Cortex™-M0 Devices

Generic User Guide

ARM®

ARM DUI 0497A (ID112109)

Cortex-M0 Devices
Generic User Guide

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Confidentiality	Change
08 October 2009	A	Non-Confidential, Unrestricted Access	First release

Proprietary Notice

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Unrestricted Access is an ARM internal classification.

Product Status

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

Web Address

`http://www.arm.com`

Contents
Cortex-M0 Devices Generic User Guide

List of Tables
Cortex-M0 Devices Generic User Guide

Preface

This preface introduces the *Cortex-M0 Devices Generic User Guide*. It contains the following sections:

About this book

This book is a generic user guide for devices that implement the ARM Cortex-M0 processor. Implementers of Cortex-M0 designs make a number of implementation choices, that can affect the functionality of the device. This means that, in this book:

- some information is described as implementation-defined
- some features are described as optional.

See the documentation from the supplier of your Cortex-M0 device for more information about these features.

In this book, unless the context indicates otherwise:

Processor Refers to the Cortex-M0 processor, as supplied by ARM.

Device Refers to an implemented device, supplied by an ARM partner, that incorporates a Cortex-M0 processor. In particular, *your device* refers to the particular implementation of the Cortex-M0 that you are using. Some features of your device depend on the implementation choices made by the ARM partner that made the device.

Product revision status

The r*n*p*n* identifier indicates the revision status of the product described in this book, where:

r*n* Identifies the major revision of the product.

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

Intended audience

This book is written for application and system-level software developers, familiar with programming, who want to program a device that includes the Cortex-M0 processor.

Using this book

This book is organized into the following chapters:

Chapter 1 *Introduction*

Read this for an introduction to the Cortex-M0 processor and its features.

Chapter 2 *The Cortex-M0 Processor*

Read this for information about how to program the processor, the processor memory model, exception and fault handling, and power management.

Chapter 3 ***The Cortex-M0 Instruction Set***

Read this for information about the processor instruction set.

Chapter 4 ***Cortex-M0 Peripherals***

Read this for information about Cortex-M0 peripherals.

Appendix A ***Cortex-M0 Options***

Read this for information about the processor implementation and configuration options.

Glossary

Read this for definitions of terms used in this book.

Typographical conventions

The typographical conventions used in this document are:

italic	Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
bold	Used for terms in descriptive lists, where appropriate.
`monospace`	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace italic`	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
< and >	Enclose replaceable terms for assembler syntax where they appear in code or code fragments. For example: `CMP Rn, <Rm|imm>`

Additional reading

This section lists publications by ARM and by third parties.

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

See onARM, `http://onarm.com`, for embedded software development resources including the *Cortex Microcontroller Software Interface Standard* (CMSIS).

ARM publications

This book contains information that is specific to this product. See the following documents for other relevant information:

- *Cortex-M0 Technical Reference Manual* (ARM DDI 0432)
- *ARMv6-M Architecture Reference Manual* (ARM DDI 0419).

Other publications

This guide only provides generic information for devices that implement the ARM Cortex-M0 processor. For information about your device see the documentation published by the device manufacturer.

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 0497A
- the page numbers to which your comments apply
- a concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Chapter 1
Introduction

This chapter introduces the Cortex-M0 processor and its features. It contains the following section:

- *About the Cortex-M0 processor and core peripherals* on page 1-2.

1.1 About the Cortex-M0 processor and core peripherals

The Cortex™-M0 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:

- simple, easy-to-use programmers model
- highly efficient ultra-low power operation
- excellent code density
- deterministic, high-performance interrupt handling
- upward compatibility with the rest of the Cortex-M processor family.

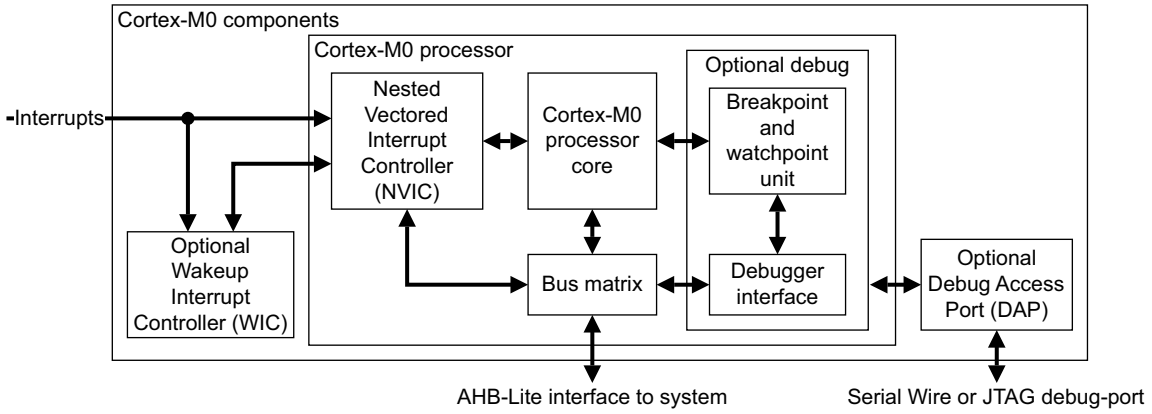


Figure 1-1 Cortex-M0 implementation

The Cortex-M0 processor is built on a high-performance processor core, with a 3-stage pipeline von Neumann architecture, making it ideal for demanding embedded applications. The processor is extensively optimized for low power and area, and delivers exceptional power efficiency through its efficient instruction set, providing high-end processing hardware including either:

- a single-cycle multiplier, in designs optimized for high performance
- a 32-cycle multiplier, in designs optimized for low area.

The Cortex-M0 processor implements the ARMv6-M architecture, that implements the ARMv6-M Thumb® instruction set, including 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-M0 processor closely integrates a configurable *Nested Vectored Interrupt Controlle*r (NVIC), to deliver industry-leading interrupt performance. The NVIC:

- includes a *non-maskable interrupt* (NMI)

- provides:
  - a zero-jitter interrupt option
  - four interrupt priority levels.

The tight integration of the processor core and NVIC provides fast execution of *interrupt service routines* (ISRs), dramatically reducing the interrupt latency. This is achieved through the hardware stacking of registers, and the ability to abandon and restart 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 sleep mode. Optionally, sleep mode support can include a deep sleep function that enables the entire device to be rapidly powered down.

1.1.1 System-level interface

The Cortex-M0 processor provides a single system-level interface using AMBA® technology to provide high speed, low latency memory accesses.

1.1.2 Optional integrated configurable debug

The Cortex-M0 processor can implement a complete hardware debug solution, with extensive hardware breakpoint and watchpoint options. This provides high system visibility of the processor, memory and peripherals through a 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 implemented debug features, and therefore this is device-dependent.

1.1.3 Cortex-M0 processor features summary

- high code density with 32-bit performance
- tools and binaries upwards-compatible across the Cortex-M processor family
- integrated low-power sleep modes
- fast code execution permits slower processor clock or increases sleep mode time
- hardware multiplier
- zero-jitter interrupt handling
- extensive debug capabilities.

1.1.4 Cortex-M0 core peripherals

The Cortex-M0 core peripherals are:

NVIC An embedded interrupt controller that supports low latency interrupt processing.

System Control Block

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

Optional system timer

The optional system timer, SysTick, is a 24-bit count-down timer. If implemented, use this as a *Real Time Operating System* (RTOS) tick timer or as a simple counter.

Chapter 2
The Cortex-M0 Processor

The following sections describe the Cortex-M0 processor:

2.1 Programmers model

This section describes the Cortex-M0 programmers model. In addition to the individual core register descriptions, it contains information about the processor modes and stacks.

2.1.1 Processor modes

The processor modes are:

Thread mode Used to execute application software. The processor enters Thread mode when it comes out of reset.

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

2.1.2 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, 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 and stack use options

Processor mode	Used to execute	Stack used
Thread	Applications	Main stack or process stack[a]
Handler	Exception handlers	Main stack

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

2.1.3 Core registers

The processor core registers are:

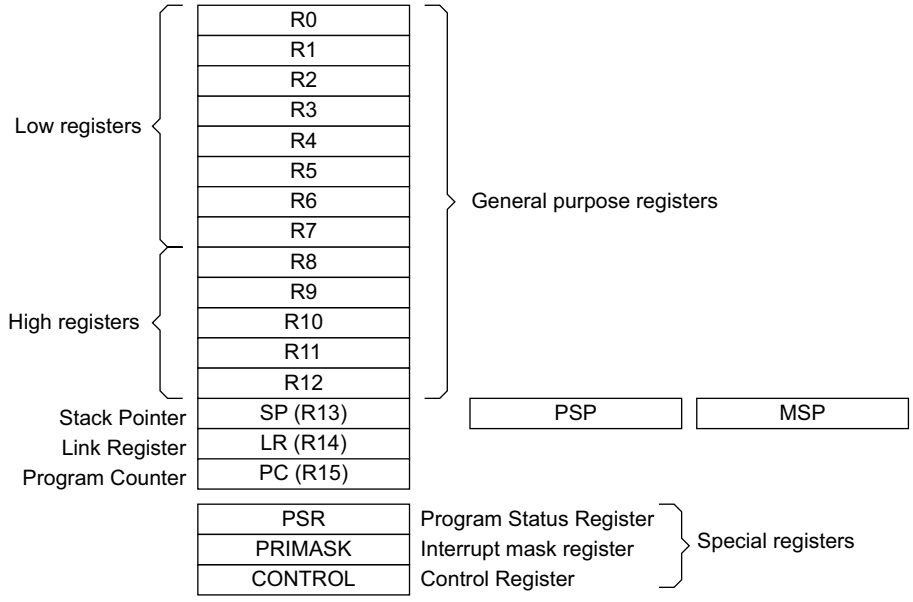


Table 2-2 Core register set summary

Name	Type[a]	Reset value	Description
R0-R12	RW	Unknown	*General-purpose registers* on page 2-4
MSP	RW	See description	*Stack Pointer* on page 2-4
PSP	RW	Unknown	*Stack Pointer* on page 2-4
LR	RW	Unknown	*Link Register* on page 2-4
PC	RW	See description	*Program Counter* on page 2-4
PSR	RW	Unknown[b]	*Program Status Register* on page 2-4
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[b]	*Execution Program Status Register* on page 2-7

Table 2-2 Core register set summary (continued)

Name	Type[a]	Reset value	Description
PRIMASK	RW	0x00000000	*Priority Mask Register* on page 2-9
CONTROL	RW	0x00000000	*CONTROL register* on page 2-9

a. Describes access type during program execution in thread mode and Handler mode. Debug access can differ.
b. 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. In Thread mode, bit[1] 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).

