**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, LCD, 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 ~ 85°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/stm32l152vdt6tr |

### Interrupt program status register

The IPSR contains the exception type number of the current *Interrupt Service Routine* (ISR). See the register summary in *Table 2 on page 14* for its attributes. The bit assignments are:

**Table 5. IPSR bit definitions**

| Bits | Description |
|---|---|
| Bits 31:9 | Reserved |
| Bits 8:0 | **ISR_NUMBER**:<br>This is the number of the current exception:<br>0: Thread mode<br>1: Reserved<br>2: NMI<br>3: Hard fault<br>4: Memory management fault<br>5: Bus fault<br>6: Usage fault<br>7: Reserved<br>....<br>10: Reserved<br>11: SVCall<br>12: Reserved for Debug<br>13: Reserved<br>14: PendSV<br>15: SysTick<br>16: IRQ0(1)<br>....<br>....<br>83: IRQ67(1)<br>see *Exception types on page 33* for more information. |

1. See STM32 product reference manual/datasheet for more information on interrupt mapping

A word access to the SRAM or peripheral bit-band alias regions map to a single bit in the SRAM or peripheral bit-band region.

The following formula shows how the alias region maps onto the bit-band region:

```
bit_word_offset = (byte_offset x 32) + (bit_number x 4)
```

```
bit_word_addr = bit_band_base + bit_word_offset
```

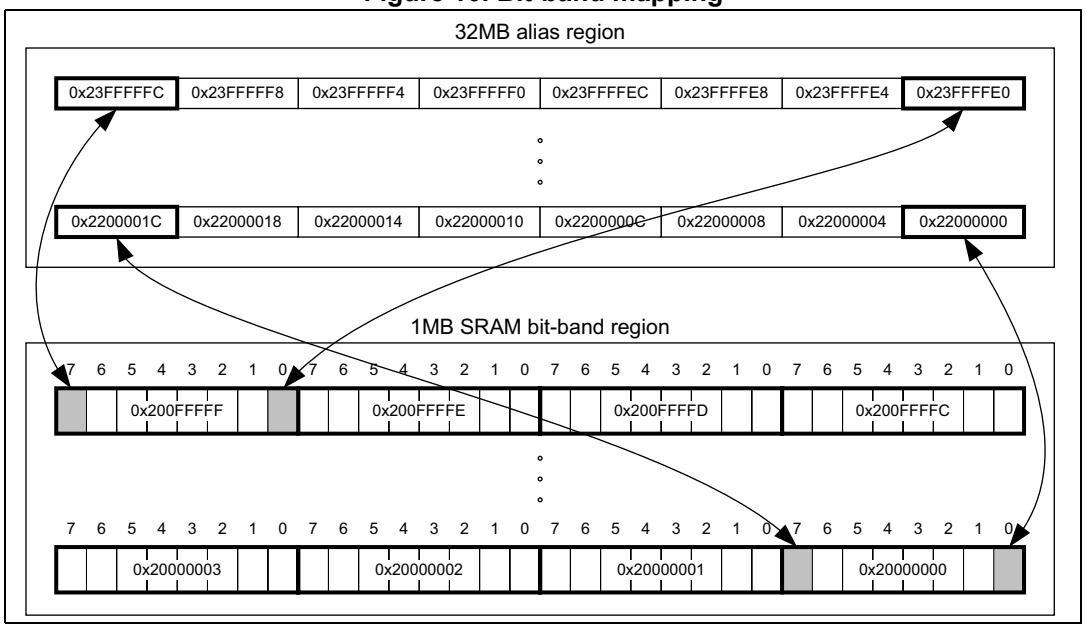
Where:

- Bit_word_offset is the position of the target bit in the bit-band memory region.
- Bit_word_addr is the address of the word in the alias memory region that maps to the targeted bit.
- Bit_band_base is the starting address of the alias region.
- Byte_offset is the number of the byte in the bit-band region that contains the targeted bit.
- Bit_number is the bit position, 0-7, of the targeted bit.

*Figure 10 on page 29* shows examples of bit-band mapping between the SRAM bit-band alias region and the SRAM bit-band region:

- The alias word at 0x23FFFFE0 maps to bit[0] of the bit-band byte at 0x200FFFFF: 0x23FFFFE0 = 0x22000000 + (0xFFFFF*32) + (0*4).
- The alias word at 0x23FFFFFC maps to bit[7] of the bit-band byte at 0x200FFFFF: 0x23FFFFFC = 0x22000000 + (0xFFFFF*32) + (7*4).
- The alias word at 0x22000000 maps to bit[0] of the bit-band byte at 0x20000000: 0x22000000 = 0x22000000 + (0*32) + (0 *4).
- The alias word at 0x2200001C maps to bit[7] of the bit-band byte at 0x20000000: 0x2200001C = 0x22000000+ (0*32) + (7*4).

