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®-M3
Core Size	32-Bit Single-Core
Speed	32MHz
Connectivity	I²C, IrDA, LINbus, SPI, UART/USART, USB
Peripherals	Brown-out Detect/Reset, Cap Sense, DMA, I²S, POR, PWM, WDT
Number of I/O	83
Program Memory Size	384KB (384K x 8)
Program Memory Type	FLASH
EEPROM Size	12K x 8
RAM Size	48K x 8
Voltage - Supply (Vcc/Vdd)	1.8V ~ 3.6V
Data Converters	A/D 25x12b; D/A 2x12b
Oscillator Type	Internal
Operating Temperature	-40°C ~ 105°C (TA)
Mounting Type	Surface Mount
Package / Case	100-LQFP
Supplier Device Package	100-LQFP (14x14)
Purchase URL	https://www.e-xfl.com/product-detail/stmicroelectronics/stm32l151vdt7x

Execution program status register

The EPSR contains the Thumb state bit, and the execution state bits for either the:

- *If-Then* (IT) instruction
- *Interruptible-Continuable Instruction* (ICI) field for an interrupted load multiple or store multiple instruction.

See the register summary in *Table 2 on page 14* for the EPSR attributes. The bit assignments are:

Table 6. EPSR bit definitions

Bits	Description
Bits 31:27	Reserved.
Bits 26:25, 15:10	**ICI**: Interruptible-continuable instruction bits See *Interruptible-continuable instructions on page 19*.
Bits 26:25, 15:10	**IT**: Indicates the execution state bits of the IT instruction, see *IT on page 94*.
Bit 24	Always set to 1.
Bits 23:16	Reserved.
Bits 9:0]	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 2.3.7: Exception entry and return on page 38*

Interruptible-continuable instructions

When an interrupt occurs during the execution of an LDM or STM instruction, the processor:

- Stops the load multiple or store multiple instruction operation temporarily
- Stores the next register operand in the multiple operation to EPSR bits[15:12].

After servicing the interrupt, the processor:

- Returns to the register pointed to by bits[15:12]
- Resumes execution of the multiple load or store instruction.

When the EPSR holds ICI execution state, bits[26:25,11:10] are zero.

If-Then block

The If-Then block contains up to four instructions following a 16-bit IT instruction. Each instruction in the block is conditional. The conditions for the instructions are either all the same, or some can be the inverse of others. See *IT on page 94* for more information.

