitor, that tags the fact that the processor has executed a Load-Exclusive instruction. If the processor is part of a multiprocessor system, includes a global monitor, and the address is in a shared region of memory, then the system also globally tags the memory locations that are addressed by exclusive accesses by each processor.

Note

Shared region of memory: Accesses to Device regions in the ranges 0x40000000-0x5fffffff and 0xc0000000-0xffffffff do not use the Global Exclusive Monitor when ACTLR.EXTEXCLALL is 0 and the default memory map is used.

The processor removes its exclusive access tag if:

- It executes a CLREX instruction.
- It executes a STREX instruction, regardless of whether the write succeeds.

- An exception occurs. This means that the processor can resolve semaphore conflicts between different threads.

In a multiprocessor implementation:

- Executing a CLREX instruction removes only the local exclusive access tag for the processor.
- Executing a STREX instruction, or an exception, removes the local exclusive access tags for the processor.
- Executing a STREX instruction to a Shareable memory region can also remove the global exclusive access tags for the processor in the system.

For more information about the synchronization primitive instructions, see *LDREX and STREX* on page 3-24 and *CLREX* on page 3-17.

Global monitor access can be done:

- In a Shared region if the MPU is implemented, or in the default memory map.

  Note

  Default memory map: Accesses to Device regions in the ranges 0x40000000-0x5fffffff and 0xc0000000-0xffffffff do not use the Global Exclusive Monitor when ACTLR.EXTEXCLALL is 0 and the default memory map is used.

- By setting ACTLR.EXTEXLALL. In this case, exclusive information is always sent externally.

In any other case, exclusive information is not sent on the AHB bus, HEXCL is 0, and only the local monitor is used.

If HEXCL is sent externally and there is no exclusive monitor for the corresponding memory region, then STREX fails.

2.2.8 Programming hints for the synchronization primitives

ISO/IEC C cannot directly generate the exclusive access instructions. CMSIS provides intrinsic functions for generation of these instructions:

Table 2-12 CMSIS functions for exclusive access instructions

Instruction	CMSIS function
LDAEX	`uint16_t __LDAEX (volatile uint16_t * ptr)`
LDAEXB	`uint8_t __LDAEXB (volatile uint8_t * ptr)`
LDAEXH	`uint16_t __LDAEXH (volatile uint16_t * ptr)`
LDREX	`uint32_t __LDREXW (uint32_t *addr)`
LDREXB	`uint8_t __LDREXB (uint8_t *addr)`
LDREXH	`uint16_t __LDREXH (uint16_t *addr)`
STLEX	`uint16_t __STLEX (uint16_t value, volatile uint16_t * ptr)`
STLEXB	`uint8_t __STLEXB (uint8_t value, volatile uint8_t * ptr)`
STLEXH	`uint16_t __STLEXH (uint16_t value, volatile uint16_t * ptr)`

Table 2-12 CMSIS functions for exclusive access instructions (continued)

Instruction	CMSIS function
STREX	uint32_t __STREXW (uint32_t value, uint32_t *addr)
STREXB	uint8_t __STREXB (uint8_t value, uint8_t *addr)
STREXH	uint16_t __STREXH (uint16_t value, uint16_t *addr)
CLREX	void __CLREX (void)

For example:

```
uint16_t  value;
uint16_t *address = 0x20001002;
value = __LDREXH (address);    // load 16-bit value from memory address 0x20001002
```

2.3 Exception model

This section describes the exception model.

2.3.1 Exception states

Each exception is in one of the following states:

Inactive The exception is not active and not pending.

Pending The exception is waiting to be serviced by the processor.

An interrupt request from a peripheral or from software can change the state of the corresponding interrupt to pending.

Active An exception that is being serviced by the processor but has not completed.

Note

An exception handler can interrupt the execution of another exception handler. In this case, both exceptions are in the active state.

Active and pending

The exception is being serviced by the processor and there is a pending exception from the same source.

2.3.2 Exception types

The exception types are:

Reset Reset is invoked on powerup or a Warm reset. The exception model treats reset as a special form of exception. When reset is asserted, the operation of the processor stops, potentially at any point in an instruction. When reset is deasserted, execution restarts from the address provided by the reset entry in the vector table. Execution restarts as privileged execution in Thread mode.

This exception is not banked between security states. With the Security Extension implemented, the processor starts in Secure state.

NMI A *Non-Maskable Interrupt* (NMI) can be signaled by a peripheral or triggered by software.

NMIs are superseded by Secure HardFault at priority -3.

This exception is not banked between security states.

If AICR.BFHFNMINS=0, then the NMI is Secure.

If AICR.BFHFNMINS=1, then NMI is Non-secure.

HardFault Priority -1. A HardFault is an exception that occurs because of an error during normal or exception processing. HardFaults have a fixed priority of -1, meaning they have higher priority than any exception with configurable priority.

This exception is banked between security states.

If BFHFNMINS=0, HardFault handles all Secure and Non-secure faults, and the handler is Secure.

If BFHFNMINS=1, HardFault handles Non-secure faults, the handler is Non-secure, and bus faults are Non-secure, even if they are caused by Secure code.

Secure HardFault Priority -3. A Secure HardFault is only enabled when BFHFNMINS=1. Secure HardFault handles faults caused by Secure code or faults to Secure regions, except bus faults.

SVCall A *Supervisor Call* (SVC) is an exception that is triggered by the SVC instruction. In an OS environment, applications can use SVC instructions to access OS kernel functions and device drivers.

This exception is banked between security states.

PendSV PendSV is an interrupt-driven request for system-level service. In an OS environment, use PendSV for context switching when no other exception is active.

This exception is banked between security states.

SysTick A SysTick exception is an exception the system timer generates when it reaches zero. Software can also generate a SysTick exception. In an OS environment, the processor can use this exception as system tick.

This exception is banked between security states.

Interrupt (IRQ) An interrupt, or IRQ, is an exception signaled by a peripheral, or generated by a software request. All interrupts are asynchronous to instruction execution. In the system, peripherals use interrupts to communicate with the processor.

This exception is not banked between security states. Secure code can assign each interrupt to Secure or Non-secure state. By default all interrupts are assigned to Secure state.

Table 2-13 Properties of the different exception types

Exception number[a]	IRQ number[a]	Exception type	Priority	Vector address[b]	Activation
1	-	Reset	-4, the highest	0x00000004	Asynchronous
2	-14	NMI	-2	0x00000008	Asynchronous
3	-13	Secure HardFault when AIRCR.BFHFNMINS is 1	-3	0x0000000C	Synchronous
		Non-secure HardFault or HardFault when AIRCR.BFHFNMINS is 0.	-1		
4-10	-	Reserved	-	-	-
11	-5	SVCall	Configurable	0x0000002C	Synchronous
12-13	-	Reserved	-	-	-

Table 2-13 Properties of the different exception types (continued)

Exception number[a]	IRQ number[a]	Exception type	Priority	Vector address[b]	Activation
14	-2	PendSV	Configurable[c]	0x00000038	Asynchronous
15	-1	SysTick	Configurable[c]	0x0000003C	Asynchronous
16 and above	0 and above	Interrupt (IRQ)	Configurable[c]	0x00000040 and above[d]	Asynchronous

a. To simplify the software layer, the CMSIS only uses IRQ numbers. It uses negative values for exceptions other than interrupts. The IPSR returns the Exception number, see *Interrupt Program Status Register* on page 2-7.

b. See *Vector table* for more information.

c. See *Interrupt Priority Registers* on page 4-8.

d. Increasing in steps of 4.

For an asynchronous exception, other than reset, the processor can execute extra instructions between the moment the exception is triggered and the moment the processor enters the exception handler.

Privileged software can disable the exceptions that have configurable priority, as shown in Table 2-13 on page 2-23. See *Interrupt Clear-enable Registers* on page 4-5 for more information.

An exception that targets Secure state cannot be disabled by Non-secure code.

For more information about HardFaults, see *Fault handling* on page 2-32.

2.3.3 Exception handlers

The processor handles exceptions using:

Interrupt Service Routines (ISRs)

Interrupts IRQ0 to IRQ239 are the exceptions handled by ISRs.

Each interrupt is configured by Secure software in Secure or Non-secure state, using ITNS.

Fault handler

HardFault is the only exception handled by the fault handler.

There can be separate fault handlers in Secure and Non-secure state.

System handlers

NMI, PendSV, SVCall, SysTick, and HardFault are all system exceptions handled by system handlers.

Most system handlers can be banked with separate handlers between Secure and Non-secure state.

2.3.4 Vector table

When the Security Extension is implemented, there are two vector tables and two Vector Table Offset Registers, VTOR_S and VTOR_NS.

The vector table contains the reset value of the Stack Pointer, and the start addresses, also called exception vectors, for all exception handlers. Figure 2-1 on page 2-25 shows the order of the exception vectors in the vector table, in Secure and Non-secure state. The least-significant bit of each vector must be 1, indicating that the exception handler is written in Thumb code.

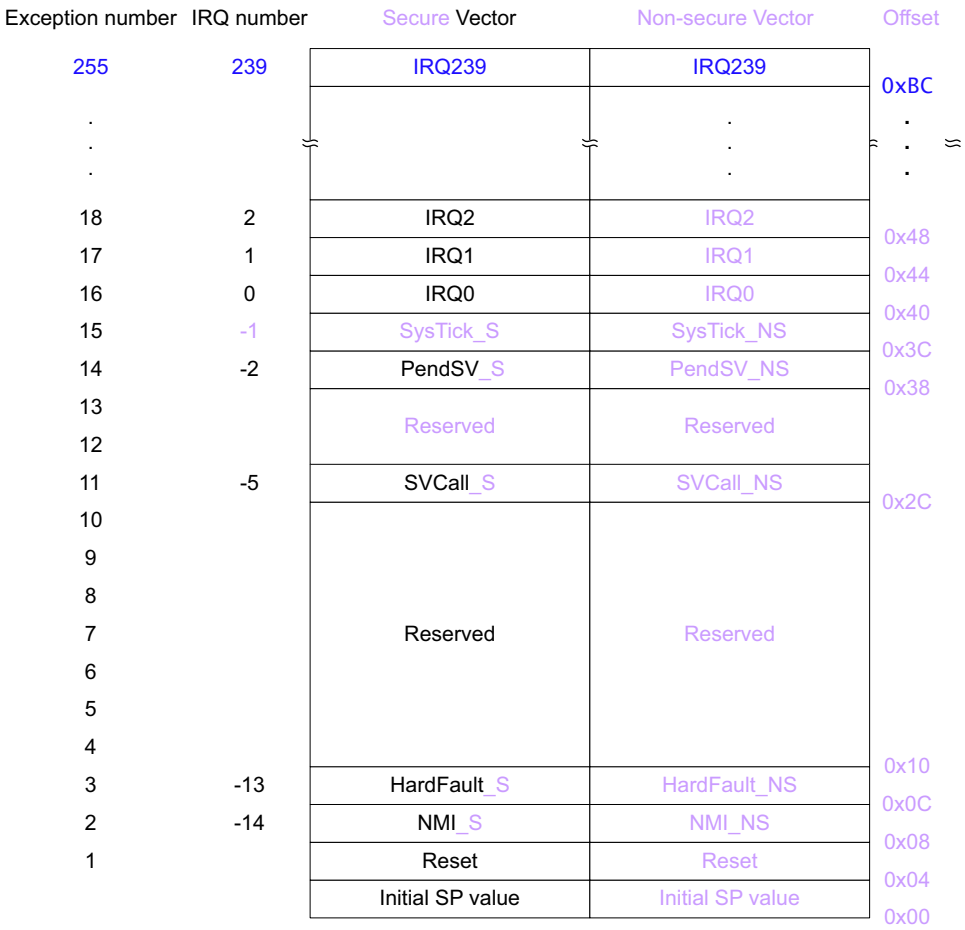


Figure 2-1 Vector table

There are two vector tables and the one that is used depends on the target state of the exception.

The Non-secure handler address of IRQs is used only if the exception targets Non-secure state (ITNS).

If only one SysTick is implemented, then its Non-secure handler address is used only if the exception targets Non-secure state (STTNS).

If AIRCR.BFHFNMINS is 0, then HardFault and NMI are only present in Secure state.

If AIRCR.BFHFNMINS is 1, then HardFault and NMI are present in Non-secure state and Secure HardFault is in Secure state.

On system reset, the vector table is fixed at address `0x00000000`.

Privileged software can write to the VTOR to relocate the vector table start address to a different memory location, in the range `0x00000000` to `0xFFFFFF80`.

The silicon vendor must configure the required alignment, which depends on the number of interrupts implemented. The minimum alignment is 32 words, enough for up to 16 interrupts. For more interrupts, adjust the alignment by rounding up to the next power of two. For example,

if you require 21 interrupts, the alignment must be on a 64-word boundary because the required table size is 37 words, and the next power of two is 64, see *Vector Table Offset Register* on page 4-15.

2.3.5 Exception priorities

As Table 2-13 on page 2-23 shows, all exceptions have an associated priority, with:

- A lower priority value indicating a higher priority.
- Configurable priorities for all exceptions except Reset, HardFault, and NMI.

If software does not configure any priorities, then all exceptions with a configurable priority have a priority of 0. For information about configuring exception priorities, see:

- *System Handler Priority Registers* on page 4-20.
- *Interrupt Priority Registers* on page 4-8.

Note

Configurable priority values are in the range 0-192, in steps of 64. The Reset, HardFault, and NMI exceptions, with fixed negative priority values, always have higher priority than any other exception.

The security state defines the priority. Depending on the value of PRIS, the priority can be extended.

Table 2-14 Extended priority

Priority value [7:6]	Secure priority	Non-secure priority when PRIS = 0	Non-secure priority when PRIS = 1
0	0	0	128
1	64	64	160
2	128	128	192
3	192	192	224

Assigning a higher priority value to IRQ[0] and a lower priority value to IRQ[1] means that IRQ[1] has higher priority than IRQ[0]. If both IRQ[1] and IRQ[0] are asserted, IRQ[1] is processed before IRQ[0].

If multiple pending exceptions have the same priority, the pending exception with the lowest exception number takes precedence. For example, if both IRQ[0] and IRQ[1] are pending and have the same priority, then IRQ[0] is processed before IRQ[1].

When the processor is executing an exception handler, the exception handler is preempted if a higher priority exception occurs. If an exception occurs with the same priority as the exception being handled, the handler is not preempted, irrespective of the exception number. However, the status of the new interrupt changes to pending.

2.3.6 Exception entry and return

Descriptions of exception handling use the following terms:

Preemption When the processor is executing an exception handler, an exception can preempt the exception handler if its priority is higher than the priority of the exception being handled.

When one exception preempts another, the exceptions are called nested exceptions. See *Exception entry* for more information.

Return This occurs when the exception handler is completed, and:

- There is no pending exception with sufficient priority to be serviced.
- The completed exception handler was not handling a late-arriving exception.

The processor pops the stack and restores the processor state to the state it had before the interrupt occurred. See *Exception return* on page 2-29 for more information.

Tail-chaining This mechanism speeds up exception servicing. On completion of an exception handler, if there is a pending exception that meets the requirements for exception entry, the stack pop is skipped and control transfers to the new exception handler.

Late-arriving This mechanism speeds up preemption. If a higher priority exception occurs during state saving for a previous exception, the processor switches to handle the higher priority exception and initiates the vector fetch for that exception. State saving is not affected by late arrival because the state saved would be the same for both exceptions. On return from the exception handler of the late-arriving exception, the normal tail-chaining rules apply.

Exception entry

Exception entry occurs when there is a pending exception which is enabled and has sufficient priority and either:

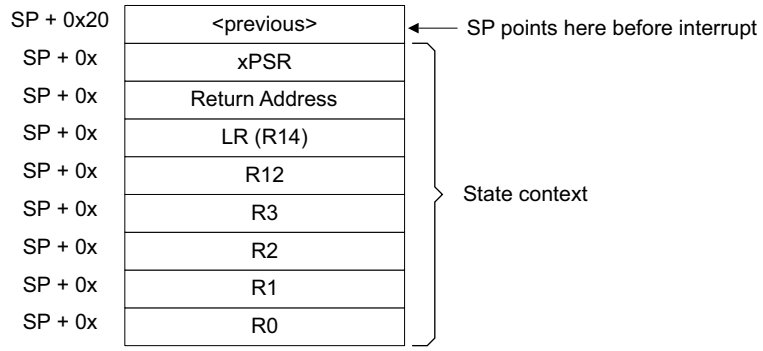
- The processor is in Thread mode.
- The new exception is of higher priority than the exception being handled, in which case the new exception preempts the exception being handled.

When one exception preempts another, the exceptions are nested.

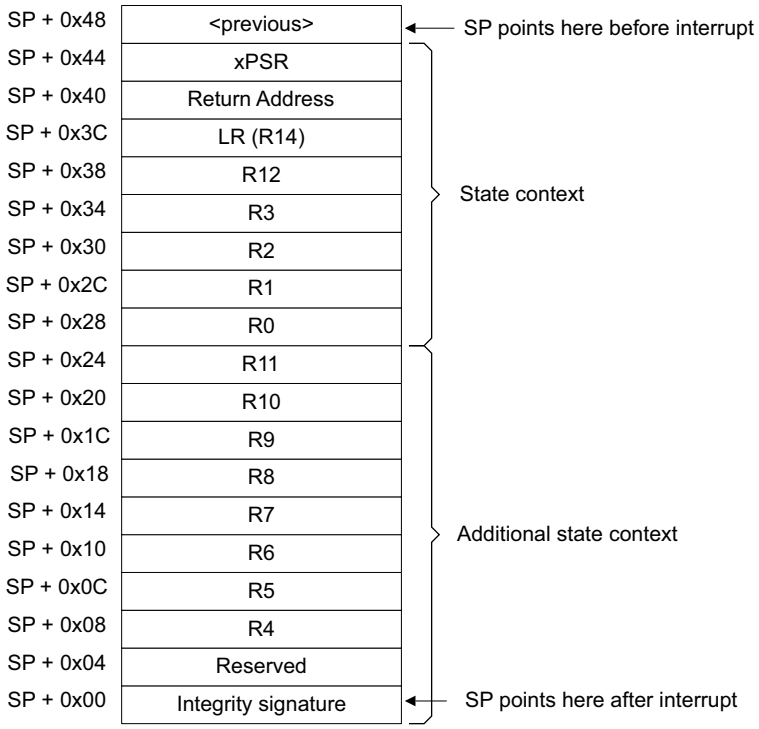
Sufficient priority means the exception has greater priority than any limit set by the mask register, see *Exception mask register* on page 2-8. An exception with less priority than this is pending but is not handled by the processor.

When the processor takes an exception, unless the exception is a tail-chained or a late-arriving exception, the processor pushes information onto the current stack. This operation is referred to as *stacking* and the structure is referred to as a *stack frame*.

The following figure shows the short stack frame. The short stack frame is used when the extended stack frame is not required, for exceptions taken from Non-secure state, or when the Security extension is not implemented.



Hardware saves the state context onto the stack that the Stack Pointer register points to. The extented stack frame shown in the following figure is used when Non-secure code preempts Secure code. The extented stack frame is also used in case of late arrival of exceptions and the final exception is Secure. In case of tail-chaining, some stacking might be required to extend the stack if it was not already full.



Immediately after stacking, the Stack Pointer indicates the lowest address in the stack frame. The stack frame is aligned to a doubleword address.

The stack frame includes the return address. This is the address of the next instruction in the interrupted program. This value is restored to the PC at exception return so that the interrupted program resumes.

The processor performs a vector fetch that reads the exception handler start address from the vector table. When stacking is complete, the processor starts executing the exception handler. At the same time, the processor writes an EXC_RETURN value to the LR. This indicates which Stack Pointer corresponds to the stack frame and what operation mode the processor was in before the entry occurred.

If no higher priority exception occurs during exception entry, the processor starts executing the exception handler and automatically changes the status of the corresponding pending interrupt to active.

If another higher priority exception occurs during exception entry, the processor starts executing the exception handler for this exception and does not change the pending status of the earlier exception. This is the late arrival case.

Exception return

Exception return occurs when the processor is in Handler mode and execution of one of the following instructions attempts to set the PC to an EXC_RETURN value:

- A `POP` instruction that loads the PC.
- A `BX` instruction using any register.

The processor saves an EXC_RETURN value to the LR on exception entry. The exception mechanism relies on this value to detect when the processor has completed an exception handler. When the processor loads a value matching this pattern to the PC it detects that the operation is not a normal branch operation and, instead, that the exception is complete. As a result, it starts the exception return sequence. Bit[3], bit[2], and bit[0] of the EXC_RETURN value indicate the required return stack and processor mode, as Table 2-15 shows.

Table 2-15 Exception return behavior

Bits	Name	Function
[31:24]	PREFIX	Indicates that this is an EXC_RETURN value. This field reads as `0b11111111`.
[23:7]	-	Reserved, RES1.
[6]	S	Indicates whether registers have been pushed to a Secure or Non-secure stack. 0 = Non-secure stack used. 1 = Secure stack used. If the Security Extension is not implemented, this bit is *RES0*.
[5]	DCRS	Indicates whether the default stacking rules apply, or whether the callee registers are already on the stack. 0 = Stacking of the callee saved registers is skipped. 1 = Default rules for stacking the callee registers are followed. If the Security Extension is not implemented, this bit is *RES1*.

Table 2-15 Exception return behavior (continued)

Bits	Name	Function
[4]	-	Reserved, RES1.
[3]	Mode	Indicates the mode that was stacked from. 0 = Handler mode. 1 = Thread mode.
[2]	SPSEL	Indicates which Stack Pointer the exception frame resides on. 0 = Main Stack Pointer. 1 = Process Stack Pointer.
[1]	-	Reserved.
[0]	ES	Indicates the security state the exception was taken to. 0 = Non-secure. 1 = Secure. If the Security Extension is not implemented, this bit is *RES0*.

2.4 Security state switches

The following table shows the branch instructions that can be used to switch between security states.

Table 2-16 Security state transitions

Current security state	Security attribute of the branch target address	Security state change
Secure	Non-secure	Change to Non-secure state if the branch was a BXNS or BLXNS instruction, with the lsb of its target address set to 0. If the branch instruction is not BXNS or BLXNS, and the branch target adress is Non-secure, then a Secure HardFault is generated.
Non-secure	Secure and Non-secure callable	Change to Secure state if the branch target address contains an SG instruction. Otherwise, a Secure Hardfault is generated.
Non-secure	Secure and not Non-secure callable	A Secure HardFault is generated.
Non-secure	Secure	Returning to Secure using `BX <reg>` or `POP {...,pc}` if the data loaded to the PC is FNC_RETURN.

Any scenario not listed in the table above triggers a Secure HardFault. For example:

- Sequential instructions that cross security attributes.
- A 32-bit instruction fetch that crosses regions with different security attributes.

When an exception is taken to the other Security state, the processor automatically transitions to that other Security state.

Secure software can call a Non-secure function using a BXNS instruction. In this case, the LR is set to a special value called FNC_RETURN, and the actual return address is saved in the Secure stack. When the Non-secure function triggers a return using the FNC_RETURN value, the processor automatically switches back to Secure state and restores the Secure PC from the Secure stack.