**Figure 10. Bit-band mapping**



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

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

The Cortex-M3 includes an exclusive access monitor, that tags the fact that the processor has executed a Load-Exclusive instruction.

The processor removes its exclusive access tag if:

- It executes a CLREX instruction
- It executes a Store-Exclusive instruction, regardless of whether the write succeeds.
- An exception occurs. This means the processor can resolve semaphore conflicts between different threads.

For more information about the synchronization primitive instructions, see *LDREX and STREX on page 70* and *CLREX on page 71*.

### 2.2.8 Programming hints for the synchronization primitives

ANSI C cannot directly generate the exclusive access instructions. Some C compilers provide intrinsic functions for generation of these instructions:

**Table 15. C compiler intrinsic functions for exclusive access instructions**

| Instruction | Intrinsic function |
|---|---|
| LDREX, LDREXH, or LDREXB | `unsigned int __ldrex(volatile void *ptr)` |
| STREX, STREXH, or STREXB | `int __strex(unsigned int val, volatile void *ptr)` |
| CLREX | `void __clrex(void)` |

The actual exclusive access instruction generated depends on the data type of the pointer passed to the intrinsic function. For example, the following C code generates the require LDREXB operation:

```
__ldrex((volatile char *) 0xFF);
```

## 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.<br>*Note: An exception handler can interrupt the execution of another exception handler. In this case both exceptions are in the active state.* |
| Active and pending | The exception is being serviced by the processor and there is a pending exception from the same source. |

### 2.3.2 Exception types

The exception types are:

| | |
|---|---|
| Reset | Reset is invoked on power up or a warm reset. The exception model treats reset as a special form of exception. When reset is asserted, the operation of the processor stops, potentially at any point in an instruction. When reset is deasserted, execution restarts from the address provided by the reset entry in the vector table. Execution restarts as privileged execution in Thread mode. |
| 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:<br>• Masked or prevented from activation by any other exception<br>• Preempted by any exception other than Reset. |
| Hard fault | A hard fault is an exception that occurs because of an error during exception processing, or because an exception cannot be managed by any other exception mechanism. Hard faults have a fixed priority of -1, meaning they have higher priority than any exception with configurable priority. |
| Memory management fault | A memory management fault is an exception that occurs because of a memory protection related fault. The fixed memory protection constraints determines this fault, for both instruction and data memory transactions. This fault is used to abort instruction accesses to *Execute Never* (XN) memory regions. |

### 2.3.7 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. See *Section 2.3.6: Interrupt priority grouping* for more information about preemption by an interrupt.

When one exception preempts another, the exceptions are called nested exceptions. See *Exception entry on page 38* 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 39* 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. The processor can accept a late arriving exception until the first instruction of the exception handler of the original exception enters the execute stage of the processor. 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 19*. 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

### Condition flags

These instructions do not change the flags.

### Examples

```
LDRR8, [R10]; loads R8 from the address in R10.
LDRNER2, [R5, #960]!; loads (conditionally) R2 from a word
    ; 960 bytes above the address in R5, and
    ; increments R5 by 960.
STRR2, [R9,#const-struc]; const-struc is an expression evaluating
    ; to a constant in the range 0-4095.
STRHR3, [R4], #4; Store R3 as halfword data into address in
    ; R4, then increment R4 by 4
LDRD R8, R9, [R3, #0x20]; Load R8 from a word 32 bytes above the
    ; address in R3, and load R9 from a word 36
    ; bytes above the address in R3
STRDR0, R1, [R8], #-16; Store R0 to address in R8, and store R1 to
    ; a word 4 bytes above the address in R8,
    ; and then decrement R8 by 16.
```

## 3.4.3 LDR and STR, register offset

Load and store with register offset.

### Syntax

```
op{type}{cond} Rt, [Rn, Rm {, LSL #n}]
```

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
- '*Rm*' is a register containing a value to be used as the offset
- '*LSL #n*' is an optional shift, with *n* in the range 0 to 3

### Operation

LDR instructions load a register with a value from memory. STR instructions store a register value into memory.

The memory address to load from or store to is at an offset from the register *Rn*. The offset is specified by the register *Rm* and can be shifted left by up to 3 bits using LSL.