On reset, the processor loads the MSP with the value from address 0x00000000.

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

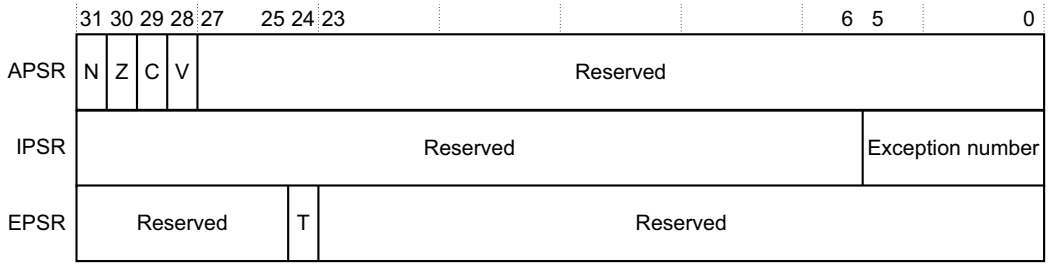
The *Program Counter* (PC) is register R15. It contains the current program address. On reset, the processor loads the PC with the value of the reset vector, at address 0x00000004. Bit[0] of the value is loaded into the EPSR T-bit at reset and must be 1.

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 mutually exclusive bitfields in 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 of the registers using `PSR` with the `MRS` instruction
- write to the APSR using `APSR` with the `MSR` instruction.

The PSR combinations and attributes are:

Table 2-3 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 the these bits

See the instruction descriptions *MRS* on page 3-54 and *MSR* on page 3-55 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 2-4 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 3-14 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 2-5 IPSR bit assignments

Bits	Name	Function
[31:6]	-	Reserved
[5:0]	Exception number	This is the number of the current exception: 0 = Thread mode 1 = Reserved 2 = NMI 3 = HardFault 4-10 = Reserved 11 = SVCall 12, 13 = Reserved 14 = PendSV 15 = SysTick, if implemented[a] 16 = IRQ0 . . . n+15 = IRQ(*n*-1)[b] (*n*+16) to 63 = Reserved. see *Exception types* on page 2-19 for more information.

a. If the device does not implement the SysTick timer, exception number 15 is reserved.
b. The number of interrupts, *n*, is implementation-defined, in the range 1-32.

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 2-6 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. Fault handlers can examine the EPSR value in the stacked PSR to determine the cause of the fault. See *Exception entry and return* on page 2-23. 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 2-27 for more information.

Interruptible-restartable instructions

When an interrupt occurs during the execution of an `LDM` or `STM` instruction, the processor abandons the load multiple or store multiple operation, and similarly abandons a multiply instruction if it implements this as a 32-cycle instruction.

After servicing the interrupt, the processor restarts execution of the abandoned 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 3-54, *MSR* on page 3-55, and *CPS* on page 3-50 for more information.

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:

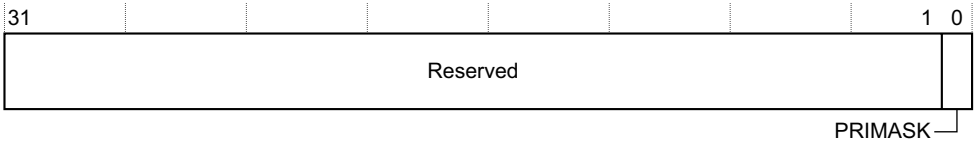


Table 2-7 PRIMASK register bit assignments

Bits	Name	Function
[31:1]	-	Reserved
[0]	PRIMASK	0 = no effect 1 = prevents the activation of all exceptions with configurable priority.

CONTROL register

The CONTROL register controls the stack used when the processor is in Thread mode. See the register summary in Table 2-2 on page 2-3 for its attributes. The bit assignments are:

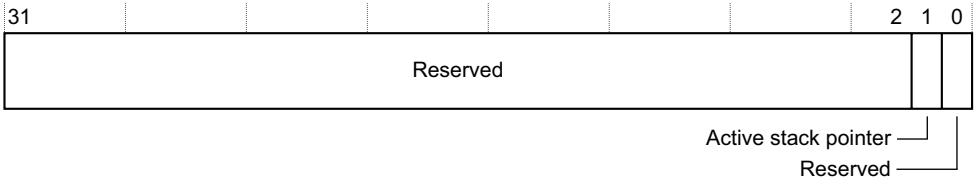


Table 2-8 CONTROL register bit assignments

Bits	Name	Function
[31:2]	-	Reserved
[1]	Active stack pointer	Defines the current stack: 0 = MSP is the current stack pointer 1 = PSP is the current stack pointer. In Handler mode this bit reads as zero and ignores writes.
[0]	-	Reserved.

Handler mode always uses the MSP, so the processor ignores explicit writes to the active stack pointer bit of the CONTROL register when in Handler mode. The exception entry and return mechanisms update the CONTROL register.

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 3-54.

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 3-53.

2.1.4 Exceptions and interrupts

The Cortex-M0 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 2-24 and *Exception return* on page 2-25 for more information.

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

2.1.5 Data types

The processor:

- supports the following data types:
  - 32-bit words
  - 16-bit halfwords
  - 8-bit bytes
- manages all data memory accesses as either little-endian or big-endian, depending on the device implementation. Instruction memory and *Private Peripheral Bus* (PPB) accesses are always little-endian. See *Memory regions, types and attributes* on page 2-13 for more information.

2.1.6 The Cortex Microcontroller Software Interface Standard

ARM provides the *Cortex Microcontroller Software Interface Standard* (CMSIS) for programming Cortex-M0 microcontrollers. The CMSIS is an integrated part of the device driver library. For a Cortex-M0 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 a Cortex-M0 processor. It also includes optional interfaces for middleware components comprising a TCP/IP stack and a Flash file system.

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

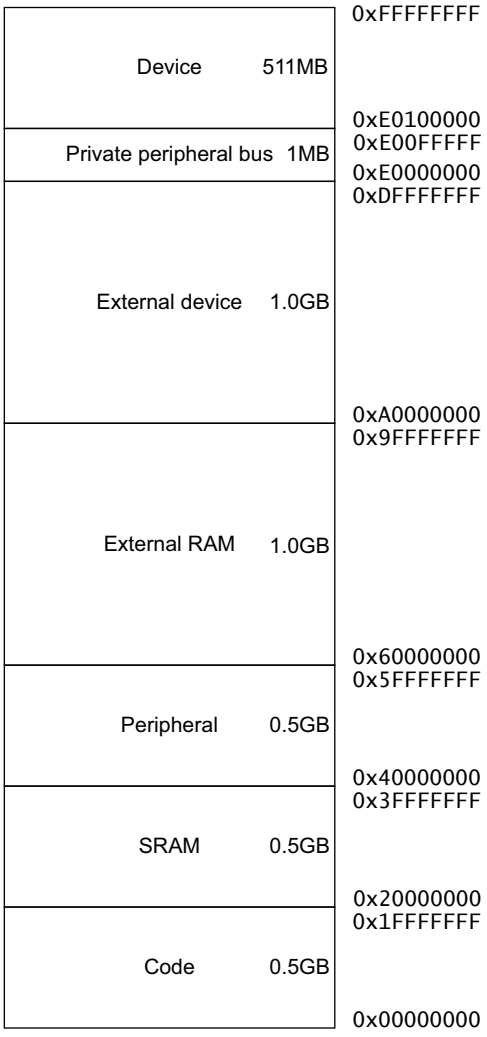
This document uses the register short names defined by the CMSIS. In a few cases these differ from the architectural short names that might be used in other documents.

The following sections give more information about the CMSIS:

- *Power management programming hints* on page 2-30
- *Intrinsic functions* on page 3-5
- *Accessing the Cortex-M0 NVIC registers using CMSIS* on page 4-3
- *NVIC programming hints* on page 4-9.

2.2 Memory model

This section describes the memory map of a Cortex-M0 device 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 PPB address range for core peripheral registers, see *About the Cortex-M0 processor and core peripherals* on page 1-2.

2.2.1 Memory regions, types and attributes

The memory map is split 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 Strongly-ordered memory.

Strongly-ordered The processor preserves transaction order relative to all other transactions.

The different ordering requirements for Device and Strongly-ordered memory mean that the memory system can buffer a write to Device memory, but must not buffer a write to Strongly-ordered memory.

The additional memory attributes include.

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

Strongly-ordered memory is always shareable.

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

Note

This attribute is relevant only if the device is likely to be used in systems where memory is shared between multiple processors.

***Execute Never* (XN)**

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

2.2.2 Memory system ordering of memory accesses

For most memory accesses caused by explicit memory access instructions, the memory system does not guarantee that the order in which the accesses complete matches the program order of the instructions, providing any re-ordering does not affect the behavior of the instruction sequence. Normally, if correct program execution depends on two memory accesses completing in program order, software must insert a memory barrier instruction between the memory access instructions, see *Software ordering of memory accesses* on page 2-16.

However, the memory system does guarantee some ordering of accesses to Device and Strongly-ordered memory. For two memory access instructions A1 and A2, if A1 occurs before A2 in program order, the ordering of the memory accesses caused by two instructions is:

A1 \ A2	Normal access	Device access		Strongly-ordered access
		Non-shareable	Shareable	
Normal access	-	-	-	-
Device access, non-shareable	-	<	-	<
Device access, shareable	-	-	<	<
Strongly-ordered access	-	<	<	<

Where:

- Means that the memory system does not guarantee the ordering of the accesses.

< Means that accesses are observed in program order, that is, A1 is always observed before A2.

2.2.3 Behavior of memory accesses

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

Table 2-9 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	External RAM	Normal	-	Executable region for data.
0xA0000000-0xDFFFFFFF	External device	Device	XN	External device memory.
0xE0000000-0xE00FFFFF	Private Peripheral Bus	Strongly-ordered	XN	This region includes the NVIC, System timer, and System Control Block. Only word accesses can be used in this region.
0xE0100000-0xFFFFFFFF	Device	Device	XN	Implementation-specific.

a. See *Memory regions, types and attributes* on page 2-13 for more information.

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

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 2-10 shows:

Table 2-10 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	-	-

Table 2-10 Memory region shareability and cache policies (continued)

Address range	Memory region	Memory type[a]	Shareability[a]	Cache policy[b]
0x60000000- 0x7FFFFFFF	External RAM	Normal	-	WBWA
0x80000000- 0x9FFFFFFF				WT
0xA0000000- 0xBFFFFFFF	External device	Device	Shareable	-
0xC0000000- 0xDFFFFFFF			Non-shareable	
0xE0000000- 0xE00FFFFF	Private Peripheral Bus	Strongly-ordered	Shareable	-
0xE0100000- 0xFFFFFFFF	Device	Device	-	-

a. See *Memory regions, types and attributes* on page 2-13 for more information.
b. WT = Write through, no write allocate. WBWA = Write back, write allocate. See the *Glossary* for more information.