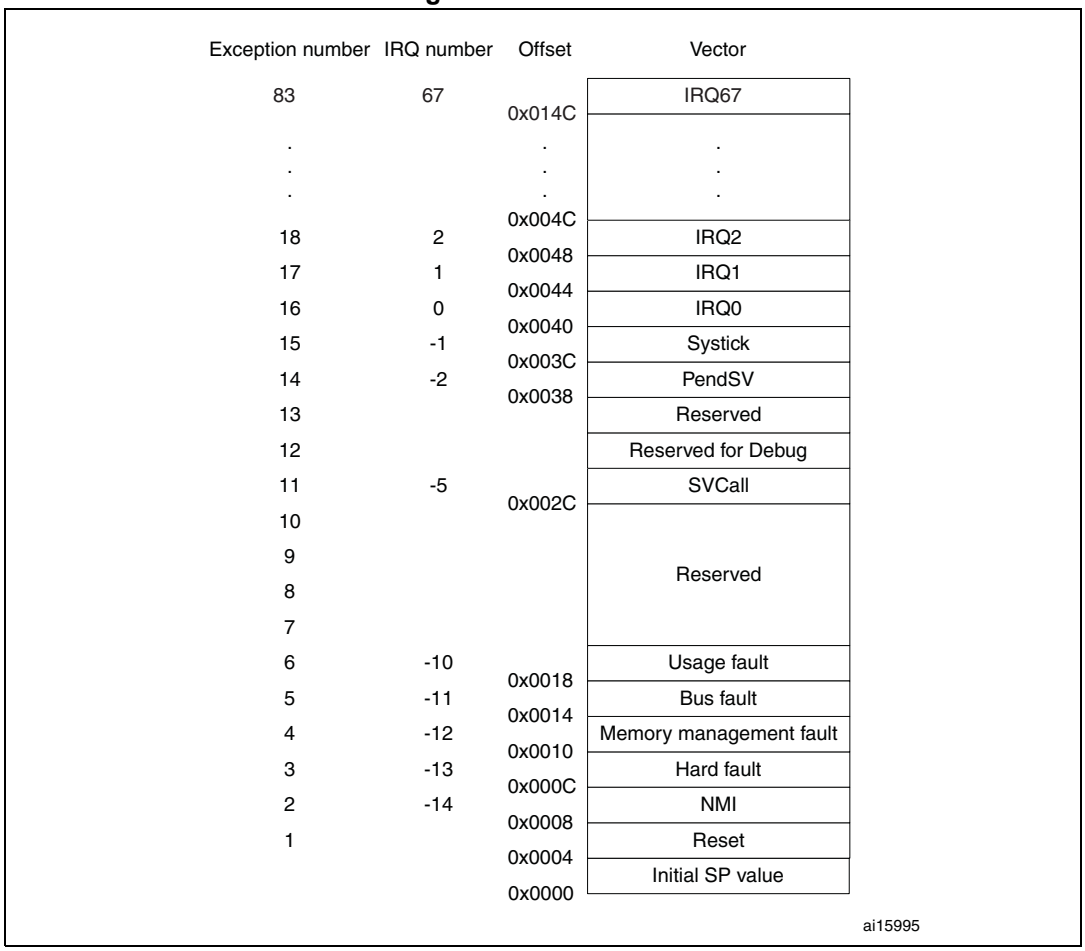
Exception mask registers

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

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 12 on page 36* 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 12. Vector table



On system reset, the vector table is fixed at address 0x00000000. Privileged software can write to the VTOR to relocate the vector table start address to a different memory location, in the range 0x00000080 to 0x3FFFFF80, see *Vector table offset register (SCB_VTOR) on page 133*.

2.3.5 Exception priorities

As *Table 16 on page 34* 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.

Table 18. Faults (continued)

Fault	Handler	Bit name	Fault status register
Bus error:	Bus fault	-	-
During exception stacking		STKERR	*Configurable fault status register (SCB_CFSR) on page 142*
During exception unstacking		UNSTKERR	
During instruction prefetch		IBUSERR	
Precise data bus error		PRECISERR	
Imprecise data bus error		IMPRECISERR	
Attempt to access a coprocessor	Usage fault	NOCP	*Configurable fault status register (SCB_CFSR) on page 142*
Undefined instruction		UNDEFINSTR	
Attempt to enter an invalid instruction set state(1)		INVSTATE	
Invalid EXC_RETURN value		INVPC	
Illegal unaligned load or store		UNALIGNED	
Divide By 0		DIVBYZERO	

1. Attempting to use an instruction set other than the Thumb instruction set.

2.4.2 Fault escalation and hard faults

All faults exceptions except for hard fault have configurable exception priority, see *System handler priority registers (SHPRx) on page 138*. Software can disable execution of the handlers for these faults, see *System handler control and state register (SCB_SHCSR) on page 140*.

Usually, the exception priority, together with the values of the exception mask registers, determines whether the processor enters the fault handler, and whether a fault handler can preempt another fault handler. as described in *Section 2.3: Exception model on page 33*.

In some situations, a fault with configurable priority is treated as a hard fault. This is called *priority escalation*, and the fault is described as *escalated to hard fault*. Escalation to hard fault occurs when:

- A fault handler causes the same kind of fault as the one it is servicing. This escalation to hard fault occurs because a fault handler cannot preempt itself because it must have the same priority as the current priority level.
- A fault handler causes a fault with the same or lower priority as the fault it is servicing. This is because the handler for the new fault cannot preempt the currently executing fault handler.
- An exception handler causes a fault for which the priority is the same as or lower than the currently executing exception.
- A fault occurs and the handler for that fault is not enabled.

If a bus fault occurs during a stack push when entering a bus fault handler, the bus fault does not escalate to a hard fault. This means that if a corrupted stack causes a fault, the fault handler executes even though the stack push for the handler failed. The fault handler operates but the stack contents are corrupted.

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

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. When the processor executes a WFI instruction it stops executing instructions and enters sleep mode. See *WFI on page 104* for more information.

Wait for event

The *wait for event* instruction, WFE, causes entry to sleep mode conditional on the value of an one-bit event register. When the processor executes a WFE instruction, it checks this register:

- If the register is 0 the processor stops executing instructions and enters sleep mode
- If the register is 1 the processor clears the register to 0 and continues executing instructions without entering sleep mode.

See *WFE on page 103* for more information.