The value to load or store can be a byte, halfword, or word. For load instructions, bytes and halfwords can either be signed or unsigned (see *Address alignment on page 56*).

## 3.5 General data processing instructions

*Table 27* shows the data processing instructions.

**Table 27. Data processing instructions**

| Mnemonic | Brief description | See |
|---|---|---|
| ADC | Add with carry | *ADD, ADC, SUB, SBC, and RSB on page 73* |
| ADD | Add | *ADD, ADC, SUB, SBC, and RSB on page 73* |
| ADDW | Add | *ADD, ADC, SUB, SBC, and RSB on page 73* |
| AND | Logical AND | *AND, ORR, EOR, BIC, and ORN on page 75* |
| ASR | Arithmetic shift right | *ASR, LSL, LSR, ROR, and RRX on page 76* |
| BIC | Bit clear | *AND, ORR, EOR, BIC, and ORN on page 75* |
| CLZ | Count leading zeros | *CLZ on page 77* |
| CMN | Compare negative | *CMP and CMN on page 78* |
| CMP | Compare | *CMP and CMN on page 78* |
| EOR | Exclusive OR | *AND, ORR, EOR, BIC, and ORN on page 75* |
| LSL | Logical shift left | *ASR, LSL, LSR, ROR, and RRX on page 76* |
| LSR | Logical shift right | *ASR, LSL, LSR, ROR, and RRX on page 76* |
| MOV | Move | *MOV and MVN on page 79* |
| MOVT | Move top | *MOVT on page 80* |
| MOVW | Move 16-bit constant | *MOV and MVN on page 79* |
| MVN | Move NOT | *MOV and MVN on page 79* |
| ORN | Logical OR NOT | *AND, ORR, EOR, BIC, and ORN on page 75* |
| ORR | Logical OR | *AND, ORR, EOR, BIC, and ORN on page 75* |
| RBIT | Reverse bits | *REV, REV16, REVSH, and RBIT on page 81* |
| REV | Reverse byte order in a word | *REV, REV16, REVSH, and RBIT on page 81* |
| REV16 | Reverse byte order in each halfword | *REV, REV16, REVSH, and RBIT on page 81* |
| REVSH | Reverse byte order in bottom halfword and sign extend | *REV, REV16, REVSH, and RBIT on page 81* |
| ROR | Rotate right | *ASR, LSL, LSR, ROR, and RRX on page 76* |
| RRX | Rotate right with extend | *ASR, LSL, LSR, ROR, and RRX on page 76* |
| RSB | Reverse subtract | *ADD, ADC, SUB, SBC, and RSB on page 73* |
| SBC | Subtract with carry | *ADD, ADC, SUB, SBC, and RSB on page 73* |
| SUB | Subtract | *ADD, ADC, SUB, SBC, and RSB on page 73* |
| SUBW | Subtract | *ADD, ADC, SUB, SBC, and RSB on page 73* |
| TEQ | Test equivalence | *TST and TEQ on page 82* |
| TST | Test | *TST and TEQ on page 82* |

### Examples

```
CLZR4,R9
CLZNER2,R3
```

## 3.5.5 CMP and CMN

Compare and compare negative.

### Syntax

```
CMP{cond} Rn, Operand2
```

```
CMN{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 compare the value in a register with *operand2*. They update the condition flags on the result, but do not write the result to a register.

The CMP instruction subtracts the value of *operand2* from the value in *Rn*. This is the same as a SUBS instruction, except that the result is discarded.

The CMN instruction adds the value of *operand2* to the value in *Rn*. This is the same as an ADDS instruction, except that the result is discarded.

### Restrictions

In these instructions:

- Do not use PC
- *Operand2* must not be SP

### Condition flags

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

### Examples

```
CMPR2, R9
CMNR0, #6400
CMPGTSP, R7, LSL #2
```

### 3.8.1 BFC and BFI

Bit Field Clear and Bit Field Insert.

#### Syntax

```
BFC{cond} Rd, #lsb, #width
BFI{cond} Rd, Rn, #lsb, #width
```

where:

- '*cond*' is an optional condition code, see *Conditional execution on page 57*.
- '*Rd*' is the destination register.
- '*Rn*' is the source register.
- *'lsb'* is the position of the least significant bit of the bitfield. *lsb* must be in the range 0 to 31.
- *'width'* is the width of the bitfield and must be in the range 1 to 32-*lsb*.

#### Operation

BFC clears a bitfield in a register. It clears width bits in *Rd*, starting at the low bit position *lsb*. Other bits in *Rd* are unchanged.

BFI copies a bitfield into one register from another register. It replaces width bits in *Rd* starting at the low bit position *lsb*, with width bits from *Rn* starting at bit[0]. Other bits in *Rd* are unchanged.

#### Restrictions

Do not use SP and do not use PC.

#### Condition flags

These instructions do not affect the flags.

#### Examples