2.2.4 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:

- the processor can reorder some memory accesses to improve efficiency, providing this does not affect the behavior of the instruction sequence
- memory or devices in the memory map might have different wait states
- some memory accesses are buffered or speculative.

Memory system ordering of memory accesses on page 2-14 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 3-51.

`DSB` The *Data Synchronization Barrier* (DSB) instruction ensures that outstanding memory transactions complete before subsequent instructions execute. See *DSB* on page 3-52.

`ISB` The *Instruction Synchronization Barrier* (ISB) ensures that the effect of all completed memory transactions is recognizable by subsequent instructions. See *ISB* on page 3-53.

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 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.

Memory accesses to Strongly-ordered memory, such as the System Control Block, do not require the use of `DMB` instructions.

2.2.5 Memory endianness

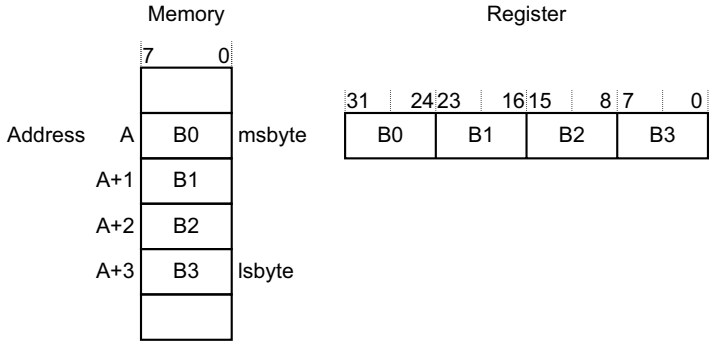
The processor views memory as a linear collection of bytes numbered in ascending order from zero. For example, bytes 0-3 hold the first stored word, and bytes 4-7 hold the second stored word.The memory endianness used is implementation-defined, and the following subsections describe the possible implementations:

- *Byte-invariant big-endian format* on page 2-18
- *Little-endian format* on page 2-18.

Read the AIRCR.ENDIANNESS field to find the implemented endianness, see *Application Interrupt and Reset Control Register* on page 4-15.

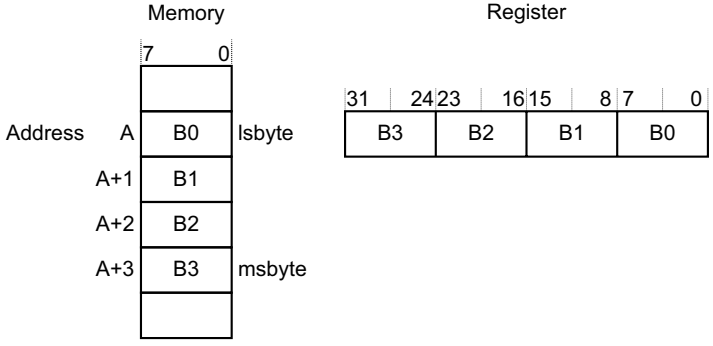
Byte-invariant big-endian format

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 lsbyte of a word at the lowest-numbered byte, and the msbyte at the highest-numbered byte. For example:



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 power up 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 in Thread mode.

NMI An NMI can be signalled by a peripheral or triggered by software. This is the highest priority exception other than reset. It is permanently enabled and has a fixed priority of -2. NMIs cannot be:

- masked or prevented from activation by any other exception
- preempted by any exception other than Reset.

HardFault 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.

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.

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.

SysTick If the device implements the SysTick timer, 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 device can use this exception as system tick.

Interrupt (IRQ) An interrupt, or IRQ, is an exception signalled 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.

Table 2-11 Properties of the different exception types

Exception number[a]	IRQ number[a]	Exception type	Priority	Vector address[b]	Activation
1	-	Reset	-3, the highest	0x00000004	Asynchronous
2	-14	NMI	-2	0x00000008	Asynchronous
3	-13	HardFault	-1	0x0000000C	Synchronous
4-10	-	Reserved	-	-	-
11	-5	SVCall	Configurable[e]	0x0000002C	Synchronous
12-13	-	Reserved	-	-	-
14	-2	PendSV	Configurable[e]	0x00000038	Asynchronous
15	-1	SysTick[c]	Configurable[e]	0x0000003C	Asynchronous
15	-	Reserved	-	-	-
16 and above[d]	0 and above	IRQ	Configurable[e]	0x00000040 and above[f]	Asynchronous

a. To simplify the software layer, the CMSIS only uses IRQ numbers and therefore 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. If your device does not implement the SysTick timer, exception number 15 is reserved.
d. The number of IRQ interrupts is implementation-defined, in the range 1-32. Unimplemented IRQ exception numbers are reserved, for example if the device implements only one IRQ, exception numbers 17 and above are reserved.
e. See *Interrupt Priority Registers* on page 4-7.
f. Increasing in steps of 4.

For an asynchronous exception, other than reset, the processor can continue executing instructions between when the exception is triggered and when the processor enters the exception handler.

Privileged software can disable the exceptions that Table 2-11 on page 2-20 shows as having configurable priority, see *Interrupt Clear-enable Register* on page 4-5.

For more information about HardFault, see *Fault handling* on page 2-27.

2.3.3 Exception handlers

The processor handles exceptions using:

ISRs The IRQ interrupts are the exceptions handled by ISRs.

Fault handler HardFault is the only exception handled by the fault handler.

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

2.3.4 Vector table

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 2-22 shows the order of the exception vectors in the vector table. The least-significant bit of each vector must be 1, indicating that the exception handler is written in Thumb code.

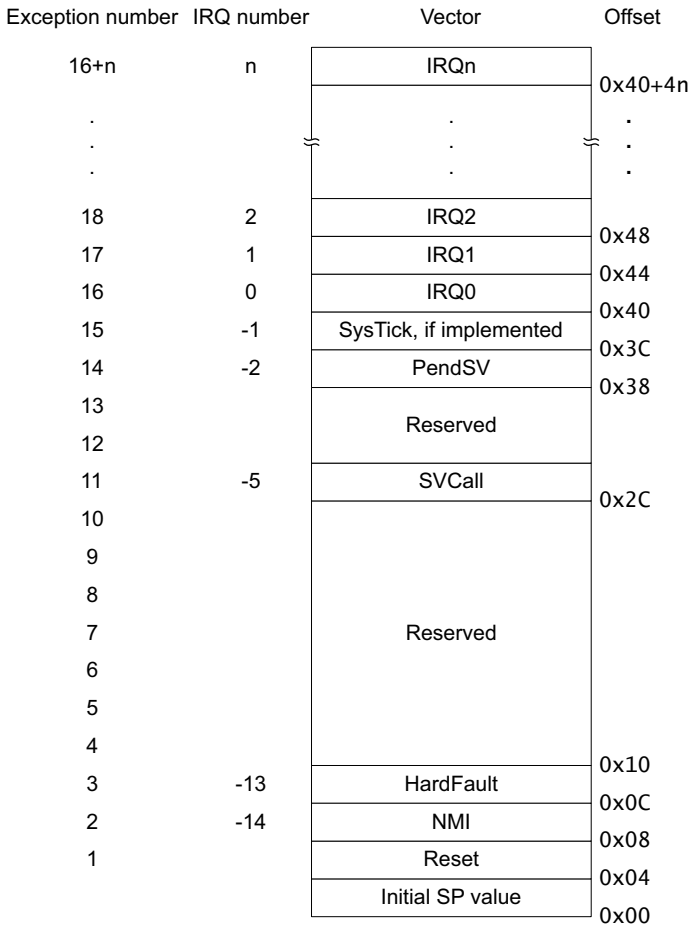


Figure 2-1 Vector table

The vector table is fixed at address `0x00000000`.

2.3.5 Exception priorities

As Table 2-11 on page 2-20 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 4-18
- *Interrupt Priority Registers* on page 4-7.

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.

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* on page 2-24 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 2-25 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

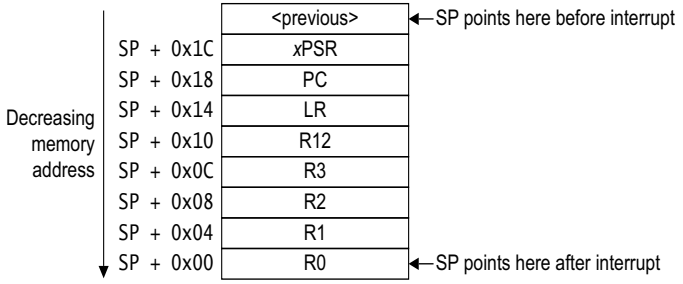
Exception entry occurs when there is a pending exception with sufficient priority and either:

- 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 of eight data words is referred to as a *stack frame*. The stack frame contains the following information:



Immediately after stacking, the stack pointer indicates the lowest address in the stack frame. The stack frame is aligned to a double-word 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 the stack pointer corresponding to the stack frame and the 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. Bits[31:4] of an EXC_RETURN value are `0xFFFFFFF`. When the processor loads a value matching this pattern to the PC it detects that the operation is a not a normal branch operation and, instead, that the exception is complete. Therefore, it starts the exception return sequence. Bits[3:0] of the EXC_RETURN value indicate the required return stack and processor mode, as Table 2-12 on page 2-26 shows.

Table 2-12 Exception return behavior

EXC_RETURN	Description
0xFFFFFFF1	Return to Handler mode. Exception return gets state from the main stack. Execution uses MSP after return.
0xFFFFFFF9	Return to Thread mode. Exception return gets state from MSP. Execution uses MSP after return.
0xFFFFFFFD	Return to Thread mode. Exception return gets state from PSP. Execution uses PSP after return.
All other values	Reserved.

2.4 Fault handling

Faults are a subset of exceptions, see *Exception model* on page 2-19. All faults result in the HardFault exception being taken or cause lockup if they occur in the NMI or HardFault handler. The faults are:

- execution of an `SVC` instruction at a priority equal or higher than SVCall
- execution of a `BKPT` instruction without a debugger attached
- 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.

Note

Only Reset and NMI can preempt the fixed priority HardFault handler. A HardFault can preempt any exception other than Reset, NMI, or another HardFault.

2.4.1 Lockup

The processor enters a lockup state if a fault occurs when executing the NMI or HardFault handlers, or if the system generates a bus error when unstacking the PSR on an exception return using the MSP. 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
- an NMI occurs and the current lockup is in the HardFault handler.

Note

If lockup state occurs in the NMI handler a subsequent NMI does not cause the processor to leave lockup state.