If the event register is 1, this indicate 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 102*. 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 and the FAULTMASK bit to 0. 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 and FAULTMASK see *Exception mask registers on page 19*.

Wakeup from WFE

The processor wakes up if:

- it detects an exception with sufficient priority to cause exception entry
- it detects an external event signal, see *The external event input on page 44*

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 (SCB_SCR) on page 136*.

2.5.3 The external event input

The processor provides an external event input signal. This signal can be generated by the up to 16 external input lines, by the PVD, RTC alarm or by the USB wakeup event, configured through the external interrupt/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 43*. Fore more details please refer to the STM32 reference manual, section 4.3 Low power modes.

2.5.4 Power management programming hints

ANSI C cannot directly generate the WFI and WFE instructions. The CMSIS provides the following intrinsic functions for these instructions:

```
void __WFE(void) // Wait for Event

void __WFE(void) // Wait for Interrupt
```

The CMSIS also provides a number of functions for accessing the special registers using MRS and MSR instructions (see *Table 22*).

Table 22. 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)
FAULTMASK	Read	uint32_t __get_FAULTMASK (void)
	Write	void __set_FAULTMASK (uint32_t value)
BASEPRI	Read	uint32_t __get_BASEPRI (void)
	Write	void __set_BASEPRI (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:

- *Operands on page 51*
- *Restrictions when using PC or SP on page 52*
- *Flexible second operand on page 52*
- *Shift operations on page 53*
- *Address alignment on page 56*
- *PC-relative expressions on page 56*
- *Conditional execution on page 57*
- *Instruction width selection on page 59*.

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 *Flexible second operand*).

Most instructions update the status flags only if the S suffix is specified. 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 {*cond*}. Conditional execution requires a preceding IT instruction. An instruction with a condition code is only executed if the condition code flags in the APSR meet the specified condition. *Table 23* 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 23 also shows the relationship between condition code suffixes and the N, Z, C, and V flags.

Table 23. Condition code suffixes

Suffix	Flags	Meaning
EQ	Z = 1	Equal
NE	Z = 0	Not equal
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.

Specific example 1: Absolute value shows the use of a conditional instruction to find the absolute value of a number. R0 = ABS(R1).

Specific example 1: Absolute value

```
MOVSR0, R1; R0 = R1, setting flags
IT MI; IT instruction for the negative condition
RSBMIR0, R1, #0; If negative, R0 = -R1
```

Note: You might have to use the .W suffix to get the maximum offset range or to generate addresses that are not word-aligned (see Instruction width selection on page 59).

Restrictions

Rd must be neither SP nor 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
```

3.4.2 LDR and STR, immediate offset

Load and store with immediate offset, pre-indexed immediate offset, or post-indexed immediate offset.

Syntax

```
op{type}{cond} Rt, [Rn {, #offset}]; immediate offset
op{type}{cond} Rt, [Rn, #offset]!; pre-indexed
op{type}{cond} Rt, [Rn], #offset; post-indexed
opD{cond} Rt, Rt2, [Rn {, #offset}]; immediate offset, two words
opD{cond} Rt, Rt2, [Rn, #offset]!; pre-indexed, two words
opD{cond} Rt, Rt2, [Rn], #offset; post-indexed, two words
```

where:

- '*op*' is either LDR (load register) or STR (store register)
- '*type*' is one of the following:
  B: Unsigned byte, zero extends to 32 bits on loads
  SB: Signed byte, sign extends to 32 bits (LDR only)
  H: Unsigned halfword, zero extends to 32 bits on loads
  SH: Signed halfword, sign extends to 32 bits (LDR only)
  —: Omit, for word
- '*cond*' is an optional condition code (see Conditional execution on page 57)
- '*Rt*' is the register to load or store
- '*Rn*' is the register on which the memory address is based
- '*offset*' is an offset from *Rn*. If *offset* is omitted, the address is the contents of *Rn*
- '*Rt2*' is the additional register to load or store for two-word operations

Restrictions

You can use SP and PC only in the MOV instruction, with the following restrictions:

- The second operand must be a register without shift
- You must not specify the S suffix

When *Rd* is PC in a MOV instruction:

- bit[0] of the value written to the PC is ignored
- A branch occurs to the address created by forcing bit[0] of that value to 0.

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
- Can update the C flag during the calculation of *operand2* (see *Flexible second operand on page 52*).
- Do not affect the V flag

Example

