PM0215
Programming manual

STM32F0xxx Cortex-M0 programming manual

Introduction

This programming manual provides information for application and system-level software developers. It gives a full description of the STM32 Cortex™-M0 processor programming model, instruction set and core peripherals.

The STM32 Cortex™-M0 processor is a high performance 32-bit processor designed for the microcontroller market. It offers significant benefits to developers, including:

- Outstanding processing performance combined with fast interrupt handling
- Enhanced system debug with extensive breakpoint and trace capabilities
- Efficient processor core, system and memories
- Ultra-low power consumption with integrated sleep modes
- Platform security

Table 1. Applicable products

Type	Part numbers
Microcontroller	STM32F0xxx

Contents

List of tables

List of figures

1 About this document

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

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

1.1 Typographical conventions

The typographical conventions used in this document are:

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

1.2 List of abbreviations for registers

The following abbreviations are used in register descriptions:

read/write (rw)	Software can read and write to these bits.
read-only (r)	Software can only read these bits.
write-only (w)	Software can only write to this bit. Reading the bit returns the reset value.
read/clear (rc_w1)	Software can read as well as clear this bit by writing 1. Writing ‘0’ has no effect on the bit value.
read/clear (rc_w0)	Software can read as well as clear this bit by writing 0. Writing ‘1’ has no effect on the bit value.
toggle (t)	Software can only toggle this bit by writing ‘1’. Writing ‘0’ has no effect.
Reserved (Res.)	Reserved bit, must be kept at reset value.

1.3 About the STM32 Cortex-M0 processor and core peripherals

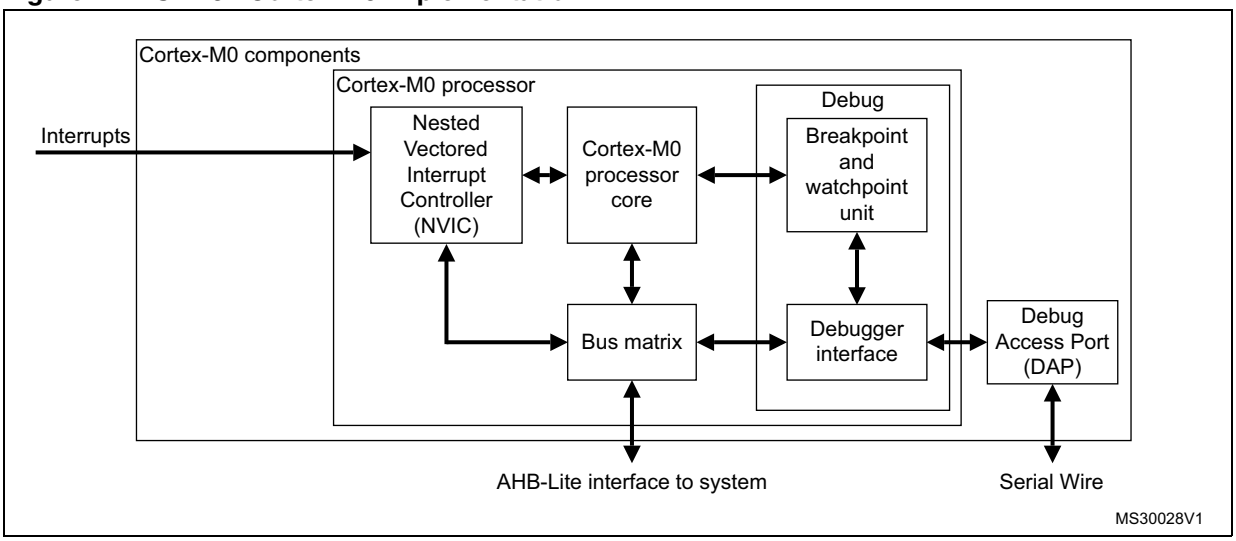
The Cortex-M0 processor is an entry-level 32-bit ARM Cortex processor designed for a broad range of embedded applications. It offers significant benefits to developers, including:

- a simple architecture that is easy to learn and program
- ultra-low power, energy efficient operation
- excellent code density
- deterministic, high-performance interrupt handling
- upward compatibility with Cortex-M processor family.

The Cortex-M0 processor is built on a highly area and power optimized 32-bit processor core, with a 3-stage pipeline von Neumann architecture. The processor delivers exceptional energy efficiency through a small but powerful instruction set and extensively optimized design, providing high-end processing hardware including a single-cycle multiplier.

The Cortex-M0 processor implements the ARMv6-M architecture, which is based on the 16-bit Thumb® instruction set and includes Thumb-2 technology. This provides the exceptional performance expected of a modern 32-bit architecture, with a higher code density than other 8-bit and 16-bit microcontrollers.

Figure 1. STM32 Cortex-M0 implementation



The Cortex-M0 processor closely integrates a configurable nested vectored interrupt controller (NVIC), to deliver industry-leading interrupt performance. The NVIC:

- includes a non-maskable interrupt (NMI)
- provides zero jitter interrupt option
- provides 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 the sleep modes, including a deep sleep function that enables the entire device to be rapidly powered down.

1.3.1 System level interface

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

1.3.2 Integrated configurable debug

The Cortex-M0 processor implements 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 2-pin Serial Wire Debug (SWD) port that is ideal for small package devices.

1.3.3 Cortex-M0 processor features and benefits summary

- High code density with 32-bit performance
- Tools and binary upwards compatible with Cortex-M processor family
- Integrated ultra low-power sleep modes
- Efficient code execution permits slower processor clock or increases sleep mode time
- Single-cycle 32-bit hardware multiplier
- Zero jitter interrupt handling
- Extensive debug capabilities

1.3.4 Cortex-M0 core peripherals

The peripherals are:

- Nested vectored interrupt controller: The NVIC is an embedded interrupt controller that supports low latency interrupt processing.
- System control block: The 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.
- System timer: SysTick is a 24-bit count-down timer. Use this as a Real Time Operating System (RTOS) tick timer or as a simple counter.

2 The STM32 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 exception processing.

The Cortex-M0 does not support multiple privilege levels. It can always use all instructions and access all resources.

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, with independent copies of the stack pointer,(see *Stack pointer (SP) register R13 on page 13*):

- the main stack and
- the process *stack*,

In Thread mode, the CONTROL register controls whether the processor uses the main stack or the process stack, see *Control register on page 16*.

In Handler mode, the processor always uses the main stack.

The options for processor operations are:

Table 2. Summary of processor mode and stack usage

Processor mode	Used to execute	Stack used
Thread	Applications	Main stack or process stack (1)
Handler	Exception handlers	Main stack

1. See *Control register on page 16*.

2.1.3 Core registers

Figure 2. Processor core registers

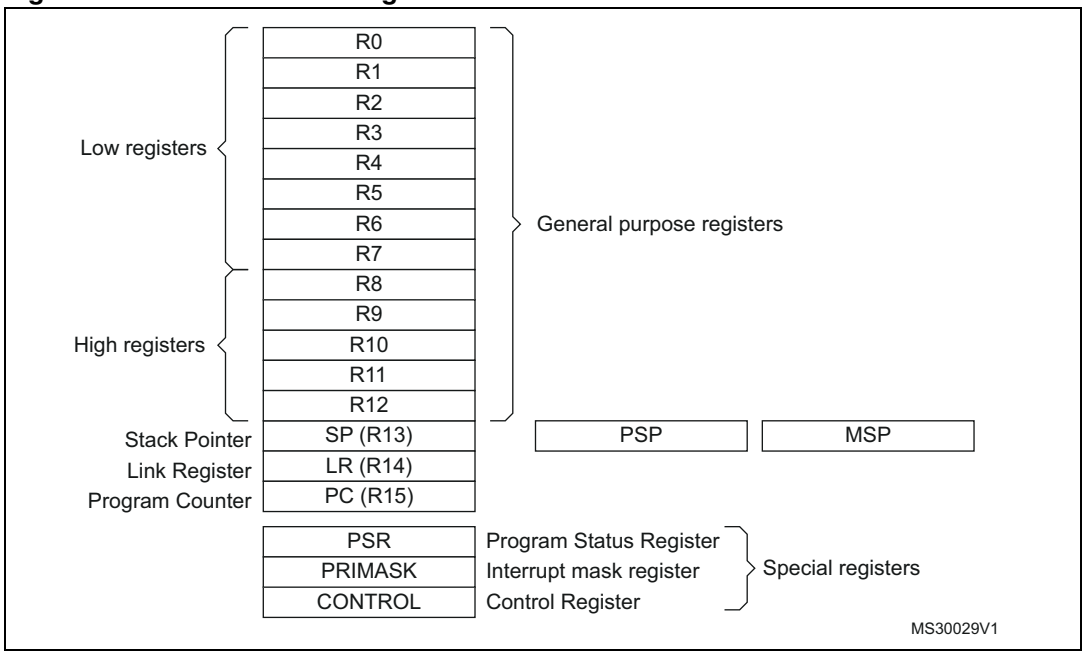


Table 3. Core register set summary

Name	Type(1)	Reset value	Description
R0-R12	read-write	Unknown	*General-purpose registers on page 12*
MSP	read-write	See description	*Stack pointer (SP) register R13 on page 13*
PSP	read-write	Unknown	*Stack pointer (SP) register R13 on page 13*
LR	read-write	Unknown	*Link register (LR) register R14 on page 13*
PC	read-write	See description	*Program counter (PC) register R15 on page 13*
PSR	read-write	Unknown(2)	*Program status register on page 13*
ASPR	read-write	Unknown	*Application program status register on page 14*
IPSR	read-only	0x00000000	*Interrupt program status register on page 14*
EPSR	read-only	Unknown(2)	*Execution program status register on page 15*
PRIMASK	read-write	0x00000000	*Priority mask register on page 15*
CONTROL	read-write	0x00000000	*Control register on page 16*

1. Describes access type during program execution in Thread and Handler modes. Debug access can differ.
2. 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 (SP) register R13

In Thread mode, bit[1] of the CONTROL register indicates the stack pointer to use:

- 0: Main Stack Pointer (MSP)(reset value). On reset, the processor loads the MSP with the value from address 0x00000000.
- 1: Process Stack Pointer (PSP).

Link register (LR) register R14

Stores return information for subroutines, function calls, and exceptions. On reset, the processor loads the LR value 0xFFFFFFFF.

Program counter (PC) register R15

Contains the current program address. On reset, the processor loads the PC with the value of the reset vector, which is at address 0x00000004. Bit[0] of the value is loaded into the EPSR T-bit at reset and must be 1.