2.5 Power management

The Cortex-M0 processor sleep modes reduce power consumption. The sleep modes your device implements are implementation-defined, but they might be one or both of the following:

- 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.

If your device implements two sleep modes providing different levels of power saving, the SLEEPDEEP bit of the SCR selects which sleep mode is used, see *System Control Register* on page 4-16. For more information about the behavior of the sleep modes see the documentation supplied by your device vendor.

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

2.5.1 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. Therefore 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 instruction, `WFI`, 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 3-60 for more information.

Wait for event

The Wait For Event instruction, `WFE`, 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 3-59 for more information.

If the event register is 1, this 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 signal, or, in a multiprocessor system, because another processor has executed an `SEV` instruction, see *SEV* on page 3-57. 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.

2.5.2 Wakeup from sleep mode

The conditions for the processor to wakeup 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.

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 bit to 1. If an interrupt arrives that is enabled and has a higher priority than current exception priority, the processor wakes up but does not execute the interrupt handler until the processor sets PRIMASK to zero. 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 *The external event signal* on page 2-30
- in a multiprocessor system, another processor in the system executes an `SEV` instruction.

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 4-16.

2.5.3 The optional Wakeup Interrupt Controller

Your device might include a *Wakeup Interrupt Controller* (WIC), an optional peripheral that can detect an interrupt and wake the processor from deep sleep mode. The WIC is enabled only when the DEEPSLEEP bit in the SCR is set to 1, see *System Control Register* on page 4-16.

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

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-M0 processor. This has the side effect of stopping the SysTick timer. When the WIC receives an interrupt, it takes a number of clock cycles to wakeup the processor and restore its state, before it can process the interrupt. This means interrupt latency is increased in deep sleep mode.

2.5.4 The external event signal

Your device might include an external event signals, that device peripherals can use to signal the processor, to wakeup the processor from WFE, or set the internal WFE event register to one to indicate that the processor must not enter sleep mode on a later `WFE` instruction. For more information see *Wait for event* on page 2-28. The documentation supplied by your device vendor might give more information about this signal.

2.5.5 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-M0 Instruction Set

This chapter describes the Cortex-M0 instruction set. It contains the following sections:

3.1 Instruction set summary

The processor implements a version of the Thumb instruction set. Table 3-1 lists the supported instructions.

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-M0 instructions

Mnemonic	Operands	Brief description	Flags	See
ADCS	*{Rd,} Rn, Rm*	Add with Carry	N,Z,C,V	page 3-29
ADD{S}	*{Rd,} Rn, <Rm|#imm>*	Add	N,Z,C,V	page 3-29
ADR	*Rd, label*	PC-relative Address to Register	-	page 3-17
ANDS	*{Rd,} Rn, Rm*	Bitwise AND	N,Z	page 3-29
ASRS	*{Rd,} Rm, <Rs|#imm>*	Arithmetic Shift Right	N,Z,C	page 3-35
B{cc}	*label*	Branch {conditionally}	-	page 3-46
BICS	*{Rd,} Rn, Rm*	Bit Clear	N,Z	page 3-33
BKPT	*#imm*	Breakpoint	-	page 3-49
BL	*label*	Branch with Link	-	page 3-46
BLX	*Rm*	Branch indirect with Link	-	page 3-46
BX	*Rm*	Branch indirect	-	page 3-46
CMN	*Rn, Rm*	Compare Negative	N,Z,C,V	page 3-37
CMP	*Rn, <Rm|#imm>*	Compare	N,Z,C,V	page 3-37
CPSID	*i*	Change Processor State, Disable Interrupts	-	page 3-50
CPSIE	*i*	Change Processor State, Enable Interrupts	-	page 3-50

Table 3-1 Cortex-M0 instructions (continued)

Mnemonic	Operands	Brief description	Flags	See
DMB	-	Data Memory Barrier	-	page 3-51
DSB	-	Data Synchronization Barrier	-	page 3-52
EORS	*{Rd,} Rn, Rm*	Exclusive OR	N,Z	page 3-33
ISB	-	Instruction Synchronization Barrier	-	page 3-53
LDM	*Rn{!}, reglist*	Load Multiple registers, increment after	-	page 3-23
LDR	*Rt, label*	Load Register from PC-relative address	-	page 3-16
LDR	*Rt, [Rn, <Rm|#imm>]*	Load Register with word	-	page 3-16
LDRB	*Rt, [Rn, <Rm|#imm>]*	Load Register with byte	-	page 3-16
LDRH	*Rt, [Rn, <Rm|#imm>]*	Load Register with halfword	-	page 3-16
LDRSB	*Rt, [Rn, <Rm|#imm>]*	Load Register with signed byte	-	page 3-16
LDRSH	*Rt, [Rn, <Rm|#imm>]*	Load Register with signed halfword	-	page 3-16
LSLS	*{Rd,} Rn, <Rs|#imm>*	Logical Shift Left	N,Z,C	page 3-35
LSRS	*{Rd,} Rn, <Rs|#imm>*	Logical Shift Right	N,Z,C	page 3-35
MOV{S}	*Rd, Rm*	Move	N,Z	page 3-39
MRS	*Rd, spec_reg*	Move to general register from special register	-	page 3-54
MSR	*spec_reg, Rm*	Move to special register from general register	N,Z,C,V	page 3-55
MULS	*Rd, Rn, Rm*	Multiply, 32-bit result	N,Z	page 3-41
MVNS	*Rd, Rm*	Bitwise NOT	N,Z	page 3-39
NOP	-	No Operation	-	page 3-56
ORRS	*{Rd,} Rn, Rm*	Logical OR	N,Z	page 3-33
POP	*reglist*	Pop registers from stack	-	page 3-25
PUSH	*reglist*	Push registers onto stack	-	page 3-25
REV	*Rd, Rm*	Byte-Reverse word	-	page 3-42
REV16	*Rd, Rm*	Byte-Reverse packed halfwords	-	page 3-42
REVSH	*Rd, Rm*	Byte-Reverse signed halfword	-	page 3-42

Table 3-1 Cortex-M0 instructions (continued)

Mnemonic	Operands	Brief description	Flags	See
RORS	*{Rd,} Rn, Rs*	Rotate Right	N,Z,C	page 3-35
RSBS	*{Rd,} Rn, #0*	Reverse Subtract	N,Z,C,V	page 3-29
SBCS	*{Rd,} Rn, Rm*	Subtract with Carry	N,Z,C,V	page 3-29
SEV	-	Send Event	-	page 3-57
STM	*Rn!, reglist*	Store Multiple registers, increment after	-	page 3-23
STR	*Rt, [Rn, <Rm|#imm>]*	Store Register as word	-	page 3-16
STRB	*Rt, [Rn, <Rm|#imm>]*	Store Register as byte	-	page 3-16
STRH	*Rt, [Rn, <Rm|#imm>]*	Store Register as halfword	-	page 3-16
SUB{S}	*{Rd,} Rn, <Rm|#imm>*	Subtract	N,Z,C,V	page 3-29
SVC	*#imm*	Supervisor Call	-	page 3-58
SXTB	*Rd, Rm*	Sign extend byte	-	page 3-43
SXTH	*Rd, Rm*	Sign extend halfword	-	page 3-43
TST	*Rn, Rm*	Logical AND based test	N,Z	page 3-44
UXTB	*Rd, Rm*	Zero extend a byte	-	page 3-43
UXTH	*Rd, Rm*	Zero extend a halfword	-	page 3-43
WFE	-	Wait For Event	-	page 3-59
WFI	-	Wait For Interrupt	-	page 3-60

3.2 Intrinsic functions

ISO/IEC C code cannot directly access some Cortex-M0 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-M0 instructions

Instruction	CMSIS intrinsic function
CPSIE i	void __enable_irq(void)
CPSID i	void __disable_irq(void)
ISB	void __ISB(void)
DSB	void __DSB(void)
DMB	void __DMB(void)
NOP	void __NOP(void)
REV	uint32_t __REV(uint32_t int value)
REV16	uint32_t __REV16(uint32_t int value)
REVSH	uint32_t __REVSH(uint32_t int value)
SEV	void __SEV(void)
WFE	void __WFE(void)
WFI	void __WFI(void)

The CMSIS also provides a number of 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)`

3.3 About the instruction descriptions

The following sections give more information about using the instructions:

3.3.1 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.

3.3.2 Restrictions when using PC or SP

Many instructions are unable to use, or have restrictions on whether you can use, the PC or 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-M0 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.

3.3.3 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 sub-sections 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.

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.

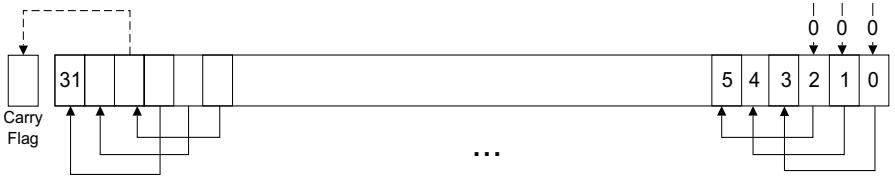


Figure 3-3 LSL #3

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 on page 3-11.

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.

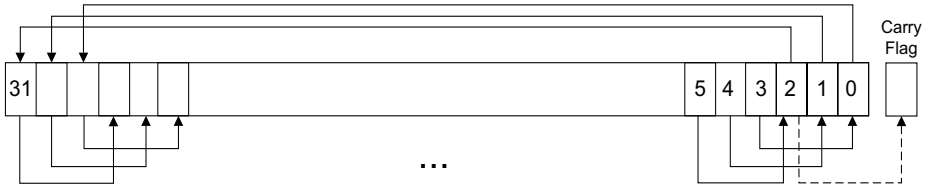


Figure 3-4 ROR #3

3.3.4 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-M0 processor. Any attempt to perform an unaligned memory access operation results in a HardFault exception.

3.3.5 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]`.

3.3.6 Conditional execution

Most data processing instructions update the condition flags in the 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-M0 processor, conditional execution is available by using conditional branches.

This section describes:

- *The condition flags* on page 3-14
- *Condition code suffixes* on page 3-15.

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 2-4.

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 3-4 shows the condition codes to use.

Table 3-4 also shows the relationship between condition code suffixes and the N, Z, C, and V flags.

Table 3-4 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 and N != V	Less than or equal, signed
`AL`	Can have any value	Always. This is the default when no suffix is specified.