```
BFC   R4, #8, #12       ; Clear bit 8 to bit 19 (12 bits) of R4 to 0
BFI   R9, R2, #8, #12   ; Replace bit 8 to bit 19 (12 bits) of R9 with
                        ; bit 0 to bit 11 from R2
```

### 3.8.2 SBFX and UBFX

Signed Bit Field Extract and Unsigned Bit Field Extract.

#### Syntax

```
SBFX{cond} Rd, Rn, #lsb, #width
UBFX{cond} Rd, Rn, #lsb, #width
```

### 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* |

### 3.8.5 B, BL, BX, and BLX

Branch instructions.

#### Syntax

`B{cond} label`

`BL{cond} label`

`BX{cond} Rm`

`BLX{cond} Rm`

where:

- 'B' is branch (immediate).
- 'BL' is branch with link (immediate).
- 'BX' is branch indirect (register).
- 'BLX' is branch indirect with link (register).
- *'cond'* is an optional condition code, see *Conditional execution on page 57*.
- *'label'* is a PC-relative expression. See *PC-relative expressions on page 56*.
- *'Rm'* is a register that indicates an address to branch to. Bit[0] of the value in *Rm* must be 1, but the address to branch to is created by changing bit[0] to 0.

#### 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 UsageFault exception if bit[0] of *Rm* is 0.

B *cond label* is the only conditional instruction that can be either inside or outside an IT block. All other branch instructions must be conditional inside an IT block, and must be unconditional outside the IT block, see *IT on page 94*.

*Table 31* shows the ranges for the various branch instructions.

**Table 31. Branch ranges**

| Instruction | Branch range |
|---|---|
| B label | –16 MB to +16 MB |
| B*cond* label (outside IT block) | –1 MB to +1 MB |
| B*cond* label (inside IT block) | –16 MB to +16 MB |
| BL{*cond*} label | –16 MB to +16 MB |
| BX{*cond*} Rm | Any value in register |
| BLX{*cond*} Rm | Any value in register |

You might have to use the .W suffix to get the maximum branch range. See *Instruction width selection on page 59*.

```
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.

Table 32. Miscellaneous instructions (continued)

| Mnemonic | Brief description | See |
|---|---|---|
| NOP | No Operation | *NOP on page 102* |
| SEV | Send Event | *SEV on page 102* |
| SVC | Supervisor Call | *SVC on page 103* |
| WFE | Wait For Event | *WFE on page 103* |
| WFI | Wait For Interrupt | *WFI on page 104* |

### 3.9.1 BKPT

Breakpoint.

#### Syntax

```
BKPT #imm
```

where:

- *'imm'* is an expression evaluating to an integer in the range 0-255 (8-bit value).

#### Operation

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

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

The BKPT instruction can be placed inside an IT block, but it executes unconditionally, unaffected by the condition specified by the IT instruction.

#### Condition flags

This instruction does not change the flags.

#### Examples

```
BKPT 0xAB    ; Breakpoint with immediate value set to 0xAB (debugger can
             ; extract the immediate value by locating it using the PC)
```

### 3.9.2 CPS

Change Processor State.

#### Syntax

```
CPSeffect iflags
```

### 4.4.3 Interrupt control and state register (SCB_ICSR)

Address offset: 0x04

Reset value: 0x0000 0000

Required privilege: Privileged

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 | VECTPENDING[9:4] | | | | | |
| rw | | | rw | w | rw | w | | | r | r | r | r | r | r | r |

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

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.