Program status register

The *Program Status Register* (PSR) combines:

- *Application program status register* (APSR)
- *Interrupt program status register* (IPSR)
- *Execution program status register* (EPSR)

These registers are mutually exclusive bitfields in the 32-bit PSR. They can be accessed 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.

Figure 3. APSR, IPSR and EPSR bit assignments

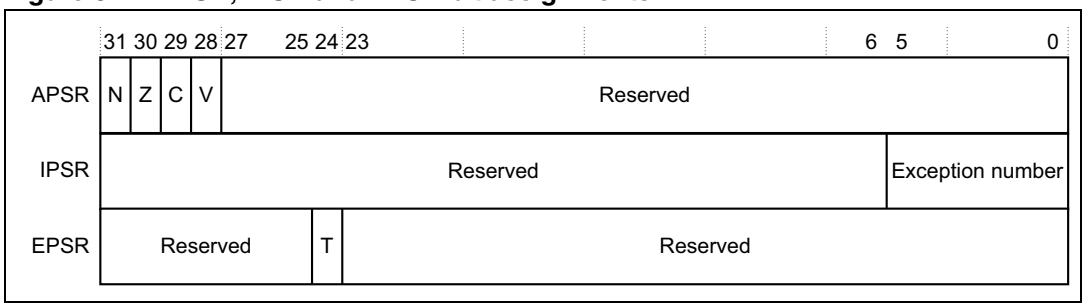


Table 4. PSR register combinations and attributes

Register	Type	Combination
PSR	read-write(1), (2)	APSR, EPSR, and IPSR
IEPSR	read-only	EPSR and IPSR
IAPSR	read-write(1)	APSR and IPSR
EAPSR	read-write(2)	APSR and EPSR

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

Application program status register

Contains the current state of *The condition flags* from previous instruction executions. See the register summary in *Table 3 on page 12* for its attributes.

Table 5. APSR bit definitions

Bits	Description
Bit 31	**N:** Negative or less than flag: 0: Operation result was positive, zero, greater than, or equal 1: Operation result was negative or less than.
Bit 30	**Z:** Zero flag: 0: Operation result was not zero 1: Operation result was zero.
Bit 29	**C:** Carry or borrow flag: 0: Add operation did not result in a carry bit or subtract operation resulted in a borrow bit 1: Add operation resulted in a carry bit or subtract operation did not result in a borrow bit.
Bit 28	**V:** Overflow flag: 0: Operation did not result in an overflow 1: Operation resulted in an overflow.
Bits 27:0	Reserved.

Interrupt program status register

Contains the exception type number of the current *Interrupt Service Routine* (ISR). See the register summary in *Table 3 on page 12* for its attributes.

Table 6. IPSR bit definitions

Bits	Description
Bits 31:6	Reserved
Bits 5:0	**ISR_NUMBER**: This is the number of the current exception, see *Exception types on page 22* for more information: 0: Thread mode 1: Resetrved 2: NMI 3: Hard fault 4-10: Reserved 11: SVCall 12: Reserved 13: Reserved 14: PendSV 15: SysTick/Reserved 16: IRQ0 .... 47: IRQ31 (see STM32 product reference manual/datasheet for interrupt mapping information) 48-63: Reserved

Execution program status register

The EPSR contains the Thumb state bit.

See the register summary in *Table 3 on page 12* for the EPSR attributes. The bit assignments are:

Table 7. EPSR bit definitions

Bits	Description
Bits 31:25	Reserved.
Bit 24	T: Thumb state bit.
Bits 23:0	Reserved.

Attempts to read the EPSR directly through application software using the MSR instruction always return zero. Attempts to write the EPSR using the MSR instruction in application software are ignored. Fault handlers can examine EPSR value in the stacked PSR to indicate the operation that is at fault. See *Section 2.3.6: Exception entry and return on page 25*.

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 28* for more information.

Interruptable-restartable instructions

LDM and STM are interruptable-restartable instructions. If an interrupt occurs during the execution of one of these instructions, the processor abandons execution of the instruction. After servicing the interrupt, the processor restarts execution of the instruction from the beginning.

Exception mask registers

The exception mask registers disable the handling of exceptions by the processor. Disable exceptions where they might impact on timing critical tasks or code sequences.

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 64*, *MSR on page 65*, and *CPSID CPSIE on page 62* for more information.

Priority mask register

The PRIMASK register prevents activation of all exceptions with configurable priority. See the register summary in *Table 3 on page 12* for its attributes.

Figure 4. PRIMASK register bit assignments

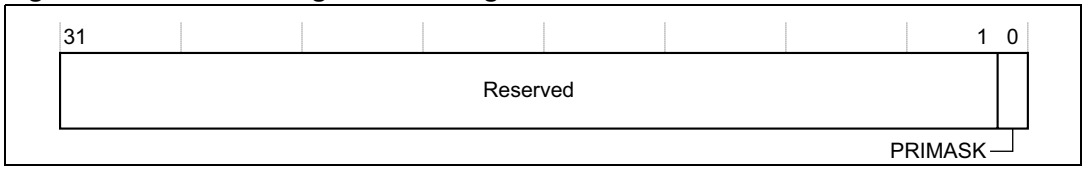


Table 8. PRIMASK register bit definitions

Bits	Description
Bits 31:1	Reserved
Bit 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 3 on page 12* for its attributes.

Figure 5. CONTROL register bit assignments

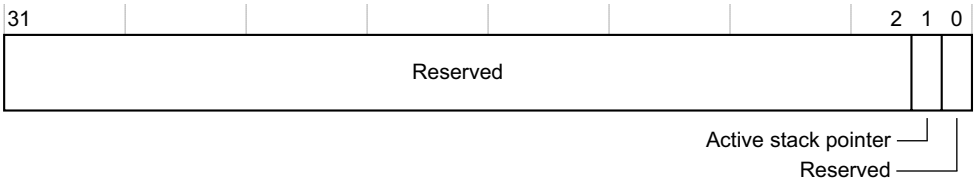


Table 9. CONTROL register bit definitions

Bits	Function
Bits 31:2	Reserved
Bit 1	**ASPSEL**: Active stack pointer selection. Selects 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.
Bit 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, it is recommended 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 *MSR on page 65*. 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 64*

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 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 26* and *Exception return on page 27* for more information. The NVIC registers control interrupt handling. See *Nested vectored interrupt controller (NVIC) on page 70* for more information.

2.1.5 Data types

The processor manages all memory accesses as little-endian. See *Memory regions, types and attributes on page 19* for more information. It supports the following data types:

- 32-bit words
- 16-bit halfwords
- 8-bit bytes

2.1.6 The Cortex microcontroller software interface standard (CMSIS)

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, the *Cortex Microcontroller Software Interface Standard* (CMSIS) defines:

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

The CMSIS includes address definitions and data structures for the core peripherals in the Cortex-M0 processor.

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

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

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

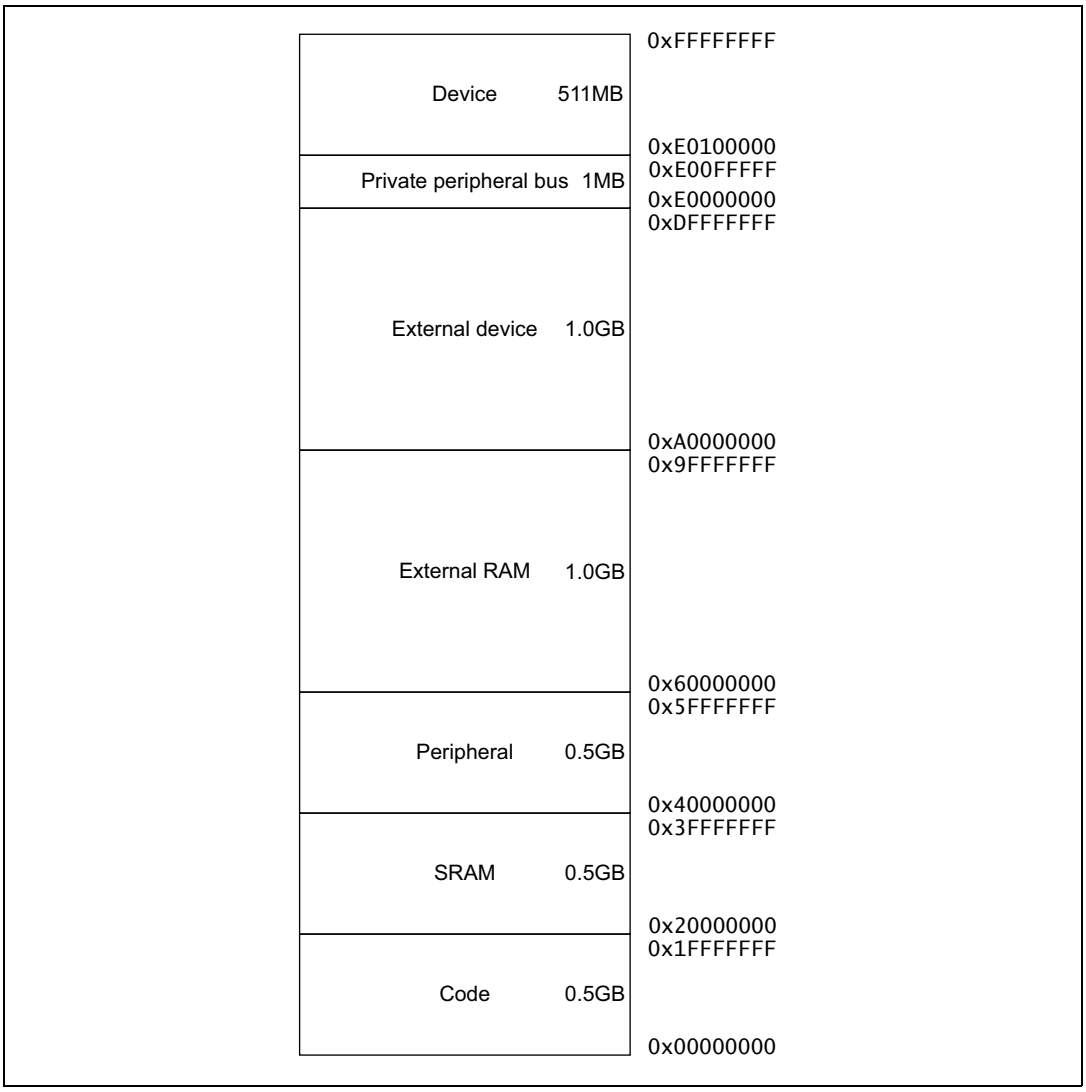
The following sections give more information about the CMSIS:

- *Power management programming hints on page 30*
- *CMSIS intrinsic functions on page 35*
- *Interrupt set-enable register (ISER) on page 71*
- *NVIC programming hints on page 75*

2.2 Memory model

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

Figure 6. Memory map



The processor reserves regions of the *Private peripheral bus* (PPB) address range for core peripheral registers, see *Section 4.1: About the STM32 Cortex-M0 core peripherals on page 69*.