2.5 Fault handling

Faults are a subset of exceptions, see *Exception model* on page 2-22. All the faults that occur in the NMI or HardFault handler might result in the HardFault exception being taken or cause lockup. See the [ARM®v8-M Architecture Reference Manual for M profile]. The faults can be divided into three categories:

Execution faults

- Execution of an `SVC` instruction at a priority equal to or higher than SVCall.
- Execution of a `BKPT` instruction when instruction debug is not authenticated for the current security state.
- A system-generated bus error on a load or store.
- Execution of an instruction from an XN memory address.
- Execution of an instruction from a location for which the system generates a bus fault.
- A system-generated bus error on a vector fetch.
- Execution of an `UNDEFINED` instruction.
- Execution of an instruction when not in Thumb state as a result of the T-bit being previously cleared to 0.
- An attempted load or store to an unaligned address.
- An MPU fault because of a privilege violation or an attempt to access an unmanaged region.
- Execution of an unpredictable instruction.
- `LDREX/STREX` instructions that target the I/O port.

Security switches

- An SAU fault because of Non-secure access to Secure data.
- A change of security memory attributes on a sequential stream of instructions.
- A branch from Secure to Non-secure state without a correct `BXNS` or `BLXNS` instruction.
- Non-secure code moving to a Secure Non-secure callable region without a branch to a Secure gateway instruction.
- A Stack Pointer Limit fault when running in Secure state.
- A fault on integrity data on return from an exception.

Exception entries and returns

- A bus fault, MPU fault, or SAU fault during Non-secure stacking.
- An error in Return From Exception data.
- An Interrupt Program Status Register mismatch on Thread and Handler mode.
- Returning from an exception that is not active in the current security state.

Note

Only Reset and NMI can preempt the fixed priority HardFault handler. A HardFault at priority -3 (when BFHFNMINS is set to 1) can preempt NMI or a HardFault at priority -1.

2.5.1 Lockup

Lockup is a processor state where the processor stops executing instructions in response to an error for which escalation to an appropriate HardFault handler is not possible because of the current exception priority. When the processor is in Lockup state, it does not execute any instructions. The processor remains in Lockup state until one of the following occurs:

- It is reset.
- A debugger halts it when instruction debug is authenticated for the current security state.
- An NMI occurs and the current Lockup is in the HardFault handler at priority -1.

Note

ARM recommends a reset to exit lockup state.

2.6 Power management

The Cortex-M23 processor has two sleep modes that reduce power consumption:

- A sleep mode, that stops the processor clock.
- A deep sleep mode, that stops the system clock and switches off the PLL and flash memory.

In Non-secure state, deep sleep mode is authorized depending on the value of SCR.SLEEPDEEPS.

The SLEEPDEEP bit of the SCR selects which sleep mode is used, see *System Control Register* on page 4-17. For more information about the behavior of the sleep modes, see <insert reference to your description of wakeup latency, and any other relevant information>.

This section describes the mechanisms for entering sleep mode, and the conditions for waking up from sleep mode.

2.6.1 Entering sleep mode

This section describes the mechanisms software can use to put the processor into sleep mode.

The system can generate spurious wakeup events. For example, a debug operation wakes up the processor. For this reason, software must be able to put the processor back into sleep mode after such an event. A program might have an idle loop to put the processor back in to sleep mode.

Wait For Interrupt

The *Wait For Interrupt* (`WFI`) instruction causes immediate entry to sleep mode. When the processor executes a `WFI` instruction, it stops executing instructions and enters sleep mode. See *WFI* on page 3-68 for more information.

Wait For Event

The *Wait For Event* (`WFE`) instruction causes entry to sleep mode conditional on the value of a one-bit event register. When the processor executes a `WFE` instruction, it checks the value of the event register:

0 The processor stops executing instructions and enters sleep mode.

1 The processor sets the register to zero and continues executing instructions without entering sleep mode.

See *WFE* on page 3-67 for more information.

If the event register is 1, it indicates that the processor must not enter sleep mode on execution of a `WFE` instruction. Typically, this is because of the assertion of an external event, an interrupt entry, an interrupt exit, a halt entry, or because another processor in the system has executed a `SEV` instruction, see *SEV* on page 3-62. Software cannot access this register directly.

Sleep-on-exit

If the SLEEPONEXIT bit of the SCR is set to 1, when the processor completes the execution of an exception handler and returns to Thread mode, it immediately enters sleep mode. Use this mechanism in applications that only require the processor to run when an interrupt occurs.

Note

Sleep-on-exit is banked between security states. If returning to Secure state, use the Secure instance. If returning to Non-secure state, use the Non-secure instance.

2.6.2 Wakeup from sleep mode

The conditions for the processor to wake up depend on the mechanism that caused it to enter sleep mode.

Wakeup from WFI or sleep-on-exit

Normally, the processor wakes up only when it detects an exception with sufficient priority to cause exception entry, ignoring the value of PRIMASK.

Some embedded systems might have to execute system restore tasks after the processor wakes up, and before it executes an interrupt handler. To achieve this, set the PRIMASK.PM bit to 1. If an enabled interrupt arrives and has a higher priority than the current exception priority, the processor wakes up but does not execute the interrupt handler. For more information about PRIMASK, see *Exception mask register* on page 2-8.

Wakeup from WFE

The processor wakes up if:

- It detects an exception with sufficient priority to cause exception entry.
- It detects an external event signal, see *External event input*.

In addition, if the SEVONPEND bit in the SCR is set to 1, any new pending interrupt triggers an event and wakes up the processor, even if the interrupt is disabled or has insufficient priority to cause exception entry. For more information about the SCR, see *System Control Register* on page 4-17.

SEVONPEND is banked between security states, and only the exceptions that target the corresponding security state are counted.

2.6.3 Wakeup Interrupt Controller

The *Wakeup Interrupt Controller* (WIC) is a peripheral that can detect an interrupt and wake the processor from deep sleep mode. The WIC is enabled each time the processor goes to sleep mode or deep sleep mode.

The WIC is not programmable and does not have any registers or user interface. It operates entirely from hardware signals and is transparent to software.

When the WIC is enabled and the processor enters deep sleep mode, the power management unit in the system can power down most of the Cortex-M23 processor. This has the side effect of stopping the SysTick timer.

2.6.4 External event input

The processor provides an external event input signal. This signal can be generated by peripherals. Tie this signal LOW if it is not used.

This signal can wakeup the processor from WFE, or set the internal WFE event register to 1 to indicate that the processor must not enter sleep mode on a later `WFE` instruction, see *Wait For Event* on page 2-34.

You can use any WFE wakeup event to set the Event Register, even if the processor is not in WFE mode, so there is no guarantee that WFE causes a sleep.

WFE can be called inside a loop to check the wakeup condition.

2.6.5 Power management programming hints

ISO/IEC C cannot directly generate the `WFI`, `WFE`, and `SEV` instructions. The CMSIS provides the following intrinsic functions for these instructions:

```
void __WFE(void) // Wait for Event
void __WFI(void) // Wait for Interrupt
void __SEV(void) // Send Event
```

Chapter 3
The Cortex-M23 Instruction Set, Reference Material

This chapter is the reference material for the Cortex-M23 instruction set description in a User Guide. The following sections give general information:

- *Instruction set summary* on page 3-2.
- *CMSIS functions* on page 3-5.
- *CMSE* on page 3-7.
- *About the instruction descriptions* on page 3-8.

Each of the following sections describes a functional group of Cortex-M23 instructions. Together they describe all the instructions that are supported by the Cortex-M23 processor:

- *Memory access instructions* on page 3-15.
- *General data processing instructions* on page 3-31.
- *Branch and control instructions* on page 3-48.
- *Miscellaneous instructions* on page 3-53.

3.1 Instruction set summary

The processor implements a version of the Thumb instruction set. Table 3-1 shows the instructions that the Cortex-M23 processor supports.

Note

In Table 3-1:

- Angle brackets, <>, enclose alternative forms of the operand.
- Braces, {}, enclose optional operands and mnemonic parts.
- The Operands column is not exhaustive.

For more information on the instructions and operands, see the instruction descriptions.

Table 3-1 Cortex-M23 instructions

Mnemonic	Operands	Brief description	Flags	Page
ADCS	*{Rd,} Rn, Rm*	Add with Carry	N,Z,C,V	page 3-32
ADD{S}	*{Rd,} Rn, <Rm|#imm>*	Add	N,Z,C,V	page 3-32
ADR	*Rd, label*	PC-relative Address to Register	-	page 3-16
ANDS	*{Rd,} Rn, Rm*	Bitwise AND	N,Z	page 3-32
ASRS	*{Rd,} Rm, <Rs|#imm>*	Arithmetic Shift Right	N,Z,C	page 3-38
B{cond}	*label*	Branch {conditionally}	-	page 3-49
BICS	*{Rd,} Rn, Rm*	Bit Clear	N,Z	page 3-36
BKPT	*#imm*	Breakpoint	-	page 3-54
BL	*label*	Branch with Link	-	page 3-49
BLX	*Rm*	Branch indirect with Link	-	page 3-49
BLXNS	Rm	Branch indirect with Link to Non-secure		page 3-51
BX	*Rm*	Branch indirect	-	page 3-49
BXNS	Rm	Branch indirect to Non-secure	-	page 3-51
CBZ	*Rn,label*	Compare and Branch on Zero	-	page 3-52
CBNZ	*Rn,label*	Compare and Branch on Non-Zero	-	page 3-52
CLREX	-	Clear Exclusive Monitor	-	page 3-17
CMN	*Rn, Rm*	Compare Negative	N,Z,C,V	page 3-40
CMP	*Rn, <Rm|#imm>*	Compare	N,Z,C,V	page 3-40
CPSID	*i*	Change Processor State, Disable Interrupts	-	page 3-55
CPSIE	*i*	Change Processor State, Enable Interrupts	-	page 3-55
DMB	-	Data Memory Barrier	-	page 3-56
DSB	-	Data Synchronization Barrier	-	page 3-57
EORS	*{Rd,} Rn, Rm*	Exclusive OR	N,Z	page 3-36

Table 3-1 Cortex-M23 instructions (continued)

Mnemonic	Operands	Brief description	Flags	Page
ISB	-	Instruction Synchronization Barrier	-	page 3-58
LDA	LDA Rt, [Rn]	Load-Acquire Word		page 3-27
LDAB	*Rt, [Rn]*	Load-Acquire Byte	-	page 3-27
LDAH	*Rt, [Rn]*	Load-Acquire Halfword	-	page 3-27
LDAEX	*Rt, [Rn]*	Load-Acquire Exclusive Word	-	page 3-28
LDAEXB	*Rt, [Rn]*	Load-Acquire Exclusive Byte	-	page 3-28
LDAEXH	*Rt, [Rn]*	Load-Acquire Exclusive Halfword	-	page 3-28
LDM	*Rn{!}, reglist*	Load Multiple registers, increment after	-	page 3-22
LDR	*Rt, label*	Load Register from PC-relative address	-	page 3-15
LDR	*Rt, [Rn, <Rm|#imm>]*	Load Register with Word	-	page 3-15
LDRB	*Rt, [Rn, <Rm|#imm>]*	Load Register Byte	-	page 3-15
LDRH	*Rt, [Rn, <Rm|#imm>]*	Load Register with Halfword	-	page 3-15
LDRSB	*Rt, [Rn, <Rm|#imm>]*	Load Register Signed Byte	-	page 3-15
LDRSH	*Rt, [Rn, <Rm|#imm>]*	Load Register Signed Halfword	-	page 3-15
LSLS	*{Rd,} Rn, <Rs|#imm>*	Logical Shift Left	N,Z,C	page 3-38
LSRS	*{Rd,} Rn, <Rs|#imm>*	Logical Shift Right	N,Z,C	page 3-38
MOV{S}	*Rd, Rm*	Move	N,Z	page 3-41
MRS	*Rd, spec_reg*	Move to general register from special register	-	page 3-59
MSR	*spec_reg, Rm*	Move to special register from general register	N,Z,C,V	page 3-60
MULS	*Rd, Rn, Rm*	Multiply, 32-bit result	N,Z	page 3-43
MVNS	*Rd, Rm*	Bitwise NOT	N,Z	page 3-41
NOP	-	No Operation	-	page 3-61
ORRS	*{Rd,} Rn, Rm*	Logical OR	N,Z	page 3-36
POP	*reglist*	Pop registers from stack	-	page 3-30
PUSH	*reglist*	Push registers onto stack	-	page 3-30
REV	*Rd, Rm*	Byte-Reverse word	-	page 3-44
REV16	*Rd, Rm*	Byte-Reverse packed halfword	-	page 3-44
REVSH	*Rd, Rm*	Byte-Reverse signed halfword	-	page 3-44
RORS	*{Rd,} Rn, Rs*	Rotate Right	N,Z,C	page 3-38
RSBS	*{Rd,} Rn, #0*	Reverse Subtract	N,Z,C,V	page 3-32
SBCS	*{Rd,} Rn, Rm*	Subtract with Carry	N,Z,C,V	page 3-32
SDIV	*{Rd,} Rn, Rm*	Signed Divide		page 3-45
SEV	-	Send Event	-	page 3-62

Table 3-1 Cortex-M23 instructions (continued)

Mnemonic	Operands	Brief description	Flags	Page
SG	-	Secure Gateway	-	page 3-63
STL	STL Rt, [Rn]	Store Release	-	page 3-27
STLB	*Rt, [Rn]*	Store Release Byte	-	page 3-27
STLH	*Rt, [Rn]*	Store Release Halfword	-	page 3-27
STLEX	*Rd, Rt, [Rn]*	Store Release Exclusive	-	page 3-28
STLEXB	*Rd, Rt, [Rn]*	Store Release Exclusive Byte	-	page 3-28
STLEXH	*Rd, Rt, [Rn]*	Store Release Exclusive Halfword	-	page 3-28
STREX	*Rd, Rt, [Rn]*	Store Register Exclusive	-	page 3-24
STREXB	*Rd, Rt, [Rn]*	Store Register Exclusive Byte	-	page 3-24
STREXH	*Rd, Rt, [Rn]*	Store Register Exclusive Halfword	-	page 3-24
STM	*Rn!, reglist*	Store Multiple registers, increment after	-	page 3-22
STR	*Rt, [Rn, <Rm|#imm>]*	Store Register Word	-	page 3-15
STRB	*Rt, [Rn, <Rm|#imm>]*	Store Register Byte	-	page 3-15
STRH	*Rt, [Rn, <Rm|#imm>]*	Store Register Halfword	-	page 3-15
SUB{S}	*{Rd,} Rn, <Rm|#imm>*	Subtract	N,Z,C,V	page 3-32
SVC	*#imm*	Supervisor Call	-	page 3-64
SXTB	*Rd, Rm*	Signed Extended Byte	-	page 3-46
SXTH	*Rd, Rm*	Signed Extended Halfword	-	page 3-46
TST	*Rn, Rm*	Logical AND based test	N,Z	page 3-47
TT	*Rd, [Rn]*	Test Target	-	page 3-65
TTT	Rd, [Rn]	Test Target Unprivileged	-	page 3-65
TTA	Rd, [Rn]	Test Target Alternate Domain	-	page 3-65
TTAT	*Rd, [Rn]*	Test Target Alternate Domain Unprivileged	-	page 3-65
UDIV	*{Rd,} Rn, Rm*	Unsigned Divide	-	page 3-45
UXTB	*Rd, Rm*	Unsigned Extend Byte	-	page 3-46
UXTH	*Rd, Rm*	Unsigned Extend Halfword	-	page 3-46
WFE	-	Wait For Event	-	page 3-67
WFI	-	Wait For Interrupt	-	page 3-68

3.2 CMSIS functions

ISO/IEC C code cannot directly access some Cortex-M23 instructions. This section describes intrinsic functions that can generate these instructions, provided by the CMSIS and that might be provided by a C compiler. If a C compiler does not support an appropriate intrinsic function, you might have to use inline assembler to access the relevant instruction.

The CMSIS provides the following intrinsic functions to generate instructions that ISO/IEC C code cannot directly access:

Table 3-2 CMSIS intrinsic functions to generate some Cortex-M23 instructions

Instruction	CMSIS intrinsic function
BKPT	void __BKPT
CLREX	void __CLREX
CLZ	uint8_t __CLZ (uint32_t value)
CPSIE i	void __enable_irq (void)
CPSID i	void __disable_irq (void)
ISB	void __ISB (void)
DSB	void __DSB (void)
DMB	void __DMB (void)
LDA	uint32_t __LDA (volatile uint32_t * ptr)
LDAB	uint8_t __LDAB (volatile uint8_t * ptr)
LDAEX	uint32_t __LDAEX (volatile uint32_t * ptr)
LDAEXB	uint8_t __LDAEXB (volatile uint32_t * ptr)
LDAEXH	uint16_t __LDAEXH (volatile uint32_t * ptr)
LDAH	uint32_t __LDAH (volatile uint32_t * addr)
LDRT	uint32_t __LDRT (uint32_t ptr)
NOP	void __NOP (void)
RBIT	uint32_t __RBIT (uint32_t int value)
REV	uint32_t __REV (uint32_t int value)
REV16	uint32_t __REV16 (uint32_t int value)
REVSH	uint32_t __REVSH(uint32_t int value)
ROR	uint32_t __ROR (uint32_t value, uint32_t shift)
RRX	uint32_t __RRX (uint32_t value)
STL	void __STL (uint32_t value, volatile uint32_t * ptr)
STLEX	uint32_t __STLEX (uint16_t value, volatile uint16_t * ptr)
STLEXB	uint32_t __STLEXB (uint16_t value, volatile uint16_t * ptr)
STLEXH	uint32_t __STLEXH (uint16_t value, volatile uint16_t * ptr)
STLH	void __STLH (uint16_t value, volatile uint16_t * ptr)

Table 3-2 CMSIS intrinsic functions to generate some Cortex-M23 instructions

Instruction	CMSIS intrinsic function
STREX	uint32_t __STREXW (uint32_t value, uint32_t *addr)
STREXH	uint32_t __STREXH (uint16_t value, uint16_t *addr)
STREXB	uint32_t __STREXB (uint8_t value, uint8_t *addr)
SEV	void __SEV (void)
WFE	void __WFE (void)
WFI	void __WFI (void)

The CMSIS provides several functions for accessing the special registers using MRS and MSR instructions:

Table 3-3 CMSIS intrinsic functions to access the special registers

Special register	Access	CMSIS function
PRIMASK	Read	uint32_t __get_PRIMASK (void)
	Write	void __set_PRIMASK (uint32_t value)
CONTROL	Read	uint32_t __get_CONTROL (void)
	Write	void __set_CONTROL (uint32_t value)
MSP	Read	uint32_t __get_MSP (void)
	Write	void __set_MSP (uint32_t TopOfMainStack)
PSP	Read	uint32_t __get_PSP (void)
	Write	void __set_PSP (uint32_t TopOfProcStack)

The CMSIS also provides several functions for accessing the Non-secure special registers using MRS and MSR instructions:

Table 3-4 CMSIS intrinsic functions to access the Non-secure special registers

Special register	Access	CMSIS function
PRIMASK_NS	Read	uint32_t __TZ_get_PRIMASK_NS (void)
	Write	void __TZ_set_PRIMASK_NS (uint32_t value)
CONTROL_NS	Read	uint32_t __TZ_get_CONTROL_NS (void)
	Write	void __TZ_set_CONTROL_NS (uint32_t value)
MSP_NS	Read	uint32_t __TZ_get_MSP_NS (void)
	Write	void __TZ_set_MSP_NS (uint32_t TopOfMainStack)
PSP_NS	Read	uint32_t __TZ_get_PSP_NS (void)
	Write	void __TZ_set_PSP_NS (uint32_t TopOfProcStack)

3.3 CMSE

CMSE is the compiler support for the ARMv8-M Security Extension (architecture intrinsics and options) and is part of the ARM C Language (ACLE) specification.

Using CMSE features is required when developing software running in Secure state. This provides mechanisms to define Secure entry points and enable the tool chain to generate correct instructions or support functions in the program image.

The CMSE features are accessed using various attributes and intrinsics. Additional macros are also defined as part of the CMSE.

3.4 About the instruction descriptions

The following sections give more information about using the instructions:

- *Operands*.
- *Restrictions when using PC or SP*.
- *Shift Operations*.
- *Address alignment* on page 3-11.
- *PC-relative expressions* on page 3-11.
- *Conditional execution* on page 3-12.

3.4.1 Operands

An instruction operand can be an ARM register, a constant, or another instruction-specific parameter. Instructions act on the operands and often store the result in a destination register. When there is a destination register in the instruction, it is usually specified before the other operands.

3.4.2 Restrictions when using PC or SP

Many instructions are unable to use, or have restrictions on whether you can use, the *Program Counter* (PC) or *Stack Pointer* (SP) for the operands or destination register. See instruction descriptions for more information.

Note

When you update the PC with a `BX`, `BLX`, or `POP` instruction, bit[0] of any address must be 1 for correct execution. This is because this bit indicates the destination instruction set, and the Cortex-M23 processor only supports Thumb instructions. When a `BL` or `BLX` instruction writes the value of bit[0] into the LR it is automatically assigned the value 1. There is an exception on `BXNS` and `BLXNS` where bit 0 with value 0 means that a switch to Non-secure is permitted.

3.4.3 Shift Operations

Register shift operations move the bits in a register left or right by a specified number of bits, the *shift length*. Register shift can be performed directly by the instructions `ASR`, `LSR`, `LSL`, and `ROR` and the result is written to a destination register.

The permitted shift lengths depend on the shift type and the instruction, see the individual instruction description. If the shift length is 0, no shift occurs. Register shift operations update the carry flag except when the specified shift length is 0. The following subsections describe the various shift operations and how they affect the carry flag. In these descriptions, *Rm* is the register containing the value to be shifted, and *n* is the shift length.

ASR

Arithmetic shift right by *n* bits moves the left-hand 32-*n* bits of the register *Rm*, to the right by *n* places, into the right-hand 32-*n* bits of the result, and it copies the original bit[31] of the register into the left-hand *n* bits of the result. See Figure 3-1 on page 3-9.

You can use the `ASR` operation to divide the signed value in the register *Rm* by 2^n, with the result being rounded towards negative-infinity.

When the instruction is `ASRS` the carry flag is updated to the last bit shifted out, bit[*n*-1], of the register *Rm*.

Note

- If *n* is 32 or more, then all the bits in the result are set to the value of bit[31] of *Rm*.
- If *n* is 32 or more and the carry flag is updated, it is updated to the value of bit[31] of *Rm*.

Figure 3-1 ASR #3

LSR

Logical shift right by *n* bits moves the left-hand 32-*n* bits of the register *Rm*, to the right by *n* places, into the right-hand 32-*n* bits of the result, and it sets the left-hand *n* bits of the result to 0. See Figure 3-2.

You can use the LSR operation to divide the value in the register *Rm* by 2^n, if the value is regarded as an unsigned integer.

When the instruction is LSRS, the carry flag is updated to the last bit shifted out, bit[*n*-1], of the register *Rm*.

Note

- If *n* is 32 or more, then all the bits in the result are cleared to 0.
- If *n* is 33 or more and the carry flag is updated, it is updated to 0.

Figure 3-2 LSR #3

LSL

Logical shift left by *n* bits moves the right-hand 32-*n* bits of the register *Rm*, to the left by *n* places, into the left-hand 32-*n* bits of the result, and it sets the right-hand *n* bits of the result to 0. See Figure 3-3 on page 3-10.

You can use the LSL operation to multiply the value in the register *Rm* by 2^n, if the value is regarded as an unsigned integer or a two's complement signed integer. Overflow can occur without warning.

When the instruction is LSLS the carry flag is updated to the last bit shifted out, bit[32-*n*], of the register *Rm*. These instructions do not affect the carry flag when used with LSL #0.