```
MOVSR11, #0x000B; write value of 0x000B to R11, flags get updated
MOVR1, #0xFA05; write value of 0xFA05 to R1, flags are not updated
MOVSR10, R12; write value in R12 to R10, flags get updated
MOVR3, #23; write value of 23 to R3
MOVR8, SP; write value of stack pointer to R8
MVNSR2, #0xF; write value of 0xFFFFFFF0 (bitwise inverse of 0xF)
    ; to the R2 and update flags
```

3.5.7 MOVT

Move top.

Syntax

```
MOVT{cond} Rd, #imm16
```

where:

- '*cond*' is an optional condition code (see *Conditional execution on page 57*)
- '*Rd*' is the destination register
- '*imm16*' is a 16-bit immediate constant

Operation

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

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

```
REVSH R0, R5 ; reverse Signed Halfword
REVHS R3, R7 ; reverse with Higher or Same condition
RBIT R7, R8 ; reverse bit order of value in R8 and write the result to R7
```

3.5.9 TST and TEQ

Test bits and test equivalence.

Syntax

```
TST{cond} Rn, Operand2
```

```
TEQ{cond} Rn, Operand2
```

where:

- '*cond*' is an optional condition code (see *Conditional execution on page 57*)
- '*Rn*' is the register holding the first operand
- '*Operand2*' is a flexible second operand (see *Flexible second operand on page 52*) for details of the options.

Operation

These instructions test the value in a register against *operand2*. They update the condition flags based on the result, but do not write the result to a register.

The TST instruction performs a bitwise AND operation on the value in *Rn* and the value of *operand2*. 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 an *operand2* constant that has that bit set to 1 and all other bits cleared to 0.

The TEQ instruction performs a bitwise exclusive OR operation on the value in *Rn* and the value of *operand2*. This is the same as the EORS instruction, except that it discards the result.

Use the TEQ instruction to test if two values are equal without affecting the V or C flags.

TEQ is also useful for testing the sign of a value. After the comparison, the N flag is the logical exclusive OR of the sign bits of the two operands.

Restrictions

Do not use either SP or PC.

Condition flags

These instructions:

- Update the N and Z flags according to the result
- Can update the C flag during the calculation of *operand2* (see *Flexible second operand on page 52*).
- Do not affect the V flag

Examples

```
TSTR0, #0x3F8; perform bitwise AND of R0 value to 0x3F8,
    ; APSR is updated but result is discarded
TEQEQR10, R9; conditionally test if value in R10 is equal to
    ; value in R9, APSR is updated but result is discarded
```

Operation

These instructions do the following:

1. Rotate the value from *Rm* right by 0, 8, 16 or 24 bits.
2. Extract bits from the resulting value:
   - SXTB extracts bits[7:0] and sign extends to 32 bits.
   - UXTB extracts bits[7:0] and zero extends to 32 bits.
   - SXTH extracts bits[15:0] and sign extends to 32 bits.
   - UXTH extracts bits[15:0] and zero extends to 32 bits.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions do not affect the flags.

Examples