### System handler priority register 1 (SCB_SHPR1)

Address offset: 0x18

Reset value: 0x0000 0000

Required privilege: Privileged

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Reserved | | | | | | | | PRI_6[7:4] | | | | PRI_6[3:0] | | | |
| | | | | | | | | rw | rw | rw | rw | r | r | r | r |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| PRI_5[7:4] | | | | PRI_5[3:0] | | | | PRI_4[7:4] | | | | PRI_4[7:4] | | | |
| rw | rw | rw | rw | r | r | r | r | rw | rw | rw | rw | r | r | r | r |

Bits 31:24 Reserved, must be kept cleared

Bits 23:16 PRI_6[7:0]: Priority of system handler 6, usage fault

Bits 15:8 PRI_5[7:0]: Priority of system handler 5, bus fault

Bits 7:0 PRI_4[7:0]: Priority of system handler 4, memory management fault

### System handler priority register 2 (SCB_SHPR2)

Address offset: 0x1C

Reset value: 0x0000 0000

Required privilege: Privileged

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| PRI_11[7:4] | | | | PRI_11[3:0] | | | | 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[7:0]:** Priority of system handler 11, SVCall

Bits 23:0 Reserved, must be kept cleared

**Table 48. 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 | SCB_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 | SCB_ICSR | NMIPENDSET | Reserved | | PENDSVSET | PENDSVCLR | PENDSTSET | PENDSTCLR | Reserved | ISRPENDING | VECTPENDING[9:0] | | | | | | | | | | | RETOBASE | Reserved | VECTACTIVE[8:0] | | | | | | | | | |
| | 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 |
| 0x08 | SCB_VTOR | Reserved | TABLEOFF[29:9] | | | | | | | | | | | | | | | | | | | | | | Reserved | | | | | | | | |
| | Reset Value | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | | | | | | | |
| 0x0C | SCB_AIRCR | VECTKEY[15:0] | | | | | | | | | | | | | | | | ENDIANESS | Reserved | | | PRIG ROUP [2:0] | | | Reserved | | | | | SYSRESETREQ | VECTCLRACTIVE | VECTRESET | |
| | Reset Value | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | | | 0 | 0 | 0 | | | | | | 0 | 0 | 0 | |
| 0x10 | SCB_SCR | Reserved | | | | | | | | | | | | | | | | | | | | | | | | | | | SEVONPEND | Reserved | SLEEPDEEP | SLEEPONEXIT | Reserved | |
| | Reset Value | | | | | | | | | | | | | | | | | | | | | | | | | | | | 0 | | 0 | 0 | | |
| 0x14 | SCB_CCR | Reserved | | | | | | | | | | | | | | | | | | | | | | | | STKALIGN | BFHFNIGN | Res. | DIV_0_TRP | UNALIGN_TRP | Reserved | USERSETMPEND | NONBASETHRDENA | |
| | Reset Value (STM32F1 series) | | | | | | | | | | | | | | | | | | | | | | | | | 1 | 0 | | | | | | | |
| | Reset Value (STM32F2 and STM32L series) | | | | | | | | | | | | | | | | | | | | | | | | | 0 | 0 | | 0 | 0 | | 0 | 0 | |
| 0x18 | SCB_SHPR1 | Reserved | | | | | | | | PRI6 | | | | | | | | PRI5 | | | | | | | | | | | PRI4 | | | | | |
| | 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 |
| 0x1C | SCB_SHPR2 | PRI11 | | | | | | | | Reserved | | | | | | | | | | | | | | | | | | | | | | | | |
| | Reset Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0x20 | SCB_SHPR3 | PRI15 | | | | | | | | PRI14 | | | | | | | | Reserved | | | | | | | | | | | | | | | | |
| | Reset Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | | | | | | | | | | | | | | | |

### 4.5.2 SysTick reload value register (STK_LOAD)

Address offset: 0x04

Reset value: 0x0000 0000

Required privilege: Privileged

| 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[23:0]:** RELOAD value

The LOAD register specifies the start value to load into the VAL register 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.

The RELOAD value is calculated according to its use:

- 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.
- To deliver a single SysTick interrupt after a delay of N processor clock cycles, use a RELOAD of value N. For example, if a SysTick interrupt is required after 400 clock pulses, set RELOAD to 400.

Bit 30 **SKEW:** SKEW flag

Reads as one. Calibration value for the 1 ms 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

Indicates the calibration value when the SysTick counter runs on HCLK max/8 as external clock. The value is product dependent, please refer to the Product Reference Manual, SysTick Calibration Value section. When HCLK is programmed at the maximum frequency, the SysTick period is 1ms.

If calibration information is not known, calculate the calibration value required from the frequency of the processor clock or external clock.

### 4.5.5 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.

### 4.5.6 SysTick register map

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

**Table 49. 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_CTRL | Reserved | | | | | | | | | | | | | | | COUNTFLAG | Reserved | | | | | | | | | | | | | CLKSOURCE | TICK INT | EN ABLE |
| | Reset Value | | | | | | | | | | | | | | | | 0 | | | | | | | | | | | | | | 1 | 0 | 0 |
| 0x04 | STK_LOAD | 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_VAL | 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 |