Note

- If *n* is 32 or more, then all the bits in the result are cleared to 0.
- If *n* is 33 or more and the carry flag is updated, it is updated to 0.

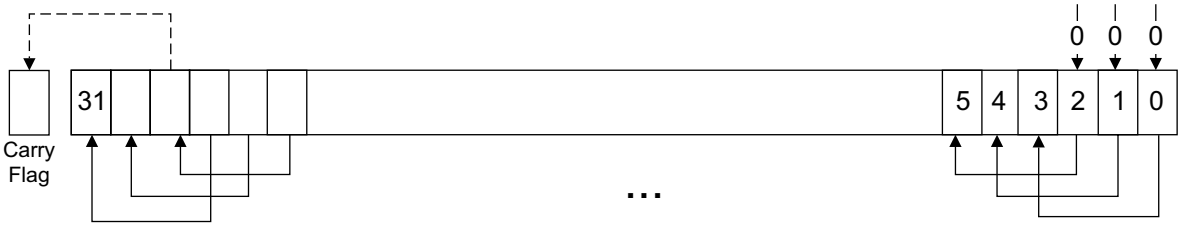


Figure 3-3 LSL #3

ROR

Rotate right by *n* bits moves the left-hand 32-*n* bits of the register *Rm*, to the right by *n* places, into the right-hand 32-*n* bits of the result, and it moves the right-hand *n* bits of the register into the left-hand *n* bits of the result. See Figure 3-4.

When the instruction is RORS the carry flag is updated to the last bit rotation, bit[*n*-1], of the register *Rm*.

Note

- If *n* is 32, then the value of the result is same as the value in *Rm*, and if the carry flag is updated, it is updated to bit[31] of *Rm*.
- ROR with shift length, *n*, greater than 32 is the same as ROR with shift length *n*-32.

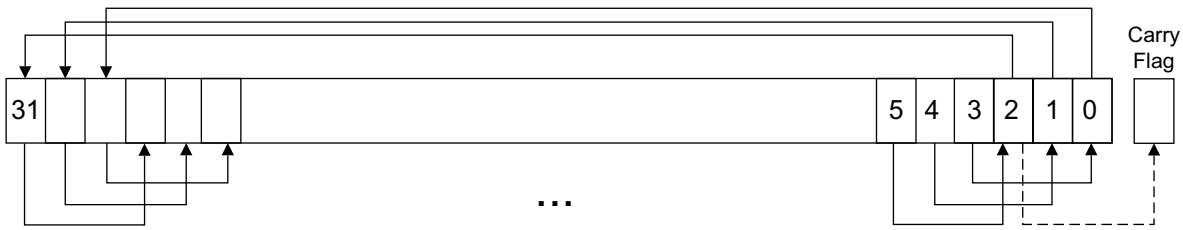


Figure 3-4 ROR #3

3.4.4 Address alignment

An aligned access is an operation where a word-aligned address is used for a word, or multiple word access, or where a halfword-aligned address is used for a halfword access. Byte accesses are always aligned.

There is no support for unaligned accesses on the Cortex-M23 processor. Any attempt to perform an unaligned memory access operation results in a HardFault exception.

3.4.5 PC-relative expressions

A PC-relative expression or *label* is a symbol that represents the address of an instruction or literal data. It is represented in the instruction as the PC value plus or minus a numeric offset. The assembler calculates the required offset from the label and the address of the current instruction. If the offset is too big, the assembler produces an error.

Note

- For most instructions, the value of the PC is the address of the current instruction plus 4 bytes.
- Your assembler might permit other syntaxes for PC-relative expressions, such as a label plus or minus a number, or an expression of the form `[PC, #imm]`.

3.4.6 Conditional execution

Most data processing instructions update the condition flags in the *Application Program Status Register* (APSR) according to the result of the operation, see *Application Program Status Register* on page 2-6. Some instructions update all flags, and some only update a subset. If a flag is not updated, the original value is preserved. See the instruction descriptions for the flags they affect.

You can execute a conditional branch instruction, based on the condition flags set in another instruction, either:

- Immediately after the instruction that updated the flags.
- After any number of intervening instructions that have not updated the flags.

On the Cortex-M23 processor, conditional execution is available by using conditional branches.

This section describes:

- *The condition flags* on page 3-13.
- *Condition code suffixes* on page 3-14.

The condition flags

The APSR contains the following condition flags:

N Set to 1 when the result of the operation was negative, cleared to 0 otherwise.

Z Set to 1 when the result of the operation was zero, cleared to 0 otherwise.

C Set to 1 when the operation resulted in a carry, cleared to 0 otherwise.

V Set to 1 when the operation caused overflow, cleared to 0 otherwise.

For more information about the APSR, see *Program Status Register* on page 2-5.

A carry occurs:

- If the result of an addition is greater than or equal to 2^{32}.
- If the result of a subtraction is positive or zero.
- As the result of a shift or rotate instruction.

Overflow occurs when the sign of the result in bit[31] does not match the sign of the result, had the operation been performed at infinite precision. For example:

- If adding two negative values results in a positive value.
- If adding two positive values results in a negative value.
- If subtracting a positive value from a negative value generates a positive value.
- If subtracting a negative value from a positive value generates a negative value.

The Compare operations are identical to subtracting, for `CMP`, or adding, for `CMN`, except that the result is discarded. See the instruction descriptions for more information.

Condition code suffixes

Conditional branch is shown in syntax descriptions as `B{`*cond*`}`. A branch instruction with a condition code is only taken if the condition code flags in the APSR meet the specified condition, otherwise the branch instruction is ignored. Table 3-5 shows the condition codes to use.

Table 3-5 also shows the relationship between condition code suffixes and the N, Z, C, and V flags.

Table 3-5 Condition code suffixes

Suffix	Flags	Meaning
`EQ`	Z = 1	Equal, last flag setting result was zero.
`NE`	Z = 0	Not equal, last flag setting result was non-zero.
`CS or HS`	C = 1	Higher or same, unsigned.
`CC or LO`	C = 0	Lower, unsigned.
`MI`	N = 1	Negative.
`PL`	N = 0	Positive or zero.
`VS`	V = 1	Overflow.
`VC`	V = 0	No overflow.
`HI`	C = 1 and Z = 0	Higher, unsigned.
`LS`	C = 0 or Z = 1	Lower or same, unsigned.
`GE`	N = V	Greater than or equal, signed.
`LT`	N != V	Less than, signed.
`GT`	Z = 0 and N = V	Greater than, signed.
`LE`	Z = 1 or N != V	Less than or equal, signed.
`AL`	Can have any value	Always. This is the default when no suffix is specified.

3.5 Memory access instructions

Table 3-6 shows the memory access instructions:

Table 3-6 Memory access instructions

Mnemonic	Brief description	See
ADR	Generate PC-relative address	*ADR* on page 3-16.
CLREX	Clear Exclusive	*CLREX* on page 3-17.
LDA{type}	Load-Acquire	*LDA and STL* on page 3-27.
LDAEX{type}	Load-Acquire Exclusive	*LDAEX and STLEX* on page 3-28.
LDM	Load Multiple registers	*LDM and STM* on page 3-22.
LDREX{type}	Load-Exclusive	*LDREX and STREX* on page 3-24.
LDR{type}	Load Register using immediate offset	*LDR and STR, immediate offset* on page 3-18.
LDR{type}	Load Register using register offset	*LDR and STR, register offset* on page 3-19.
LDR	Load Register from PC-relative address	*LDR, PC-relative* on page 3-21.
POP	Pop registers from stack	*PUSH and POP* on page 3-30.
PUSH	Push registers onto stack	*PUSH and POP* on page 3-30.
STL{type}	Store-Release	*LDA and STL* on page 3-27.
STLEX{type}	Store-Acquire Exclusive	*LDAEX and STLEX* on page 3-28.
STM	Store Multiple registers	*LDM and STM* on page 3-22.
STREX{type}	Store Register Exclusive	*LDREX and STREX* on page 3-24.
STR{type}	Store Register using immediate offset	*LDR and STR, immediate offset* on page 3-18.
STR{type}	Store Register using register offset	*LDR and STR, register offset* on page 3-19.

Semaphore data shared between multiple processes in software, and between multiple processors, use exclusive accesses to handle the read-modify-write sequence as required. For an exclusive read-modify-write sequence to succeed, no other process or processor can modify the variable between the exclusive read and exclusive write cycles.

If there is an access conflict of the exclusive Read-Modify-Write sequence:

- The exclusive store fails.
- The memory location does not update.

A local monitor inside the processor is responsible for the detection and management of access conflicts. A global monitor in the system is responsible for the detection and management of access conflicts between multiple processors.

3.5.1 ADR

Generates a PC-relative address.

Syntax

`ADR Rd, label`

where:

Rd	Is the destination register.
label	Is a PC-relative expression. See *PC-relative expressions* on page 3-11.

Operation

`ADR` generates an address by adding an immediate value to the PC, and writes the result to the destination register.

`ADR` facilitates the generation of position-independent code, because the address is PC-relative.

If you use `ADR` to generate a target address for a `BX` or `BLX` instruction, you must ensure that bit[0] of the address you generate is set to 1 for correct execution.

Restrictions

In this instruction *Rd* must specify R0-R7. The data-value addressed must be word aligned and within 1020 bytes of the current PC.

Condition flags

This instruction does not change the flags.

Examples

```
ADR    R1, TextMessage   ; Write address value of a location labelled as
                         ; TextMessage to R1

ADR    R3, [PC,#996]     ; Set R3 to value of PC + 996.
```

3.5.2 CLREX

Clear Exclusive.

Syntax

`CLREX{cond}`

Where:

cond Is an optional condition code. See *Conditional execution* on page 3-12.

Operation

Use `CLREX` to make the next `STREX`, `STREXB`, or `STREXH` instruction write 1 to its destination register and fail to perform the store. However, if there is an `LDREX` instruction between the `CLREX` instruction and the next `STREX`, `STREXB`, or `STREXH` instruction, then the `LDREX` instruction is valid and does not fail.

`CLREX` enables compatibility with other ARM Cortex processors that have to force the failure of the store exclusive if the exception occurs between a load-exclusive instruction and the matching store-exclusive instruction in a synchronization operation. In Cortex-M processors, the local exclusive access monitor clears automatically on an exception boundary, so exception handlers using CLREX are optional.

See *Synchronization primitives* on page 2-18 for more information.

Condition flags

This instruction does not change the flags.

Examples

```
CLREX
```

3.5.3 LDR and STR, immediate offset

Load and Store with immediate offset.

Syntax

```
LDR Rt, [<Rn | SP> {, #imm}]
LDR<B|H> Rt, [Rn {, #imm}]
STR Rt, [<Rn | SP>, {,#imm}]
STR<B|H> Rt, [Rn {,#imm}]
```

where:

Rt	Is the register to load or store.
Rn	Is the register on which the memory address is based.
imm	Is an offset from *Rn*. If *imm* is omitted, it is assumed to be zero.

Operation

LDR, LDRB, and LDRH instructions load the register specified by *Rt* with either a word, byte or halfword data value from memory. Sizes less than word are zero extended to 32-bits before being written to the register specified by *Rt*.

STR, STRB, and STRH instructions store the word, least-significant byte, or lower halfword contained in the single register specified by *Rt* into memory. The memory address to load from or store to is the sum of the value in the register specified by either *Rn* or SP and the immediate value *imm*.

Restrictions

In these instructions:

- *Rt* and *Rn* must only specify R0-R7.
- *imm* must be between:
  - 0 and 1020 and an integer multiple of four for LDR and STR using SP as the base register.
  - 0 and 124 and an integer multiple of four for LDR and STR using R0-R7 as the base register.
  - 0 and 62 and an integer multiple of two for LDRH and STRH.
  - 0 and 31 for LDRB and STRB.
- The computed address must be divisible by the number of bytes in the transaction, see *Address alignment* on page 3-11.

Condition flags

These instructions do not change the flags.

Examples

```
LDR     R4, [R7]                ; Loads R4 from the address in R7.
STR     R2, [R0,#const-struc]   ; const-struc is an expression evaluating
                                ; to a constant in the range 0-1020.
```

3.5.4 LDR and STR, register offset

Load and Store with register offset.

Syntax

`LDR Rt, [Rn, Rm]`

`LDR<B|H> Rt, [Rn, Rm]`

`LDR<SB|SH> Rt, [Rn, Rm]`

`STR Rt, [Rn, Rm]`

`STR<B|H> Rt, [Rn, Rm]`

where:

Rt	Is the register to load or store.
Rn	Is the register on which the memory address is based.
Rm	Is a register containing a value to be used as the offset.

Operation

LDR, LDRB, LDRH, LDRSB, and LDRSH load the register specified by *Rt* with either a word, zero extended byte, zero extended halfword, sign extended byte, or sign extended halfword value from memory.

STR, STRB, and STRH store the word, least-significant byte, or lower halfword contained in the single register specified by *Rt* into memory.

The memory address to load from or store to is the sum of the values in the registers specified by *Rn* and *Rm*.

Restrictions

In these instructions:

- *Rt*, *Rn*, and *Rm* must only specify R0-R7.
- The computed memory address must be divisible by the number of bytes in the load or store, see *Address alignment* on page 3-11.

Condition flags

These instructions do not change the flags.

Examples

```
STR    R0, [R5, R1]         ; Store value of R0 into an address equal to
                            ; sum of R5 and R1
LDRSH  R1, [R2, R3]         ; Load a halfword from the memory address
                            ; specified by (R2 + R3), sign extend to 32-bits
                            ; and write to R1.
```

3.5.5 LDR, PC-relative

Load register (literal) from memory.

Syntax

`LDR Rt, label`

where:

Rt	Is the register to load.
label	Is a PC-relative expression. See *PC-relative expressions* on page 3-11.

Operation

Loads the register specified by *Rt* from the word in memory specified by *label*.

Restrictions

In these instructions, *label* must be within 1020 bytes of the current PC and word aligned.

Condition flags

These instructions do not change the flags.

Examples

```
LDR     R0, LookUpTable   ; Load R0 with a word of data from an address
                          ; labelled as LookUpTable.
LDR     R3, [PC, #100]    ; Load R3 with memory word at (PC + 100).
```

3.5.6 LDM and STM

Load and Store Multiple registers.

Syntax

`LDM` *`Rn`*`{!}, reglist`

`STM` *`Rn`*`!, reglist`

where:

`Rn` Is the register on which the memory addresses are based.

`!` Writeback suffix.

`reglist` Is a list of one or more registers to be loaded or stored, enclosed in braces. It can contain register ranges. It must be comma separated if it contains more than one register or register range, see *Examples* on page 3-23.

`LDMIA` and `LDMFD` are synonyms for `LDM`. `LDMIA` refers to the base register being Incremented After each access. `LDMFD` refers to its use for popping data from Full Descending stacks.

`STMIA` and `STMEA` are synonyms for `STM`. `STMIA` refers to the base register being Incremented After each access. `STMEA` refers to its use for pushing data onto Empty Ascending stacks.

Operation

`LDM` instructions load the registers in *`reglist`* with word values from memory addresses based on *`Rn`*.

`STM` instructions store the word values in the registers in *`reglist`* to memory addresses based on *`Rn`*.

The memory addresses used for the accesses are at 4-byte intervals ranging from the value in the register specified by *`Rn`* to the value in the register specified by *`Rn`* + 4 * (*n*-1), where *n* is the number of registers in *`reglist`*. The accesses happen in order of increasing register numbers, with the lowest numbered register using the lowest memory address and the highest number register using the highest memory address. If the Write-Back suffix is specified, the value in the register specified by *`Rn`* + 4 **n* is written back to the register specified by *`Rn`*.

Restrictions

In these instructions:

- *reglist* and *Rn* are limited to R0-R7.
- The Write-Back suffix must always be used unless the instruction is an LDM where reglist also contains *Rn*, in which case the Write-Back suffix must not be used.
- The value in the register specified by *Rn* must be word aligned. See *Address alignment* on page 3-11 for more information.
- For STM, if *Rn* appears in *reglist*, then it must be the first register in the list.

Condition flags

These instructions do not change the flags.

Examples

```
LDM     R0,{R0,R3,R4}      ; LDMIA is a synonym for LDM
STMIA   R1!,{R2-R4,R6}
```

Incorrect examples

```
STM     R5!,{R4,R5,R6} ; Value stored for R5 is unpredictable
LDM     R2,{}          ; There must be at least one register in the list
```

3.5.7 LDREX and STREX

Load and Store Register Exclusive.

Syntax

`LDREX Rt, [Rn {, #offset}]`

`STREX Rd, Rt, [Rn {, #offset}]`

`LDREXB Rt, [Rn]`

`STREXB Rd, Rt, [Rn]`

`LDREXH Rt, [Rn]`

`STREXH Rd, Rt, [Rn]`

Where:

Rd Is the destination register for the returned status.

Rt Is the register to load or store.

Rn Is the register on which the memory address is based.

offset Is an optional offset applied to the value in *Rn*. If *offset* is omitted, the address is the value in *Rn*.

Operation

LDREX, LDREXB, and LDREXH load a word, byte, and halfword respectively from a memory address.

STREX, STREXB, and STREXH attempt to store a word, byte, and halfword respectively to a memory address. The address used in any Store-Exclusive instruction must be the same as the address in the most recently executed Load-exclusive instruction. The value stored by the Store-Exclusive instruction must also have the same data size as the value loaded by the preceding Load-exclusive instruction. This means software must always use a Load-exclusive instruction and a matching Store-Exclusive instruction to perform a synchronization operation, see *Synchronization primitives* on page 2-18.

If a Store-Exclusive instruction performs the store, it writes 0 to its destination register. If it does not perform the store, it writes 1 to its destination register. If the Store-Exclusive instruction writes 0 to the destination register, it is guaranteed that no other process in the system has accessed the memory location between the Load-exclusive and Store-Exclusive instructions.

For reasons of performance, keep the number of instructions between corresponding Load-Exclusive and Store-Exclusive instruction to a minimum.

Exclusive accesses are not supported in the I/O memory space.

The local monitor does not tag the address or the size. It means that a LDREX or STREX instruction completes even if the address, the size or the attributes do not match.

The global monitor is used in addition to the local monitor when:

- The target address is a shared location in the default memory map with no MPU hint, or hits in a shared MPU region.

Note

Default memory map: Accesses to Device regions in the ranges 0x40000000-0x5fffffff and 0xc0000000-0xffffffff do not use the Global Exclusive Monitor when ACTLR.EXTEXCLALL is 0 and the default memory map is used.

- ACTLR.EXCLEXTALL is set. In this case, any memory location uses the exclusive monitor. This is particularly useful when there is no MPU implemented or the MPU is disabled.

The silicon vendor must specify which memory regions have a global monitor. If an `STREX` instruction uses the global monitor whereas there is no global monitor present, then the instruction always fails.

The silicon vendor must specify how many addresses are supported, and how many processors are present. `LDREX` and `STREX` instructions that target the I/O port always trigger a HardFault.

Restrictions

In these instructions:

- Do not use PC.
- Do not use SP for *Rd* and *Rt*.
- For STREX, *Rd* must be different from both *Rt* and *Rn*.
- The value of *offset* must be a multiple of four in the range 0-1020.

Condition flags

These instructions do not change the flags.

Examples

```
    MOV     R1, #0x1            ; Initialize the 'lock taken' value
try
    LDREX   R0, [LockAddr]      ; Load the lock value
    CMP     R0, #0              ; Is the lock free?
    BNE     try                 ; No - try again
    STREX   R0, R1, [LockAddr]  ; Try and claim the lock
    CMP     R0, #0              ; Did this succeed?
    BNE     try                 ; No - try again
    ....                        ; Yes - we have the lock.
```

For higher efficiency, in a system with multiple cores, WFE can be used before BNE try and SEV after the last BNE try.

3.5.8 LDA and STL

Load-Acquire and Store-Release.

Syntax

LDA Rt, [Rn]

STLH Rt, [Rn]

STL Rt, [Rn]

LDAB Rt, [Rn]

STLB Rt, [Rn]

LDAH Rt, [Rn]

where:

Rt Is the register to load or store,

Rn Is the register on which the memory address is based,

Operation

LDA, LDAB, and LDAH loads word, byte, and halfword data respectively from a memory address. If any loads or stores appear after a load-acquire in program order, then all observers are guaranteed to see the load-acquire before the loads and stores. Loads and stores appearing before a load-acquire are unaffected.

STL, STLB, and STLH stores word, byte, and halfword data respectively to a memory address. If any loads or stores appear before a store-release in program order, then all observers are guaranteed to see the loads and stores before observing the store-release. Loads and stores appearing after a store-release are unaffected.

In addition, if a store-release is followed by a load-acquire, each observer is guaranteed to see them in program order.

There is no requirement that a load-acquire and store-release be paired.

All store-release operations are multi-copy atomic, meaning that in a multiprocessing system, if one observer sees a write to memory because of a store-release operation, then all observers see it. Also, all observers see all writes to the same location in the same order.

Restrictions

The address specified must be naturally aligned, or an alignment fault is generated.

The PC must not use SP for *Rt*.

Condition flags

These instructions do not change the flags.

Examples

```
STR r1, [r0] # Write a memory location
STL r3, [r2] # Memory location at r0 is guaranteed to be visible when update location
at address r2 is visible
```

3.5.9 LDAEX and STLEX

Load-Acquire and Store-Release Exclusive.

Syntax

`LDAEX` *`Rt, [Rn]`*

`LDAEXB` *`Rt, [Rn]`*

`LDAEXH` *`Rt, [Rn]`*

`STLEX` *`Rd, Rt, [Rn]`*

`STLEXB` *`Rd, Rt, [Rn]`*

`STLEXH` *`Rd, Rt, [Rn]`*

where:

`Rd` Is the destination register into which the status result of the store exclusive is written.

`Rt` Is the register to load or store.

`Rn` Is the register on which the memory address is based.

Operation

`LDAEX`, `LDAEXB`, `LDAEXH`, and `LDAEXD` calculate an address from a base register value and an immediate offset, loads a word from memory, writes it to a register, and:

- If the address has the Shareable memory attribute, marks the physical address as exclusive access for the executing core in a global monitor.
- Causes the core that executes to indicate an active exclusive access in the local monitor.

If any loads or stores appear after an `LDAEX`, `LDAEXB`, `LDAEXH`, or `LDAEXD` instruction in program order, then all observers are guaranteed to observe the `LDAEX`, `LDAEXB`, `LDAEXH` ,or `LDAEXD` instruction before observing the loads and stores. Loads and stores appearing before an `LDAEX`, `LDAEXB`, `LDAEXH`, or `LDAEXD` instruction are unaffected.

`STLEX`, `STLEXB`, `STLEXH` and `STLEXD` calculate an address from a base register value and an immediate offset, and stores a word from a register to memory. If the executing core has exclusive access to the memory addressed:

- *`Rd`* is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the *`Rd`* field. The value returned is:
  - **0** If the operation updates memory.
  - **1** If the operation fails to update memory.

If any loads or stores appear before an `STLEX`, `STLEXB`, `STLEXH`, or `STLEXD` instruction in program order, then all observers are guaranteed to observe the loads and stores before observing the store-release. Loads and stores appearing after an `STLEX`, `STLEXB`, `STLEXH`, or `STLEXD` instruction are unaffected.

Note

All store-release operations are multi-copy atomic.

Condition flags

These instructions do not change the flags.