```
SXTH  R4, R6, ROR #16  ; Rotate R6 right by 16 bits, then obtain the lower
                       ; halfword of the result and then sign extend to
                       ; 32 bits and write the result to R4.
UXTB  R3, R10          ; Extract lowest byte of the value in R10 and zero
                       ; extend it, and write the result to R3
```

3.8.4 Branch and control instructions

Table 30 shows the branch and control instructions:

Table 30. Branch and control instructions

Mnemonic	Brief description	See
B	Branch	*B, BL, BX, and BLX on page 92*
BL	Branch with Link	*B, BL, BX, and BLX on page 92*
BLX	Branch indirect with Link	*B, BL, BX, and BLX on page 92*
BX	Branch indirect	*B, BL, BX, and BLX on page 92*
CBNZ	Compare and Branch if Non Zero	*CBZ and CBNZ on page 93*
CBZ	Compare and Branch if Non Zero	*CBZ and CBNZ on page 93*
IT	If-Then	*IT on page 94*
TBB	Table Branch Byte	*TBB and TBH on page 96*
TBH	Table Branch Halfword	*TBB and TBH on page 96*

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

Restrictions

The restrictions are:

- *Rn* must be in the range of R0 to R7
- The branch destination must be within 4 to 130 bytes after the instruction
- These instructions must not be used inside an IT block.

Condition flags

These instructions do not change the flags.

Examples

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

3.8.7 IT

If-Then condition instruction.

Syntax

`IT{x{y{z}}} cond`

where:

- *'x'* specifies the condition switch for the second instruction in the IT block.
- *'y'* specifies the condition switch for the third instruction in the IT block.
- *'z'* specifies the condition switch for the fourth instruction in the IT block.
- *'cond'* specifies the condition for the first instruction in the IT block.

The condition switch for the second, third and fourth instruction in the IT block can be either:

T: Then. Applies the condition *cond* to the instruction.

E: Else. Applies the inverse condition of *cond* to the instruction.

a) It is possible to use AL (the *always* condition) for *cond* in an IT instruction. If this is done, all of the instructions in the IT block must be unconditional, and each of *x*, *y*, and *z* must be T or omitted but not E.

Operation

The IT instruction makes up to four following instructions conditional. The conditions can be all the same, or some of them can be the logical inverse of the others. The conditional instructions following the IT instruction form the *IT block*.

The instructions in the IT block, including any branches, must specify the condition in the *{cond}* part of their syntax.

where:

- *'cond'* is an optional condition code, see *Conditional execution on page 57*.
- *'Rd'* is the destination register.
- *'spec_reg'* can be any of: APSR, IPSR, EPSR, IEPSR, IAPSR, EAPSR, PSR, MSP, PSP, PRIMASK, BASEPRI, BASEPRI_MAX, FAULTMASK, or CONTROL.

Operation

Use MRS in combination with MSR as part of a read-modify-write sequence for updating a PSR, for example to clear the Q flag.

In process swap code, the programmers model state of the process being swapped out must be saved, including relevant PSR contents. Similarly, the state of the process being swapped in must also be restored. These operations use MRS in the state-saving instruction sequence and MSR in the state-restoring instruction sequence.

BASEPRI_MAX is an alias of BASEPRI when used with the MRS instruction.

See *MSR on page 101*.

Restrictions

Rd must not be SP and must not be PC.

Condition flags

This instruction does not change the flags.

Examples

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

3.9.7 MSR

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

Syntax

`MSR{cond} spec_reg, Rn`

where:

- *'cond'* is an optional condition code, see *Conditional execution on page 57*.
- *'Rn'* is the source register.
- *'spec_reg'* can be any of: APSR, IPSR, EPSR, IEPSR, IAPSR, EAPSR, PSR, MSP, PSP, PRIMASK, BASEPRI, BASEPRI_MAX, FAULTMASK, or CONTROL.

Operation

The register access operation in MSR depends on the privilege level. Unprivileged software can only access the APSR, see *Table 4: APSR bit definitions on page 17*. Privileged software can access all special registers.

In unprivileged software writes to unallocated or execution state bits in the PSR are ignored.

4.2.9 MPU region attribute and size register (MPU_RASR)

Address offset: 0x10

Reset value: 0x0000 0000

Required privilege: Privileged

The MPU_RASR register defines the region size and memory attributes of the MPU region specified by the MPU_RNR, and enables that region and any subregions.

MPU_RASR is accessible using word or halfword accesses:

- The most significant halfword holds the region attributes
- The least significant halfword holds the region size and the region and subregion enable bits.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved			XN	Res.	AP[2:0]			Reserved		TEX[2:0]			S	C	B
			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
SRD[7:0]								Reserved		SIZE					ENABLE
rw	rw	rw	rw	rw	rw	rw	rw			rw	rw	rw	rw	rw	rw

Bits 31:29 **Reserved, forced by hardware to 0.**

Bit 28 **XN**: Instruction access disable bit:
0: Instruction fetches enabled