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.

Additional memory attributes include:

Execute Never (XN)	Means the processor prevents instruction accesses. Any attempt to fetch an instruction from an XN region causes a HardFault exception.

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 this 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 *Section 2.2.4: Software ordering of memory accesses on page 20*.

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:

Table 10. Ordering of memory accesses(1)

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

1. - 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 11. Memory access behavior

Address range	Memory region	Memory type(1)	XN(1)	Description
0x00000000-0x1FFFFFFF	Code	Normal	-	Executable region for program code. Can also put data here.
0x20000000-0x3FFFFFFF	SRAM	Normal	-	Executable region for data. 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
0xED000000-0xED0FFFFF	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.
0xED100000-0xFFFFFFFF	Device	Device	XN	This region includes all the STM32 standard peripherals.

1. See *Memory regions, types and attributes on page 19* for more information.

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

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 have different wait states
- Some memory accesses are buffered or speculative.

Section 2.2.2: Memory system ordering of memory accesses on page 19 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* instruction ensures that outstanding memory transactions complete before subsequent memory transactions. See *DMB on page 63*.

DSB The *Data Synchronization Barrier* instruction ensures that outstanding memory transactions complete before subsequent instructions execute. See *DSB on page 63*.

ISB The *Instruction Synchronization Barrier* ensures that the effect of all completed memory transactions is recognizable by subsequent instructions. See *ISB on page 64*.

Use memory barrier instructions in, for example:

- **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 in the program. 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.

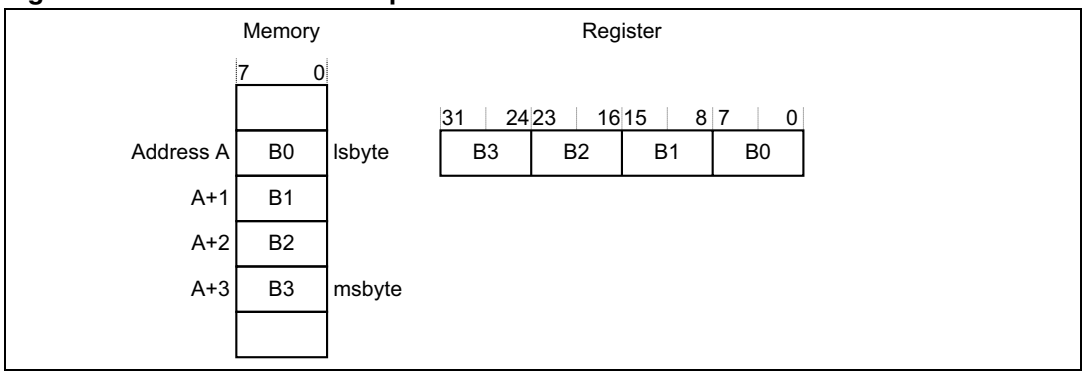
2.2.5 Memory endianness

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.

Little-endian format

In little-endian format, the processor stores the least significant byte (lsbyte) of a word at the lowest-numbered byte, and the most significant byte (msbyte) at the highest-numbered byte. See *Figure 7* for an example.

Figure 7. Little-endian 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 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 in Thread mode from the address provided by the reset entry in the vector table.
NMI	A *NonMaskable Interrupt* (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.
Hard fault	A hard fault is an exception that occurs because of an error during normal exception processing. Hard faults 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	A SysTick exception is an exception the system timer generates when it reaches zero. Software can also generate a SysTick exception. In an OS environment, the processor can use this exception as system tick.

Interrupt (IRQ) A 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 12. Properties of the different exception types

Exception number[1]	IRQ number[1]	Exception type	Priority	Vector address or offset[2]	Activation
1	-	Reset	-3, the highest	0x00000004	Asynchronous
2	-14	NMI	-2	0x00000008	Asynchronous
3	-13	Hard fault	-1	0x0000000C	Synchronous
4-10	-	Reserved	-	-	-
11	-5	SVCall	Configurable[3]	0x0000002C	Synchronous
12-13	-	Reserved	-	-	-
14	-2	PendSV	Configurable[3]	0x00000038	Asynchronous
15	-1	SysTick	Configurable[3]	0x0000003C	Asynchronous
16 - 47	0 - 31	Interrupt (IRQ)	Configurable[3]	0x00000040 and above[4]	Asynchronous

1. 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 14*.
2. See *Vector table on page 24* for more information.
3. See *Interrupt priority register (IPR0-IPR7) on page 73*.
4. Increasing in steps of 4.

For an asynchronous exception other than reset, the processor can execute another instruction between when the exception is triggered and when the processor enters the exception handler.

Software can disable the exceptions that *Table 12 on page 23* shows as having configurable priority, see:*Interrupt clear-enable register (ICER) on page 71*.

For more information about hard faults, see *Section 2.4: Fault handling on page 28*.

2.3.3 Exception handlers

The processor handles exceptions using:

Interrupt Service Routines (ISRs) Interrupts IRQ0 to IRQ31 are the exceptions handled by ISRs.

Fault handlers Hard fault is the only fault exception handled by the fault handlers.

System handlers NMI, PendSV, SVCall SysTick, and Hard fault exceptions are all system exceptions that are handled by system handlers.

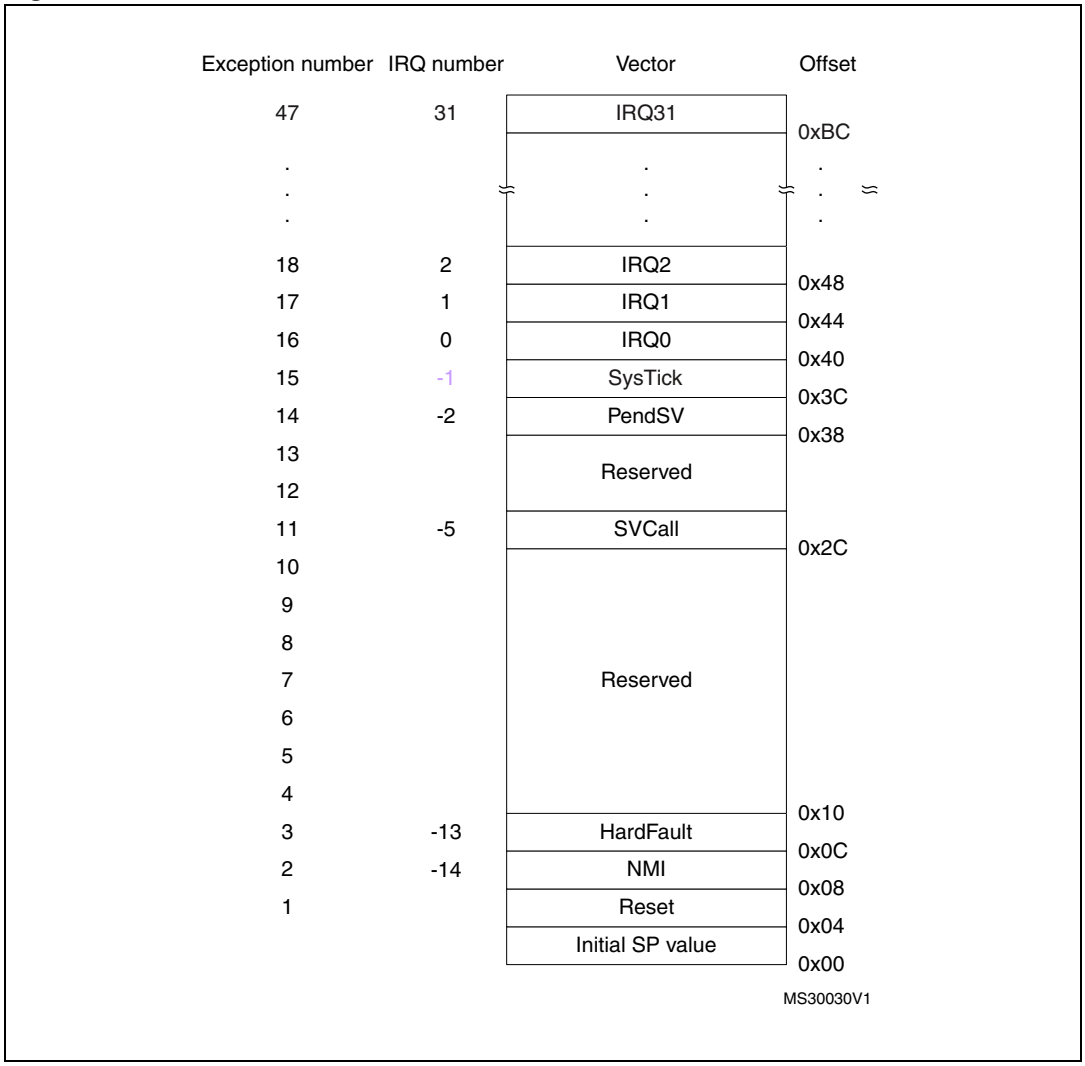
2.3.4 Vector table

The vector table contains the reset value of the stack pointer, and the start addresses, also called exception vectors, for all exception handlers.

Figure 8 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 Thumb code.