3.4 Memory access instructions

Table 3-5 shows the memory access instructions:

Table 3-5 Memory access instructions

Mnemonic	Brief description	See
ADR	Generate PC-relative address	*ADR* on page 3-17
LDM	Load Multiple registers	*LDM and STM* on page 3-23
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 3-20
LDR	Load Register from PC-relative address	*LDR, PC-relative* on page 3-22
POP	Pop registers from stack	*PUSH and POP* on page 3-25
PUSH	Push registers onto stack	*PUSH and POP* on page 3-25
STM	Store Multiple registers	*LDM and STM* on page 3-23
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 3-20

3.4.1 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 3-12.

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.
```

3.4.2 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* in to 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 3-12.

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.
```

3.4.3 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 3-12.

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.
```

3.4.4 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 3-12.

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).
```

3.4.5 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 3-24.

`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 happens 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 writeback 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 writeback suffix must always be used unless the instruction is an LDM where reglist also contains *Rn*, in which case the writeback suffix must not be used.
- the value in the register specified by *Rn* must be word aligned. See *Address alignment* on page 3-12 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
```

3.4.6 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 is LR for a `PUSH` and PC for a `POP`.

Condition flags

These instructions do not change the flags.

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.
```

3.5 General data processing instructions

Table 3-6 shows the data processing instructions:

Table 3-6 Data processing instructions

Mnemonic	Brief description	See
ADCS	Add with Carry	*ADC, ADD, RSB, SBC, and SUB* on page 3-29
ADD{S}	Add	*ADC, ADD, RSB, SBC, and SUB* on page 3-29
ANDS	Logical AND	*AND, ORR, EOR, and BIC* on page 3-33
ASRS	Arithmetic Shift Right	*ASR, LSL, LSR, and ROR* on page 3-35
BICS	Bit Clear	*AND, ORR, EOR, and BIC* on page 3-33
CMN	Compare Negative	*CMP and CMN* on page 3-37
CMP	Compare	*CMP and CMN* on page 3-37
EORS	Exclusive OR	*AND, ORR, EOR, and BIC* on page 3-33
LSLS	Logical Shift Left	*ASR, LSL, LSR, and ROR* on page 3-35
LSRS	Logical Shift Right	*ASR, LSL, LSR, and ROR* on page 3-35
MOV{S}	Move	*MOV and MVN* on page 3-39
MULS	Multiply	*MULS* on page 3-41
MVNS	Move NOT	*MOV and MVN* on page 3-39
ORRS	Logical OR	*AND, ORR, EOR, and BIC* on page 3-33
REV	Reverse byte order in a word	*REV, REV16, and REVSH* on page 3-42
REV16	Reverse byte order in each halfword	*REV, REV16, and REVSH* on page 3-42
REVSH	Reverse byte order in bottom halfword and sign extend	*REV, REV16, and REVSH* on page 3-42
RORS	Rotate Right	*ASR, LSL, LSR, and ROR* on page 3-35
RSBS	Reverse Subtract	*ADC, ADD, RSB, SBC, and SUB* on page 3-29
SBCS	Subtract with Carry	*ADC, ADD, RSB, SBC, and SUB* on page 3-29
SUBS	Subtract	*ADC, ADD, RSB, SBC, and SUB* on page 3-29
SXTB	Sign extend a byte	*SXT and UXT* on page 3-43
SXTH	Sign extend a halfword	*SXT and UXT* on page 3-43

Table 3-6 Data processing instructions (continued)

Mnemonic	Brief description	See
UXTB	Zero extend a byte	*SXT and UXT* on page 3-43
UXTH	Zero extend a halfword	*SXT and UXT* on page 3-43
TST	Test	*TST* on page 3-44

3.5.1 ADC, ADD, RSB, SBC, and SUB

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

Note

Cortex-M0 supports the ADC, RSB, and SBC instructions only as instructions that update the flags, that is, as ADCS, RSBS, and SBCS.

Syntax

```
ADCS   {Rd,} Rn, Rm
ADD{S} {Rd,} Rn, <Rm|#imm>
RSBS   {Rd,} Rn, Rm, #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 a further 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 a further 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*. 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 3-32.

See also *ADR* on page 3-17.

Restrictions

Table 3-7 lists the legal combinations of register specifiers and immediate values that can be used with each instruction.

Table 3-7 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-R15	-	Rd and *Rn* must specify the same register. *Rn* and *Rm* must not both specify the PC (R15).
	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
```

3.5.2 AND, ORR, EOR, and BIC

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

Note

Cortex-M0 supports the `AND`, `ORR`, `EOR`, and `BIC` instructions only as instructions that update the flags, that is, as `ANDS`, `ORRS`, `EORS`, and `BICS`.

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 3-14.

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
```

3.5.3 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.
```

3.5.4 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
```

3.5.5 MOV and MVN

Move and Move NOT.

Syntax

`MOV{S} Rd, Rm`

`MOVS Rd, #imm`

`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 3-13.

`Rd` is the destination register.

`Rm` is a register.

`imm` is any value in the range 0-255.

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.

When *`Rd`* is the PC in a `MOV` instruction:

- Bit[0] of the result is discarded.
- A branch occurs to the address created by forcing bit[0] of the result to 0. The T-bit remains unmodified.

Note

Though it is possible to use `MOV` as a branch instruction, ARM strongly recommends the use of a `BX` or `BLX` instruction to branch for software portability.

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
```

3.5.6 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 3-13.

The results 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
```

3.5.7 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
```

3.5.8 SXT and UXT

Sign extend and Zero extend.

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
```

3.5.9 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
```

3.6 Branch and control instructions

Table 3-8 shows the branch and control instructions:

Table 3-8 Branch and control instructions

Mnemonic	Brief description	See
B{cc}	Branch {conditionally}	*B, BL, BX, and BLX* on page 3-46
BL	Branch with Link	*B, BL, BX, and BLX* on page 3-46
BLX	Branch indirect with Link	*B, BL, BX, and BLX* on page 3-46
BX	Branch indirect	*B, BL, BX, and BLX* on page 3-46

3.6.1 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 3-13.
label	is a PC-relative expression. See *PC-relative expressions* on page 3-12.
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 HardFault 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 3-9 shows the ranges for the various branch instructions.

Table 3-9 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-M0 processor.

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
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 Miscellaneous instructions

Table 3-10 shows the remaining Cortex-M0 instructions:

Table 3-10 Miscellaneous instructions

Mnemonic	Brief description	See
BKPT	Breakpoint	*BKPT* on page 3-49
CPSID	Change Processor State, Disable Interrupts	*CPS* on page 3-50
CPSIE	Change Processor State, Enable Interrupts	*CPS* on page 3-50
DMB	Data Memory Barrier	*DMB* on page 3-51
DSB	Data Synchronization Barrier	*DSB* on page 3-52
ISB	Instruction Synchronization Barrier	*ISB* on page 3-53
MRS	Move from special register to register	*MRS* on page 3-54
MSR	Move from register to special register	*MSR* on page 3-55
NOP	No Operation	*NOP* on page 3-56
SEV	Send Event	*SEV* on page 3-57
SVC	Supervisor Call	*SVC* on page 3-58
WFE	Wait For Event	*WFE* on page 3-59
WFI	Wait For Interrupt	*WFI* on page 3-60

3.7.1 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 in to lockup if a debugger is not attached when a `BKPT` instruction is executed. See *Lockup* on page 2-27 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.
```

3.7.2 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` cause interrupts to be enabled by clearing PRIMASK.See *Exception mask register* on page 2-8 for more information about these registers.

Restrictions

There are no restrictions.

Condition flags

This instruction does not change the condition flags.

Examples

```
CPSID i ; Disable all interrupts except NMI (set PRIMASK)
CPSIE i ; Enable interrupts (clear PRIMASK)
```

3.7.3 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
```

3.7.4 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
```

3.7.5 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
```

3.7.6 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`.

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 that are suitable for modifying a specific flag in the PSR.

See *MSR* on page 3-55.

Restrictions

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

Condition flags

This instruction does not change the flags.

Examples

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

3.7.7 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`.

Operation

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

See *MRS* on page 3-54.

Restrictions

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

Condition flags

This instruction updates the flags explicitly based on the value in *`Rn`*.

Examples

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

3.7.8 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.

Use `NOP` for padding, for example to place the subsequent instructions on a 64-bit boundary.

Restrictions

There are no restrictions.

Condition flags

This instruction does not change the flags.

Examples

```
NOP  ; No operation
```

3.7.9 SEV

Send Event.

Syntax

`SEV`

Operation

`SEV` causes an event to be signaled to all processors in a multiprocessor system. It also sets the local event register, see *Power management* on page 2-28.

See also *WFE* on page 3-59.

Restrictions

There are no restrictions.

Condition flags

This instruction does not change the flags.

Examples

```
SEV ; Send Event
```

3.7.10 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

There are no restrictions.

Condition flags

This instruction does not change the flags.

Examples

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

3.7.11 WFE

Wait For Event.

Syntax

```
WFE
```

Operation

If the event register is 0, `WFE` suspends execution until one of the following events occurs:

- an exception, unless masked by the exception mask registers or the current priority level
- an exception enters the Pending state, if `SEVONPEND` in the System Control Register is set
- a Debug Entry request, if debug is enabled
- an event signaled by a peripheral or another processor in a multiprocessor system using the `SEV` instruction.

If the event register is 1, `WFE` clears it to 0 and completes immediately.

For more information see *Power management* on page 2-28.

