ARM Cortex-M4 User Guide (Interrupts, exceptions, NVIC)
Cortex-M structure

CMSIS = Cortex Microcontroller Software Interface Standard
Cortex CPU core registers

- Two processor modes:
  - Thread mode for User tasks
  - Handler mode for O/S tasks and exceptions
- Stack-based exception model
- Vector table contains addresses

Process SP (handler or thread mode – select in CONTROL reg.)
Main SP (selected at reset – always used in handler mode)

Convention:
PSP in thread mode,
MSP in O/S & handler mode
Cortex-M4 processor operating modes

- **Thread** mode – normal processing
- **Handler** mode – interrupt/exception processing
- Privilege levels = **User** and **Privileged**
  - Supports basic “security” & memory access protection
  - Supervisor/operating system usually privileged
Cortex-M4 interrupts/exceptions

- Interrupts/exceptions managed by Nested Vectored Interrupt Controller (NVIC)
- CPU state/context (subset of registers) saved on the stack: R0-R3, R12, LR, PC, PSR
- PC loaded from a vector table, located at 0x0000_0000
  - Vector fetched (Flash memory) while saving state (SRAM)
  - Typical latency = 12 cycles
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.
  - **Active**: The exception is being serviced by the processor but has not completed.
  - **Active and pending**: The exception is being serviced by the processor and there is a pending exception from the same source.

- An interrupt request from a peripheral or from software can change the state of the corresponding interrupt to pending.
- An exception handler can interrupt (preempt) the execution of another exception handler. In this case both exceptions are in the active state.
## Cortex-M CPU and peripheral exceptions

<table>
<thead>
<tr>
<th>Priority¹</th>
<th>IRQ#²</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>-3</td>
<td>Power-up or warm reset</td>
</tr>
<tr>
<td>NMI</td>
<td>-2</td>
<td>-14 Non-maskable interrupt from peripheral or software</td>
</tr>
<tr>
<td>HardFault</td>
<td>-1</td>
<td>-13 Error during exception processing or no other handler</td>
</tr>
<tr>
<td>MemManage</td>
<td>Config</td>
<td>-12 Memory protection fault (MPU-detected)</td>
</tr>
<tr>
<td>BusFault</td>
<td>Config</td>
<td>-11 AHB data/prefetch aborts</td>
</tr>
<tr>
<td>UsageFault</td>
<td>Config</td>
<td>-10 Instruction execution fault - undefined instruction, illegal unaligned access</td>
</tr>
<tr>
<td>SVCcall</td>
<td>Config</td>
<td>-5 System service call (SVC) instruction</td>
</tr>
<tr>
<td>DebugMonitor</td>
<td>Config</td>
<td></td>
</tr>
<tr>
<td>PendSV</td>
<td>Config</td>
<td>-2 Interrupt-driven request for system service</td>
</tr>
<tr>
<td>SysTick</td>
<td>Config</td>
<td>-1 System tick timer reaches 0</td>
</tr>
<tr>
<td>IRQ0</td>
<td>Config</td>
<td>0 Signaled by peripheral or by software request</td>
</tr>
<tr>
<td>IRQ1 (etc.)</td>
<td>Config</td>
<td>1 Signaled by peripheral or by software request</td>
</tr>
</tbody>
</table>

¹ Lowest priority # = highest priority
² IRQ# used in CMSIS function calls
Vector table

- 32-bit vector (handler address) loaded into PC, while saving CPU context.
- Reset vector includes initial stack pointer
- Peripherals use positive IRQ #s
- CPU exceptions use negative IRQ #s
- IRQ # used in CMSIS function calls
- Cortex-M4 allows up to 240 IRQs
- IRQ priorities user-programmable
- NMI & HardFault priorities fixed