Examples

```
lock

    MOV R1, #0x1 ; Initialize the 'lock taken' value try
    LDAEX R0, [LockAddr] ; Load the lock value
    CMP R0, #0  ; Is the lock free?
    BNE try     ; No - try again
    STREX R0, R1, [LockAddr] ; Try and claim the lock
    CMP R0, #0  ; Did this succeed?
    BNE try     ; No - try again
                    ; Yes - we have the lock.
unlock
    MOV     r1, #0
    STL     r1, [r0]
```

3.5.10 PUSH and POP

Push registers onto, and pop registers off a full-descending stack.

Syntax

`PUSH` *`reglist`*

`POP` *`reglist`*

where:

`reglist` Is a non-empty list of registers, enclosed in braces. It can contain register ranges. It must be comma separated if it contains more than one register or register range.

Operation

`PUSH` stores registers on the stack, with the lowest numbered register using the lowest memory address and the highest numbered register using the highest memory address.

`POP` loads registers from the stack, with the lowest numbered register using the lowest memory address and the highest numbered register using the highest memory address.

`PUSH` uses the value in the SP register minus four as the highest memory address, `POP` uses the value in the SP register as the lowest memory address, implementing a full-descending stack. On completion, `PUSH` updates the SP register to point to the location of the lowest store value, `POP` updates the SP register to point to the location above the highest location loaded.

If a `POP` instruction includes PC in its *`reglist`*, a branch to this location is performed when the `POP` instruction has completed. Bit[0] of the value read for the PC is used to update the APSR T-bit. This bit must be 1 to ensure correct operation.

Restrictions

In these instructions:

- *`reglist`* must use only R0-R7.
- The exception to this rule is LR for a `PUSH` and PC for a `POP`.

Condition flags

These instructions do not change the flags.

A `POP` instruction that contains the PC can be used as an Exception Return or Function Return instruction, depending on the value of the loaded PC.

Examples

```
PUSH    {R0,R4-R7}      ; Push R0,R4,R5,R6,R7 onto the stack
PUSH    {R2,LR}         ; Push R2 and the link-register onto the stack
POP     {R0,R6,PC}      ; Pop r0,r6 and PC from the stack, then branch to
                        ; the new PC.
```

3.6 General data processing instructions

Table 3-7 shows the data processing instructions:

Table 3-7 Data processing instructions

Mnemonic	Brief description	See
ADCS	Add with Carry	*ADC, ADD, RSB, SBC, and SUB* on page 3-32.
ADD{S}	Add	*ADC, ADD, RSB, SBC, and SUB* on page 3-32.
ANDS	Logical AND	*AND, ORR, EOR, and BIC* on page 3-36.
ASRS	Arithmetic Shift Right	*ASR, LSL, LSR, and ROR* on page 3-38.
BICS	Bit Clear	*AND, ORR, EOR, and BIC* on page 3-36.
CMN	Compare Negative	*CMP and CMN* on page 3-40.
CMP	Compare	*CMP and CMN* on page 3-40.
EORS	Exclusive OR	*AND, ORR, EOR, and BIC* on page 3-36.
LSLS	Logical Shift Left	*ASR, LSL, LSR, and ROR* on page 3-38.
LSRS	Logical Shift Right	*ASR, LSL, LSR, and ROR* on page 3-38.
MOV{S}	Move	*MOV and MVN* on page 3-41.
MULS	Multiply	*MULS* on page 3-43.
MVNS	Move NOT	*MOV and MVN* on page 3-41.
ORRS	Logical OR	*AND, ORR, EOR, and BIC* on page 3-36.
REV	Reverse byte order in a word	*REV, REV16, and REVSH* on page 3-44.
REV16	Reverse byte order in each halfword	*REV, REV16, and REVSH* on page 3-44.
REVSH	Reverse byte order in bottom halfword and sign extend	*REV, REV16, and REVSH* on page 3-44.
RORS	Rotate Right	*ASR, LSL, LSR, and ROR* on page 3-38.
RSBS	Reverse Subtract	*ADC, ADD, RSB, SBC, and SUB* on page 3-32.
SBCS	Subtract with Carry	*ADC, ADD, RSB, SBC, and SUB* on page 3-32.
SDIV	Signed Divide	*SDIV and UDIV* on page 3-45.
SUBS	Subtract	*ADC, ADD, RSB, SBC, and SUB* on page 3-32.
SXTB	Signed extend Byte	*SXT and UXT* on page 3-46.
SXTH	Signed extend Halfword	*SXT and UXT* on page 3-46.
UDIV	Unsigned Divide	*SDIV and UDIV* on page 3-45.
UXTB	Unsigned Extend Byte	*SXT and UXT* on page 3-46.
UXTH	Unsigned Extend Halfword	*SXT and UXT* on page 3-46.
TST	Test	*TST* on page 3-47.

3.6.1 ADC, ADD, RSB, SBC, and SUB

Add with carry, Add, Reverse Subtract, Subtract with carry, and Subtract.

Syntax

```
ADCS   {Rd,} Rn, Rm
ADD{S} {Rd,} Rn, <Rm|#imm>
RSBS   {Rd,} Rn, #0
SBCS   {Rd,} Rn, Rm
SUB{S} {Rd,} Rn, <Rm|#imm>
```

where:

`S`	Causes an `ADD` or `SUB` instruction to update flags.
`Rd`	Specifies the result register.
`Rn`	Specifies the first source register.
`Rm`	Specifies the second source register.
`imm`	Specifies a constant immediate value.

When the optional `Rd` register specifier is omitted, it is assumed to take the same value as *`Rn`*, for example `ADDS R1,R2` is identical to `ADDS R1,R1,R2`.

Operation

The ADCS instruction adds the value in *Rn* to the value in *Rm*, adding another one if the carry flag is set, places the result in the register specified by Rd and updates the N, Z, C, and V flags.

The ADD instruction adds the value in *Rn* to the value in *Rm* or an immediate value specified by *imm* and places the result in the register specified by Rd.

The ADDS instruction performs the same operation as ADD and also updates the N, Z, C, and V flags.

The RSBS instruction subtracts the value in *Rn* from zero, producing the arithmetic negative of the value, and places the result in the register specified by Rd and updates the N, Z, C, and V flags.

The SBCS instruction subtracts the value of *Rm* from the value in *Rn*, deducts another one if the carry flag is set. It places the result in the register specified by Rd and updates the N, Z, C, and V flags.

The SUB instruction subtracts the value in *Rm* or the immediate specified by *imm* from *Rn*. It places the result in the register specified by Rd.

The SUBS instruction performs the same operation as SUB and also updates the N, Z, C, and V flags.

Use ADC and SBC to synthesize multiword arithmetic, see *Examples* on page 3-35.

See also *ADR* on page 3-16.

Restrictions

Table 3-8 lists the legal combinations of register specifiers and immediate values that can be used with each instruction.

Table 3-8 ADC, ADD, RSB, SBC and SUB operand restrictions

Instruction	Rd	Rn	Rm	imm	Restrictions
ADCS	R0-R7	R0-R7	R0-R7	-	Rd and *Rn* must specify the same register.
ADD	R0-R15	R0-R15	R0-PC	-	Rd and *Rn* must specify the same register. *Rn* and *Rm* must not both specify PC.
	R0-R7	SP or PC	-	0-1020	Immediate value must be an integer multiple of four.
	SP	SP	-	0-508	Immediate value must be an integer multiple of four.
ADDS	R0-R7	R0-R7	-	0-7	-
	R0-R7	R0-R7	-	0-255	Rd and *Rn* must specify the same register.
	R0-R7	R0-R7	R0-R7	-	-
RSBS	R0-R7	R0-R7	-	-	-
SBCS	R0-R7	R0-R7	R0-R7	-	Rd and *Rn* must specify the same register.
SUB	SP	SP	-	0-508	Immediate value must be an integer multiple of four.
SUBS	R0-R7	R0-R7	-	0-7	-
	R0-R7	R0-R7	-	0-255	Rd and *Rn* must specify the same register.
	R0-R7	R0-R7	R0-R7	-	-

Examples

Example 3-1 shows two instructions that add a 64-bit integer contained in R0 and R1 to another 64-bit integer contained in R2 and R3, and place the result in R0 and R1.

Example 3-1 64-bit addition

```
ADDS    R0, R0, R2    ; add the least significant words
ADCS    R1, R1, R3    ; add the most significant words with carry
```

Multiword values do not have to use consecutive registers. Example 3-2 shows instructions that subtract a 96-bit integer contained in R1, R2, and R3 from another contained in R4, R5, and R6. The example stores the result in R4, R5, and R6.

Example 3-2 96-bit subtraction

```
SUBS    R4, R4, R1    ; subtract the least significant words
SBCS    R5, R5, R2    ; subtract the middle words with carry
SBCS    R6, R6, R3    ; subtract the most significant words with carry
```

Example 3-3 shows the RSBS instruction used to perform a 1's complement of a single register.

Example 3-3 Arithmetic negation

```
RSBS    R7, R7, #0    ; subtract R7 from zero
```

3.6.2 AND, ORR, EOR, and BIC

Logical AND, OR, Exclusive OR, and Bit Clear.

Syntax

`ANDS {Rd,} Rn, Rm`

`ORRS {Rd,} Rn, Rm`

`EORS {Rd,} Rn, Rm`

`BICS {Rd,} Rn, Rm`

where:

Rd Is the destination register.

Rn Is the register holding the first operand and is the same as the destination register.

Rm Second register.

Operation

The `AND`, `EOR`, and `ORR` instructions perform bitwise AND, exclusive OR, and inclusive OR operations on the values in *Rn* and *Rm*.

The `BIC` instruction performs an AND operation on the bits in *Rn* with the logical negation of the corresponding bits in the value of *Rm*.

The condition code flags are updated on the result of the operation, see *The condition flags* on page 3-13.

Restrictions

In these instructions, *Rd*, *Rn*, and *Rm* must only specify R0-R7.

Condition flags

These instructions:

- Update the N and Z flags according to the result.
- Do not affect the C or V flag.

Examples

```
ANDS    R2, R2, R1
ORRS    R2, R2, R5
ANDS    R5, R5, R8
EORS    R7, R7, R6
BICS    R0, R0, R1
```

3.6.3 ASR, LSL, LSR, and ROR

Arithmetic Shift Right, Logical Shift Left, Logical Shift Right, and Rotate Right.

Syntax

```
ASRS {Rd,} Rm, Rs
ASRS {Rd,} Rm, #imm
LSLS {Rd,} Rm, Rs
LSLS {Rd,} Rm, #imm
LSRS {Rd,} Rm, Rs
LSRS {Rd,} Rm, #imm
RORS {Rd,} Rm, Rs
```

where:

Rd Is the destination register. If Rd is omitted, it is assumed to take the same value as *Rm*.

Rm Is the register holding the value to be shifted.

Rs Is the register holding the shift length to apply to the value in *Rm*.

imm Is the shift length. The range of shift length depends on the instruction:

- ASR shift length from 1 to 32
- LSL shift length from 0 to 31
- LSR shift length from 1 to 32.

Note

MOVS Rd, *Rm* is a pseudonym for LSLS Rd, *Rm*, #0.

Operation

ASR, LSL, LSR, and ROR perform an arithmetic-shift-left, logical-shift-left, logical-shift-right, or a right-rotation of the bits in the register *Rm* by the number of places specified by the immediate *imm* or the value in the least-significant byte of the register specified by *Rs*.

For details on what result is generated by the different instructions, see *Shift Operations* on page 3-8.

Restrictions

In these instructions, *Rd*, *Rm*, and *Rs* must only specify R0-R7. For non-immediate instructions, Rd and *Rm* must specify the same register.

Condition flags

These instructions update the N and Z flags according to the result.

The C flag is updated to the last bit shifted out, except when the shift length is 0, see *Shift Operations* on page 3-8. The V flag is left unmodified.

Examples

```
ASRS    R7, R5, #9  ; Arithmetic shift right by 9 bits
LSLS    R1, R2, #3  ; Logical shift left by 3 bits with flag update
LSRS    R4, R5, #6  ; Logical shift right by 6 bits
RORS    R4, R4, R6  ; Rotate right by the value in the bottom byte of R6.
```

3.6.4 CMP and CMN

Compare and Compare Negative.

Syntax

`CMN Rn, Rm`

`CMP Rn, #imm`

`CMP Rn, Rm`

where:

Rn	Is the register holding the first operand.
Rm	Is the register to compare with.
imm	Is the immediate value to compare with.

Operation

These instructions compare the value in a register with either the value in another register or an immediate value. They update the condition flags on the result, but do not write the result to a register.

The `CMP` instruction subtracts either the value in the register specified by *Rm*, or the immediate *imm* from the value in *Rn* and updates the flags. This is the same as a `SUBS` instruction, except that the result is discarded.

The `CMN` instruction adds the value of *Rm* to the value in *Rn* and updates the flags. This is the same as an `ADDS` instruction, except that the result is discarded.

Restrictions

For the:

- `CMN` instruction, *Rn* and *Rm* must only specify R0-R7.
- `CMP` instruction:
  - *Rn* and *Rm* can specify R0-R14.
  - Immediate must be in the range 0-255.

Condition flags

These instructions update the N, Z, C, and V flags according to the result.

Examples

```
CMP     R2, R9
CMN     R0, R2
```

3.6.5 MOV and MVN

Move and Move NOT.

Syntax

`MOV{S} Rd, Rm`

`MOVS Rd, #imm8`

`MOV{W} Rd, #imm16`

`MVNS Rd, Rm`

where:

`S` Is an optional suffix. If `S` is specified, the condition code flags are updated on the result of the operation, see *Conditional execution* on page 3-12.

`Rd` Is the destination register.

`Rm` Is a register.

`imm8` Is any value in the range 0-255.

`imm16` Is any value in the range 0-65535.

Operation

The `MOV` instruction copies the value of *`Rm`* into *`Rd`*.

The `MOVS` instruction performs the same operation as the `MOV` instruction, but also updates the N and Z flags.

The `MVNS` instruction takes the value of *`Rm`*, performs a bitwise logical negate operation on the value, and places the result into *`Rd`*.

Restrictions

In these instructions, *`Rd`* and *`Rm`* must only specify R0-R7. The exception to this rule is *`MOV RD, Rm`* for which *`Rm`* can be either PC or R0-R14.

Condition flags

If `S` is specified, these instructions:

- Update the N and Z flags according to the result.
- Do not affect the C or V flags.

Example

```
MOVS  R0, #0x000B     ; Write value of 0x000B to R0, flags get updated
MOVS  R1, #0x0        ; Write value of zero to R1, flags are updated
MOV   R10, R12        ; Write value in R12 to R10, flags are not updated
MOVS  R3, #23         ; Write value of 23 to R3
MOV   R8, SP          ; Write value of stack pointer to R8
MVNS  R2, R0          ; Write inverse of R0 to the R2 and update flags
```

3.6.6 MOVT

Move Top.

Syntax

`MOVT Rd, #imm16`

Where:

Rd	Is the destination register.
imm16	Is a 16-bit immediate constant and must be in the range 0-65535.

Operation

`MOVT` writes a 32-bit immediate value, *imm16*, to the top halfword, *Rd*[31:16], of its destination register. The write does not affect *Rd*[15:0].

The `MOV`, `MOVT` instruction pair enables you to generate any 32-bit constant.

Restrictions

Rd must not be SP and must not be PC.

Condition flags

This instruction does not change the flags.

Examples

```
MOV R3, #0x4567
MOVT R3, #F123 ; R3 is now F1234567.
```

3.6.7 MULS

Multiply using 32-bit operands, and producing a 32-bit result.

Syntax

`MULS Rd, Rn, Rm`

where:

Rd	Is the destination register.
Rn, Rm	Are registers holding the values to be multiplied.

Operation

The `MUL` instruction multiplies the values in the registers specified by *Rn* and *Rm*, and places the least significant 32 bits of the result in *Rd*. The condition code flags are updated on the result of the operation, see *Conditional execution* on page 3-12.

The result of this instruction does not depend on whether the operands are signed or unsigned.

Restrictions

In this instruction:

- *Rd*, *Rn*, and *Rm* must only specify R0-R7.
- *Rd* must be the same as *Rm*.

Condition flags

This instruction:

- Updates the N and Z flags according to the result.
- Does not affect the C or V flags.

Examples

```
MULS    R0, R2, R0      ; Multiply with flag update, R0 = R0 x R2
```

In SMUL configurations, the MUL instruction takes 32 cycles. Depending on the data, it can be faster to do the multiplication in software using `ADD` instructions.

3.6.8 REV, REV16, and REVSH

Reverse bytes.

Syntax

`REV Rd, Rn`

`REV16 Rd, Rn`

`REVSH Rd, Rn`

Where:

Rd Is the destination register.

Rn Is the source register.

Operation

Use these instructions to change endianness of data:

`REV` Converts 32-bit big-endian data into little-endian data or 32-bit little-endian data into big-endian data.

`REV16` Converts two packed 16-bit big-endian data into little-endian data or two packed 16-bit little-endian data into big-endian data.

`REVSH` Converts 16-bit signed big-endian data into 32-bit signed little-endian data or 16-bit signed little-endian data into 32-bit signed big-endian data.

Restrictions

In these instructions, *Rd*, and *Rn* must only specify R0-R7.

Condition flags

These instructions do not change the flags.

Examples

```
REV    R3, R7  ; Reverse byte order of value in R7 and write it to R3
REV16  R0, R0  ; Reverse byte order of each 16-bit halfword in R0
REVSH  R0, R5  ; Reverse signed halfword
```

3.6.9 SDIV and UDIV

Signed Divide and Unsigned Divide.

Syntax

`SDIV {Rd,} Rn, Rm`

`UDIV {Rd,} Rn, Rm`

Where:

Rd Is the destination register. If *Rd* is omitted, the destination register is *Rn*.

Rn Is the register holding the value to be divided.

Rm Is a register holding the divisor.

Operation

The SDIV instruction performs a signed integer division of the value in *Rn* by the value in *Rm*.

The UDIV instruction performs an unsigned integer division of the value in *Rn* by the value in *Rm*.

For both instructions, if the value in *Rn* is not divisible by the value in *Rm*, the result is rounded towards zero.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions do not change the flags.

Examples

```
SDIV  R0, R2, R4  ; Signed divide, R0 = R2/R4
UDIV  R8, R8, R1  ; Unsigned divide, R8 = R8/R1
```

Depending on the SDIV parameter, SDIV or UDIV takes either 17 or 34 cycles.

Depending on the value of the operands, it can be faster to do the division in software.

3.6.10 SXT and UXT

Signed Extend and Unsigned Extend Byte/Halfword.

Syntax

`SXTB Rd, Rm`

`SXTH Rd, Rm`

`UXTB Rd, Rm`

`UXTH Rd, Rm`

Where:

Rd Is the destination register.

Rm Is the register holding the value to be extended.

Operation

These instructions extract bits from the resulting value:

- `SXTB` extracts bits[7:0] and sign extends to 32 bits.
- `UXTB` extracts bits[7:0] and zero extends to 32 bits.
- `SXTH` extracts bits[15:0] and sign extends to 32 bits.
- `UXTH` extracts bits[15:0] and zero extends to 32 bits.

Restrictions

In these instructions, *Rd* and *Rm* must only specify R0-R7.

Condition flags

These instructions do not affect the flags.

Examples

```
    SXTH  R4, R6          ; Obtain the lower halfword of the
                          ; value in R6 and then sign extend to
                          ; 32 bits and write the result to R4.
    UXTB  R3, R1          ; Extract lowest byte of the value in R10 and zero
                          ; extend it, and write the result to R3
```

3.6.11 TST

Test bits.

Syntax

`TST Rn, Rm`

Where:

Rn	Is the register holding the first operand.
Rm	The register to test against.

Operation

This instruction tests the value in a register against another register. It updates the condition flags based on the result, but does not write the result to a register.

The `TST` instruction performs a bitwise AND operation on the value in *Rn* and the value in *Rm*. This is the same as the `ANDS` instruction, except that it discards the result.

To test whether a bit of *Rn* is 0 or 1, use the `TST` instruction with a register that has that bit set to 1 and all other bits cleared to 0.

Restrictions

In these instructions, *Rn* and *Rm* must only specify R0-R7.

Condition flags

This instruction:

- Updates the N and Z flags according to the result.
- Does not affect the C or V flags.

Examples

```
TST     R0, R1  ; Perform bitwise AND of R0 value and R1 value,
                ; condition code flags are updated but result is discarded
```

3.7 Branch and control instructions

Table 3-9 shows the branch and control instructions:

Table 3-9 Branch and control instructions

Mnemonic	Brief description	See
B{cc}	Branch {conditionally}	*B, BL, BX, and BLX* on page 3-49.
BL	Branch with Link	*B, BL, BX, and BLX* on page 3-49.
BLX	Branch indirect with Link	*B, BL, BX, and BLX* on page 3-49.
BLXNS	Branch with Link and Exchange Non-secure	*BXNS and BLXNS* on page 3-51.
BX	Branch indirect	*B, BL, BX, and BLX* on page 3-49.
BXNS	Branch indirect Non Secure	*BXNS and BLXNS* on page 3-51.
CBNZ	Compare and Branch if Non-Zero	*CBZ and CBNZ* on page 3-52.
CBZ	Compare and Branch if Zero	*CBZ and CBNZ* on page 3-52.

3.7.1 B, BL, BX, and BLX

Branch instructions.

Syntax

`B{cond} label`

`BL label`

`BX Rm`

`BLX Rm`

Where:

cond	Is an optional condition code, see *Conditional execution* on page 3-12.
label	Is a PC-relative expression. See *PC-relative expressions* on page 3-11.
Rm	Is a register providing the address to branch to.

Operation

All these instructions cause a branch to the address indicated by *label* or contained in the register specified by *Rm*. In addition:

- The BL and BLX instructions write the address of the next instruction to LR, the link register R14.
- The BX and BLX instructions result in a UsageFault exception if bit[0] of *Rm* is 0.

BL and BLX instructions also set bit[0] of the LR to 1. This ensures that the value is suitable for use by a subsequent POP {PC} or BX instruction to perform a successful return branch.

Table 3-10 shows the ranges for the various branch instructions.

Table 3-10 Branch ranges

Instruction	Branch range
B *label*	−2KB to +2KB.
B*cond label*	−256 bytes to +254 bytes.
BL *label*	−16MB to +16MB.
BX *Rm*	Any value in register.
BLX *Rm*	Any value in register.

Restrictions

In these instructions:

- Do not use SP or PC in the BX or BLX instruction.
- For BX and BLX, bit[0] of *Rm* must be 1 for correct execution. Bit[0] is used to update the EPSR T-bit and is discarded from the target address.

Note

B*cond* is the only conditional instruction on the Cortex-M23 processor.

BX can be used an Exception or Function return.

Condition flags

These instructions do not change the flags.

Examples

```
B      loopA  ; Branch to loopA
BL     funC   ; Branch with link (Call) to function funC, return address
              ; stored in LR
BX     LR     ; Return from function call if LR contains a FUNC_RETURN value.
BLX    R0     ; Branch with link and exchange (Call) to a address stored
              ; in R0
BEQ    labelD ; Conditionally branch to labelD if last flag setting
              ; instruction set the Z flag, else do not branch.
```

3.7.2 BXNS and BLXNS

Branch and Exchange Non-secure, Branch with Link and Exchange Non-secure

Syntax

`BXNS <Rm>`

`BLXNS <Rm>`

Where:

`Rm` Is a register containing an address to branch to.

Operation

The `BXNS` instruction causes a branch to an address contained in *Rm* and conditionally causes a transition from the Secure to the Non-secure state.