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
```

3.7.12 WFI

Wait for Interrupt.

Syntax

```
WFI
```

Operation

`WFI` suspends execution until one of the following events occurs:

- an exception
- an interrupt becomes pending, which would preempt if PRIMASK was clear
- a Debug Entry request, regardless of whether debug is enabled.

Note

`WFI` is intended for power saving only. When writing software assume 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-M0 Peripherals

This chapter describes the ARM Cortex-M0 core peripherals. It contains the following sections:

4.1 About the Cortex-M0 peripherals

The address map of the PPB is:

Table 4-1 Core peripheral register regions

Address	Core peripheral	Description
0xE000E008-0xE000E00F	System Control Block	Table 4-10 on page 4-11
0xE000E010-0xE000E01F	Reserved	-
0xE000E010-0xE000E01F	SysTick system timer[a]	Table 4-19 on page 4-21
0xE000E100-0xE000E4EF	Nested Vectored Interrupt Controller	Table 4-2 on page 4-3
0xE000ED00-0xE000ED3F	System Control Block	Table 4-10 on page 4-11
0xE000EF00-0xE000EF03	Nested Vectored Interrupt Controller	Table 4-2 on page 4-3

a. This address range is reserved if your device does not implement SysTick.

In register descriptions, the register *type* is described as follows:

RW Read and write.

RO Read-only.

WO Write-only.

4.2 Nested Vectored Interrupt Controller

This section describes the NVIC and the registers it uses. The NVIC supports:

- An implementation-defined number of interrupts, in the range 1-32.
- A programmable priority level of 0-192 in steps of 64 for each interrupt. A higher level corresponds to a lower priority, so level 0 is the highest interrupt priority.
- Level and pulse detection of interrupt signals.
- Interrupt tail-chaining.
- An external NMI.

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	ISER	RW	0x00000000	*Interrupt Set-enable Register* on page 4-4
0xE000E180	ICER	RW	0x00000000	*Interrupt Clear-enable Register* on page 4-5
0xE000E200	ISPR	RW	0x00000000	*Interrupt Set-pending Register* on page 4-5
0xE000E280	ICPR	RW	0x00000000	*Interrupt Clear-pending Register* on page 4-6
0xE000E400-0xE000E41C	IPR0-7	RW	0x00000000	*Interrupt Priority Registers* on page 4-7

4.2.1 Accessing the Cortex-M0 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 interrupt or exception to 1.

Table 4-3 CMSIS access NVIC functions (continued)

CMSIS function	Description
`void NVIC_ClearPendingIRQ(IRQn_Type IRQn)`[a]	Clears the pending status of interrupt or exception to 0.
`uint32_t NVIC_GetPendingIRQ(IRQn_Type IRQn)`[a]	Reads the pending status of interrupt or exception. This function returns 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.
`uint32_t NVIC_GetPriority(IRQn_Type IRQn)`[a]	Reads the priority of an interrupt or exception with configurable priority level. This function return the current priority level.

a. The input parameter `IRQn` is the IRQ number, see Table 2-11 on page 2-20 for more information.

4.2.2 Interrupt Set-enable Register

The ISER enables interrupts, and shows the interrupts that are enabled. See the register summary in Table 4-2 on page 4-3 for the register attributes.

The bit assignments are:

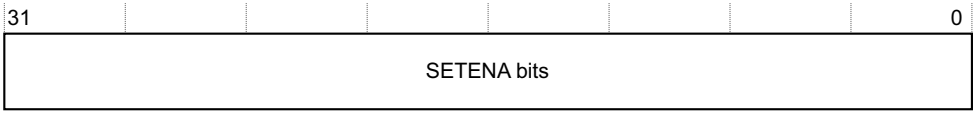


Table 4-4 ISER bit assignments

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 Register

The ICER disables interrupts, and shows the interrupts that are enabled. See the register summary in Table 4-2 on page 4-3 for the register attributes.

The bit assignments are:

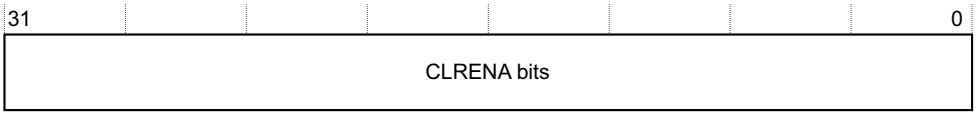


Table 4-5 ICER bit assignments

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 Register

The ISPR forces interrupts into the pending state, and shows the interrupts that are pending. See the register summary in Table 4-2 on page 4-3 for the register attributes.

The bit assignments are:

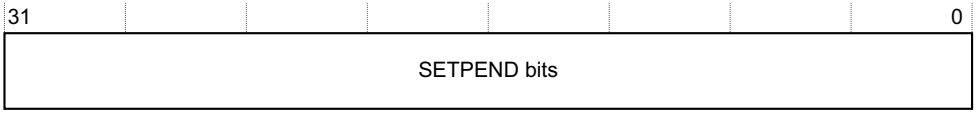


Table 4-6 ISPR bit assignments

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 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 Register

The ICPR removes the pending state from interrupts, and shows the interrupts that are pending. See the register summary in Table 4-2 on page 4-3 for the register attributes.

The bit assignments are:

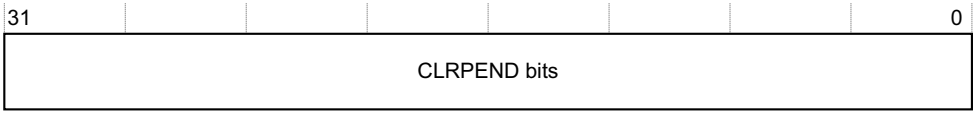


Table 4-7 ICPR bit assignments

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

Note

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

4.2.6 Interrupt Priority Registers

The interrupt priority registers provide an 8-bit priority field for each interrupt, and each register holds four priority fields. This means the number of registers is implementation-defined, and corresponds to the number of implemented interrupts. These registers are only word-accessible. See the register summary in Table 4-2 on page 4-3 for their attributes. For an implementation that supports 32 interrupts the registers are IPR0-IPR7,as shown:

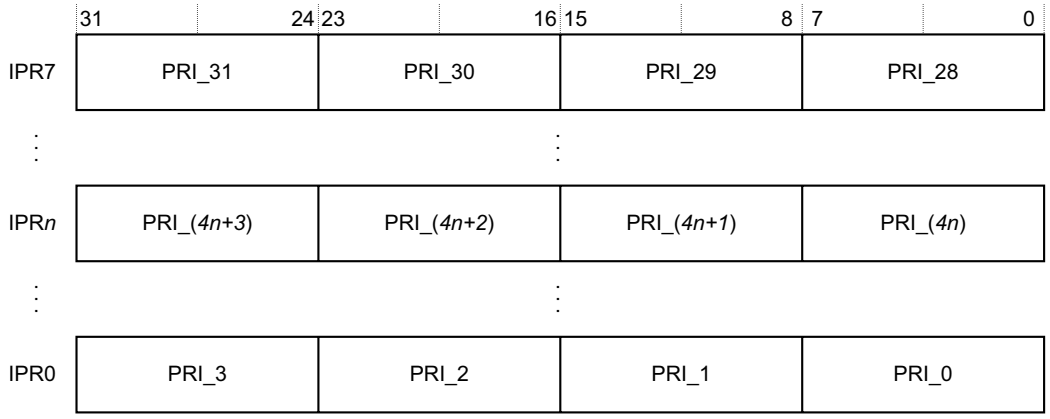


Table 4-8 IPR bit assignments

Bits	Name	Function
[31:24]	Priority, byte offset 3	Each priority field holds a priority value, 0-192. 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-M0 NVIC registers using CMSIS* on page 4-3 for more information about the access to the interrupt priority array, which provides the software view of the interrupt priorities.

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

- the corresponding 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].

4.2.7 Level-sensitive and pulse interrupts

A Cortex-M0 device can support 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 the documentation supplied by your device vendor for details of which interrupts are level-sensitive and which are pulsed.

Hardware and software control of interrupts

The Cortex-M0 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 Register* on page 4-5.

A pending interrupt remains pending until one of the following:

- 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 re-enter 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 re-enter 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.

4.2.8 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.

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 a number of functions for NVIC control, including:

Table 4-9 CMSIS functions for NVIC control

CMSIS interrupt control function	Description
`void NVIC_EnableIRQ(IRQn_t IRQn)`	Enable IRQn
`void NVIC_DisableIRQ(IRQn_t IRQn)`	Disable IRQn
`uint32_t NVIC_GetPendingIRQ (IRQn_t IRQn)`	Return true (1) if IRQn is pending
`void NVIC_SetPendingIRQ (IRQn_t IRQn)`	Set IRQn pending
`void NVIC_ClearPendingIRQ (IRQn_t IRQn)`	Clear IRQn pending status
`void NVIC_SetPriority (IRQn_t IRQn, uint32_t priority)`	Set priority for IRQn
`uint32_t NVIC_GetPriority (IRQn_t IRQn)`	Read priority of IRQn
`void NVIC_SystemReset (void)`	Reset the system

The input parameter `IRQn` is the IRQ number, see Table 2-11 on page 2-20 for more information. For more information about these functions, see the CMSIS documentation.

4.3 System Control Block

The SCB provides system implementation information, and system control. This includes configuration, control, and reporting of the system exceptions. The SCB registers are:

Table 4-10 Summary of the SCB registers

Address	Name	Type	Reset value	Description
0xE000ED00	CPUID	RO	0x410CC200	*CPUID Register*
0xE000ED04	ICSR	RW[a]	0x00000000	*Interrupt Control and State Register* on page 4-12
0xE000ED0C	AIRCR	RW[a]	0xFA050000	*Application Interrupt and Reset Control Register* on page 4-15
0xE000ED10	SCR	RW	0x00000000	*System Control Register* on page 4-16
0xE000ED14	CCR	RO	0x00000208	*Configuration and Control Register* on page 4-17
0xE000ED1C	SHPR2	RW	0x00000000	*System Handler Priority Register 2* on page 4-19
0xE000ED20	SHPR3	RW	0x00000000	*System Handler Priority Register 3* on page 4-20

a. See the register description for more information.

4.3.1 The CMSIS mapping of the Cortex-M0 SCB registers

To improve software efficiency, the CMSIS simplifies the SCB register presentation. In the CMSIS, the array `SHP[1]` corresponds to the registers SHPR2-SHPR3.

4.3.2 CPUID Register

The CPUID register contains the processor part number, version, and implementation information. See the register summary in Table 4-10 for its attributes. The reset value depends on the variant and patch values of the implemented device. The bit assignments are:

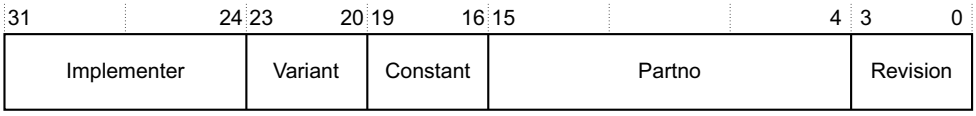


Table 4-11 CPUID register bit assignments

Bits	Name	Function
[31:24]	Implementer	Implementer code: 0x41 corresponds to ARM
[23:20]	Variant	Variant number, the r value in the r*n*p*n* product revision identifier: 0x0 corresponds to revision 0
[19:16]	Constant	Constant that defines the architecture of the processor: 0xC corresponds to ARMv6-M architecture
[15:4]	Partno	Part number of the processor: 0xC20 corresponds to Cortex-M0
[3:0]	Revision	Revision number, the p value in the r*n*p*n* product revision identifier: 0x0 corresponds to patch 0

4.3.3 Interrupt Control and State Register

The ICSR:

- provides:
  - a set-pending bit for the NMI exception
  - set-pending and clear-pending bits for the PendSV and SysTick exceptions
- indicates:
  - the exception number of the exception being processed
  - whether there are preempted active exceptions
  - the exception number of the highest priority pending exception
  - whether any interrupts are pending.

See the register summary in Table 4-10 on page 4-11 for the ICSR attributes. The bit assignments are:

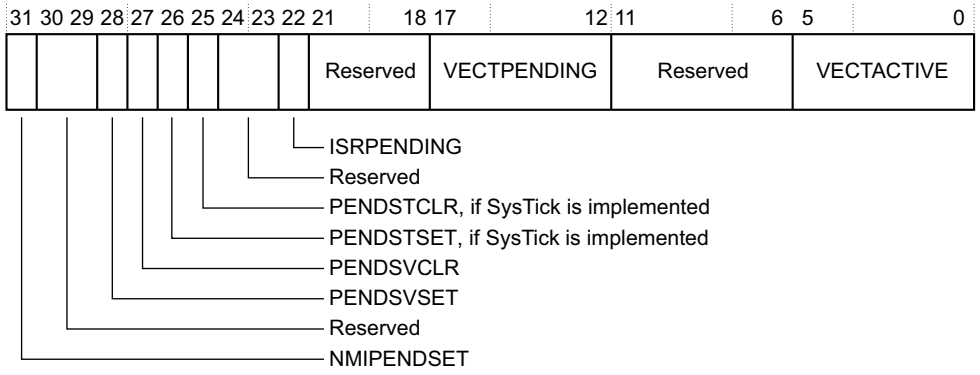


Table 4-12 ICSR bit assignments

Bits	Name	Type	Function
[31]	NMIPENDSET	RW	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. Because NMI is the highest-priority exception, normally the processor enters the NMI exception handler as soon as it detects a write of 1 to this bit. Entering the handler then clears this bit to 0. This means a read of this bit by the NMI exception handler returns 1 only if the NMI signal is reasserted while the processor is executing that handler.
[30:29]	-	-	Reserved.
[28]	PENDSVSET	RW	PendSV set-pending bit. Write: 0 = no effect 1 = changes PendSV exception state to pending. Read: 0 = PendSV exception is not pending 1 = PendSV exception is pending. Writing 1 to this bit is the only way to set the PendSV exception state to pending.

Table 4-12 ICSR bit assignments (continued)

Bits	Name	Type	Function
[27]	PENDSVCLR	WO	PendSV clear-pending bit. Write: 0 = no effect 1 = removes the pending state from the PendSV exception.
[26]	PENDSTSET	RW	SysTick exception set-pending bit. Write: 0 = no effect 1 = changes SysTick exception state to pending. Read: 0 = SysTick exception is not pending 1 = SysTick exception is pending. If your device does not implement the SysTick timer, this bit is Reserved.
[25]	PENDSTCLR	WO	SysTick exception clear-pending bit. Write: 0 = no effect 1 = removes the pending state from the SysTick exception. This bit is WO. On a register read its value is Unknown. If your device does not implement the SysTick timer, this bit is Reserved.
[24:23]	-	-	Reserved.
[22]	ISRPENDING	RO	Interrupt pending flag, excluding NMI and Faults: 0 = interrupt not pending 1 = interrupt pending.
[21:18]	-	-	Reserved.
[17:12]	VECTPENDING	RO	Indicates the exception number of the highest priority pending enabled exception: 0 = no pending exceptions Nonzero = the exception number of the highest priority pending enabled exception.
[11:6]	-	-	Reserved.

Table 4-12 ICSR bit assignments (continued)

Bits	Name	Type	Function
[5:0]	VECTACTIVE[a]	RO	Contains the active exception number: 0 = Thread mode Nonzero = The exception number[a] of the currently active exception. **Note** Subtract 16 from this value to obtain the CMSIS IRQ number that identifies the corresponding bit in the Interrupt Clear-Enable, Set-Enable, Clear-Pending, Set-pending, and Priority Register, see Table 2-5 on page 2-7.

a. This is the same value as IPSR bits[5:0], see *Interrupt Program Status Register* on page 2-7.

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 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 4-10 on page 4-11 and Table 4-13 on page 4-16 for its attributes.

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

The bit assignments are:

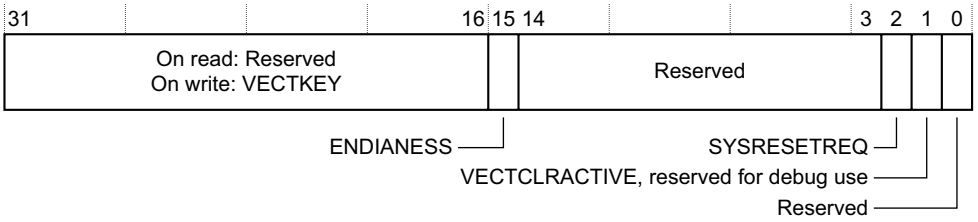


Table 4-13 AIRCR bit assignments

Bits	Name	Type	Function
[31:16]	Read: Reserved Write: VECTKEY	RW	Register key: Reads as Unknown On writes, write `0x05FA` to VECTKEY, otherwise the write is ignored.
[15]	ENDIANESS	RO	Data endianness implemented: 0 = Little-endian 1 = Big-endian.
[14:3]	-	-	Reserved
[2]	SYSRESETREQ	WO	System reset request: 0 = no effect 1 = requests a system level reset. This bit reads as 0.
[1]	VECTCLRACTIVE	WO	Reserved for debug use. This bit reads as 0. When writing to the register you must write 0 to this bit, otherwise behavior is Unpredictable.
[0]	-	-	Reserved.

4.3.5 System Control Register

The SCR controls features of entry to and exit from low power state. See the register summary in Table 4-10 on page 4-11 for its attributes. The bit assignments are:

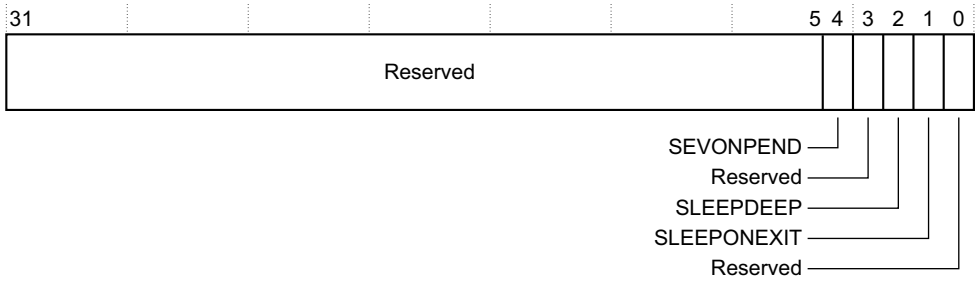


Table 4-14 SCR bit assignments

Bits	Name	Function
[31:5]	-	Reserved.
[4]	SEVONPEND	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 enters pending state, 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 on execution of an `SEV` instruction or an external event.
[3]	-	Reserved.
[2]	SLEEPDEEP	Controls whether the processor uses sleep or deep sleep as its low power mode: 0 = sleep 1 = deep sleep. If your device does not support two sleep modes, the effect of changing the value of this bit is implementation-defined.
[1]	SLEEPONEXIT	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.

4.3.6 Configuration and Control Register

The CCR is a read-only register and indicates some aspects of the behavior of the Cortex-M0 processor. See the register summary in Table 4-10 on page 4-11 for the CCR attributes.

The bit assignments are:

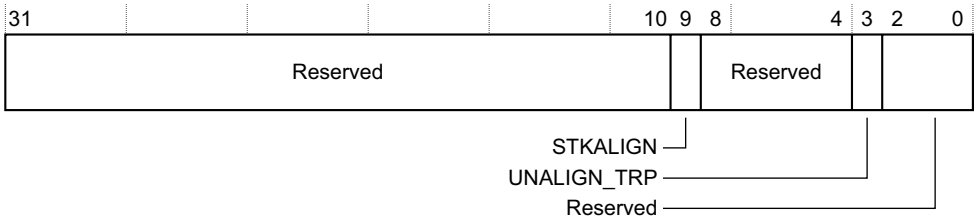


Table 4-15 CCR bit assignments

Bits	Name	Function
[31:10]	-	Reserved.
[9]	STKALIGN	Always reads as one, indicates 8-byte stack alignment on exception entry. On exception entry, the processor uses bit[9] of the stacked PSR to indicate the stack alignment. On return from the exception it uses this stacked bit to restore the correct stack alignment.
[8:4]	-	Reserved.
[3]	UNALIGN_TRP	Always reads as one, indicates that all unaligned accesses generate a HardFault.
[2:0]	-	Reserved.

4.3.7 System Handler Priority Registers

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

SHPR2-SHPR3 are word accessible. See the register summary in Table 4-10 on page 4-11 for their attributes.

To access to 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 2-11 on page 2-20 for more information.

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

Table 4-16 System fault handler priority fields

Handler	Field	Register description
SVCall	PRI_11	*System Handler Priority Register 2*
PendSV	PRI_14	*System Handler Priority Register 3* on page 4-20
SysTick	PRI_15	

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

System Handler Priority Register 2

The bit assignments are:

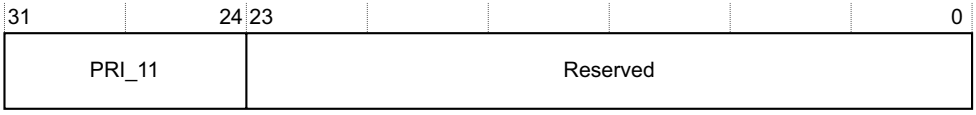


Table 4-17 SHPR2 register bit assignments

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

System Handler Priority Register 3

The bit assignments are:

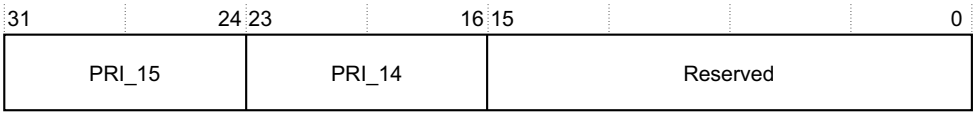


Table 4-18 SHPR3 register bit assignments

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 field is Reserved if your device does not implement the SysTick timer.

4.3.8 SCB usage hints and tips

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

4.4 Optional system timer, SysTick

If implemented, 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 4-19 System timer registers summary

Address	Name	Type	Reset value	Description
0xE000E010	SYST_CSR	RW	-[a]	*SysTick Control and Status Register* on page 4-22
0xE000E014	SYST_RVR	RW	Unknown	*SysTick Reload Value Register* on page 4-23
0xE000E018	SYST_CVR	RW	Unknown	*SysTick Current Value Register* on page 4-23
0xE000E01C	SYST_CALIB	RO	-[a]	*SysTick Calibration Value Register* on page 4-24

a. See the register description for more information.

4.4.1 SysTick Control and Status Register

The SYST_CSR enables the SysTick features. See the register summary in Table 4-19 on page 4-21 for its attributes. The register resets to `0x00000000`, or `0x00000002` if your device does not implement a reference clock. The bit assignments are:

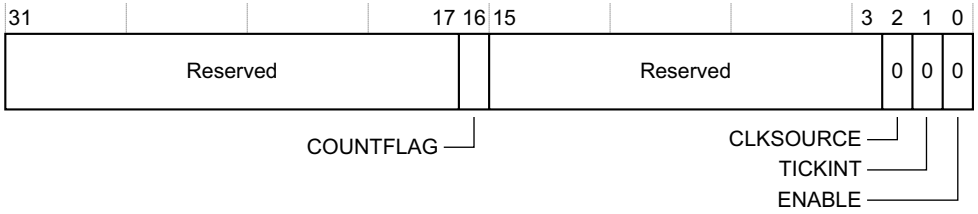


Table 4-20 SYST_CSR bit assignments

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 = reference clock 1 = processor clock. If your device does not implement a reference clock, this bit reads-as-one and ignores writes.
[1]	TICKINT	Enables SysTick exception request: 0 = counting down to zero does not assert the SysTick exception request 1 = counting down to zero to 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 start value to load into the SYST_CVR. See the register summary in Table 4-19 on page 4-21 for its attributes. The bit assignments are:

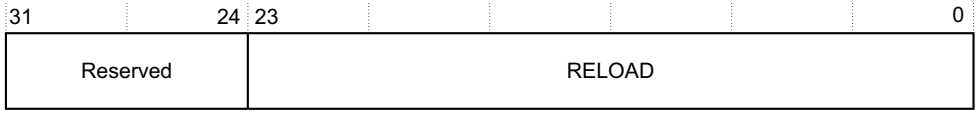


Table 4-21 SYST_RVR bit assignments

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*.

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 4-19 on page 4-21 for its attributes. The bit assignments are:

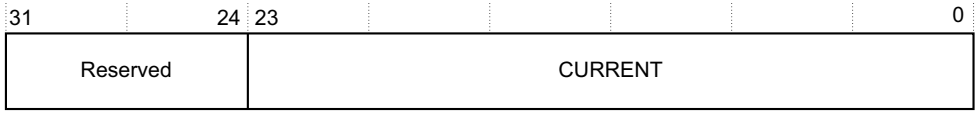


Table 4-22 SYST_CVR bit assignments

Bits	Name	Function
[31:24]	-	Reserved.
[23:0]	CURRENT	Reads return the current value of the SysTick counter. 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 properties. See the register summary in Table 4-19 on page 4-21 for its attributes. The reset value of this register is implementation-defined. See the documentation supplied by your device vendor for more information about the meaning of the SYST_CALIB field values.

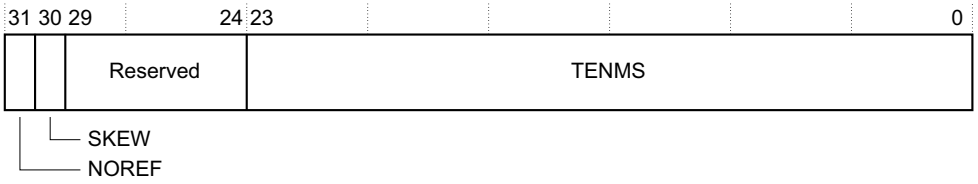


Table 4-23 SYST_CALIB register bit assignments

Bits	Name	Function
[31]	NOREF	Indicates whether the device provides a reference clock to the processor: 0 = reference clock provided 1 = reference clock provided. If your device does not provide a reference clock, the SYST_CSR.CLKSOURCE bit reads-as-one and ignores writes.
[30]	SKEW	Indicates whether the TENMS value is exact: 0 = TENMS value is exact 1 = TENMS value is inexact, or not given. An inexact TENMS value can affect the suitability of SysTick as a software real time clock.