1: Instruction fetches disabled.

Bit 27 **Reserved, forced by hardware to 0.**

Bits 26:24 **AP[2:0]**: Access permission
For information about access permission, see *Section 4: Core peripherals*
For the description of the encoding of the AP bits refer to *Table 37 on page 108*.

Bits 23:22 **Reserved, forced by hardware to 0.**

Bits 21:19 **TEX[2:0]: memory attribute**
For the description of the encoding of the TEX bits refer to *Table 35 on page 107*

Bit 18 **S: Shareable memory attribute**
For the description of the encoding of the S bits refer to *Table 35 on page 107*

Bit 17 **C: memory attribute**

Bit 16 **B: memory attribute**

Bits 15:8 **SRD**: Subregion disable bits.
For each bit in this field:
0: corresponding sub-region is enabled
1: corresponding sub-region is disabled
See *Subregions on page 110* for more information.
Region sizes of 128 bytes and less do not support subregions. When writing the attributes for such a region, write the SRD field as 0x00.

Bits 7:6 **Reserved, forced by hardware to 0.**

Bits 5:1 **SIZE**: Size of the MPU protection region.
The minimum permitted value is 3 (b00010), see *SIZE field values* for more information.

Bit 0 **ENABLE**: Region enable bit.

SIZE field values

The SIZE field defines the size of the MPU memory region specified by the MPU_RNR register as follows:

(Region size in bytes) = 2(SIZE+1)

The smallest permitted region size is 32B, corresponding to a SIZE value of 4. Table 4-45 gives example SIZE values, with the corresponding region size and value of N in the RBAR.

Table 39. Example SIZE field values

SIZE value	Region size	Value of N(1)	Note
b00100 (4)	32B	5	Minimum permitted size
b01001 (9)	1KB	10	-
b10011 (19)	1MB	20	-
b11101 (29)	1GB	30	-
b11111 (31)	4GB	b01100	Maximum possible size

1. In the MPU_RBAR register see *Section 4.2.8 on page 114*

Table 40. MPU 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	MPU_TYPER	Reserved								IREGION[7:0]								DREGION[7:0]							Reserved								SEPARATE
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0x04	MPU_CR	Reserved																													PRIVDEFENA	HFNMIENA	ENABLE
	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
0x08	MPU_RNR	Reserved																													REGION[7:0]		
	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
0x0C	MPU_RBAR	ADDR[31:N]...																											VALID	REGION [3:0]			
	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

4.3.8 Software trigger interrupt register (NVIC_STIR)

Address offset: 0xE00

Reset value: 0x0000 0000

Required privilege: When the USERSETMPEND bit in the SCR is set to 1, unprivileged software can access the STIR, see *Section 4.4.6: System control register (SCB_SCR)*. Only privileged software can enable unprivileged access to the STIR.

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							INTID[8:0]								
							w	w	w	w	w	w	w	w	w

Bits 31:9 Reserved, must be kept cleared.

Bits 8:0 **NTID[8:0]** Software generated interrupt ID

Write to the STIR to generate a Software Generated Interrupt (SGI). The value to be written is the Interrupt ID of the required SGI, in the range 0-239. For example, a value of 0b000000011 specifies interrupt IRQ3.

4.3.9 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-M3 latches all interrupts. A peripheral interrupt becomes pending for one of the following reasons:

- The NVIC detects that the interrupt signal is HIGH 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 4.3.4: Interrupt set-pending registers (NVIC_ISPRx)*, or to the STIR to make an SGI pending, see *Section 4.3.8: Software trigger interrupt register (NVIC_STIR)*.

Bit 3 UNALIGN_ TRP

Enables unaligned access traps:

0: Do not trap unaligned halfword and word accesses

1: Trap unaligned halfword and word accesses.

If this bit is set to 1, an unaligned access generates a usage fault.

Unaligned LDM, STM, LDRD, and STRD instructions always fault irrespective of whether UNALIGN_TRP is set to 1.

Bit 2 Reserved, must be kept cleared

Bit 1 USERSETMPEND

Enables unprivileged software access to the STIR, see *Software trigger interrupt register (NVIC_STIR) on page 126*:

0: Disable

1: Enable.

Bit 0 NONBASETHRDENA

Configures how the processor enters Thread mode.

0: Processor can enter Thread mode only when no exception is active.

1: Processor can enter Thread mode from any level under the control of an EXC_RETURN value, see *Exception return on page 39*.

4.4.8 System handler priority registers (SHPRx)

The SHPR1-SHPR3 registers set the priority level, 0 to 15 of the exception handlers that have configurable priority.

SHPR1-SHPR3 are byte accessible.

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

Table 46. System fault handler priority fields

Handler	Field	Register description
Memory management fault	PRI_4	*System handler priority register 1 (SCB_SHPR1)*
Bus fault	PRI_5	
Usage fault	PRI_6	
SVCall	PRI_11	*System handler priority register 2 (SCB_SHPR2) on page 139*
PendSV	PRI_14	*System handler priority register 3 (SCB_SHPR3) on page 140*
SysTick	PRI_15	

Each PRI_N field is 8 bits wide, but the processor implements only bits[7:4] of each field, and bits[3:0] read as zero and ignore writes.