The `BLXNS` instruction calls a subroutine at an address contained in *Rm* and conditionally causes a transition from the Secure to the Non-secure state.

For both `BXNS` and `BLXNS`, bit[0] indicates a transition to Non-secure state if value is 0, otherwise the target state remains Secure. `BLXNS` pushes the return address and partial PSR to the Secure stack and assigns R14 to a FNC_RETURN value.

These instructions are available for Secure state only. When the processor is in Non-secure state, these instructions are UNDEFINED and triggers a HardFault if executed.

Restrictions

PC and SP cannot be used for *Rm*.

Condition flags

These instructions do not change the flags.

Examples

```
LDR r0, =non_secure_function
MOVS r1, #1
BICS r0, r1 # Clear bit 0 of address in r0
BLXNS r0 ; Call Non-Secure function. This sets r14 to FUNC_RETURN value
BX
```

Note

For information about how to build a Secure image that uses a previously generated import library, see the *ARM® Compiler Software Development Guide*.

3.7.3 CBZ and CBNZ

Compare and Branch on Zero, Compare and Branch on Non-Zero.

Syntax

CB{N}Z <Rn>,< label>

Where:

cond	Is an optional condition code. See see *Conditional execution* on page 3-12.
Rn	Is the register holding the operand.
label	Is the branch destination.

Operation

Use the CBZ or CBNZ instructions to avoid changing the condition code flags and to reduce the number of instructions.

CBZ *Rn*, label does not change condition flags but is otherwise equivalent to:

```
CMP     Rn, #0
BEQ     label
```

CBNZ Rn, label does not change condition flags but is otherwise equivalent to:

```
CMP     Rn, #0
BNE     label
```

Restrictions

The restrictions are:

- *Rn* must be in the range of R0-R7.
- The branch destination must be within 4 to 130 bytes after the instruction.

Condition flags

These instructions do not change the flags.

Examples

```
CBZ    R5, target ; Forward branch if R5 is zero
CBNZ   R0, target ; Forward branch if R0 is not zero
```

3.8 Miscellaneous instructions

Table 3-11 shows the remaining Cortex-M23 instructions:

Table 3-11 Miscellaneous instructions

Mnemonic	Brief description	See
BKPT	Breakpoint	*BKPT* on page 3-54.
CPSID	Change Processor State, Disable Interrupts	*CPS* on page 3-55.
CPSIE	Change Processor State, Enable Interrupts	*CPS* on page 3-55.
DMB	Data Memory Barrier	*DMB* on page 3-56.
DSB	Data Synchronization Barrier	*DSB* on page 3-57.
ISB	Instruction Synchronization Barrier	*ISB* on page 3-58.
MRS	Move from special register to register	*MRS* on page 3-59.
MSR	Move from register to special register	*MSR* on page 3-60.
NOP	No Operation	*NOP* on page 3-61.
SEV	Send Event	*SEV* on page 3-62.
SG	Secure Gateway	*SG* on page 3-63.
SVC	Supervisor Call	*SVC* on page 3-64.
TT	Test Target	*TT, TTT, TTA, and TTAT* on page 3-65.
TTT	Test Target Unprivileged	*TT, TTT, TTA, and TTAT* on page 3-65.
TTA	Test Target Alternate Domain	*TT, TTT, TTA, and TTAT* on page 3-65.
TTAT	Test Target Alternate Domain Unprivileged	*TT, TTT, TTA, and TTAT* on page 3-65.
WFE	Wait For Event	*WFE* on page 3-67.
WFI	Wait For Interrupt	*WFI* on page 3-68.

3.8.1 BKPT

Breakpoint.

Syntax

`BKPT #imm`

where:

imm Is an integer in the range 0-255.

Operation

The `BKPT` instruction causes the processor to enter Debug state. Debug tools can use this to investigate system state when the instruction at a particular address is reached.

imm is ignored by the processor. If required, a debugger can use it to store additional information about the breakpoint.

The processor might also produce a HardFault or go into Lockup if a debugger is not attached or if debug is not enabled when a `BKPT` instruction is executed. See *Lockup* on page 2-33 for more information.

Restrictions

There are no restrictions.

Condition flags

This instruction does not change the flags.

Examples

```
BKPT #0     ; Breakpoint with immediate value set to 0x0.
```

3.8.2 CPS

Change Processor State.

Syntax

```
CPSID i

CPSIE i
```

Operation

`CPS` changes the PRIMASK special register values. `CPSID` causes interrupts to be disabled by setting PRIMASK. `CPSIE` causes interrupts to be enabled by clearing PRIMASK. See *Exception mask register* on page 2-8 for more information about these registers.

Restrictions

If the current mode of execution is not privileged, then this instruction behaves as a `NOP` and does not change the current state of PRIMASK.

Condition flags

This instruction does not change the condition flags.

Examples

```
    CPSID i ; Disable all interrupts except NMI and Hardfault.
If PRIS is set, PRIMASK_NS.PM rises the security level to 0x80, and does  not mask
Secure interrupts with a lower priority value.


    CPSIE i ; Enable interrupts (clear PRIMASK.PM)
```

3.8.3 DMB

Data Memory Barrier.

Syntax

```
DMB
```

Operation

`DMB` acts as a data memory barrier. It ensures that all explicit memory accesses that appear in program order before the `DMB` instruction are observed before any explicit memory accesses that appear in program order after the `DMB` instruction. `DMB` does not affect the ordering of instructions that do not access memory.

Restrictions

There are no restrictions.

Condition flags

This instruction does not change the flags.

Examples

```
DMB  ; Data Memory Barrier
```

3.8.4 DSB

Data Synchronization Barrier.

Syntax

```
DSB
```

Operation

`DSB` acts as a special data synchronization memory barrier. Instructions that come after the `DSB`, in program order, do not execute until the `DSB` instruction completes. The `DSB` instruction completes when all explicit memory accesses before it complete.

Restrictions

There are no restrictions.

Condition flags

This instruction does not change the flags.

Examples

```
DSB ; Data Synchronisation Barrier
```

3.8.5 ISB

Instruction Synchronization Barrier.

Syntax

`ISB`

Operation

`ISB` acts as an Instruction Synchronization Barrier. It flushes the pipeline of the processor, so that all instructions following the `ISB` are fetched from cache or memory again, after the `ISB` instruction has been completed.

Restrictions

There are no restrictions.

Condition flags

This instruction does not change the flags.

Examples

```
ISB  ; Instruction Synchronisation Barrier
```

3.8.6 MRS

Move the contents of a special register to a general-purpose register.

Syntax

`MRS Rd, spec_reg`

Where:

Rd Is the general-purpose destination register.

spec_reg Is one of the special-purpose registers: APSR, IPSR, EPSR, IEPSR, IAPSR, EAPSR, PSR, MSP, PSP, PRIMASK, or CONTROL. *spec_reg* can also be MSP_NS, PSP_NS, MSPLIM, PSPLIM, CONTROL_NS, PRIMASK_NS in Secure state.

Operation

MRS stores the contents of a special-purpose register to a general-purpose register. The MRS instruction can be combined with the MSR instruction to produce read-modify-write sequences, which are suitable for modifying a specific flag in the PSR.

See *MSR* on page 3-60.

Restrictions

In this instruction, *Rd* must not be SP or PC.

If the current mode of execution is not privileged, then the values of all registers other than the APSR read as zero.

If Non-secure code tries to access a register reserved to Secure state, then it reads as zero.

Condition flags

This instruction does not change the flags.

Examples

```
MRS  R0, PRIMASK ; Read PRIMASK value and write it to R0
```

3.8.7 MSR

Move the contents of a general-purpose register into the specified special register.

Syntax

`MSR spec_reg, Rn`

Where:

Rn Is the general-purpose source register.

spec_reg Is the special-purpose destination register: APSR, IPSR, EPSR, IEPSR, IAPSR, EAPSR, PSR, MSP, PSP, PRIMASK, or CONTROL. *spec_reg* can also be MSP_NS, PSP_NS, MSPLIM, PSPLIM, CONTROL_NS, PRIMASK_NS in Secure state.

Operation

MSR updates one of the special registers with the value from the register specified by *Rn*.

See *MRS* on page 3-59.

Restrictions

In this instruction, *Rn* must not be SP and must not be PC.

If the current mode of execution is not privileged, then all attempts to modify any register other than the APSR are ignored.

A write in Non-secure state to a register that is reserved to Secure is ignored.

Condition flags

This instruction updates the flags explicitly based on the value in *Rn* when PASR is written.

Examples

```
MSR  CONTROL, R1 ; Read R1 value and write it to the CONTROL register
```

3.8.8 NOP

No Operation.

Syntax

`NOP`

Operation

`NOP` performs no operation and is not guaranteed to be time consuming. The processor might remove it from the pipeline before it reaches the execution stage.

Restrictions

There are no restrictions.

Condition flags

This instruction does not change the flags.

Examples

```
NOP  ; No operation
```

3.8.9 SEV

Send Event.

Syntax

SEV

Operation

SEV sets the local event register, see *Power management* on page 2-34. This depends on your system. You can connect TXEV from other processors, in this case it can depends on SEV. However, peripherals might be connected to RXEV.

See also *WFE* on page 3-67.

Restrictions

There are no restrictions.

Condition flags

This instruction does not change the flags.

Examples

```
SEV ; Send Event
```

3.8.10 SG

Secure Gateway.

Syntax

`SG`

Operation

Secure Gateway marks a valid branch target for branches from Non-secure code that wants to call Secure code.

A linker is expected to generate a Secure Gateway operation as a part of the branch table for the *Non-secure Callable* (NSC) region.

There is no C intrinsic function for `SG`. ARM does not expect software developers to insert a Secure Gateway instruction inside C or C++ program code. It is expected that a linker generates the branch veneers that contain `SG` instructions and branches.

Note

For information about how to build a Secure image that uses a previously generated import library, see the *ARM® Compiler Software Development Guide*.

3.8.11 SVC

Supervisor Call.

Syntax

`SVC #imm`

Where:

imm Is an integer in the range 0-255.

Operation

The `SVC` instruction causes the `SVC` exception.

imm is ignored by the processor. If required, it can be retrieved by the exception handler to determine what service is being requested.

Restrictions

Executing the `SVC` instruction, while the current execution priority level is greater than or equal to that of the SVCall handler, results in a fault being generated.

Condition flags

This instruction does not change the flags.

Examples

```
SVC  #0x32 ; Supervisor Call (SVC handler can extract the immediate value
           ; by locating it through the stacked PC)
```

3.8.12 TT, TTT, TTA, and TTAT

Test Target (Alternate Domain, Unprivileged).

Syntax

{op} Rd, Rn, label

Where:

op Is one of:

- TT — *Test Target* (TT) queries the security state and access permissions of a memory location.
- TTT — *Test Target Unprivileged* (TTT) queries the security state and access permissions of a memory location for an unprivileged access to that location.
- TTA — *Test Target Alternate Domain* (TTA) queries the security state and access permissions of a memory location for a Non-secure access to that location. These instructions are only valid when executing in Secure state, and are UNDEFINED if used from Non-secure state.
- TTAT — *Test Target Alternate Domain Unprivileged* (TTAT) queries the security state and access permissions of a memory location for a Non-secure and unprivileged access to that location. These instructions are only valid when executing in Secure state, and are UNDEFINED if used from Non-secure state.

Rd Is the destination general-purpose register into which the status result of the target test is written.

Rn Is the general-purpose base register.

Operation

The instruction returns the security state and access permissions in the destination register, the contents of which are as follows:

Table 3-12 Security state and access permissions in the destination register

Bits	Name	Description
[7:0]	MREGION	The MPU region that the address maps to. This field is 0 if MRVALID is 0.
[15:8]	SREGION	The SAU region that the address maps to. This field is only valid if the instruction is executed from Secure state. This field is 0 if SRVALID is 0.
[16]	MRVALID	Set to 1 if the MREGION content is valid. Set to 0 if the MREGION content is invalid.
[17]	SRVALID	Set to 1 if the SREGION content is valid. Set to 0 if the SREGION content is invalid.
[18]	R	Read accessibility. Set to 1 if the memory location can be read according to the permissions of the selected MPU when operating in the current mode. For TTT and TTAT, this bit returns the permissions for unprivileged access, regardless of whether the current mode is privileged or unprivileged.

Table 3-12 Security state and access permissions in the destination register **(continued)**

Bits	Name	Description
[19]	RW	Read/write accessibility. Set to 1 if the memory location can be read and written according to the permissions of the selected MPU when operating in the current mode.
[20]	NSR	Equal to R AND NOT S. Can be used in combination with the LSLS (immediate) instruction to check both the MPU and SAU or IDAU permissions. This bit is only valid if the instruction is executed from Secure state and the R field is valid.
[21]	NSRW	Equal to RW AND NOT S. Can be used in combination with the LSLS (immediate) instruction to check both the MPU and SAU or IDAU permissions. This bit is only valid if the instruction is executed from Secure state and the RW field is valid.
[22]	S	Security. A value of 1 indicates the memory location is Secure, and a value of 0 indicates the memory location is Non-secure. This bit is only valid if the instruction is executed from Secure state.
[23]	IRVALID	IREGION valid flag. For a Secure request, indicates the validity of the IREGION field. Set to 1 if the IREGION content is valid. Set to 0 if the IREGION content is invalid. This bit is always 0 if the IDAU cannot provide a region number, the address is exempt from security attribution, or if the requesting TT instruction is executed from the Non-secure state.
[31:24]	IREGION	IDAU region number. Indicates the IDAU region number containing the target address. This field is 0 if IRVALID is 0.

Invalid fields are 0.

The MREGION field is invalid and 0 if any of the following conditions are true:

- The MPU is not present or MPU_CTRL.ENABLE is 0.
- The address did not match any enabled MPU regions.
- The address matched multiple MPU regions.
- TT was executed from an unprivileged mode, or TTA is executed and Non-secure state is unprivileged.

The R, RW, NSR, and NSRW bits are invalid and 0 if any of the following conditions are true:

- The address matched multiple MPU regions.
- TT is executed from an unprivileged mode, or TTA is executed and Non-secure state is unprivileged..

3.8.13 WFE

Wait For Event.

Syntax

```
WFE
```

Operation

See *Power management* on page 2-34.

Note

`WFE` is intended for power saving only. When writing software assume that `WFE` might behave as `NOP`.

Restrictions

There are no restrictions.

Condition flags

This instruction does not change the flags.

Examples

```
WFE  ; Wait for event
```

3.8.14 WFI

Wait for Interrupt.

Syntax

```
WFI
```

Operation

See *Power management* on page 2-34.

Note

`WFI` is intended for power saving only. When writing, software assumes that `WFI` might behave as a `NOP` operation.

Restrictions

There are no restrictions.

Condition flags

This instruction does not change the flags.

Examples

```
WFI ; Wait for interrupt
```

Chapter 4
Cortex-M23 Peripherals, Reference Material

The following sections are the reference material for the ARM Cortex-M23 core peripherals descriptions in a User Guide:

4.1 About the Cortex-M23 peripherals

The address map of the *Private Peripheral Bus* (PPB) is:

Table 4-1 Core peripheral register regions

Address	Core peripheral	Description
0xE000E008-0xE000E00F	System Control Space	Table 4-11 on page 4-11.
0xE000E010-0xE000E01F	Reserved	-
0xE000E010-0xE000E01F	System timer	Table 4-23 on page 4-25.
0xE000E100-0xE000E4EF	Nested Vectored Interrupt Controller	Table 4-2 on page 4-3.
0xE000ED00-0xE000ED3F	System Control Space	Table 4-11 on page 4-11.
0xE000ED90-0xE000EDCF	Memory Protection Unit[a]	Table 4-35 on page 4-35.
0xE000EF00-0xE000EF03	Nested Vectored Interrupt Controller	Table 4-2 on page 4-3.
0XE000ED00-0XE00EDEF	Security Attribution Unit	Table 4-28 on page 4-29.

a. Software can read the MPU Type Register at 0xE000ED90 to test for the presence of a *Memory Protection Unit* (MPU).

In register descriptions:

- The register *type* is described as follows:
  RW Read and write.
  RO Read-only.
  WO Write-only.
- The *required privilege* gives the privilege level required to access the register, as follows:
  Privileged Only privileged software can access the register.
  Unprivileged Both unprivileged and privileged software can access the register.

4.2 Nested Vectored Interrupt Controller

This section describes the *Nested Vectored Interrupt Controller* (NVIC) and the registers it uses. The NVIC supports:

- 0 to 239 interrupts.
- A programmable priority level of 0-192 in steps of 64 for each interrupt in Secure state. A higher level corresponds to a lower priority, so level 0 is the highest interrupt priority. In Non-secure state, this depends on the value of PRIS. See *Extended priority* on page 2-26.
- Level and pulse detection of interrupt signals.
- Interrupt tail-chaining.
- An external *Non-Maskable Interrupt* (NMI).
- An optional *Wake-up Interrupt Controller* (WIC).

The processor automatically stacks its state on exception entry and unstacks this state on exception exit, with no instruction overhead. This provides low latency exception handling. The hardware implementation of the NVIC registers is:

Table 4-2 NVIC register summary

Address	Name	Type	Reset value	Description
0xE000E100-0xE000E13C	NVIC_ISER0 - NVIC_ISER7	RW	0x00000000	*Interrupt Set-enable Registers* on page 4-5.
		-	-	
0xE002E100-0xE002E13C	NVIC_ISER0_NS - NVIC_ISER7_NS	-	0x00000000	
		-	-	Depending on NVIC_ITTNS, bits can be RAZ/WI from Non-secure state.
0xE000E180-0xE000E1BC	NVIC_ICER0 - NVIC_ICER7	RW	0x00000000	*Interrupt Clear-enable Registers* on page 4-5.
		-	-	
0xE002E180-0xE002E1BC	NVIC_ICER0_NS - NVIC_ICER7_NS	-	0x00000000	
		-	-	Depending on NVIC_ITTNS, bits can be RAZ/WI from Non-secure state.
0xE000E200-0xE000E23C	NVIC_ISPR0 - NVIC_ISPR7	RW	0x00000000	*Interrupt Set-pending Registers* on page 4-6.
		-	-	
0xE002E200-0xE002E23C	NVIC_ISPR0_NS - NVIC_ISPR7_NS	-	0x00000000	
		-	-	Depending on NVIC_ITTNS, bits can be RAZ/WI from Non-secure state.

Table 4-2 NVIC register summary (continued)

Address	Name	Type	Reset value	Description
0xE002E280-0xE000E2BC	NVIC_ICPR0 - NVIC_ICPR7	RW	0x00000000	*Interrupt Clear-pending Registers* on page 4-7.
		-	-	
0xE000E280-0xE002E2BC	NVIC_ICPR0_NS - NVIC_ICPR7_NS	-	0x00000000	Depending on NVIC_ITTNS, bits can be RAZ/WI from Non-secure state.
		-	-	
0xE000E300-0xE000E33C	NVIC_IABR0 - NVIC_ISABR7	RO	0x00000000	*Interrupt Active Bit Registers* on page 4-7.
		-	-	
0xE002E300-0xE002E33C	NVIC_IABR0_NS - NVIC_IABR7_NS	-	0x00000000	
		-	-	Depending on NVIC_ITTNS, bits can be RAZ/WI from Non-secure state.
0xE000E380-0xE000E3BC	NVIC_ITNS0 - NVIC_ITNS7	RW	0x00000000	*Interrupt Target Non-secure Registers* on page 4-8.
0xE000E400-0xE000E5DC	NVIC_IPR0 - NVIC_IPR119	RW	0x00000000	*Interrupt Priority Registers* on page 4-8.
		-	-	
0xE002E400-0xE002E5DC	NVIC_IPR0_NS - NVIC_IPR119_NS	-	0x00000000	
		-	-	Depending on NVIC_ITTNS, bits can be RAZ/WI from Non-secure state.

4.2.1 Accessing the Cortex-M23 NVIC registers using CMSIS

CMSIS functions enable software portability between different Cortex-M profile processors.

To access the NVIC registers when using CMSIS, use the following functions:

Table 4-3 CMSIS access NVIC functions

CMSIS function	Description
`void NVIC_EnableIRQ(IRQn_Type IRQn)`[a]	Enables an interrupt or exception.
`void NVIC_DisableIRQ(IRQn_Type IRQn)`[a]	Disables an interrupt or exception.
`void NVIC_SetPendingIRQ(IRQn_Type IRQn)`[a]	Sets the pending status of an interrupt or exception to 1.
`void NVIC_ClearPendingIRQ(IRQn_Type IRQn)`[a]	Clears the pending status of an interrupt or exception to 0.
`uint32_t NVIC_GetPendingIRQ(IRQn_Type IRQn)`[a]	Reads the pending status of an interrupt or exception. This function returns a non-zero value if the pending status is set to 1.
`void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)`[a]	Sets the priority of an interrupt or exception with configurable priority level to 1.

Table 4-3 CMSIS access NVIC functions (continued)

CMSIS function	Description
`uint32_t NVIC_GetPriority(IRQn_Type IRQn)`[a]	Reads the priority of an interrupt or exception with configurable priority level. This function returns the current priority level.
`uint32_t SetTargetState(IRQn_Type IRQn)`[a]	Sets the interrupt target field in the NVIC.
`uint32_t NVIC_GETTargetState(IRQn_Type IRQn)`[a]	Gets interrupt target state.
`uint32_t ClearTargetState(IRQn_Type IRQn)`[a]	Clears the interrupt target field in the Non-secure NVIC when in Secure state.

a. The input parameter `IRQn` is the IRQ number, see Table 2-13 on page 2-23 for more information.

4.2.2 Interrupt Set-enable Registers

The NVIC_ISER0-NVIC_ISER7 enable interrupts, and shows which interrupts are enabled. See the register summary in Table 4-2 on page 4-3 for the register attributes.

Register bits can be RAZ/WI depending on the value of ITNS.

The bit assignments are:

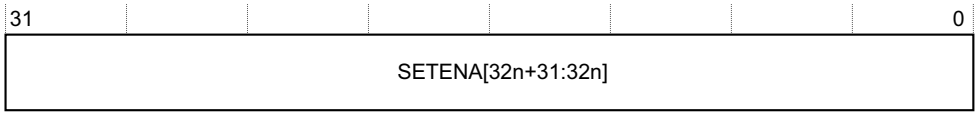


Table 4-4 NVIC_ISERn bit assignments

Bits	Name	Function
[31:0]	SETENA	Interrupt set-enable bits. Write: 0 = No effect. 1 = Enable interrupt. Read: 0 = Interrupt disabled. 1 = Interrupt enabled.

If a pending interrupt is enabled, the NVIC activates the interrupt based on its priority. If an interrupt is not enabled, asserting its interrupt signal changes the interrupt state to pending, but the NVIC never activates the interrupt, regardless of its priority.

4.2.3 Interrupt Clear-enable Registers

The NVIC_ICER0-NVIC_ICER7 disable interrupts, and show which interrupts are enabled. See the register summary in Table 4-2 on page 4-3 for the register attributes.

Register bits can be RAZ/WI depending on the value of ITNS.

The bit assignments are:

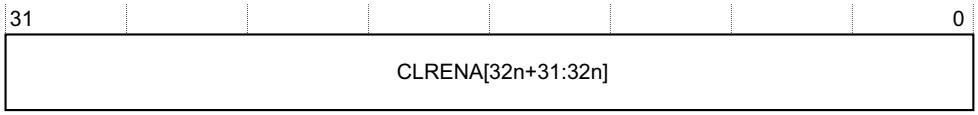


Table 4-5 NVIC_ICERn bit assignments

Bits	Name	Function
[31:0]	CLRENA	Interrupt clear-enable bits. Write: 0 = No effect. 1 = Disable interrupt. Read: 0 = Interrupt disabled. 1 = Interrupt enabled.