[29:24]	-	Reserved.
[23:0]	TENMS	Reload value for 10ms (100Hz) timing, subject to system clock skew errors. If the value reads as zero, the 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. Some implementations stop this clock signal for low power mode. If this happens, the SysTick counter stops.

Ensure software uses word accesses to access the SysTick registers.

The SysTick counter reload and current value are not initialized by hardware. This means the correct initialization sequence for the SysTick counter is:

1. Program the reload value.
2. Clear the current value.
3. Program the Control and Status register.

Appendix A
Cortex-M0 Options

This appendix describes the configuration options for a Cortex-M0 processor implementation. This means it shows what features of a Cortex-M0 implementation are determined by the device manufacturer. It contains the following section:

- *Cortex-M0 implementation options* on page A-2.

A.1 Cortex-M0 implementation options

Table A-1 shows the Cortex-M0 implementation options:

Table A-1 Effects of the Cortex-M0 implementation options

Option	Description, and affected documentation
Number of interrupts	The implementer decides how many interrupts the Cortex-M0 implementation supports, in the range 1-32. This affects the range of IRQ values in Table 2-5 on page 2-7.
Inclusion of the WIC	The implementer decides whether to include the *Wakeup interrupt Controller* (WIC), see *The optional Wakeup Interrupt Controller* on page 2-30.
Sleep mode power-saving	The implementer decides what sleep modes to implement, and the power-saving measures associated with any implemented mode, see *Power management* on page 2-28.
Endianness	The implementer decides whether the memory system is little-endian or big-endian, see *Data types* on page 2-10 and *Memory endianness* on page 2-17.
Memory features	Some features of the memory system are implementation-specific. This means that *Memory model* on page 2-12 cannot completely describe the memory map for a specific Cortex-M0 implementation.
SysTick timer	The SysTick timer and its SYST_CALIB register are implementation-defined. This can affect: • System timer, *Optional system timer, SysTick* on page 4-21 • The entry for SYST_CALIB in Table 4-19 on page 4-20 • SysTick Calibration Value Register on page 4-22.

Glossary

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

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

Base register In instruction descriptions, a register specified by a load or store instruction that is used to hold the base value for the instruction's address calculation. Depending on the instruction and its addressing mode, an offset can be added to or subtracted from the base register value to form the address that is sent to memory.

See also Index register.

Big-endian (BE) Byte ordering scheme in which bytes of decreasing significance in a data word are stored at increasing addresses in memory.

See also Byte-invariant, Endianness, Little-endian.

Big-endian memory Memory in which:

- a byte or halfword at a word-aligned address is the most significant byte or halfword within the word at that address

- a byte at a halfword-aligned address is the most significant byte within the halfword at that address.

See also Little-endian memory.

Breakpoint A breakpoint is a mechanism provided by debuggers to identify an instruction at which program execution is to be halted. Breakpoints are inserted by the programmer to enable inspection of register contents, memory locations, variable values at fixed points in the program execution to test that the program is operating correctly. Breakpoints are removed after the program is successfully tested.

Byte-invariant In a byte-invariant system, the address of each byte of memory remains unchanged when switching between little-endian and big-endian operation. When a data item larger than a byte is loaded from or stored to memory, the bytes making up that data item are arranged into the correct order depending on the endianness of the memory access. An ARM byte-invariant implementation also supports unaligned halfword and word memory accesses. It expects multi-word accesses to be word-aligned.

Cache A block of on-chip or off-chip fast access memory locations, situated between the processor and main memory, used for storing and retrieving copies of often used instructions, data, or instructions and data. This is done to greatly increase the average speed of memory accesses and so improve processor performance.

Conditional execution
If the condition code flags indicate that the corresponding condition is true when the instruction starts executing, it executes normally. Otherwise, the instruction does nothing.

Context The environment that each process operates in for a multitasking operating system. In ARM processors, this is limited to mean the physical address range that it can access in memory and the associated memory access permissions.

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

Direct Memory Access (DMA)
An operation that accesses main memory directly, without the processor performing any accesses to the data concerned.

Endianness Byte ordering. The scheme that determines the order that successive bytes of a data word are stored in memory. An aspect of the system's memory mapping.

See also Little-endian and Big-endian

Exception An event that interrupts program execution. When an exception occurs, the processor suspends the normal program flow and starts execution at the address indicated by the corresponding exception vector. The indicated address contains the first instruction of the handler for the exception.

An exception can be an interrupt request, a fault, or a software-generated system exception. Faults include attempting an invalid memory access, attempting to execute an instruction in an invalid processor state, and attempting to execute an undefined instruction.

Exception vector *See* Interrupt vector.

Halfword A 16-bit data item.

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

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

Index register In some load and store instruction descriptions, the value of this register is used as an offset to be added to or subtracted from the base register value to form the address that is sent to memory. Some addressing modes optionally enable the index register value to be shifted prior to the addition or subtraction.

See also Base register.

Interrupt handler A program that control of the processor is passed to when an interrupt occurs.

Interrupt vector One of a number of fixed addresses in low memory, or in high memory if high vectors are configured, that contains the first instruction of the corresponding interrupt handler.

Little-endian (LE) Byte ordering scheme in which bytes of increasing significance in a data word are stored at increasing addresses in memory.

See also Big-endian, Byte-invariant, Endianness.

Little-endian memory
Memory in which:

- a byte or halfword at a word-aligned address is the least significant byte or halfword within the word at that address

- a byte at a halfword-aligned address is the least significant byte within the halfword at that address.

See also Big-endian memory.

Read Reads are defined as memory operations that have the semantics of a load. Reads include the Thumb instructions `LDM`, `LDR`, `LDRSH`, `LDRH`, `LDRSB`, `LDRB`, and `POP`.

Region A partition of memory space.

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

Thumb instruction One or two halfwords that specify an operation for a processor to perform. Thumb instructions must be halfword-aligned.

Unaligned A data item stored at an address that is not divisible by the number of bytes that defines the data size is said to be unaligned. For example, a word stored at an address that is not divisible by four.

Undefined Indicates an instruction that generates an Undefined instruction exception.

Unpredictable You cannot rely on the behavior. Unpredictable behavior must not represent security holes. Unpredictable behavior must not halt or hang the processor, or any parts of the system.

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

WA *See* Write-allocate.

WB *See* Write-back.

Word A 32-bit data item.

Write Writes are defined as operations that have the semantics of a store. Writes include the Thumb instructions `STM`, `STR`, `STRH`, `STRB`, and `PUSH`.

Write-allocate (WA) In a write-allocate cache, a cache miss on storing data causes a cache line to be allocated into the cache.

Write-back (WB) In a write-back cache, data is only written to main memory when it is forced out of the cache on line replacement following a cache miss. Otherwise, writes by the processor only update the cache. This is also known as copyback.

Write buffer A block of high-speed memory, arranged as a FIFO buffer, between the data cache and main memory, whose purpose is to optimize stores to main memory.

Write-through (WT) In a write-through cache, data is written to main memory at the same time as the cache is updated.

WT *See* Write-through.