<table>
<thead>
<tr>
<th>Exception number</th>
<th>IRQ number</th>
<th>Offset</th>
<th>Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>16+n</td>
<td>n</td>
<td>0x0040+4n</td>
<td>IRQ0</td>
</tr>
<tr>
<td>18</td>
<td>2</td>
<td>0x004C</td>
<td>IRQ2</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>0x0048</td>
<td>IRQ1</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>0x0044</td>
<td>IRQ0</td>
</tr>
<tr>
<td>15</td>
<td>-1</td>
<td>0x0040</td>
<td>Systick</td>
</tr>
<tr>
<td>14</td>
<td>-2</td>
<td>0x003C</td>
<td>PendSV</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>0x0038</td>
<td>Reserved</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td>Reserved for Debug</td>
</tr>
<tr>
<td>11</td>
<td>-5</td>
<td>0x002C</td>
<td>SVCalls</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td>Usage fault</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>Bus fault</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>Memory management fault</td>
</tr>
<tr>
<td>6</td>
<td>-10</td>
<td>0x0018</td>
<td>Hard fault</td>
</tr>
<tr>
<td>5</td>
<td>-11</td>
<td>0x0014</td>
<td>NMI</td>
</tr>
<tr>
<td>4</td>
<td>-12</td>
<td>0x0010</td>
<td>Reset</td>
</tr>
<tr>
<td>3</td>
<td>-13</td>
<td>0x000C</td>
<td>Initial SP value</td>
</tr>
<tr>
<td>2</td>
<td>-14</td>
<td>0x0008</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>0x0004</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x0000</td>
<td></td>
</tr>
<tr>
<td>Position</td>
<td>Priority</td>
<td>Type of priority</td>
<td>Acronym</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
<td>------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Reserved</td>
</tr>
<tr>
<td>-3</td>
<td>fixed</td>
<td>Reset</td>
<td>Reset</td>
</tr>
<tr>
<td>6</td>
<td>settable</td>
<td>SysTick</td>
<td>System tick timer</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
<td>settable</td>
<td>WWDG</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>settable</td>
<td>PVD</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>settable</td>
<td>TAMP_STAMP</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>settable</td>
<td>RTC_WKUP</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>settable</td>
<td>FLASH</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>settable</td>
<td>RCC</td>
</tr>
<tr>
<td>6</td>
<td>13</td>
<td>settable</td>
<td>EXTI0</td>
</tr>
<tr>
<td>7</td>
<td>14</td>
<td>settable</td>
<td>EXTI1</td>
</tr>
<tr>
<td>8</td>
<td>15</td>
<td>settable</td>
<td>EXTI2</td>
</tr>
<tr>
<td>9</td>
<td>16</td>
<td>settable</td>
<td>EXTI3</td>
</tr>
<tr>
<td>10</td>
<td>17</td>
<td>settable</td>
<td>EXTI4</td>
</tr>
<tr>
<td>11</td>
<td>18</td>
<td>settable</td>
<td>DMA1_Stream0</td>
</tr>
<tr>
<td>12</td>
<td>19</td>
<td>settable</td>
<td>DMA1_Stream1</td>
</tr>
<tr>
<td>13</td>
<td>20</td>
<td>settable</td>
<td>DMA1_Stream2</td>
</tr>
</tbody>
</table>
STM32F4 vector table from startup code (partial)

____Vectors DCD __initial_sp ; Top of Stack
DCD Reset_Handler ; Reset Handler
DCD NMI_Handler ; NMI Handler

......
DCD SVC_Handler ; SVCall Handler
DCD DebugMon_Handler ; Debug Monitor Handler
DCD 0 ; Reserved
DCD PendSV_Handler ; PendSV Handler
DCD SysTick_Handler ; SysTick Handler

; External Interrupts
DCD WWDG_IRQHandler ; Window WatchDog
DCD PVD_IRQHandler ; PVD via EXTI Line detection
DCD TAMPP_STAMP_IRQHandler ; Tamper/TimeStamps via EXTI
DCD RTC_WKUP_IRQHandler ; RTC Wakeup via EXTI line
DCD FLASH_IRQHandler ; FLASH
DCD RCC_IRQHandler ; RCC
DCD EXTI0_IRQHandler ; EXTI Line0
DCD EXTI1_IRQHandler ; EXTI Line1
DCD EXTI2_IRQHandler ; EXTI Line2
Special CPU registers

ARM instructions to “access special registers”
- MRS   Rd,spec ; move from special register (other than R0-R15) to Rd
- MSR   spec,Rs  ; move from register Rs to special register

Use CMSIS\(^1\) functions to clear/set PRIMASK
- \_enable_irq();  // enable interrupts (set PRIMASK=0)
- \_disable_irq(); // disable interrupts (set PRIMASK=1)
  (double-underscore at beginning)