4.2.4 Interrupt Set-pending Registers

The NVIC_ISPR0-NVIC_ISPR7 force interrupts into the pending state, and shows which interrupts are pending. See the register summary in Table 4-2 on page 4-3 for the register attributes.

Register bits can be RAZ/WI depending on the value of ITNS.

The bit assignments are:

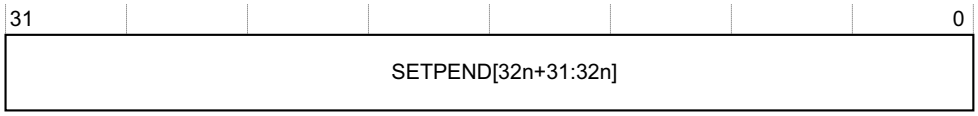


Table 4-6 NVIC_ISPRn bit assignments

Bits	Name	Function
[31:0]	SETPEND	Interrupt set-pending bits. Write: 0 = No effect. 1 = Changes interrupt state to pending. Read: 0 = Interrupt is not pending. 1 = Interrupt is pending.

Note

Writing 1 to the NVIC_ISPR bit corresponding to:

- An interrupt that is pending has no effect.
- A disabled interrupt sets the state of that interrupt to pending.

4.2.5 Interrupt Clear-pending Registers

The NVIC_ICPR0-NVIC_ICPR9 remove the pending state from interrupts, and shows which interrupts are pending. See the register summary in Table 4-2 on page 4-3 for the register attributes.

Register bits can be RAZ/WI depending on the value of ITNS.

The bit assignments are:

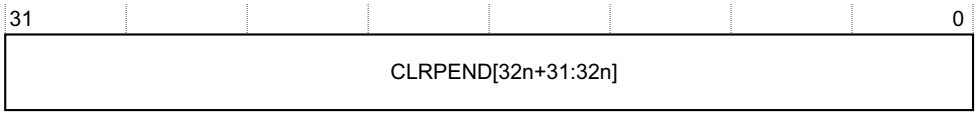


Table 4-7 NVIC_ICPRn bit assignments

Bits	Name	Function
[31:0]	CLRPEND	Interrupt clear-pending bits. Write: 0 = No effect. 1 = Removes pending state and interrupt. Read: 0 = Interrupt is not pending. 1 = Interrupt is pending.

Note

Writing 1 to an NVIC_ICPR bit does not affect the active state of the corresponding interrupt.

4.2.6 Interrupt Active Bit Registers

The NVIC_IABR0-NVIC_IABR7 indicate the active state of each interrupt. See the register summary in Table 4-2 on page 4-3 for the register attributes.

Register bits can be RAZ/WI depending on the value of ITNS.

The bit assignments are:

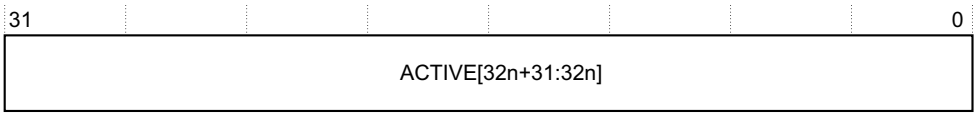


Table 4-8 NVIC_IABRn bit assignments

Bits	Name	Function
[31:0]	ACTIVE	Active state bits. 0 = The interrupt is not active. 1 = The interrupt is active.

4.2.7 Interrupt Target Non-secure Registers

The NVIC_ITNS0-NVIC_ITNS7 determine, for each group of 32 interrupts, whether each interrupt targets Non-secure or Secure state. See the register summary in Table 4-2 on page 4-3 for the register attributes.

This register is accessible from Secure state only.

The bit assignments are:

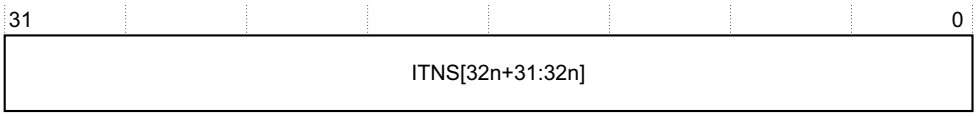


Table 4-9 NVIC_ITNSn bit assignments

Bits	Name	Function
[31:0]	ITNS	Interrupt Targets Non-secure bits. 0 = The interrupt targets Secure state. 1 = The interrupt targets Non-secure state.

4.2.8 Interrupt Priority Registers

The NVIC_IPR0-NVIC_IPR59 registers provide an 8-bit priority field for each interrupt. These registers are only word-accessible. See the register summary in Table 4-2 on page 4-3 for their attributes. Each register holds four priority fields as shown:

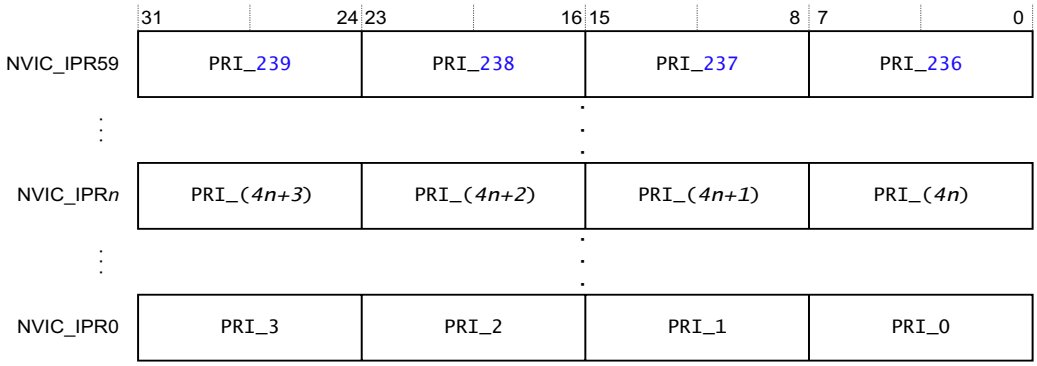


Table 4-10 NVIC_IPRn bit assignments

Bits	Name	Function
[31:24]	Priority, byte offset 3	Each priority field holds a priority value. The priority depends on the value of PRIS for exceptions targeting the Non-secure state. The lower the value, the greater the priority of the corresponding interrupt. The processor implements only bits[7:6] of each field, bits[5:0] read as zero and ignore writes. This means writing 255 to a priority register saves value 192 to the register.
[23:16]	Priority, byte offset 2	
[15:8]	Priority, byte offset 1	
[7:0]	Priority, byte offset 0	

See *Accessing the Cortex-M23 NVIC registers using CMSIS* on page 4-4 for more information about the access to the interrupt priority array, which provides the software view of the interrupt priorities.

Find the NVIC_IPR number and byte offset for interrupt *M* as follows:

- The corresponding NVIC_IPR number, *N*, is given by *N* = *N* DIV 4.
- The byte offset of the required Priority field in this register is *M* MOD 4, where:
  - Byte offset 0 refers to register bits[7:0].
  - Byte offset 1 refers to register bits[15:8].
  - Byte offset 2 refers to register bits[23:16].
  - Byte offset 3 refers to register bits[31:24].

Priority values depend on the value of PRIS as described in *Extended priority* on page 2-26.

Register bits can be RAZ/WI depending on the value of ITNS.

4.2.9 Level-sensitive and pulse interrupts

The processor supports both level-sensitive and pulse interrupts. Pulse interrupts are also described as edge-triggered interrupts.

A level-sensitive interrupt is held asserted until the peripheral deasserts the interrupt signal. Typically this happens because the ISR accesses the peripheral, causing it to clear the interrupt request. A pulse interrupt is an interrupt signal sampled synchronously on the rising edge of the processor clock. To ensure the NVIC detects the interrupt, the peripheral must assert the interrupt signal for at least one clock cycle, during which the NVIC detects the pulse and latches the interrupt.

When the processor enters the ISR, it automatically removes the pending state from the interrupt, see *Hardware and software control of interrupts*. For a level-sensitive interrupt, if the signal is not deasserted before the processor returns from the ISR, the interrupt becomes pending again, and the processor must execute its ISR again. This means that the peripheral can hold the interrupt signal asserted until it no longer requires servicing.

See <reference required> for details of which interrupts are level-sensitive and which are pulsed.

Hardware and software control of interrupts

The Cortex-M23 processor latches all interrupts. A peripheral interrupt becomes pending for one of the following reasons:

- The NVIC detects that the interrupt signal is active and the corresponding interrupt is not active.
- The NVIC detects a rising edge on the interrupt signal.
- Software writes to the corresponding interrupt set-pending register bit, see *Interrupt Set-pending Registers* on page 4-6.

A pending interrupt remains pending until one of the following occurs:

- The processor enters the ISR for the interrupt. This changes the state of the interrupt from pending to active. Then:
  - For a level-sensitive interrupt, when the processor returns from the ISR, the NVIC samples the interrupt signal. If the signal is asserted, the state of the interrupt changes to pending, which might cause the processor to immediately reenter the ISR. Otherwise, the state of the interrupt changes to inactive.

— For a pulse interrupt, the NVIC continues to monitor the interrupt signal, and if this is pulsed the state of the interrupt changes to pending and active. In this case, when the processor returns from the ISR the state of the interrupt changes to pending, which might cause the processor to immediately reenter the ISR.

If the interrupt signal is not pulsed while the processor is in the ISR, when the processor returns from the ISR the state of the interrupt changes to inactive.

- Software writes to the corresponding interrupt clear-pending register bit.

  For a level-sensitive interrupt, if the interrupt signal is still asserted, the state of the interrupt does not change. Otherwise, the state of the interrupt changes to inactive.

  For a pulse interrupt, state of the interrupt changes to:

  — Inactive, if the state was pending.

  — Active, if the state was active and pending.

4.2.10 NVIC usage hints and tips

Ensure software uses correctly aligned register accesses. The processor does not support unaligned accesses to NVIC registers.

An interrupt can enter pending state even if it is disabled. Disabling an interrupt only prevents the processor from taking that interrupt.

Before programming VTOR to relocate the vector table, ensure the vector table entries of the new vector table are set up for fault handlers, NMI, and all enabled exception like interrupts. For more information, see *Vector Table Offset Register* on page 4-15.

NVIC programming hints

Software uses the `CPSIE i` and `CPSID i` instructions to enable and disable interrupts. The CMSIS provides the following intrinsic functions for these instructions:

```
void __disable_irq(void) // Disable Interrupts
void __enable_irq(void)  // Enable Interrupts
```

In addition, the CMSIS provides functions for NVIC control, listed in *Accessing the Cortex-M23 NVIC registers using CMSIS* on page 4-4.

The input parameter `IRQn` is the IRQ number, see Table 2-13 on page 2-23 for more information. For more information about these functions, see the CMSIS documentation.

4.3 System Control Space

The *System Control Space* (SCS) provides system implementation information, and system control. This includes configuration, control, and reporting of the system exceptions. The SCS registers are:

Table 4-11 Summary of the SCS registers

Address	Name	Type	Reset value	Description
0xE000ED00	CPUID_S	RO	0x410CD200	*CPUID Register* on page 4-12.
0xE002ED00	CPUID_NS	RO	0x410CD200	
0xE000ED04	ICSR_S	RW[a]	0x00000000	*Interrupt Control and State Register* on page 4-12.
0xE002ED04	ICSR_NS		0x00000000	
0xE000ED08	VTOR	RW	0x00000000	*Vector Table Offset Register* on page 4-15.
0xE002ED08	VTOR_NS	RW	0x00000000	
0xE000ED0C	AIRCR_S	RW[b]	0xFA050000	*Application Interrupt and Reset Control Register* on page 4-16.
0xE002ED0C	AIRCR_NS		0xFA050000	
0xE000ED10	SCR_S	RW	0x00000000	*System Control Register* on page 4-17.
0xE002ED10	SCR_NS	RW	0x00000000	
0xE000ED14	CCR_S	RW	0x00000204	*Configuration and Control Register* on page 4-19.
0xE002ED14	CCR_NS	RW	0x00000204	
0xE000ED1C	SHPR2_S	RW	0x00000000	*System Handler Priority Register 2* on page 4-21.
0xE002ED1C	SHPR2_NS	RW	0x00000000	
0xE000ED20	SHPR3_S	RW	0x00000000	*System Handler Priority Register 3* on page 4-21.
0xE002ED20	SHPR3_NS	RW	0x00000000	
0xE000ED24	SHCSR_S	RW	0x00000000	*System Handler Control and State Register* on page 4-21.
0xE002ED24	SHCSR_NS	RW	0x00000000	
0xE000E008	ACTLR_S	RW	0x00000000	*Auxiliary Control Register* on page 4-23.
0xE002E008	ACTLR_NS	RW	0x00000000	

a. See the register description for more information.

b. See the register description for more information.

4.3.1 The CMSIS mapping of the Cortex-M23 SCS registers

To improve software efficiency, the CMSIS simplifies the SCS register presentation. In the CMSIS, `SHP[0]` accesses SHPR2 and `SHP[1]` accesses SHPR3.

4.3.2 CPUID Register

The CPUID register contains the processor part number, version, and implementation information. See the register summary in Table 4-11 on page 4-11 for its attributes. The bit assignments are:

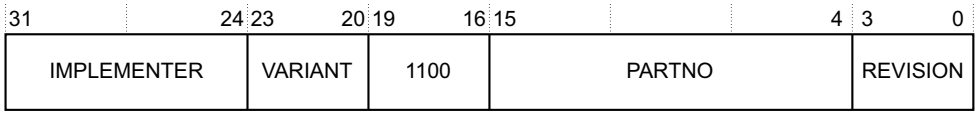


Table 4-12 CPUID register bit assignments

Bits	Name	Function
[31:24]	IMPLEMENTER	Implementer code: 0x41 = ARM.
[23:20]	VARIANT	Major revision number *n* in the r*n*p*m* revision status: 0x1 = Revision 1.
[19:16]	ARCHITECTURE	Constant that defines the architecture of the processor: 0xC = ARMv8-M architecture.
[15:4]	PARTNO	Part number of the processor: 0xD20 = Cortex-M23.
[3:0]	REVISION	Minor revision number *m* in the r*n*p*m* revision status: 0x0 = Patch 0.

4.3.3 Interrupt Control and State Register

The ICSR:

- Provides:
  - A set-pending bit for the *Non-Maskable Interrupt* (NMI) exception.
  - Set-pending and clear-pending bits for the PendSV and SysTick exceptions.
- Indicates:
  - The exception number of the highest priority pending exception.

This register is banked between Secure and Non-secure state on a bit by bit basis.

See the register summary in Table 4-11 on page 4-11 for the ICSR attributes.

The bit assignments are:

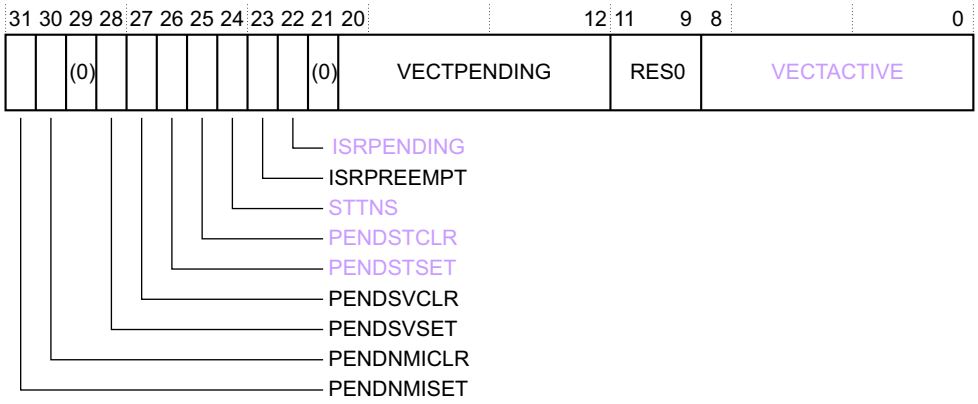


Table 4-13 ICSR bit assignments

Bits	Name	Type	Function
[31]	PENDNMISET	WO	NMI set-pending bit. Write: 0 = No effect. 1 = Changes NMI exception state to pending. Read: 0 = NMI exception is not pending. 1 = NMI exception is pending. If AIRCR.BFHFNMINS is 0 this bit is RAZ/WI from Non-secure state.
[30]	PENDNMICLR	WO/ RAZ	NMI bit-pending bit. 0 = No effect. 1 = Clear pending status. If AIRCR.BFHFNMINS is 0 this bit is RAZ/WI from Non-secure state.
[29]	-	-	Reserved.
[28]	PENDSVSET	RW	This bit is banked between security states. PendSV set-pending bit. Write: 0 = No effect. 1 = Sets the PendSV exception pending. Read: 0 = PendSV exception is not pending. 1 = PendSV exception is pending.
[27]	PENDSVCLR	WO	This bit is banked between security states. PendSV clearing-pending bit. 0 = No effect. 1 = Clear pending status.

Table 4-13 ICSR bit assignments (continued)

Bits	Name	Type	Function
[26]	PENDSTSET	RW	This bit is banked between security states if two SysTicks are implemented. This bit is RAZ/WI from Non-secure state if one SysTick is implemented and STTNS=0. SysTick set-pending bit. Write: 0 = No effect. 1 = Sets the SysTick exception pending for the selected Security state. Read: 0 = SysTick exception is not pending. 1 = SysTick exception is pending.
[25]	PENDSTCLR	WO	This bit is banked between security states if two SysTicks are implemented. This bit is RAZ/WI from Non-secure state if one SysTick is implemented and STTNS=0. SysTick clear-pending bit. 0 = No effect. 1 = Clear pending status.
[24]	STTNS	RW	SysTick Targets Non-secure bit. When one SysTick is implemented: 0 = SysTick is Secure. 1 = SysTick is Non-secure. This bit behaves as RAZ/WI when: • Accessed from Non-secure state • No SysTick is implemented. • Two SysTicks are implemented. • The Security Extension is not implemented.
[23]	ISRPREEMPT	RO	Interrupt preempt bit. 0 = Will not service. 1 = Will service a pending exception. When the debug extensions are not implemented, this bit is RAZ/WI.
[22]	ISRPENDING	RO	Interrupt pending bit. 0 = No external interrupt is pending. 1 = External interrupt is pending. When the debug extensions are not implemented, this bit is RAZ/WI.
[21]	-	-	Reserved.

Table 4-13 ICSR bit assignments (continued)

Bits	Name	Type	Function
[20:12]	VECTPENDING	RO	Vector pending bit. 0 = No pending and enabled exception. Non-zero = Exception number.
[11:9]	-	-	Reserved.
[8:0]	VECTACTIVE	RO	Vector active bit. 0 = Thread mode. Non-zero = Exception number. When the debug extensions are not implemented, this bit is RAZ/WI.

When you write to the ICSR, the effect is UNPREDICTABLE if you:

- Write 1 to the PENDSVSET bit and write 1 to the PENDSVCLR bit.
- Write 1 to the PENDSTSET bit and write 1 to the PENDSTCLR bit.

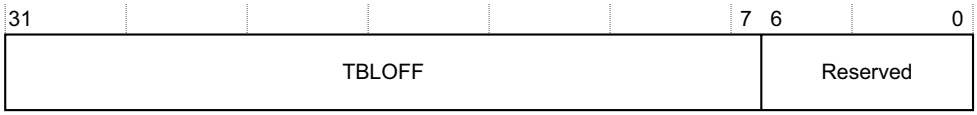
4.3.4 Vector Table Offset Register

The VTOR indicates the offset of the vector table base address from memory address `0x00000000`. See the register summary in Table 4-14 for its attributes.

With the Security Extension implemented, the following occurs:

- Exceptions that target Secure state use VTOR_S to determine the base address of the Secure vector table.
- Exceptions that target Non-secure state use VTOR_NS to determine the base address of the Non-secure vector table.

The bit assignments are:



<See the configurable information after the register description table for information about the configuration of the boundary between TBLOFF field and the [6:0] field that follows.>

Table 4-14 VTOR bit assignments

Bits	Name	Function
[31:7]	TBLOFF	Vector table base offset field. It contains bits[31:7] of the offset of the table base from the bottom of the memory map. <See the configurable information after this table for information about the configuration of this field and the [6:0] field that follows.>
[6:0]	-	Reserved.

<Configure the next statement to give the information required for your implementation, the statement reminds you of how to determine the alignment requirement.> The last bit of the Exception number bit field depends on the number of interrupts implemented.

- 0-47 interrupts = [31:7].
- 48-111 interrupts = [31:8].
- 112-239 interrupts = [31:9].

4.3.5 Application Interrupt and Reset Control Register

The AIRCR provides endian status for data accesses and reset control of the system. See the register summary in Table 4-11 on page 4-11 and Table 4-15 for its attributes.

To write to this register, you must write `0x05FA` to the VECTKEY field, otherwise the processor ignores the write.

This register is banked between Secure and Non-secure state on a bit by bit basis.

The bit assignments are:

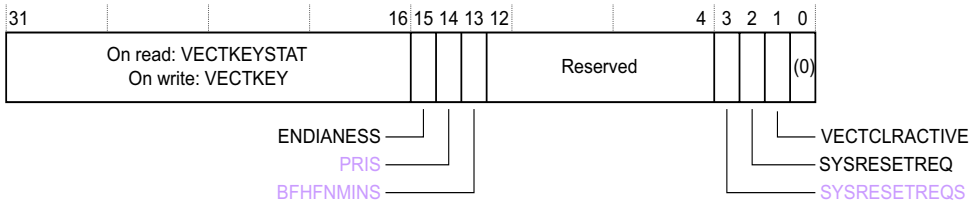


Table 4-15 AIRCR bit assignments

Bits	Name	Function
[31:16]	VECTKEY	Vector key bits. On writes, write `0x05FA` to VECTKEY, otherwise the write is ignored. This bit is not banked between Security states.
[31:16]	VECTKEYSTAT	Vector key status bits. On reads, this field reads as `0xFA05`.
[15]	ENDIANESS	Data endianness bits. 0 = Little-endian. 1 = Big-endian. This bit is not banked between Security states.
[14]	PRIS	Priority Secure exceptions bit. 0 = Priority ranges of Secure and Non-secure exceptions are identical. 1 = Non-secure exceptions are de-prioritized. This bit is not banked between Security states and it is *RES0* when the Security Extension is not implemented.

Table 4-15 AIRCR bit assignments (continued)

Bits	Name	Function
[13]	BFHFNMINS	BusFault, HardFault, and NMI Non-secure enable bit. 0 = BusFault, HardFault, and NMI are Secure. 1 = BusFault and NMI are Non-secure and exceptions can target Non-secure HardFault. This bit is not banked between Security states it is *RES0* when the Security Extension is not implemented.
[12:4]	-	Reserved.
[13]	SYSRESETREQS	System reset request Secure only bit. 0 = SYSRESETREQ functionality is available to both security states. 1 = SYSRESETREQ functionality is available to Secure state. This bit is not banked between security states. In Secure state, this bit is RAZ/WI.
[2]	SYSRESETREQ	System reset request bit. 0 = Do not request a system reset. 1 = Request a system reset. This bit is not banked between security states.
[1]	VECTCLRACTIVE	Clear active state bit. 0 = Do not clear active state. 1 = Clear active state. This bit is WO and can only be written when the processor is in Halt state. This bit is not banked between security states.
[0]	-	Reserved.

