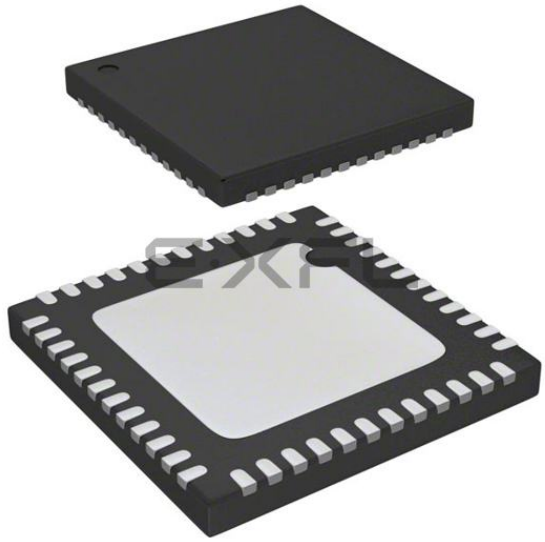


Welcome to E-XFL.COM

What is "Embedded - Microcontrollers"?

"Embedded - Microcontrollers" refer to small, integrated circuits designed to perform specific tasks within larger systems. These microcontrollers are essentially compact computers on a single chip, containing a processor core, memory, and programmable input/output peripherals. They are called "embedded" because they are embedded within electronic devices to control various functions, rather than serving as standalone computers. Microcontrollers are crucial in modern electronics, providing the intelligence and control needed for a wide range of applications.

Applications of "Embedded - Microcontrollers"

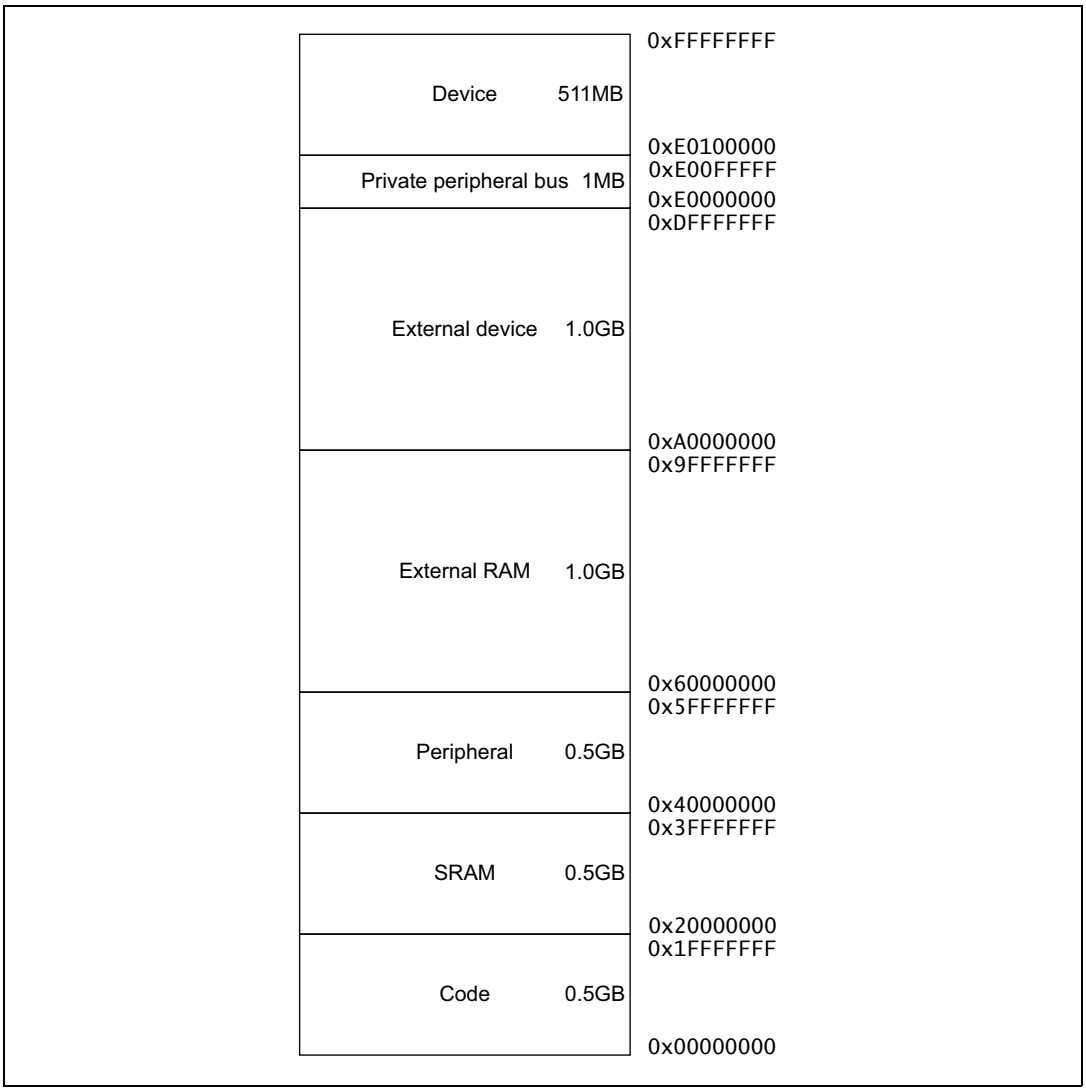
Details	
Product Status	Active
Core Processor	ARM® Cortex®-M0
Core Size	32-Bit Single-Core
Speed	48MHz
Connectivity	CANbus, HDMI-CEC, I²C, IrDA, LINbus, SPI, UART/USART, USB
Peripherals	DMA, I²S, POR, PWM, WDT
Number of I/O	38
Program Memory Size	32KB (32K x 8)
Program Memory Type	FLASH
EEPROM Size	-
RAM Size	6K x 8
Voltage - Supply (Vcc/Vdd)	1.65V ~ 1.95V
Data Converters	A/D 13x12b
Oscillator Type	Internal
Operating Temperature	-40°C ~ 85°C (TA)
Mounting Type	Surface Mount
Package / Case	48-UFQFN Exposed Pad
Supplier Device Package	48-UFQFPN (7x7)
Purchase URL	https://www.e-xfl.com/product-detail/stmicroelectronics/stm32f048c6u6