Figure 8. Vector table



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

2.3.5 Exception priorities

As *Table 12 on page 23* shows, all exceptions have an associated priority, with:

- A lower priority value indicating a higher priority
- Configurable priorities for all exceptions except Reset, Hard fault, 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 (SHPRx) on page 83*
- *Interrupt priority register (IPR0-IPR7) on page 73*

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

For example, 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 26* 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 27* 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 is the same for both exceptions. Therefore the state saving continues uninterrupted. 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:

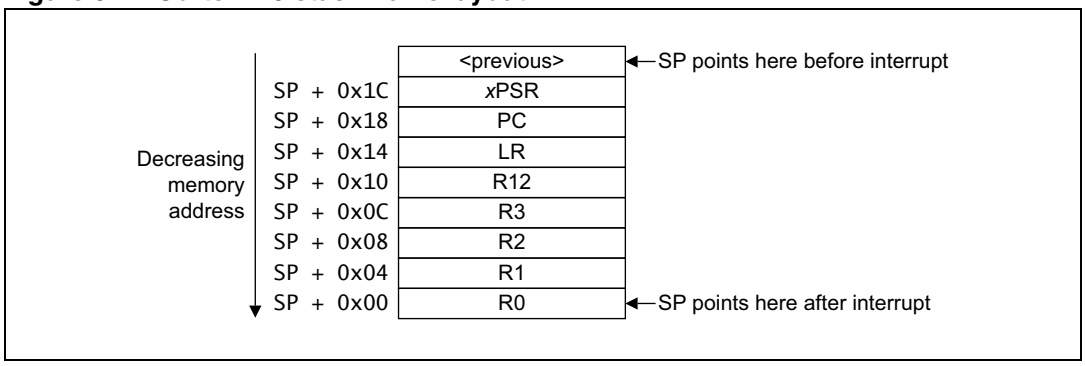
- The processor is in Thread mode
- The new exception is of higher priority than the exception being handled, in which case the new exception preempts the original exception.

When one exception preempts another, the exceptions are nested.

Sufficient priority means the exception has more priority than any limits set by the mask registers, see *Exception mask registers on page 15*. 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 as *stacking* and the structure of eight data words is referred as *stack frame*. The stack frame contains the following information:

Figure 9. Cortex-M0 stack frame layout



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 which stack pointer corresponds to the stack frame and what operation mode the was 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 executes one of the following instructions to load the EXC_RETURN value into the PC:

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

EXC_RETURN is the value loaded into 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 shown in *Table 13*.

Table 13. Exception return behavior

EXC_RETURN[31:0]	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 the exceptions, see *Exception model on page 22*. 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 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 hard fault.

Lockup

The processor enters a lockup state if a hard fault occurs when executing the NMI or hard fault 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 either:

- It is reset
- An NMI occurs and the current lockup is in the HardFault handler.
- It is halted by a debugger.

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

2.5 Power management

The STM32 and Cortex-M0 processor sleep modes reduce power consumption:

- Sleep mode stops the processor clock. All other system and peripheral clocks may still be running.
- Deep sleep mode stops most of the STM32 system and peripheral clocks. At product level, this corresponds to either the Stop or the Standby mode. For more details, please refer to the “Power modes” Section in the STM32 reference manual.

The SLEEPDEEP bit of the SCR selects which sleep mode is used, see *System control register (SCR) on page 81*. For more information about the behavior of the sleep modes see the STM32 product reference manual.

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 to sleep mode.

Wait for interrupt

The *wait for interrupt* instruction, WFI, causes immediate entry to sleep mode (unless the wake-up condition is true, see *Wakeup from WFI or sleep-on-exit on page 29*). When the processor executes a WFI instruction it stops executing instructions and enters sleep mode. See *WFI on page 68* for more information.

Wait for event

The *wait for event* instruction, WFE, causes entry to sleep mode depending 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 clears the register to 0 and continues executing instructions without entering sleep mode.

See *WFE on page 67* 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 an external event signal is asserted, or a processor in the system has executed an SEV instruction, see *SEV on page 66*. 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 it returns to Thread mode and immediately enters sleep mode. Use this mechanism in applications that only require the processor to run when an exception occurs.

2.5.2 Wakeup from sleep mode

The conditions for the processor to wakeup depend on the mechanism that cause 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 registers on page 15*.

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 *Section 2.5.3: The external event input*

In addition, if the SEVONPEND bit in the SCR is set to 1, any new pending interrupt triggers an event and wakes up the processor, even if the interrupt is disabled or has insufficient priority to cause exception entry. For more information about the SCR see *System control register (SCR) on page 81*.

2.5.3 The external event input

The processor provides an external event input signal. This signal can be generated by up to 16 external input lines and other internal asynchronous events, configured through the extended interrupt and event controller (EXTI).

This signal can 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, see *Wait for event on page 29*. Fore more details please refer to the STM32 reference manual, section 4.3 Low power modes.

2.5.4 Power management programming hints

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

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

3 The STM32 Cortex-M0 instruction set

This chapter is the reference material for the Cortex-M0 instruction set description in a User Guide. The following sections give general information:

Section 3.1: Instruction set summary on page 31

Section 3.2: CMSIS intrinsic functions on page 35

Section 3.3: About the instruction descriptions on page 36

Each of the following sections describes a functional group of Cortex-M0 instructions. Together they describe all the instructions supported by the Cortex-M0 processor:

Section 3.4: Memory access instructions on page 41

Section 3.5: General data processing instructions on page 48

Section 3.6: Branch and control instructions on page 59

Section 3.7: Miscellaneous instructions on page 61

3.1 Instruction set summary

The processor implements a version of the thumb instruction set. *Table 14* lists the supported instructions.

In *Table 14*:

- Angle brackets, <>, enclose alternative forms of the operand
- Braces, {}, enclose optional operands
- The operands column is not exhaustive
- Op2 is a flexible second operand that can be either a register or a constant
- Most instructions can use an optional condition code suffix

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

Table 14. Cortex-M0 instructions

Mnemonic	Operands	Brief description	Flags	Page
ADCS	{Rd,} Rn, Rm	Add with carry	N,Z,C,V	*3.5.1 on page 49*
ADD{S}	{Rd,} Rn, <Rm|#imm>	Add	N,Z,C,V	*3.5.1 on page 49*
ADR	Rd, label	PC-relative address to register	-	*3.4.1 on page 42*
ANDS	{Rd,} Rn, Rm	Bitwise AND	N,Z	*3.5.2 on page 51*
ASRS	{Rd,} Rm, <Rs|#imm>	Arithmetic shift right	N,Z,C	*3.5.3 on page 52*
B{cc}	label	Branch {conditionally}	-	*3.6.1 on page 59*

Table 14. Cortex-M0 instructions

Mnemonic	Operands	Brief description	Flags	Page
BICS	{Rd,} Rn, Rm	Bit clear	N,Z	*3.5.2 on page 51*
BKPT	#imm	Breakpoint	-	*3.7.1 on page 61*
BL	label	Branch with link	-	*3.6.1 on page 59*
BLX	Rm	Branch indirect with Link	-	*3.6.1 on page 59*
BX	Rm	Branch indirect	-	*3.6.1 on page 59*
CMN	Rn, Rm	Compare negative	N,Z,C,V	*3.5.4 on page 53*
CMP	Rn, <Rm|#imm>	Compare	N,Z,C,V	*3.5.4 on page 53*
CPSID	i	Change processor state, disable interrupts	-	*3.7.2 on page 62*
CPSIE	i	Change processor state, enable interrupts	-	*3.7.2 on page 62*
DMB	-	Data memory barrier	-	*3.7.3 on page 63*
DSB	-	Data synchronization barrier	-	*3.7.4 on page 63*
EORS	{Rd,} Rn, Rm	Exclusive OR	N,Z	*3.5.2 on page 51*
ISB	-	Instruction synchronization barrier	-	*3.7.5 on page 64*
LDM	Rn{!}, reglist	Load multiple registers, increment after	-	*3.4.5 on page 46*
LDR	Rt, label	Load register from PC-relative address	-	*3.4.4 on page 45*
LDR	Rt, [Rn, <Rm|#imm>]	Load register with word	-	*3.4.3 on page 44*
LDRB	Rt, [Rn, <Rm|#imm>]	Load register with byte	-	*3.4.2 on page 43*
LDRH	Rt, [Rn, <Rm|#imm>]	Load register with halfword	-	*3.4.2 on page 43*
LDRSB	Rt, [Rn, <Rm|#imm>]	Load register with signed byte	-	*3.4.2 on page 43*
LDRSH	Rt, [Rn, <Rm|#imm>]	Load register with signed halfword	-	*3.4.2 on page 43*
LSLS	{Rd,} Rn, <Rs|#imm>	Logical shift left	N,Z,C	*3.5.3 on page 52*

Table 14. Cortex-M0 instructions

Mnemonic	Operands	Brief description	Flags	Page
LSRS	{Rd,} Rn, <Rs|#imm>	Logical shift right	N,Z,C	*3.5.3 on page 52*
MOV{S}	Rd, Rm	Move	N,Z	*3.5.5 on page 54*
MRS	Rd, spec_reg	Move to general register from special register	-	*3.7.6 on page 64*
MSR	spec_reg, Rm	Move to special register from general register	N,Z,C,V	*3.7.7 on page 65*
MULS	Rd, Rn, Rm	Multiply, 32-bit result	N,Z	*3.5.6 on page 55*
MVNS	Rd, Rm	Bitwise NOT	N,Z	*3.5.5 on page 54*
NOP	-	No operation	-	*3.7.8 on page 66*
ORRS	{Rd,} Rn, Rm	Logical OR	N,Z	*3.5.2 on page 51*
POP	reglist	Pop registers from stack	-	*3.4.6 on page 47*
PUSH	reglist	Push registers onto stack	-	*3.4.6 on page 47*
REV	Rd, Rm	Byte-reverse word	-	*3.5.7 on page 56*
REV16	Rd, Rm	Byte-reverse packed halfwords	-	*3.5.7 on page 56*
REVSH	Rd, Rm	Byte-reverse signed halfword	-	*3.5.7 on page 56*
RORS	{Rd,} Rn, Rs	Rotate right	N,Z,C	*3.5.3 on page 52*
RSBS	{Rd,} Rn, #0	Reverse subtract	N,Z,C,V	*3.5.1 on page 49*
SBCS	{Rd,} Rn, Rm	Subtract with carry	N,Z,C,V	*3.5.1 on page 49*
SEV	-	Send event	-	*3.7.9 on page 66*
STM	Rn!, reglist	Store multiple registers, increment after	-	*3.4.5 on page 46*
STR	Rt, [Rn, <Rm|#imm>]	Store register as word	-	*3.4.2 on page 43*
STRB	Rt, [Rn, <Rm|#imm>]	Store register as byte	-	*3.4.2 on page 43*
STRH	Rt, [Rn, <Rm|#imm>]	Store register as halfword	-	*3.4.2 on page 43*

Table 14. Cortex-M0 instructions

Mnemonic	Operands	Brief description	Flags	Page
SUB{S}	{Rd,} Rn, <Rm|#imm>	Subtract	N,Z,C,V	*3.5.1 on page 49*
SVC	#imm	Supervisor call	-	*3.7.10 on page 67*
SXTB	Rd, Rm	Sign extend byte	-	*3.5.8 on page 57*
SXTH	Rd, Rm	Sign extend halfword	-	*3.5.8 on page 57*
TST	Rn, Rm	Logical AND based test	N,Z	*3.5.9 on page 58*
UXTB	Rd, Rm	Zero extend a byte	-	*3.5.8 on page 57*
UXTH	Rd, Rm	Zero extend a halfword	-	*3.5.8 on page 57*
WFE	-	Wait for event	-	*3.7.11 on page 67*
WFI	-	Wait for interrupt	-	*3.7.12 on page 68*

3.2 CMSIS 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 CMIS 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 an inline assembler to access some instructions.