Special Cortex-M Assembly Language Instructions
- CPSIE I ; Change Processor State/Enable Interrupts (sets PRIMASK = 0)
- CPSID I ; Change Processor State/Disable Interrupts (sets PRIMASK = 1)

Prioritized Interrupts Mask Register (PRIMASK)

| 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  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

PRIMASK = 1 prevents (masks) activation of all exceptions with configurable priority
PRIMASK = 0 permits (enables) exceptions

Processor Status Register (PSR)

| 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  |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| N   | Z   | C   | V   | Q   | ICI/IT | T   | Reserved | ICI/IT | ISR_NUMBER |  # of current exception (lower priority cannot interrupt) |

\(^1\) Cortex Microcontroller Software Interface Standard – Functions for all ARM Cortex-M CPUs, defined in project header files: core_cmFunc.h, core_cm3.h
Prioritized interrupts

- Up to 256 priority levels
- 8-bit priority value
- Implementations may use fewer bits
  - STM32F4xx uses upper 4 bits of each priority byte => 16 levels
  - STM32F4xx uses 4 bits => 16 levels
- NMI & HardFault priorities are fixed
“Tail-chaining” interrupts

- NVIC does not unstack registers and then stack them again, if going directly to another ISR.
- NVIC can halt stacking (and remember its place) if a new IRQ is received.
Exception return

- The exception mechanism detects when the processor has completed an exception handler.

- Exception return occurs when:
  1. Processor is in Handler mode
  2. EXC_RETURN loaded to PC
  3. Processor executes one of these instructions:
     - LDM or POP that loads the PC
     - LDR with PC as the destination
     - BX using any register

- EXC_RETURN value loaded into LR on exception entry (after stacking original LR)
  - Lowest 5 bits of EXC_RETURN provide information on the return stack and processor mode.
Interrupt signal: from device to CPU

In each peripheral device:
- Each potential interrupt source has a separate **arm (enable)** bit
  - Set for devices from which interrupts, are to be accepted
  - Clear to prevent the peripheral from interrupting the CPU
- Each potential interrupt source has a separate **flag** bit
  - hardware sets the flag when an “event” occurs
  - Interrupt request = (flag & enable)
  - ISR software must clear the flag to acknowledge the request
  - test flags in software if interrupts not desired

**Nested Vectored Interrupt Controller (NVIC)**
- Receives all interrupt requests
- Each has an enable bit and a priority within the VIC
- Highest priority enabled interrupt sent to the CPU

**Within the CPU:**
- Global interrupt enable bit in PRIMASK register
- Interrupt if priority of IRQ < that of current thread
- Access interrupt vector table with IRQ#
Nested Vectored Interrupt Controller

- NVIC manages and prioritizes external interrupts in Cortex-M
  - 45 IRQ sources from STM32L1xx peripherals
- NVIC interrupts CPU with IRQ# of highest-priority IRQ signal
  - CPU uses IRQ# to access the vector table & get intr. handler start address
NVIC registers

- **NVIC_ISERx/NVIC_ICERx**
  - Interrupt Set/Clear Enable Register
  - $1 = \text{Set (enable) interrupt/Clear (disable) interrupt}$

- **NVIC_ISPRx/NVIC_ICPRx**
  - Interrupt Set/Clear Pending Register
  - Read 1 from ISPR if interrupt in pending state
  - Write 1 to set interrupt to pending or clear from pending state

- **NVIC_IABRx – Interrupt Active Bit Register**
  - Read 1 if interrupt in active state

- $x = 0..7$ for each register type, with 32 bits per register, to support up to 240 IRQs (STM32F4xx has 82 channels)
  - Each bit controls one interrupt, identified by its IRQ# (0..239)
  - Register# $x = \text{IRQ# DIV 32}$
  - Bit $n$ in the register $= \text{IRQ# MOD 32}$
NVIC registers (continued)

- NVIC_IPRx (x=0..59) – Interrupt Priority Registers
  - Supports up to 240 interrupts: 0..239
  - 8-bit priority field for each interrupts (4-bit field in STM32F4)
    - 4 priority values per register
    - 0 = highest priority
    - Register# x = IRQ# DIV 4
    - Byte offset within the register = IRQ# MOD 4
    - Ex. IRQ85:
      - 85/4 = 21 with remainder 1 (register 21, byte offset 1)
      - Write priority<<8 to NVIC_IPR2
      - 85/32 = 2 with remainder 21: write 1<<21 to NVIC_SER2