List of figures

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*.

2.3 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.

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
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*

3.3 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*).

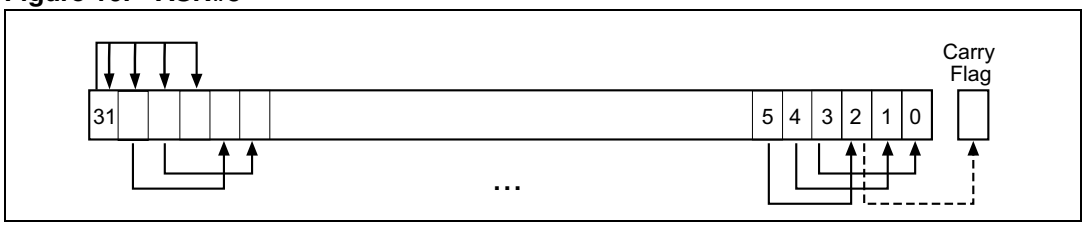
You can use the ASR operation to divide the signed value in the register *Rm* by 2^n, with the result being rounded towards negative-infinity.

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*).

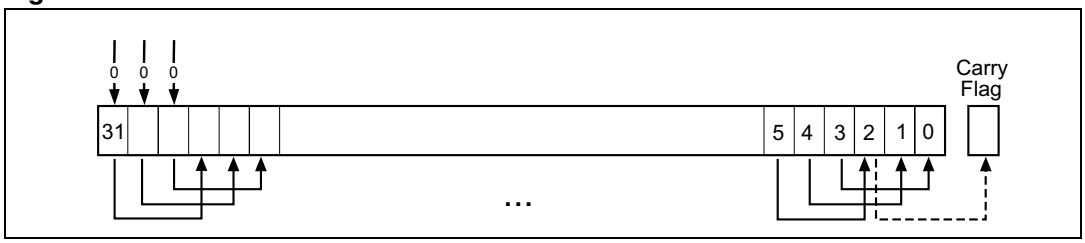
You can use the LSR #n operation to divide the value in the register *Rm* by 2^n, if the value is regarded as an unsigned integer.

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*).

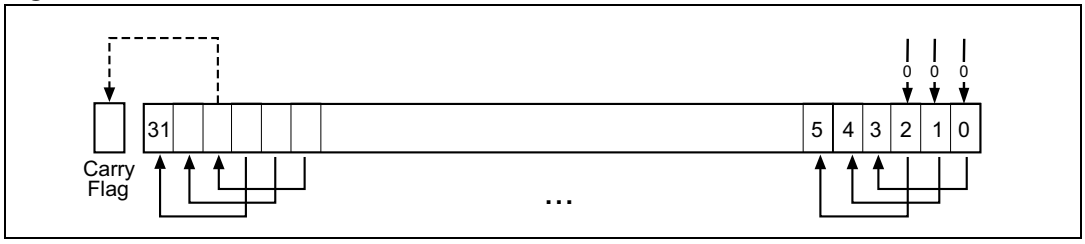
You can use the LSL #n operation to multiply the value in the register *Rm* by 2^n, if the value is regarded as an unsigned integer or a two's complement signed integer. Overflow can occur without warning.

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

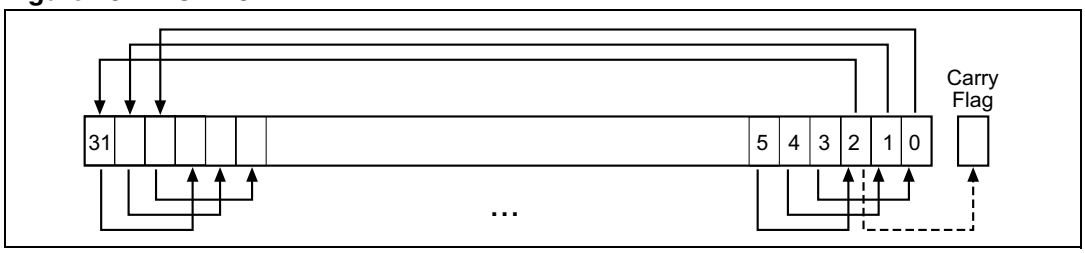
Rotate right by *n* bits moves the left-hand 32-*n* bits of the register *Rm*, to the right by *n* places, into the right-hand 32-*n* bits of the result. It also moves the right-hand *n* bits of the register into the left-hand *n* bits of the result (see *Figure 13*).

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



3.4 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*

3.4.4 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).
```

3.5 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*

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
```

3.5.5 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
```

3.7.2 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)
```

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.

4.2.2 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.

4.2.8 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.