The CMSIS provides the intrinsic functions listed in *Table 15* to generate instructions that ISO/TEC C code cannot directly access.

Table 15. 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 (see *Table 16*).

Table 16. 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 operands. Operands in some instructions are flexible in that they can either be a register or a constant (see *Shift operations*).

3.3.2 Restrictions when using PC or SP

Many instructions have restrictions on whether you can use the *program counter* (PC) or *stack pointer* (SP) for the operands or destination register. See instruction descriptions for more information.

Bit[0] of any address written to the PC with a BX, BLX or POP instruction must be 1 for correct execution, because this bit indicates the required 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. 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 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 10: ASR#3*).

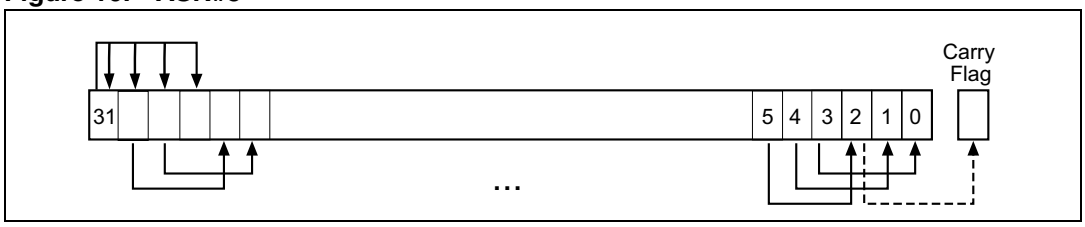
You can use the ASR operation to divide the signed value in the register *Rm* by 2^n, with the result being rounded towards negative-infinity.

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

Note: 1 If n is 32 or more, all the bits in the result are set to the value of bit[31] of Rm.

2 If n is 32 or more and the carry flag is updated, it is updated to the value of bit[31] of Rm.

Figure 10. 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 11*).

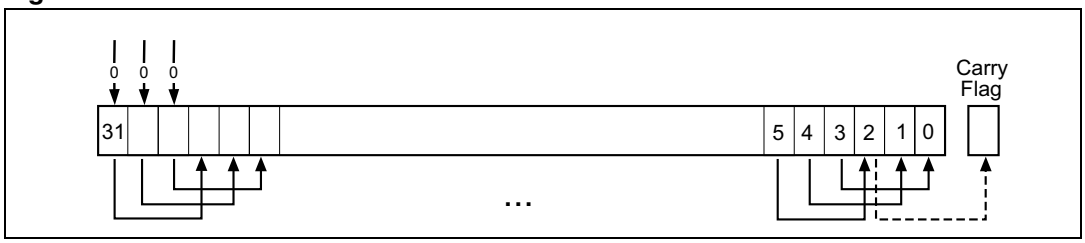
You can use the LSR #n operation to divide the value in the register *Rm* by 2^n, if the value is regarded as an unsigned integer.

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

Note: 1 If n is 32 or more, then all the bits in the result are cleared to 0.

2 If n is 33 or more and the carry flag is updated, it is updated to 0.

Figure 11. 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 12: LSL#3 on page 38*).

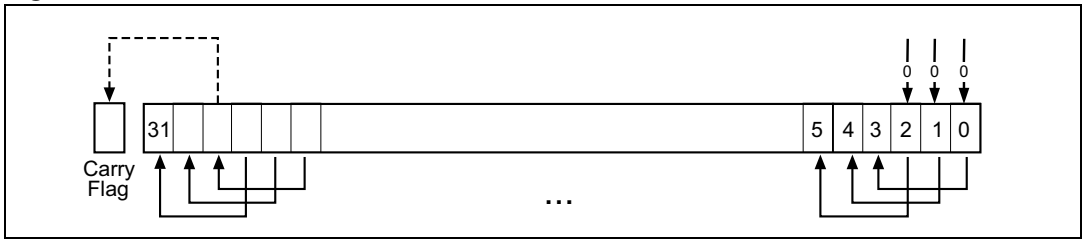
You can use the LSL #n operation to multiply the value in the register *Rm* by 2^n, if the value is regarded as an unsigned integer or a two's complement signed integer. Overflow can occur without warning.

When the instruction is LSLS or when LSL #*n*, with non-zero *n*, is used in *operand2* with the instructions MOVS, MVNS, ANDS, ORRS, ORNS, EORS, BICS, TEQ or TST, 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: 1 If n is 32 or more, then all the bits in the result are cleared to 0.

2 If n is 33 or more and the carry flag is updated, it is updated to 0.

Figure 12. LSL#3



ROR

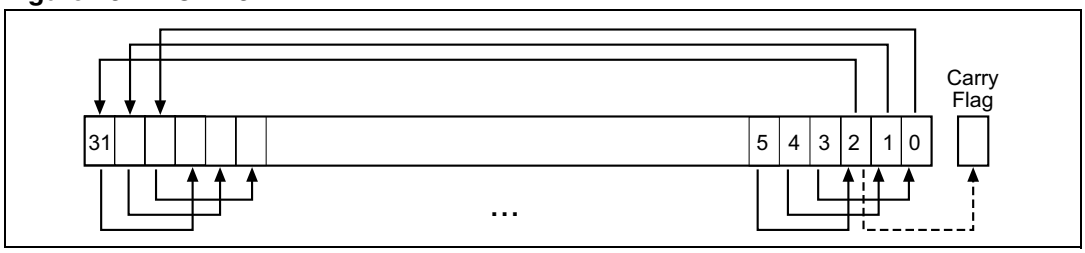
Rotate right by *n* bits moves the left-hand 32-*n* bits of the register *Rm*, to the right by *n* places, into the right-hand 32-*n* bits of the result. It also moves the right-hand *n* bits of the register into the left-hand *n* bits of the result (see *Figure 13*).

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

Note: 1 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.

2 ROR with shift length, n, more than 32 is the same as ROR with shift length n-32.

Figure 13. 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.

- For most instructions, the value of the PC is the address of the current instruction plus four 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, #number].

3.3.6 Conditional execution

Most data processing instructions can optionally update the condition flags in the *application program status register* (APSR) according to the result of the operation (see *Application program status register on page 14*). 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 an instruction conditionally, based on the condition flags set in another instruction:

- 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*
- *Condition code suffixes*

The condition flags

The APSR contains the following condition flags:

- N: Set to 1 when the result of the operation is negative, otherwise cleared to 0
- Z: Set to 1 when the result of the operation is zero, otherwise cleared to 0
- C: Set to 1 when the operation results in a carry, otherwise cleared to 0.
- V: Set to 1 when the operation causes an overflow, otherwise cleared to 0.

For more information about the APSR see *Program status register on page 13*.

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 if the sign of a 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

The instructions that can be conditional have an optional condition code, shown in syntax descriptions as B{cond}. An instruction with a condition code is only executed if the condition code flags in the APSR meet the specified condition. *Table 17* shows the condition codes to use.

You can use conditional execution with the IT instruction to reduce the number of branch instructions in code.

Table 17 also shows the relationship between condition code suffixes and the N, Z, C, and V flags.

Table 17. Condition code suffixes and their relationship with the flags

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 18 shows the memory access instructions:

Table 18. Memory access instructions

Mnemonic	Brief description	See
ADR	Load PC-relative address	*ADR on page 42*
LDM	Load multiple registers	*LDM and STM on page 46*
LDR{type}	Load register using immediate offset	*LDR and STR, immediate offset on page 43*
LDR{type}	Load register using register offset	*LDR and STR, register offset on page 44*
LDR	Load register using PC-relative address	*LDR, PC-relative on page 45*
LDRD	Load register dual	*LDR and STR, immediate offset on page 43*
POP	Pop registers from stack	*PUSH and POP on page 47*
PUSH	Push registers onto stack	*PUSH and POP on page 47*
STM	Store multiple registers	*LDM and STM on page 46*
STR{type}	Store register using immediate offset	*LDR and STR, immediate offset on page 43*
STR{type}	Store register using register offset	*LDR and STR, register offset on page 44*

3.4.1 ADR

Load 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 39*)

Operation

ADR determines the address by adding an immediate value to the PC. It writes the result to the destination register.

ADR produces 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 to1 for correct execution.

Restrictions

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

For 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 39*.

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 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 39*

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
LDRSHR1, [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 or store
- '*label*' is a PC-relative expression (see *PC-relative expressions on page 39*)

Operation

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

Restrictions

In these instructions: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
- '*!*' is an optional writeback suffix. If ! is present, the final address that is loaded from or stored to is written back into *Rn*.
- '*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 46*).

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 loads the registers in *reglist* with word values from memory addresses based on *Rn*.

STM stores the word values in the registers in *reglist* to memory addresses based on *Rn*.

The memory addresses used for accesses are at 4-byte intervals ranging from *Rn* to *Rn* + 4 * (*n*-1), where *n* is the number of registers in *reglist*. The accesses happen in order of increasing register numbers, with the lowest numbered register using the lowest memory address and the highest number register using the highest memory address. If the writeback suffix is specified, the value in the register specified by of *Rn* + 4 * (*n*) or 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 39* 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
STMIAR1!,{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 (or register ranges), enclosed in braces. Commas must separate register lists or ranges (see *Examples on page 46*).

Operation

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

See *LDM and STM on page 46* for more information.