Bit 19 **NOCP:** No coprocessor usage fault. The processor does not support coprocessor instructions:
0: No usage fault caused by attempting to access a coprocessor
1: the processor has attempted to access a coprocessor.

Bit 18 **INVPC:** Invalid PC load usage fault, caused by an invalid PC load by EXC_RETURN:
When this bit is set to 1, the PC value stacked for the exception return points to the instruction that tried to perform the illegal load of the PC.
0: No invalid PC load usage fault
1: The processor has attempted an illegal load of EXC_RETURN to the PC, as a result of an invalid context, or an invalid EXC_RETURN value.

Bit 17 **INVSTATE:** Invalid state usage fault:
When this bit is set to 1, the PC value stacked for the exception return points to the instruction that attempted the illegal use of the EPSR.
This bit is not set to 1 if an undefined instruction uses the EPSR.
0: No invalid state usage fault
1: The processor has attempted to execute an instruction that makes illegal use of the EPSR.

Bit 16 **UNDEFINSTR:** Undefined instruction usage fault:
When this bit is set to 1, the PC value stacked for the exception return points to the undefined instruction.
An undefined instruction is an instruction that the processor cannot decode.
0: No undefined instruction usage fault
1: The processor has attempted to execute an undefined instruction.

Bit 15 **BFARVALID:** *Bus Fault Address Register* (BFAR) valid flag:
The processor sets this bit to 1 after a bus fault where the address is known. Other faults can set this bit to 0, such as a memory management fault occurring later.
If a bus fault occurs and is escalated to a hard fault because of priority, the hard fault handler must set this bit to 0. This prevents problems if returning to a stacked active bus fault handler whose BFAR value has been overwritten.
0: Value in BFAR is not a valid fault address
1: BFAR holds a valid fault address.

Bits 14:13 Reserved, must be kept cleared

Bit 12 **STKERR:** Bus fault on stacking for exception entry
When the processor sets this bit to 1, the SP is still adjusted but the values in the context area on the stack might be incorrect. The processor does not write a fault address to the BFAR.
0: No stacking fault
1: Stacking for an exception entry has caused one or more bus faults.

Bit 11 **UNSTKERR:** Bus fault on unstacking for a return from exception
This fault is chained to the handler. This means that when the processor sets this bit to 1, the original return stack is still present. The processor does not adjust the SP from the failing return, does not performed a new save, and does not write a fault address to the BFAR.
0: No unstacking fault
1: Unstack for an exception return has caused one or more bus faults.

4.4.12 Memory management fault address register (SCB_MMFAR)

Address offset: 0x34

Reset value: undefined

Required privilege: Privileged

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MMFAR[31:16]															
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
MMFAR[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:0 **MMFAR[31:0]**: Memory management fault address

When the MMARVALID bit of the MMFSR is set to 1, this field holds the address of the location that generated the memory management fault.

When an unaligned access faults, the address is the actual address that faulted. Because a single read or write instruction can be split into multiple aligned accesses, the fault address can be any address in the range of the requested access size.

Flags in the MMFSR register indicate the cause of the fault, and whether the value in the MMFAR is valid. See *Configurable fault status register (SCB_CFSR) on page 142*.

4.4.13 Bus fault address register (SCB_BFAR)

Address offset: 0x38

Reset value: undefined

Required privilege: Privileged

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BFAR[31:16]															
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
BFAR[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:0 **BFAR[31:0]**: Bus fault address

When the BFARVALID bit of the BFSR is set to 1, this field holds the address of the location that generated the bus fault.

When an unaligned access faults the address in the BFAR is the one requested by the instruction, even if it is not the address of the fault.

Flags in the BFSR register indicate the cause of the fault, and whether the value in the BFAR is valid. See *Configurable fault status register (SCB_CFSR) on page 142*.

4.5.3 SysTick current value register (STK_VAL)

Address offset: 0x08

Reset value: 0x0000 0000

Required privilege: Privileged

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[23:0]:** Current counter value

The VAL register contains the current value of the SysTick counter.

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_CTRL register to 0.

4.5.4 SysTick calibration value register (STK_CALIB)

Address offset: 0x0C

Reset value: 0x0002328

Required privilege: Privileged

The CALIB register indicates the SysTick calibration properties.

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 zero. Indicates that a separate reference clock is provided. The frequency of this clock is HCLK/8.