4.3.6 System Control Register

The SCR controls features of entry to and exit from low-power state. See the register summary in Table 4-11 on page 4-11 for its attributes.

This register is banked between Secure and Non-secure state on a bit by bit basis.

The bit assignments for SCR_S and SCR_NS are:

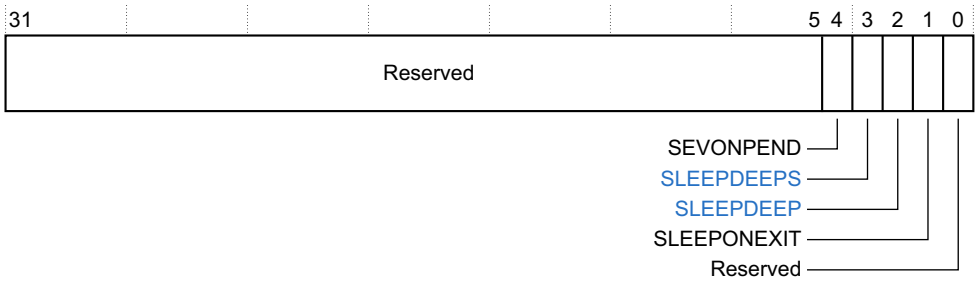


Table 4-16 SCR bit assignments

Bits	Name	Function
[31:5]	-	Reserved.
[4]	SEVONPEND	This bit is banked between security states. Send Event on Pending bit: 0 = Only enabled interrupts or events can wakeup the processor, disabled interrupts are excluded. 1 = Enabled events and all interrupts, including disabled interrupts, can wakeup the processor. When an event or interrupt becomes pending, the event signal wakes up the processor from WFE. If the processor is not waiting for an event, the event is registered and affects the next WFE. The processor also wakes up from WFE on execution of an SEV instruction or an external event.
[3]	SLEEPDEEPS	Controls whether the SLEEPDEEP bit is only accessible from the Secure state: 0 = The SLEEPDEEP bit is accessible from both security states. 1 = The SLEEPDEEP bit behaves as RAZ/WI when accessed from the Non-secure state.

Table 4-16 SCR bit assignments (continued)

Bits	Name	Function
[2]	SLEEPDEEP	Controls whether the processor uses sleep or deep sleep as its low-power mode: 0 = Sleep. 1 = Deep sleep. This bit is not banked between security states.
[1]	SLEEPONEXIT	This bit is banked between security states. Indicates sleep-on-exit when returning from Handler mode to Thread mode: 0 = Do not sleep when returning to Thread mode. 1 = Enter sleep, or deep sleep, on return from an ISR to Thread mode. Setting this bit to 1 enables an interrupt driven application to avoid returning to an empty main application.
[0]	-	Reserved.

4.3.7 Configuration and Control Register

The CCR is a read-only register and indicates some aspects of the behavior of the Cortex-M23 processor. See the register summary in Table 4-11 on page 4-11 for the CCR attributes.

This register is banked between Secure and Non-secure state.

The bit assignments for CCR_S and CCR_NS are:

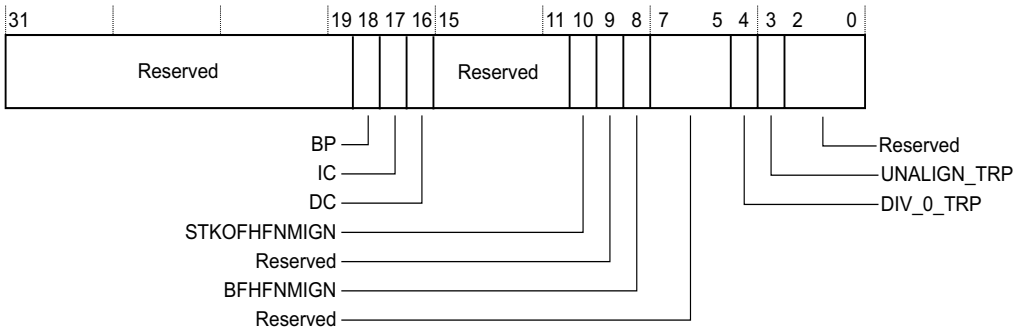


Table 4-17 CCR bit assignments

Bits	Name	Function
[31:19]	-	Reserved.
[18]	BP	RAZ/WI.
[17]	IC	RAZ/WI.
[16]	DC	RAZ/WI.

Table 4-17 CCR bit assignments (continued)

Bits	Name	Function
[15:11]	-	Reserved.
[10]	STKOFHFNMIGN	0 = RAZ/WI.
[9]	-	*RES1*.
[8]	BFHFNMIGN	0 = RAZ/WI.
[7:5]	-	Reserved.
[4]	DIV_0_TRP	RAZ/WI.
[3]	UNALIGN_TRP	1 = RAO/WI.
[2:0]	-	Reserved.

4.3.8 System Handler Priority Registers

The SHPR2-SHPR3 registers set the priority level, 0 to 192, of the system exception handlers that have configurable priority.

The SHPR2-SHPR3 registers are word accessible. See the register summary in Table 4-11 on page 4-11 for their attributes.

To access the system exception priority level using CMSIS, use the following CMSIS functions:

- `uint32_t NVIC_GetPriority(IRQn_Type IRQn)`
- `void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)`

The input parameter `IRQn` is the IRQ number, see Table 2-13 on page 2-23 for more information.

The system handlers, and the priority field and register for each handler are:

Table 4-18 System fault handler priority fields

Handler	Field	Register description
SVCall	PRI_11	*System Handler Priority Register 2* on page 4-21.
PendSV	PRI_14	*System Handler Priority Register 3* on page 4-21.
SysTick	PRI_15	

Each PRI_N field is 8 bits wide, but the processor implements only bits[7:6] of each field. Bits[5:0] read as zero and ignore writes.

If one SysTick is implemented, the SysTick handler is not banked. In this case, STTNS indicates whether it can be written by Non-secure or not.

If two SysTicks are implemented, the SysTick handler is banked between security states.

The SVCall and PendSV handlers are always banked between security states.

Priorities values depend on the value of PRIS, as described in *Extended priority* on page 2-26.

System Handler Priority Register 2

This register is banked between Secure and Non-secure state.

The bit assignments for SHPR2_S and SHPR2_NS are:

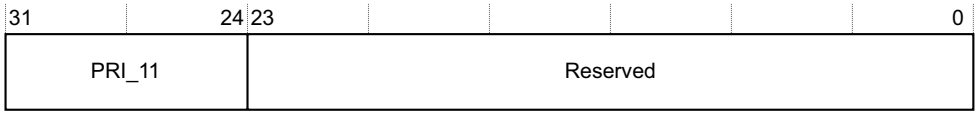


Table 4-19 SHPR2 register bit assignments

Bits	Name	Function
[31:24]	PRI_11	Priority of system handler 11, SVCall.
[23:0]	-	Reserved.

System Handler Priority Register 3

This register is banked between Secure and Non-secure state.

The bit assignments for SHPR3_S and SHPR3_NS are:

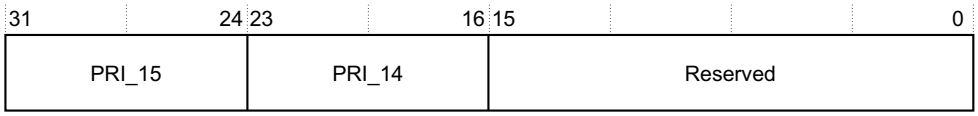


Table 4-20 SHPR3 register bit assignments

Bits	Name	Function
[31:24]	PRI_15	Priority of system handler 15, SysTick exception[a].
[23:16]	PRI_14	Priority of system handler 14, PendSV.
[15:0]	-	Reserved.

a. This is Reserved when the SysTick timer is not implemented. If the Security Extension and two SysTicks are implemented, it is banked between security states. If the Security Extension, one SysTick is implemented, and STTNIS is 1, then it is RAZ/WI from Non-secure state.

4.3.9 System Handler Control and State Register

The SHCSR provides access to the active and pending status of system exceptions.

This register is banked between Secure and Non-secure state on a bit by bit basis.

The bit assignments for SHCSR_S and SHCSR_NS are:

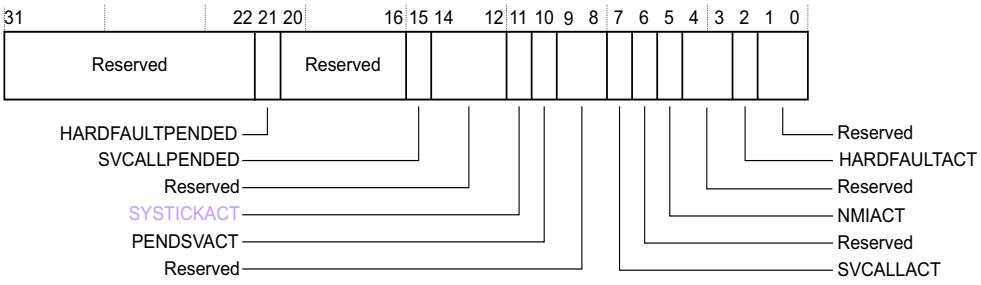


Table 4-21 SHCSR bit assignments

Bits	Name	Function
[31:22]	-	Reserved.
[21]	HARDFAULTPENDED	This bit is banked between security states. HardFault exception pended state bit. 0 = HardFault exception is not pending for the selected security state. 1 = HardFault exception is pending for the selected security state. If AIRCR.BFHFNMINS is set to zero, the Non-secure HardFault exception does not preempt.
[20:16]	-	Reserved.
[15]	SVCALLPENDED	This bit is banked between security states. SVCall exception pended state bit. 0 = SVCall exception is not pending for the selected security state. 1 = SVCall exception is pending for the selected security state.
[14:12]	-	Reserved.
[11]	SYSTICKACT	If two SysTick timers are implemented, this bit is banked between security states. SysTick exception active state bit. 0 = SysTick exception is not active for the selected security state. 1 = SysTick exception is active for the selected security state. If less than two SysTick timers are implemented when the Security Extension is implemented, this bit is not banked between Security states, and if AIRCR.STTNS is zero this bit is RAZ/WI from Non-secure state.
[10]	PENDSVACT	This bit is banked between security states. PendSV exception active state bit. 0 = PendSV exception is not active for the selected security state. 1 = PendSV exception is active for the selected security state.

Table 4-21 SHCSR bit assignments (continued)

Bits	Name	Function
[9:8]	-	Reserved.
[7]	SVCALLACT	This bit is banked between security states. SVCall exception active state bit. 0 = SVCall exception is not active for the selected security state. 1 = SVCall exception is active for the selected security state.
[6]	-	Reserved.
[5]	NMIACT	NMI exception active state bit. 0 = NMI exception is not active. 1 = NMI exception is active.
[4:3]	-	Reserved.
[2]	HARDFAULTACT	This bit is banked between security states. HardFault exception active state bit. 0 = HardFault exception is not active for the selected security state. 1 = HardFault exception is active for the selected security state.
[1:0]	-	Reserved.

4.3.10 Auxiliary Control Register

The ACTLR contains several fields that allow software to control the processor features and functionality.

The bit assignments are:

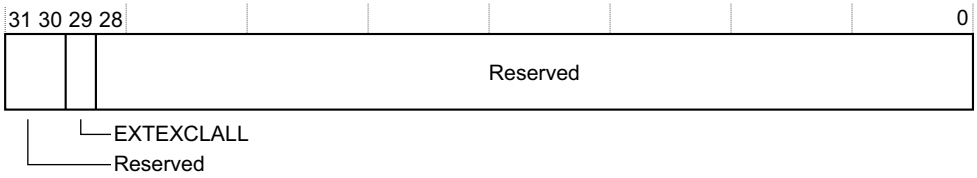


Table 4-22 ACTLR bit assignments

Bits	Name	Function
[31:30]	-	RAZ/WI.
[29]	EXTEXCLALL	0 = LDREX and STREX instructions use the global monitor when hitting in a shared region, either in the default memory map, or in a shared MPU region. **Note** Shared region: Accesses to Device regions in the ranges 0x40000000-0x5fffffff and 0xc0000000-0xffffffff do not use the Global Exclusive Monitor when ACTLR.EXTEXCLALL is 0 and the default memory map is used. 1 = LDREX and STREX instructions always use the global exclusive monitor.
[28:0]	-	RAZ/WI.

4.3.11 SCS usage hints and tips

Ensure software uses aligned 32-bit word size transactions to access all the SCS registers.

4.4 System timer, SysTick

If the Security Extension is not implemented, SysTick timers can be present or absent. You can configure your Cortex-M23 processor to have up to two SysTick timers.

If the Security Extension is implemented, SysTick timers can be present for Secure state, present for both Secure and Non-secure states, or absent.

Note

If you configure your Cortex-M23 processor to have one SysTick timer, the SysTick timer can be configured to be in Secure or Non-secure state using the STTNS bit in the ICSR register. This bit is programmable only in Secure state.

When enabled, the timer counts down from the reload value to zero, reloads (wraps to) the value in the SYST_RVR on the next clock cycle, then decrements on subsequent clock cycles. Writing a value of zero to the SYST_RVR disables the counter on the next wrap. When the counter transitions to zero, the COUNTFLAG status bit is set to 1. Reading SYST_CSR clears the COUNTFLAG bit to 0. Writing to the SYST_CVR clears the register and the COUNTFLAG status bit to 0. The write does not trigger the SysTick exception logic. Reading the register returns its value at the time it is accessed.

Note

When the processor is halted for debugging, the counter does not decrement.

The system timer registers are:

Table 4-23 System timer registers summary

Address	Name	Type	Reset value	Description
0xE000E010	SYST_CSR	RW	0x00000000	*SysTick Control and Status Register*.
0xE000E014	SYST_RVR	RW	Unknown	*SysTick Reload Value Register* on page 4-26.
0xE000E018	SYST_CVR	RW	Unknown	*SysTick Current Value Register* on page 4-27.
0xE000E01C	SYST_CALIB	RO	0xC0000000 [a]	*SysTick Calibration Value Register* on page 4-27.

a. SysTick calibration value.

4.4.1 SysTick Control and Status Register

The SYST_CSR controls the SysTick timer and provides status data for the selected Security state. See the register summary in Table 4-23 for its attributes.

This register is banked between Secure and Non-secure state if two SysTick timers are implemented.

The bit assignments for SYST_CSR_S and SYST_CSR_NS are:

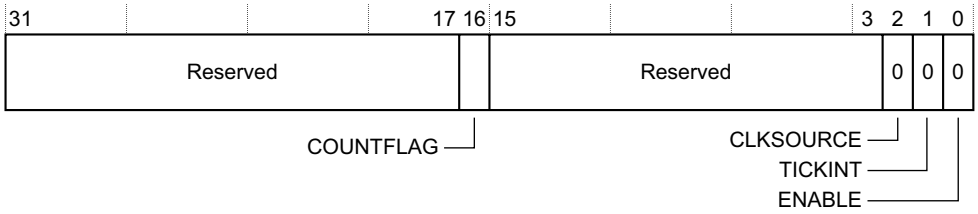


Table 4-24 SYST_CSR bit assignments

Bits	Name	Function
[31:17]	-	Reserved.
[16]	COUNTFLAG	Returns 1 if timer counted to 0 since the last read of this register.
[15:3]	-	Reserved.
[2]	CLKSOURCE	Selects the SysTick timer clock source: 0 = External reference clock. 1 = Processor clock.
[1]	TICKINT	Enables SysTick exception request: 0 = Counting down to zero does not assert the SysTick exception request. 1 = Counting down to zero asserts the SysTick exception request.
[0]	ENABLE	Enables the counter: 0 = Counter disabled. 1 = Counter enabled.

4.4.2 SysTick Reload Value Register

The SYST_RVR specifies the SysTick timer counter reload value for the selected Security state. See the register summary in Table 4-23 on page 4-25 for its attributes.

This register is banked between Secure and Non-secure state if two SysTick timers are implemented.

The bit assignments for SYST_RVR_S and SYST_RVR_NS are:

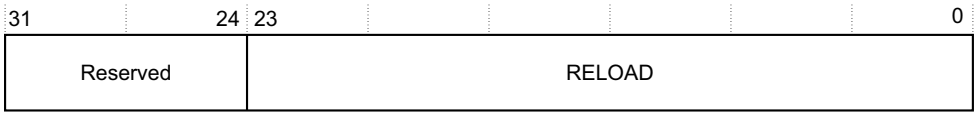


Table 4-25 SYST_RVR bit assignments

Bits	Name	Function
[31:24]	-	Reserved
[23:0]	RELOAD	Value to load into the SYST_CVR when the counter is enabled and when it reaches 0, see *Calculating the RELOAD value* on page 4-27.

Calculating the RELOAD value

The RELOAD value can be any value in the range `0x00000001-0x00FFFFFF`. You can program a value of 0, but this has no effect because the SysTick exception request and COUNTFLAG are activated when counting from 1 to 0.

To generate a multi-shot timer with a period of N processor clock cycles, use a RELOAD value of N-1. For example, if the SysTick interrupt is required every 100 clock pulses, set RELOAD to 99.

4.4.3 SysTick Current Value Register

The SYST_CVR contains the current value of the SysTick counter. See the register summary in Table 4-23 on page 4-25 for its attributes.

This register is banked between Secure and Non-secure state if two SysTick timers are implemented.

The bit assignments for SYST_CVR_S and SYST_CVR_NS are:

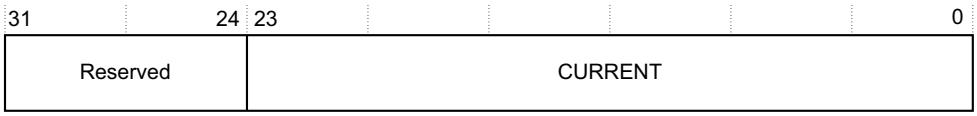


Table 4-26 SYST_CVR bit assignments

Bits	Name	Function
[31:24]	-	Reserved.
[23:0]	CURRENT	Reads return the current value of the SysTick counter. If only one SystTick timer is implemented and ICSR.STTNS is clear, this field is RAZ/WI from Non-secure. If no SystTick timer is implemented this field is reserved. A write of any value clears the field to 0, and also clears the SYST_CSR.COUNTFLAG bit to 0.

4.4.4 SysTick Calibration Value Register

The SYST_CALIB register indicates the SysTick calibration value and parameters for the selected Security state. See the register summary in Table 4-23 on page 4-25 for its attributes.

This register is banked between Secure and Non-secure state if two SysTick timers are implemented.

The bit assignments for SYST_CALIB_S and SYST_CALIB_NS are:

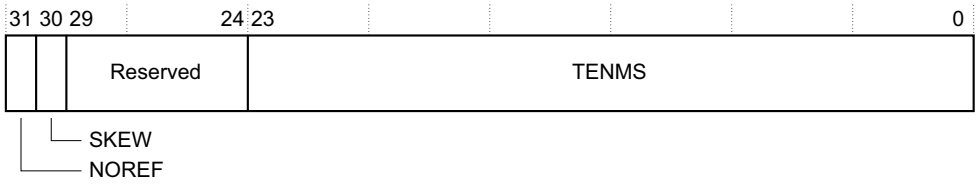


Table 4-27 SYST_CALIB register bit assignments

Bits	Name	Function
[31]	NOREF	Reads as one. Indicates that no separate reference clock is provided.
[30]	SKEW	Reads as one. Calibration value for the 10ms inexact timing is not known because TENMS is not known. This can affect the suitability of SysTick as a software real-time clock.
[29:24]	-	Reserved.
[23:0]	TENMS	Reads as zero. Indicates calibration value is not known.

If calibration information is not known, calculate the calibration value required from the frequency of the processor clock or external clock.

4.4.5 SysTick usage hints and tips

The interrupt controller clock updates the SysTick counter. If this clock signal is stopped for low-power mode, the SysTick counter stops.

Ensure software uses word accesses to access the SysTick registers.

If the SysTick counter reload and current value are undefined at reset, the correct initialization sequence for the SysTick counter is:

1. Program reload value.
2. Clear current value.
3. Program Control and Status register.

4.5 Security Attribution and Memory Protection

This section describes the security attribution and memory protection that the processor uses. The Protection Unit consists of the optional *Security Attribution Unit* (SAU) and the optional *Memory Protection Unit* (MPU).

The Cortex-M23 processor has an optional *Security Attribution Unit* (SAU) and *Memory Protection Unit* (MPU) that provide fine grain memory control, enabling applications to use multiple privilege levels, separating and protecting code, data, and stack on a task-by-task basis. Such requirements are becoming critical in many embedded applications such as automotive systems.

4.5.1 Security Attribution Unit

If the ARMv8-M Security Extension is implemented, the system can contain an SAU. The SAU determines the security of an address.

For instructions, the SAU returns the security attribute (Secure or Non-secure) and identifies whether the instruction address is in a Non-secure callable region.

For data, the SAU returns the security attribute and checks whether both the security of the core and the target address are Non-secure.

When a memory access is performed, the SAU is required. Any address that matches multiple SAU regions is marked as Secure regardless of the attributes that are specified by the regions that matched the address.

The following table shows the SAU registers.

Table 4-28 SAU registers

Address	Name	Type	Reset value	Description
0xE000EDD0	SAU_CTRL	RW	00000000[a]	See *Security Attribution Unit Control Register* on page 4-30. This is the reset value in Secure state. In Non-secure state this register is RAZ/WI.
0xE000EDD4	SAU_TYPE	RO	00000000	See *Security Attribution Unit Type Register* on page 4-31. This is the reset value in Secure state. In Non-secure state this register is RAZ/WI.

Table 4-28 SAU registers (continued)

Address	Name	Type	Reset value	Description
0xE000EDD8	SAU_RNR	RW	UNKNOWN	See *Security Attribution Unit Region Number Register* on page 4-32. In Non-secure state this register is RAZ/WI. With the Security Extension implemented, if the number of SAU regions is 0, then only SAU_CTRL.ALLNS is writable.
0xE000EDDC	SAU_RBAR	RW	UNKNOWN	See *Security Attribution Unit Region Base Address Register* on page 4-32. In Non-secure state this register is RAZ/WI.
0xE000EDE0	SAU_RLAR	RW	Bit[0] resets to 0. Other bits reset to an UNKNOWN value.	See *Security Attribution Unit Region Limit Address Register* on page 4-33. This is the reset value in Secure state. In Non-secure state this register is RAZ/WI.

a. This is the reset value when the Security Extension is implemented. If the Security Extension is not implemented, the reset value is `00000002`.

Note

- Only Privileged accesses to the SAU registers are permitted. Unprivileged accesses generate a fault.
- The SAU registers are word accessible only. Halfword and byte accesses are UNPREDICTABLE.
- The SAU registers are RAZ/WI when accessed from Non-secure state.
- The SAU registers are not banked between Security states.

4.5.2 Security Attribution Unit Control Register

The SAU_CTRL allows enabling of the Security Attribution Unit.