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 new PC.
```

3.5 General data processing instructions

Table 19 shows the data processing instructions.

Table 19. Data processing instructions

Mnemonic	Brief description	See
ADCS	Add with carry	*ADD{S}, ADCS, SUB{S}, SBCS, and RSBS on page 49*
ADD(S)	Add	*ADD{S}, ADCS, SUB{S}, SBCS, and RSBS on page 49*
ANDS	Logical AND	*ANDS, ORRS, EORS and BICS on page 51*
ASRS	Arithmetic shift right	*ASRS, LSLS, LSRS and RORS on page 52*
BICS	Bit clear	*ANDS, ORRS, EORS and BICS on page 51*
CMN	Compare negative	*CMP and CMN on page 53*
CMP	Compare	*CMP and CMN on page 53*
EORS	Exclusive OR	*ANDS, ORRS, EORS and BICS on page 51*
LSLS	Logical shift left	*ASRS, LSLS, LSRS and RORS on page 52*
LSRS	Logical shift right	*ASRS, LSLS, LSRS and RORS on page 52*
MOV(S)	Move	*MOV, MOVS and MVNS on page 54*
MULS	Multiply	*MULS on page 55*
MVNS	Move NOT	*MOV, MOVS and MVNS on page 54*
ORRS	Logical OR	*ANDS, ORRS, EORS and BICS on page 51*
REV	Reverse byte order in a word	*REV, REV16, and REVSH on page 56*
REV16	Reverse byte order in each halfword	*REV, REV16, and REVSH on page 56*
REVSH	Reverse byte order in bottom halfword and sign extend	*REV, REV16, and REVSH on page 56*
RORS	Rotate right	*ASRS, LSLS, LSRS and RORS on page 52*
RSBS	Reverse subtract	*ADD{S}, ADCS, SUB{S}, SBCS, and RSBS on page 49*
SBCS	Subtract with carry	*ADD{S}, ADCS, SUB{S}, SBCS, and RSBS on page 49*
SUBS	Subtract	*ADD{S}, ADCS, SUB{S}, SBCS, and RSBS on page 49*
SUBW	Subtract	*ADD{S}, ADCS, SUB{S}, SBCS, and RSBS on page 49*
SXTB	Sign extends to 32 bits	*SXTB, SXTH, UXTB and UXTH on page 57*
SXTH	Sign extends to 32 bits	*SXTB, SXTH, UXTB and UXTH on page 57*
UXTB	Zero extends to 32 bits	*SXTB, SXTH, UXTB and UXTH on page 57*
UXTH	Zero extends to 32 bits	*SXTB, SXTH, UXTB and UXTH on page 57*
TST	Test	*TST on page 58*

3.5.1 ADD{S}, ADCS, SUB{S}, SBCS, and RSBS

Add, add with carry, subtract, subtract with carry, and reverse subtract.

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. If omitted,this value is assumed to take the same value as *Rn*, for example ADDS R1,R2 is identical to ADDS R1,R1,R2.
- *Rn*: specifies the first source register
- *Rm*: specifies the second source register
- *imm*: specifies a constant immediate value.

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 *Multiword arithmetic examples on page 50* and *ADR on page 42*).

Restrictions

Table 20 lists the legal combinations of register specifiers and immediate values that can be used with each instruction.

Table 20. ADCS, ADD, RSBS, SBCS and SUB operand restrictions

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

Examples

Multiword arithmetic examples

Specific example: 64-bit addition 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.

Specific example: 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. *Specific example: 96-bit subtraction* 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.

Specific example: 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
```

Specific example: Arithmetic negation shows the RSBS instruction used to perform a 1's complement of a single register.

Specific example: Arithmetic negation

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

3.5.2 ANDS, ORRS, EORS and BICS

Logical AND, OR, exclusive OR and bit clear.

Syntax

```
ANDS {Rd,} Rn, Rm
ORRS {Rd,} Rn, Rm
EORS {Rd,} Rn, Rm
BICS {Rd,} Rn, Rm
```

where:

- '*Rd*' is the destination register
- '*Rn*' is the register holding the first operand and is the same as the destination register.
- '*Rm*' is the 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 39*.

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 ASRS, LSLS, LSRS and RORS

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 *Rm*.
- '*imm*' is the shift length. The range of shift lengths depend on the instruction as follows:
  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 the preferred syntax 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 36*).

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 36*).

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

In these instructions:

- CMN instruction *Rn*, and *Rm* must only specify R0-R7.
- CMP instruction:
  - *Rn* and *Rm* can specify R0-R14
  - *imm* 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, MOVS and MVNS

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 39*).
- '*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 NOT 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 ignored
- A branch occurs to the address created by forcing bit[0] of that value 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 to the ARM instruction set.

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 flag

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 39*.

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. It 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 and reverse bits.

Syntax

```
op Rd, Rn
```

where:

- '*op*' is one of:
  REV: Reverse byte order in a word
  REV16: Reverse byte order in each halfword independently
  REVSH: Reverse byte order in the bottom halfword, and sign extends to 32 bits
- '*Rd*' is the destination register
- '*Rn*' is the register holding the operand

Operation

Use these instructions to change endianness of data:

- REV**:** Converts either:
  - 32-bit big-endian data into little-endian data or
  - 32-bit little-endian data into big-endian data.
- REV16**:** Converts either:
  - 2 packed 16-bit big-endian data into little-endian data or
  - 2 packed 16-bit little-endian data into big-endian data.
- REVSH: Converts either:
  - 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 SXTB, SXTH, UXTB and UXTH

Sign extend and Zero extend.

Syntax

```
SXTB Rd, Rm
SXTH Rd, Rm
UXTB Rd, Rm
UXTH Rd, Rm
```

where:

- '*Rd*' is the destination register
- '*Rn*' ,'*Rm*' are the registers holding the first and second operands

Operation

These instructions extract bits from the resulting value:

1. SXTB extracts bits[7:0] and sign extends to 32 bits
2. UXTB extracts bits[7:0] and zero extends to 32 bits
3. SXTH extracts bits[15:0] and sign extends to 32 bits
4. 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*' is 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 flag

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 21 shows the branch and control instructions:

Table 21. Branch and control instructions

Mnemonic	Brief description	See
B{cc}	Branch {conditionally}	*B, BL, BX, and BLX on page 59*
BL	Branch with link	
BLX	Branch indirect with link	
BX	Branch indirect	

3.6.1 B, BL, BX, and BLX

Branch instructions.

Syntax

```
B{cond} label
BL label
BX Rm
BLX Rm
```

where:

- 'B' is branch (immediate).
- 'BL' is branch with link (immediate).
- 'BX' is branch indirect (register).
- 'BLX' is branch indirect with link (register).
- *'label'* is a PC-relative expression. See *PC-relative expressions on page 39*.
- *'Rm'* is a register that indicates an address to branch to.
- *'Cond'* is an optional condition code, see *Conditional execution on page 39*.

Operation

All these instructions cause a branch to *label*, or to the address indicated in *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 cause a Hard fault exception if bit[0] of *Rm* is 0.
- The 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 22 shows the ranges for the various branch instructions.

Table 22. Branch ranges

Instruction	Branch range
B label	–2 KB to +2 KB
B*cond* label	–256 bytes to +254 bytes

Table 22. Branch ranges (continued)

Instruction	Branch range
BL label	−16 MB to +16 MB
BX Rm	Any value in register
BLX Rm	Any value in register

Restrictions

The restrictions are:

- 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.

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 23 shows the remaining Cortex-M0 instructions:

Table 23. Miscellaneous instructions

Mnemonic	Brief description	See
BKPT	Breakpoint	*BKPT on page 61*
CPSID	Change Processor State, Disable Interrupts	*CPSID CPSIE on page 62*
CPSIE	Change Processor State, Enable Interrupts	*CPSID CPSIE on page 62*
DMB	Data Memory Barrier	*DMB on page 63*
DSB	Data Synchronization Barrier	*DSB on page 63*
ISB	Instruction Synchronization Barrier	*ISB on page 64*
MRS	Move from special register to register	*MRS on page 64*
MSR	Move from register to special register	*MSR on page 65*
NOP	No Operation	*NOP on page 66*
SEV	Send Event	*SEV on page 66*
SVC	Supervisor Call	*SVC on page 67*
WFE	Wait For Event	*WFE on page 67*
WFI	Wait For Interrupt	*WFI on page 68*

3.7.1 BKPT

Breakpoint.

Syntax

```
BKPT #imm
```

where: '*imm*' is an integer in the range 0-255.

Operation

BKPT 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 produce a HardFault or go in to lockup if a debugger is not attached when a BKPT instruction is executed. See *Lockup on page 28* for more information.

Restrictions: None

Condition flags

This instruction does not change the flags.

Examples

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

3.7.2 CPSID CPSIE

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 registers on page 15* for more information about these registers.

Restrictions

None

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 completed before any explicit memory accesses that appear, in program order, after the DMB instruction. DMB does not affect the ordering or execution of instructions that do not access memory.

Restrictions

None

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

None

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

None

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. MRS can be combined with the MSR instruction to produce read-modify-write sequences, which are suitable for modifying a specific flag in the PSR. See *MSR on page 65*.

Restrictions

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 64*.

Restrictions

Rn must not be SP and must not be 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 does nothing. NOP is not necessarily a time-consuming NOP. The processor might remove it from the pipeline before it reaches the execution stage.

Use NOP for padding, for example to place the following instruction on a 64-bit boundary.

Restrictions

None

Condition flags

This instruction does not change the flags.

Examples

```
NOP  ; No operation
```

3.7.9 SEV

Send event.

Syntax

```
SEV
```

Operation

SEV is a hint instruction that causes an event to be signaled to all processors within a multiprocessor system. It also sets the local event register to 1, see *Power management on page 28* and *WFE on page 67*.

Restrictions

None

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. It can be retrieved by the exception handler to determine what service is being requested.

Restrictions: None

Condition flags

This instruction does not change the flags.

Examples

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

3.7.11 WFE

Wait for event. WFE is a hint instruction.

Syntax

```
WFE
```

Operation

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

- An exception, unless masked by exception mask registers or the current priority level
- An exception enters Pending state, if SEVONPEND in 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 returns immediately. For more information see *Power management on page 28*.

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

Restrictions: None

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 is a hint instruction that 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.

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

Restrictions

None

Condition flags

This instruction does not change the flags.

Examples

```
WFI ; Wait for interrupt
```

4 Core peripherals

4.1 About the STM32 Cortex-M0 core peripherals

The address map of the *Private peripheral bus* (PPB) is:

Table 24. STM32 core peripheral register regions

Address	Core peripheral	Description
0xE000E008-0xE000E00F	*System control block (SCB)*	*Table 32 on page 84*
0xE000E010-0xE000E01F	*SysTick timer (STK)*	*Table 34 on page 89*
0xE000E100-0xE000E4EF	*Nested vectored interrupt controller (NVIC)*	*Table 29 on page 76*
0xE000ED00-0xE000ED3F	*System control block (SCB)*	*Table 32 on page 84*
0xE000EF00-0xE000EF03	*Nested vectored interrupt controller (NVIC)*	*Table 29 on page 76*

In register descriptions, register type is described as follows:

- RW: Read and write.
- RO: Read-only.
- WO: Write-only.

4.2 Nested vectored interrupt controller (NVIC)

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

- Up to 32 interrupts
- 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 *Non-maskable interrupt* (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 25. NVIC register summary

Address	Name	Type	Reset value	Description
0xE000E100	ISER	RW	0x00000000	*Table 4.2.2: Interrupt set-enable register (ISER) on page 71*
0XE000E180	ICER	RW	0x00000000	*Table 4.2.3: Interrupt clear-enable register (ICER) on page 71*
0XE000E200	ISPR	RW	0x00000000	*Table 4.2.4: Interrupt set-pending register (ISPR) on page 72*
0XE000E280	ICPR	RW	0x00000000	*Table 4.2.5: Interrupt clear-pending register (ICPR) on page 72*
0xE000E400-0xE000E41C	IPR0-IPR7	RW	0x00000000	*Table 4.2.6: Interrupt priority register (IPR0-IPR7) on page 73*

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 26. CMSIS access NVIC functions

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

1. The input parameter IRQn is the IRQ number,

4.2.2 Interrupt set-enable register (ISER)

Address offset: 0x00

Reset value: 0x0000 0000

The ISER register enables interrupts, and shows which interrupts are enabled

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SETENA[31:16]															
rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SETENA[15:0]															
rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs

Bits 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 (ICER)

Address offset: 0x080

Reset value: 0x0000 0000

The ICER register disables interrupts, and shows which interrupts are enabled.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CLRENA[31:16]															
rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CLRENA[15:0]															
rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1

Bits 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 (ISPR)

Address offset: 0x0100

Reset value: 0x0000 0000

This register forces interrupts into pending state, and shows which interrupts are pending.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SETPEND[31:16]															
rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SETPEND[15:0]															
rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs

Bits 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

Writing 1 to the ISPR bit corresponding to an interrupt that is pending:

– has no effect.

Writing 1 to the ISPR bit corresponding to a disabled interrupt:

– sets the state of that interrupt to pending.

4.2.5 Interrupt clear-pending register (ICPR)

Address offset: 0x0180

Reset value: 0x0000 0000

Removes pending state from interrupts and shows which interrupts are pending.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CLRPEND[31:16]															
rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CLRPEND[15:0]															
rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1

Bits 31:0 **CLRPEND**: Interrupt clear-pending bits

Write:

0: No effect

1: Removes the pending state of an interrupt

Read:

0: Interrupt is not pending

1: Interrupt is pending

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

4.2.6 Interrupt priority register (IPR0-IPR7)

Address offset: 0x0300

Reset value: 0x0000 0000

The IPR registers provide an 8-bit priority field for each interrupt. These registers are only word-accessible. Each register holds four priority fields, as shown in *Figure 14*.

Figure 14. IPR register mapping

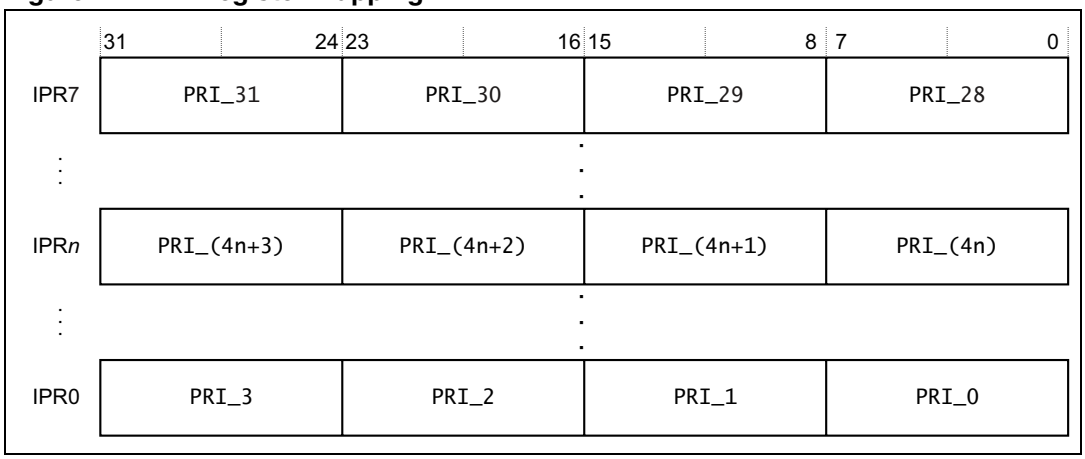


Table 27. IPR bit assignments

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 *Interrupt set-enable register (ISER) on page 71 Accessing the Cortex-M0 NVIC registers using CMSIS on page 70* for more information about the interrupt priority array, that provides the software view of the interrupt priorities.

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

- The corresponding IPR number, *M*, is given by *M* = *N* DIV 4
- The byte offset of the required Priority field in this register is *N* 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

STM32 interrupts are both level-sensitive and pulse-sensitive. 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 needs servicing.

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 HIGH (active) and the 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 *Section 4.2.4: Interrupt set-pending register (ISPR)*.

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 design hints and tips

Ensure software uses correctly aligned register accesses. The processor does not support unaligned accesses to NVIC registers. See the individual register descriptions for the supported access sizes.

An interrupt can enter pending state even 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 28. 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 12: Properties of the different exception types on page 23*. For more information about these functions see the CMSIS documentation.

4.2.9 NVIC register map

This table shows the NVIC register map and reset values. The base address of the main NVIC register block is 0xE000E100.

Table 29. NVIC register map and reset values

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x000	**NVIC_ISER**	SETENA[31:0]																															
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x080	**NVIC_ICER**	CLRENA[31:0]																															
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x100	**NVIC_ISPR**	SETPEND[31:0]																															
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x180	**NVIC_ICPR**	CLRPEND[31:0]																															
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x300	**NVIC_IPR0-7**	IP[3]								IP[2]								IP[1]								IP[0]							
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

4.3 System control block (SCB)

The *System control block* (SCB) provides system implementation information, and system control. This includes configuration, control, and reporting of the system exceptions. To improve software efficiency, the CMSIS simplifies the SCB register presentation, in the CMSIS, the array SHP[1] corresponds to the registers SHPR2-SHPR3.

Table 30. Summary of the system control block registers

Address	Name	Type	Reset value	Description
0xE000ED00	CPUID	RO	0x410CC200	*Section 4.3.1: CPUID base register (CPUID) on page 77*
0xE000ED04	ICSR	RW(1)	0x00000000	*Section 4.3.2: Interrupt control and state register (ICSR) on page 78*
0xE000ED0C	AIRCR	RW(1)	0xFA050000	*Section 4.3.3: Application interrupt and reset control register (AIRCR) on page 80*
0xE000ED10	SCR	RW	0x00000000	*Section 4.3.4: System control register (SCR) on page 81*
0xE000ED14	CCR	RW	0x00000204	*Section 4.3.5: Configuration and control register (CCR) on page 82*
0xE000ED1C	SHPR2	RW	0x00000000	*Section 4.3.6: System handler priority registers (SHPRx) on page 83*
0xE000ED20	SHPR3	RW	0x00000000	

1. See the register description for more information.

4.3.1 CPUID base register (CPUID)

Address offset: 0x00

Reset value: 0x410C C200

The CPUID register contains the processor part number, version, and implementation information.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Implementer								Variant				Constant			
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PartNo												Revision			
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Bits 31:24 **Implementer**: Implementer code
0x41: ARM

Bits 23:20 **Variant**: Variant number: The r value in the *Rnpn* product revision identifier
0x0: revision 0

Bits 19:16 **Constant:** Constant that defines the architecture of the processor:
0xC: ARMv6-M architecture

Bits 15:4 **PartNo**: Part number of the processor
0xC20: Cortex-M0

Bits 3:0 **Revision**: The p value in the *Rnpn* product revision identifier, indicates patch release.
0x0: patch 0

4.3.2 Interrupt control and state register (ICSR)

Address offset: 0x04

Reset value: 0x0000 0000

The ICSR:

- Provides:
  - A set-pending bit for the *Non-Maskable Interrupt* (NMI) exception
  - Set-pending and clear-pending bits for the PendSV and SysTick exceptions
- Indicates:
  - The exception number of the exception being processed
  - Whether there are preempted active exceptions
  - The exception number of the highest priority pending exception
  - Whether any interrupts are pending.

Caution: 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.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
NMIPENDSET	Reserved		PENDSVSET	PENDSVCLR	PENDSTSET	PENDSTCLR	Reserved		ISRPENDING	Reserved				VECTPENDING[5:4]	
rw			rw	w	rw	w			r					r	r

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VECTPENDING[3:0]				Reserved						VECTACTIVE[5:0]					
r	r	r	r							rw	rw	rw	rw	rw	rw

Bit 31 **NMIPENDSET**: NMI set-pending bit.

Write:

0: No effect

1: Change 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 enter the NMI exception handler as soon as it registers a write of 1 to this bit, and entering the handler clears this bit to 0. 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.

Bits 30:29 Reserved

Bit 28 **PENDSVSET:** PendSV set-pending bit.

Write:

0: No effect

1: Change 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.

Bit 27 **PENDSVCLR:** PendSV clear-pending bit. This bit is write-only. On a read, value is unknown.
0: No effect
1: Removes the pending state from the PendSV exception.

Bit 26 **PENDSTSET:** SysTick exception set-pending bit.
Write:
0: No effect
1: Change SysTick exception state to pending
Read:
0: SysTick exception is not pending
1: SysTick exception is pending

Bit 25 **PENDSTCLR:** SysTick exception clear-pending bit. Write-only. On a read, value is unknown.
0: No effect
1: Removes the pending state from the SysTick exception.

Bit 24:23 Reserved, must be kept cleared.

Bit 22 **ISRPENDING:** Interrupt pending flag, excluding NMI and Faults.
0: Interrupt not pending
1: Interrupt pending

Bits 21:18 Reserved, must be kept cleared.

Bits 17:12 **VECTPENDING**: Pending vector. Indicates the exception number of the highest priority pending enabled exception.
0: No pending exceptions
Other values: The exception number of the highest priority pending enabled exception.

Bits 11:6 Reserved

Bits 5:0 **VECTACTIVE** Active vector. Contains the active exception number:
0: Thread mode
Other values: The exception number(1) of the currently active exception.

Note: Subtract 16 from this value to obtain CMSIS IRQ number required to index into the Interrupt Clear-Enable, Set-Enable, Clear-Pending, Set-Pending, or Priority Registers, see Table 6 on page 14.

1. This is the same value as IPSR bits[5:0], see *Interrupt program status register on page 14*.

4.3.3 Application interrupt and reset control register (AIRCR)

Address offset: 0x0C

Reset value: 0xFA05 0000

The AIRCR provides endian status for data accesses and reset control of the system.