The SAU_CTRL bit assignments are:

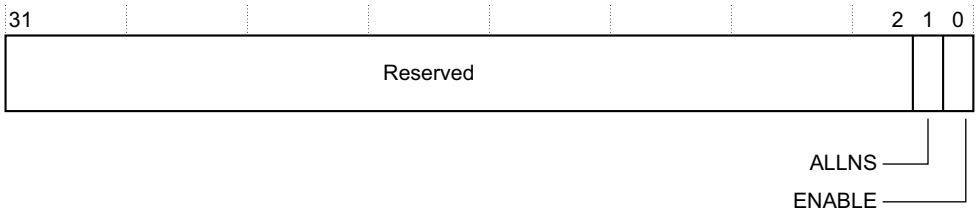


Table 4-29 SAU_CTRL bit assignments

Bits	Name	Function
[31:2]	-	Reserved.
[1]	ALLNS	All Non-secure. When SAU_CTRL.ENABLE is 0 this bit controls if the memory is marked as Non-secure or Secure. The possible values of this bit are: 0 = Memory is marked as Secure and is not Non-secure callable. 1 = Memory is marked as Non-secure. This bit is RAO/WI when the Security Extension is not implemented. This bit is writable when the Security Extension is implemented with an SAU with zero region. Write this bit after a reset to allow regions to become Non-secure, depending on the IDAU.
[0]	ENABLE	Enable. Enables the SAU. The possible values of this bit are: 0 = The SAU is disabled. 1 = The SAU is enabled. This bit is RAZ/WI when the Security Extension is not implemented or when the Security Extension is implemented without an SAU region.

4.5.3 Security Attribution Unit Type Register

The SAU_TYPE indicates the number of regions implemented by the Security Attribution Unit.

The SAU_TYPE bit assignments are:

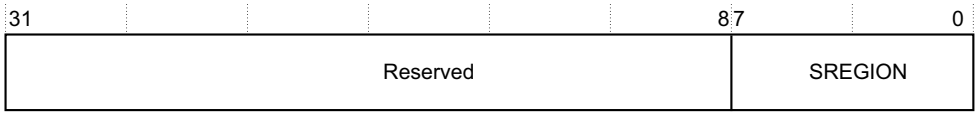


Table 4-30 SAU_TYPE bit assignments

Bits	Name	Function
[31:8]	-	Reserved.
[7:0]	SREGION	SAU regions. The number of implemented SAU regions.

4.5.4 Security Attribution Unit Region Number Register

The SAU_RNR selects the region currently accessed by SAU_RBAR and SAU_RLAR.

The SAU_RNR bit assignments are:

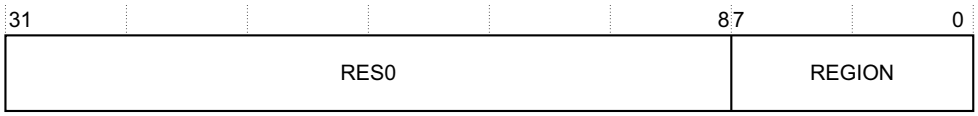


Table 4-31 SAU_RNR bit assignments

Bits	Name	Function
[31:8]	-	Reserved.
[7:0]	REGION	Region number. Indicates the SAU region accessed by SAU_RBAR and SAU_RLAR. If no SAU regions are implemented, this field is reserved. Writing a value corresponding to an unimplemented region is CONSTRAINED UNPREDICTABLE. This field resets to an UNKNOWN value on a Warm reset.

4.5.5 Security Attribution Unit Region Base Address Register

The SAU_RBAR provides indirect read and write access to the base address of the currently selected SAU region.

The SAU_RBAR bit assignments are:

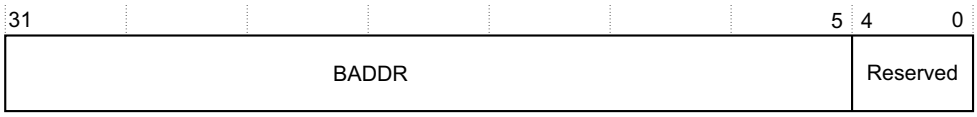


Table 4-32 SAU_RBAR bit assignments

Bits	Name	Function
[31:5]	BADDR	Base address. Holds bits [31:5] of the base address for the selected SAU region. Bits [4:0] of the base address are defined as `0x00`.
[4:0]	-	Reserved.

4.5.6 Security Attribution Unit Region Limit Address Register

The SAU_RLAR provides indirect read and write access to the limit address of the currently selected SAU region.

The SAU_RLAR bit assignments are:

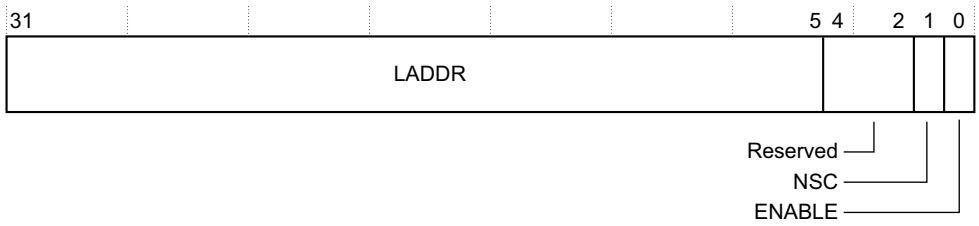


Table 4-33 SAU_RLAR bit assignments

Bits	Name	Function
[31:5]	LADDR	Limit address. Holds bits [31:5] of the limit address for the selected SAU region. Bits [4:0] of the limit address are defined as `0x1F`.
[4:2]	-	Reserved.
[1]	NSC	Non-secure callable. Controls whether Non-secure state is permitted to execute an `SG` instruction from this region. The possible values of this bit are: 0 = Region is not Non-secure callable. 1 = Region is Non-secure callable.
[0]	ENABLE	Enable. SAU region enable. The possible values of this bit are: 0 = SAU region is enabled. 1 = SAU region is disabled. This bit reset to 0 on a warm reset.

4.5.7 Memory Protection Unit

The MPU is divided into <eight regions> and defines the location, size, access permissions, and memory attributes of each region. It supports:

- Independent attribute settings for each region.
- Export of memory attributes to the system.

If the Cortex-M23 processor implements the ARMv8-M Security Extension, it contains:

- One optional Secure MPU.
- One optional Non-secure MPU.

When memory regions overlap, the processor generates a fault if a core access hits the overlapping regions.

The MPU memory map is unified. This means instruction accesses and data accesses have the same region settings.

If a program accesses a memory location that is prohibited by the MPU, the processor generates a HardFault exception.

In an OS environment, the kernel can update the MPU region setting dynamically based on the process to be executed. Typically, an embedded OS uses the MPU for memory protection.

Configuration of MPU regions is based on memory types, see *Memory regions, types, and attributes* on page 2-12.

Table 4-34 shows the possible MPU region attributes. These include Shareability and cache behavior attributes that are not relevant to most microcontroller implementations. See *MPU configuration for a microcontroller* on page 4-44 for guidelines for programming such an implementation.

Table 4-34 Memory attributes summary

Memory type	Shareability	Other attributes	Description
Device, nGnRE	-	-	All accesses to Device, nGnRE memory occur in program order. All Strongly ordered regions are assumed to be shared.
Device	Shared	-	Memory-mapped peripherals that several processors share.
Normal	Shared	Non-cacheable Write-Through Cacheable Write-Back Cacheable	Normal memory that is shared between several processors.
	Non-shared	Non-cacheable Write-Through Cacheable Write-Back Cacheable	Normal memory that only a single processor uses.

Use the MPU registers to define the MPU regions and their attributes. Table 4-35 shows the MPU registers.

Table 4-35 MPU registers summary

Address	Name	Type	Reset Value	Description
0xE000ED90	MPU_TYPE	RO	The reset value is fixed and depends on the value of bits[15:8] which depends on implementation options.	See *MPU Type Register*.
0xE000ED94	MPU_CTRL	RW	0x00000000	See *MPU Control Register* on page 4-36.
0xE000ED98	MPU_RNR	RW	UNKNOWN	See *MPU Region Number Register* on page 4-38.
0xE000ED9C	MPU_RBAR	RW	UNKNOWN	See *MPU Region Base Address Register* on page 4-38.
0xE000EDA0	MPU_RLAR	RW	UNKNOWN	See *MPU Region Limit Address Register* on page 4-39.
0xE000EDC0	MPU_MAIR0	RW	UNKNOWN	See *MPU Memory Attribute Indirection Register 0 and MPU Memory Attribute Indirection Register 1* on page 4-40.
0xE000EDC4	MPU_MAIR1	RW	UNKNOWN	

4.5.8 MPU Type Register

The MPU_TYPE register indicates whether the MPU is present, and if so, how many regions it supports.

The MPU_TYPE bit assignments are:

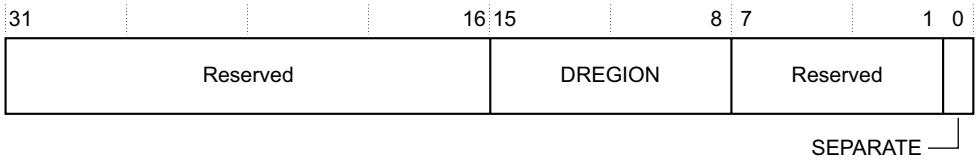


Table 4-36 MPU_TYPE bit assignments

Bits	Name	Function
[31:16]	-	Reserved.

Table 4-36 MPU_TYPE bit assignments **(continued)**

Bits	Name	Function
[15:8]	DREGION	Data regions. Number of regions supported by the MPU. `0x00` = Zero regions if your device does not include the MPU. `0x8=` Eight regions if your device includes the MPU. This value is implementation defined.
[7:1]	-	Reserved.
[0]	SEPARATE	Indicates support for unified or separate instructions and data address regions. ARMv8-M only supports unified MPU regions. 0 = Unified.

4.5.9 MPU Control Register

The MPU_CTRL register enables the MPU. When the MPU is enabled, it controls:

- Whether the default memory map is enabled as a background region for privileged accesses.
- Whether the MPU is enabled for HardFaults, and NMIs.

The MPU_CTRL bit assignments are:

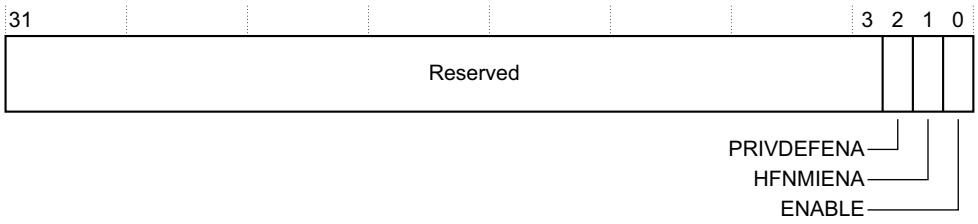


Table 4-37 MPU_CTRL bit assignments

Bits	Name	Function
[31:3]	-	Reserved.
[2]	PRIVDEFENA	Enables privileged software access to the default memory map. When the MPU is enabled: 0 = Disables use of the default memory map. Any memory access to a location not covered by any enabled region causes a fault. 1 = Enables use of the default memory map as a background region for privileged software accesses. When enabled, the background region acts as if it is region number -1. Any region that is defined and enabled has priority over this default map. If the MPU is disabled, the processor ignores this bit.
[1]	HFNMIENA	Enables the operation of MPU during HardFault and NMI handlers. When the MPU is enabled: 0 = MPU is disabled during HardFault and NMI handlers, regardless of the value of the ENABLE bit. 1 = The MPU is enabled during HardFault and NMI handlers. When the MPU is disabled, if this bit is set to 1 the behavior is UNPREDICTABLE.
[0]	ENABLE	Enables the MPU: 0 = MPU is disabled. 1 = MPU is enabled.

XN and Strongly ordered rules always apply to the System Control Space regardless of the value of the ENABLE bit.

When the ENABLE bit is set to 1, at least one region of the memory map must be enabled for the system to function unless the PRIVDEFENA bit is set to 1. If the PRIVDEFENA bit is set to 1 and no regions are enabled, then only privileged software can operate.

When the ENABLE bit is set to 0, the system uses the default memory map. This has the same behavior as if the MPU is not implemented, see *Memory access behavior* on page 2-15. The default memory map applies to accesses from both privileged and unprivileged software.

When the MPU is enabled, accesses to the System Control Space and vector table are always permitted. Other areas are accessible based on regions and whether PRIVDEFENA is set to 1.

Unless HFNMIENA is set to 1, the MPU is not enabled when the processor is executing the handler for an exception with priority –1, –2, or –3. These priorities are only possible when handling a HardFault or NMI exception. Setting the HFNMIENA bit to 1 enables the MPU when operating with these priorities.

4.5.10 MPU Region Number Register

The MPU_RNR selects the region currently accessed by MPU_RBAR and MPU_RLAR.

The MPU_RNR bit assignments are:

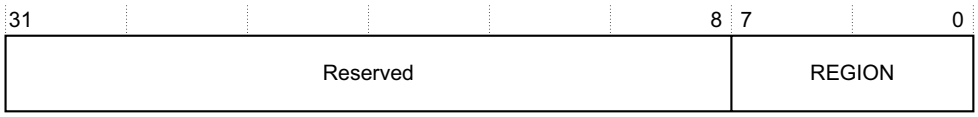


Table 4-38 MPU_RNR bit assignments

Bits	Name	Function
[31:8]	-	Reserved.
[7:0]	REGION	Regions. Indicates the memory region accessed by MPU_RBAR and PMU_RLAR. If no MPU region is implemented, this field is reserved. Writing a value corresponding to an unimplemented region is CONSTRAINED UNPREDICTABLE.

You need to write the required region number to this register before accessing the MPU_RBAR or MPU_RASR.

4.5.11 MPU Region Base Address Register

The MPU_RBAR defines the base address of the MPU region selected by the MPU_RNR, and writes to this register can update the value of the MPU_RNR.

Write MPU_RBAR with the VALID bit set to 1 to change the current region number and update the MPU_RNR.

The MPU_RBAR bit assignments are:

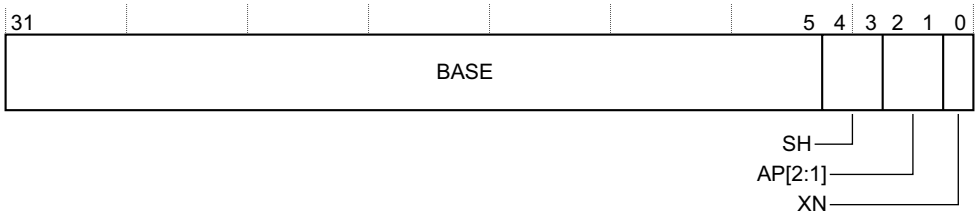


Table 4-39 MPU_RBAR bit assignments

Bits	Name	Function
[31:5]	BASE	Contains bits [31:5] of the lower inclusive limit of the selected MPU memory region. This value is zero extended to provide the base address to be checked against.
[4:3]	SH	Shareability. Defines the shareability domain of this region for Normal memory. 0b00 Non-shareable. 0b01 UNPREDICTABLE. 0b10 Outer shareable. 0b11 Inner shareable. All other values are reserved. For any type of Device memory, the value of this field is ignored.
[2:1]	AP[2:1]	Access permissions. 0b00 Read/write by privileged code only. 0b01 Read/write by any privilege level. 0b10 Read-only by privileged code only. 0b11 Read-only by any privilege level.
[0]	XN	Execute never. Defines whether code can be executed from this region. 0 Execution not permitted. 1 Execution only permitted if read permitted.

4.5.12 MPU Region Limit Address Register

The MPU_RLAR provides indirect read and write access to the limit address of the currently selected MPU region for the selected Security state.

The MPU_RLAR bit assignments are:

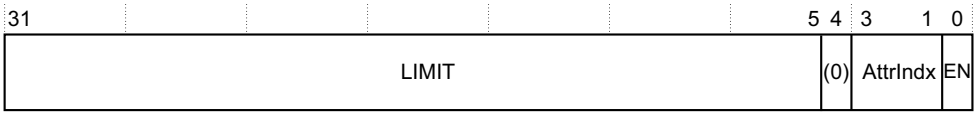


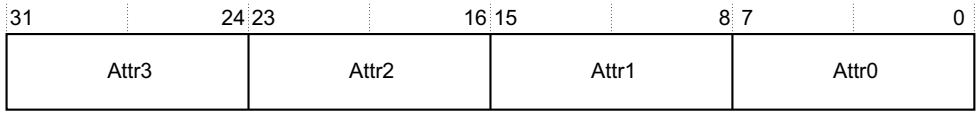
Table 4-40 MPU_RLAR bit assignments

Bits	Name	Function
[31:5]	**LIMIT**	Limit address. Contains bits[31:5] of the upper inclusive limit of the selected MPU memory region. This value is postfixed with 0x1F to provide the limit address to be checked against.
[4]	**-**	Reserved.
[3:1]	**AttrIndx**	Attribute index. Associates a set of attributes in the MPU_MAIR0 and MPU_MAIR1 fields.
[0]	**EN**	Enable. Region enable. The possible values of this bit are: **0** Region disabled. **1** Region enabled.

4.5.13 MPU Memory Attribute Indirection Register 0 and MPU Memory Attribute Indirection Register 1

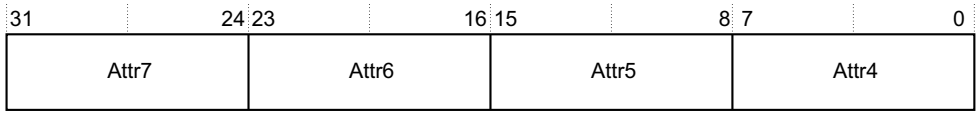
The MPU_MAIR0 and MPU_MAIR1 provide the memory attribute encodings corresponding to the AttrIndex values.

The MPU_MAIR0 bit assignments are:



Attr<*n*>, bits [8n+7:8n], for *n*= 0 to 3. Memory attribute encoding for MPU regions with an AttrIndex of *n*.

The MPU_MAIR1 bit assignments are:



Attr<*n*>, bits [8(*n*-4)+7:8(*n*-4)], for *n* = 4 to 7 Memory attribute encoding for MPU regions with an AttrIndex of *n*.

MAIR_ATTR defines the memory attribute encoding used in MPU_MAIR0 and MPU_MAIR1, and the bit assignments are:

When MAIR_ATTR[7:4] is `0000`:

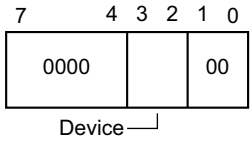


Table 4-41 MAIR_ATTR values for bits[3:2] when MAIR_ATTR[7:4] is `0000`

Bits	Name	Function
[3:2]	Device	Device attributes. Specifies the memory attributes for Device. The possible values of this field are: `0b00` Device-nGnRnE. `0b01` Device-nGnRE. `0b10` Device-nGRE. `0b11` Device-GRE.

When MAIR_ATTR[7:4] is not 0000:

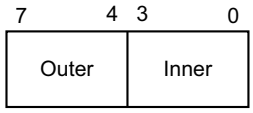


Table 4-42 MAIR_ATTR bit assignments when MAIR_ATTR[7:4] is not 0000

Bits	Name	Function
[7:4]	Outer	Outer attributes. Specifies the Outer memory attributes. The possible values of this field are: 0b0000 Device memory. In this case, refer to *MAIR_ATTR values for bits[3:2] when MAIR_ATTR[7:4] is 0000* on page 4-41. 0b00RW Normal memory, Outer write-through transient (RW is not 00). 0b0100 Normal memory, Outer non-cacheable. 0b01RW Normal memory, Outer write-back transient (RW is not 00). 0b10RW Normal memory, Outer write-through non-transient. 0b11RW Normal memory, Outer write-back non-transient. R and W specify the outer read and write allocation policy: 0 = do not allocate, 1 = allocate.
[3:0]	Inner	Inner attributes. Specifies the Inner memory attributes. The possible values of this field are: 0b0000 UNPREDICTABLE. 0b00RW Normal memory, Inner write-through transient (RW is not 00). 0b0100 Normal memory, Inner non-cacheable. 0b01RW Normal memory, Inner write-back transient (RW is not 00). 0b10RW Normal memory, Inner write-through non-transient. 0b11RW Normal memory, Inner write-back non-transient. R and W specify the outer read and write allocation policy: 0 = do not allocate, 1 = allocate.

4.5.14 MPU mismatch

When access violates the MPU permissions, the processor generates a HardFault.

If BFHFNMINS = 0, Hardfaults are always Secure.

If BFHFNMINS = 1, MPU faults are Secure or Non-secure depending on the MPU that is accessed.

This means that Non-secure code and Secure code can both access a Non-secure MPU. This depends on the SAU or IDAU programming and on the data or instruction.

If the SAU detects a fault, then this fault has priority over MPU faults.

If BFHFNMINS = 1 and the MPU fault is Secure, then this triggers a Secure HardFault.

4.5.15 Updating an MPU region

To update the attributes for an MPU region, update the MPU_RNR, MPU_RBAR and MPU_RASR registers.

Updating an MPU region

Simple code to configure one region:

```
; R1 = region number
; R2 = base address, permissions and shareability
; R3 = limit address, attributes index and enable
LDR R0,=MPU_RNR
STR R1, [R0, #0x0]   ; MPU_RNR
STR R2, [R0, #0x4]   ; MPU_RBAR
STR R2, [R0, #0x8]   ; MPU_RLAR
```

Software must use memory barrier instructions:

- Before MPU setup if there might be outstanding memory transfers, such as buffered writes, that might be affected by the change in MPU settings.
- After MPU setup if it includes memory transfers that must use the new MPU settings.

However, an `ISB` instruction is not required if the MPU setup process starts by entering an exception handler, or is followed by an exception return, because the exception entry and exception return mechanism cause memory barrier behavior.

For example, if you want all the memory access behavior to take effect immediately after the programming sequence, use a `DSB` instruction and an `ISB` instruction. A `DSB` is required after changing MPU settings, such as at the end of a context switch. An `ISB` is required if the code that programs the MPU region or regions is entered using a branch or call. If the programming sequence is entered using a return from exception, or by taking an exception, then you do not require an `ISB`.

4.5.16 MPU design hints and tips

To avoid unexpected behavior, disable the interrupts before updating the attributes of a region that the interrupt handlers might access.

When setting up the MPU, and if the MPU has previously been programmed, disable unused regions to prevent any previous region settings from affecting the new MPU setup.

MPU configuration for a microcontroller

Usually, a microcontroller system has only a single processor and no caches. In such a system, program the MPU as follows:

Table 4-43 Memory region attributes for a microcontroller

Memory region	MAIR_ATTR.Outer MAIR_ATTRInner	Shareability	Memory type and attributes
Flash memory	`0b1010`	0	Normal memory, Non-shareable, Write-Through.
Internal SRAM	`0b1010`	1	Normal memory, Shareable, Write-Through.
External SRAM	`0b1111`	1	Normal memory, Shareable, Write-Back, write-allocate.
Peripherals	`0b0000`	1	Device memory, Shareable.

In most microcontroller implementations, the cache policy attributes do not affect the system behavior. However, using these settings for the MPU regions can make the application code more portable. The values given are for typical situations. In special systems, such as multiprocessor designs or designs with a separate DMA engine, the shareability attribute might be important. In these cases, refer to the recommendations of the memory device manufacturer.

Shareability attributes define whether the global monitor is used, or only the local monitor is used, as detailed in *LDREX and STREX* on page 3-24.

4.6 I/O Port

The Cortex-M23 processor optionally implements a dedicated single-cycle I/O port for high-speed, low-latency access to peripherals. The I/O port is memory mapped and supports all the load and store instructions given in *Memory access instructions* on page 3-15. The I/O port does not support code execution and does not support all forms of exclusive Load and Store.

[Vendor should provide information about which peripherals are connected to this interface].

[If implemented, the I/O port can be protected by the MPU and SAU].