To write to this register, you must write 0x5FA to the VECTKEY field, otherwise the processor ignores the write.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved(read)/ VECTKEY[15:0](write)															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ENDIANESS	Reserved												SYSRESETREQ	VECTCLRACTIVE	Reserved
r													w	w	

Bits 31:16 **Reserved / VECTKEY** Register key
Reads as unknown
On writes, write 0x5FA to VECTKEY, otherwise the write is ignored.

Bit 15 **ENDIANESS** Data endianness bit
Reads as 0.
0: Little-endian

Bits 14:3 Reserved, must be kept cleared

Bit 2 **SYSRESETREQ** System reset request
Reads as 0.
0: No system reset request
1: Asserts a signal to the outer system that requests a reset.

Bit 1 **VECTCLRACTIVE**
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.

Bit 0 Reserved, must be kept cleared

4.3.4 System control register (SCR)

Address offset: 0x10

Reset value: 0x0000 0000

The SCR controls features of entry to and exit from low power state.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved											SEVONPEND	Res.	SLEEPDEEP	SLEEPONEXIT	Res.
											rw		rw	rw	

Bits 31:5 Reserved, must be kept cleared

Bit 4 **SEVEONPEND** Send Event on Pending bit

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

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.

Bit 3 Reserved, must be kept cleared

Bit 2 **SLEEPDEEP**

Controls whether the processor uses sleep or deep sleep as its low power mode:

0: Sleep

1: Deep sleep.

Bit 1 **SLEEPONEXIT**

Configures sleep-on-exit when returning from Handler mode to Thread mode. Setting this bit to 1 enables an interrupt-driven application to avoid returning to an empty main application.

0: Do not sleep when returning to Thread mode.

1: Enter sleep, or deep sleep, on return to Thread mode from an interrupt service routine.

Bit 0 Reserved, must be kept cleared

4.3.5 Configuration and control register (CCR)

Address offset: 0x14

Reset value: 0x0000 0204

The CCR is a read-only register and indicates some aspects of the behavior of the Cortex-M0 processor.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						STK ALIGN	Reserved					UN ALIGN_ TRP	Reserved		
						rw						rw			

Bits 31:10 Reserved, must be kept cleared

Bit 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.

Bits 8:4 Reserved, must be kept cleared

Bit 3 **UNALIGN_ TRP**

Always reads as one, indicates that all unaligned accesses generate a HardFault.

Bits 2:0 Reserved, must be kept cleared

4.3.6 System handler priority registers (SHPRx)

The SHPR2-SHPR3 registers set the priority level, 0 to 192, of the exception handlers that have configurable priority. SHPR2-SHPR3 are word accessible. To access the system exception priority level using CMSIS, use the following CMSIS functions (where the input parameter IRQn is the IRQ number):

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

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.

Table 31. System fault handler priority fields and registers

Handler	Field	Register description
SVCall	PRI_11	*System handler priority register 2 (SHPR2) on page 83*
PendSV	PRI_14	*System handler priority register 3 (SHPR3) on page 83*
SysTick	PRI_15	

System handler priority register 2 (SHPR2)

Address offset: 0x1C

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PRI_11								Reserved							
rw	rw	rw	rw	r	r	r	r								
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved															

Bits 31:24 **PRI_11:** Priority of system handler 11, SVCall

Bits 23:0 Reserved, must be kept cleared

System handler priority register 3 (SHPR3)

Address: 0xE000 ED20

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PRI_15								PRI_14							
rw	rw	rw	rw	r	r	r	r	rw	rw	rw	rw	r	r	r	r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved															

Bits 31:24 **PRI_15:** Priority of system handler 15, SysTick exception.
This is Reserved when the SysTick timer is not implemented.

Bits 23:16 **PRI_14:** Priority of system handler 14, PendSV

Bits 15:0 Reserved, must be kept cleared

4.3.7 SCB usage hints and tips

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

4.3.8 SCB register map

The table provides shows the System control block register map and reset values. The base address of the SCB register block is 0xE000 ED00 for register described in *Table 32*.

Table 32. SCB register map and reset values

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	**CPUID**	Implementer								Variant				Constant				PartNo												Revision			
	Reset Value	0	1	0	0	0	0	0	1	0	0	0	1	1	1	1	1	1	1	0	0	0	0	1	0	0	0	1	1	0	0	0	1
0x04	**ICSR**	NMIPENDSET	Reserved		PENDSVSET	PENDSVCLR	PENDSTSET	PENDSTCLR	Reserved	ISRPENDING	Reserved			VECTPENDING[5:0]								Reserved						VECTACTIVE[5:0]					
	Reset Value	0			0	0	0	0		0				0	0	0	0	0	0	0	0							0	0	0	0	0	0
0x0C	**AIRCR**	VECTKEY[15:0]																ENDIANESS	Reserved												SYSRESETREQ	VECTCLRACTIVE	VECTRESET
	Reset Value	0	0	0	0	0	0	0	1	1	1	1	1	1	0	1	0	0													0	0	0
0x10	**SCR**	Reserved																										SEVONPEND	Reserved	SLEEPDEEP	SLEEPONEXIT	Reserved	
	Reset Value																											0		0	0		
0x14	**CCR**	Reserved																						STKALIGN	Reserved				UNALIGN_TRP	Reserved			
	Reset Value																							1					1				
0x1C	**SHPR2**	PRI11								Reserved																							
	Reset Value	0	0	0	0	0	0	0	0																								
0x20	**SHPR3**	PRI15								PRI14								Reserved															
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0																

4.4 SysTick timer (STK)

When enabled, the timer:

- counts down from the reload value to zero,
- reloads (wraps to) the value in the STK_RVR on the next clock cycle,
- then decrements on subsequent clock cycles.

Writing a value of zero to the STK_RVR disables the counter on the next wrap.

When the counter transitions to zero, the COUNTFLAG status bit is set to 1.

Reading STK_CSR clears the COUNTFLAG bit to 0.

Writing to the STK_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.

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

Table 33. System timer registers summary

Address	Name	Type	Reset value	Description
0xE000E010	STK_CSR	RW	0x00000000	*SysTick control and status register (STK_CSR) on page 86*
0xE000E014	STK_RVR	RW	Unknown	*SysTick reload value register (STK_RVR) on page 87*
0xE000E018	STK_CVR	RW	Unknown	*SysTick current value register (STK_CVR) on page 87*
0xE000E01C	STK_CALIB	RO	0xC0000000	*SysTick calibration value register (STK_CALIB) on page 88*

4.4.1 SysTick control and status register (STK_CSR)

Address offset: 0x00

Reset value: 0x0000 0004

The SysTick CSR register enables the SysTick features.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															COUNT FLAG
															rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved													CLKSO URCE	TICK INT	EN ABLE
													rw	rw	rw

Bits 31:17 Reserved, must be kept cleared.

Bit 16 **COUNTFLAG:**
Returns 1 if timer counted to 0 since last time this was read.

Bits 15:3 Reserved, must be kept cleared.

Bit 2 **CLKSOURCE:** Clock source selection
Selects the timer clock source.
0: External reference clock
1: Processor clock

Bit 1 **TICKINT**: SysTick exception request enable
0: Counting down to zero does not assert the SysTick exception request
1: Counting down to zero to asserts the SysTick exception request.

Bit 0 **ENABLE**: Counter enable
Enables the counter. When ENABLE is set to 1, the counter starts counting down. On reaching 0, it sets the COUNTFLAG to 1 and optionally asserts the SysTick depending on the value of TICKINT. It then loads the RELOAD value again, and begins counting.
0: Counter disabled
1: Counter enabled

4.4.2 SysTick reload value register (STK_RVR)

Address offset: 0x04

Reset value: Unknown

The STK_RVR specifies the start value to load into the STK_CVR.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								RELOAD[23:16]							
								rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RELOAD[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:24 Reserved, must be kept cleared.

Bits 23:0 **RELOAD:** RELOAD value

Value to load into the STK_CVR when the counter is enabled and when it reaches 0.

Calculating the RELOAD value

The RELOAD value can be any value in the range 0x00000001-0x00FFFFFF. A start value of 0 is possible, but 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 (STK_CVR)

Address offset: 0x08

Reset value: Unknown

The STK_CVR contains the current value of the SysTick counter.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								CURRENT[23:16]							
								rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CURRENT[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:24 Reserved, must be kept cleared.

Bits 23:0 **CURRENT:** Current counter value

Reads return the current value of the SysTick counter.

A write of any value clears the field to 0, and also clears the COUNTFLAG bit in the STK_CSR register to 0.

4.4.4 SysTick calibration value register (STK_CALIB)

Address offset: 0x0C

Reset value: 0x0000000

The CALIB register indicates the SysTick calibration properties. If calibration information is not known, calculate the calibration value required from the frequency of the processor clock or external clock.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
NO REF	SKEW	Reserved						TENMS[23:16]							
r	r							r	r	r	r	r	r	r	r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TENMS[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Bit 31 **NOREF:** NOREF flag. Reads as one. Indicates that no separate reference clock is provided.

Bit 30 **SKEW:** SKEW flag: Reads as one. Calibration value for the 10ms inexact timing is not known because TENMS is not known. This can affect the suitability of SysTick as a software real time clock.

Bits 29:24 Reserved, must be kept cleared.

Bits 23:0 **TENMS[23:0]:** Calibration value. Reads as zero. Indicates calibration value is not known

4.4.5 SysTick design hints and tips

The SysTick counter runs on the processor clock. If this clock signal is stopped for low power mode, the SysTick counter stops.

Ensure software uses aligned word accesses to access the SysTick registers.

The SysTick counter reload and current value are undefined at reset, the correct initialization sequence for the SysTick counter is:

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

4.4.6 SysTick register map

The table provided shows the SysTick register map and reset values. The base address of the SysTick register block is 0xE000 E010.

Table 34. SysTick register map and reset values

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	**STK_CSR**	Reserved															COUNTFLAG	Reserved													CLKSOURCE	TICK INT	EN ABLE
	Reset Value																0														1	0	0
0x04	**STK_RVR**	Reserved								RELOAD[23:0]																							
	Reset Value									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x08	**STK_CVR**	Reserved								CURRENT[23:0]																							
	Reset Value									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x0C	**STK_CALIB**	Reserved								TENMS[23:0]																							
	Reset Value									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

5 Revision history

Table 35. Document revision history

Date	Revision	Changes
27-April-2012	1	Initial release.