- STIR – Software Trigger Interrupt Register
  - Write IRQ# (0..239) to trigger that interrupt from software
  - Unprivileged access to this register enabled in system control register (SCR)
NVIC register addresses

NVIC_ISER0/1/2 = 0xE000E100/104/108
NVIC_ICER0/1/2 = 0xE000E180/184/188
NVIC_IPR0/1/2/…/20 = 0xE00E400/404/408/40C/….500

;Example – Enable EXTI0 with priority 5 (EXTI0 = IRQ6)
NVIC_ISER0  EQU  0xE000E100          ;bit 6 enables EXTI0
NVIC_IPR1    EQU  0xE000E404 ;3rd byte = EXTI0 priority

  ldr  r0,=NVIC_ISER0
  mov  r1,#0x0040 ;Set bit 6 of ISER0 for EXTI0
  str  r1,[r0]
  ldr  r0,=NVIC_IPR1 ;IRQ6 priority in IPR1[23:16]
  ldr  r1,[r0] ;Read IPR1
  bic  r1,#0x00ff0000 ;Clear [23:16] for IRQ6
  orr  r1,#0x00500000 ;Bits [23:20] = 5
  str  r1,[r0] ;Upper 4 bits of byte = priority
CMSIS functions

- NVIC_EnableIRQ(IRQn_Type IRQn)
- NVIC_DisableIRQ(IRQn_Type IRQn)
- NVIC_SetPendingIRQ(IRQn_Type IRQn)
- NVIC_ClearPendingIRQ(IRQn_Type IRQn)
- NVIC_GetPendingIRQ(IRQn_Type IRQn)
- NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)
- NVIC_GetPriority(IRQn_Type IRQn)

CMSIS = Cortex Microcontroller Software Interface Standard
- Vendor-independent hardware abstraction layer for Cortex-M
- Facilitates software reuse
- Other CMSIS functions: System tick timer, Debug interface, etc.
NVIC setup: enable interrupts

- NVIC only considers IRQs whose enable bits are set
  - Each IRQ has its own `enable` bit within NVIC
- `Interrupt Set Enable Register`: each bit `enables` one interrupt
  - CMSIS function: `NVIC_EnableIRQ(n)`; //set bit to `enable` IRQn
- `Interrupt Clear Enable Register`: each bit `disables` one interrupt
  - CMSIS function: `NVIC_DisableIRQ(n)`; //set bit to `disable` IRQn
- For convenience, `stm32f4xx.h` defines symbols for each IRQn
  Examples: `EXTI0_IRQHandler = 6` ; //External interrupt EXTI0 is IRQ #6
            `TIM3_IRQHandler = 29` ; //Timer TIM3 interrupt is IRQ #29

Usage:

```
NVIC_EnableIRQ(EXTI0_IRQHandler); //enable external interrupt EXTI0
NVIC_DisableIRQ(TIM3_IRQHandler); //disable interrupt from timer TIM3
```
MDK-ARM software components

- **Device Family Pack (DFP)**
  - Support files for a specific microcontroller family
    - CMSIS system/startup
    - Drivers (GPIO, Timers, ADC, etc.)
    - Flash algorithm

- **CMSIS** – generic CMSIS components
  - CORE functions (NVIC, SysTick)
  - DSP library
  - RTOS support

- **MDK “Professional Middleware”**
  - Libraries for USB, TCPIP, Files, Graphics, etc.
CMSIS organization

- **Core Peripheral Access Layer** provides name definitions, address definitions, and helper functions to access core registers and core peripherals.
- **Device Peripheral Access Layer (MCU specific)** offers name definitions, address definitions, and driver code to access peripherals.
- **Access Functions for Peripherals (MCU specific and optional)** implements additional helper functions for peripherals.
System tick timer interrupts

- SysTick timer is a simple 24-bit down counter
  - Interrupt on count down from 1 -> 0
  - Counter rolls over from 0 to 24-bit “reload” value (determines interrupt period)
  - User provides interrupt handler: `SysTick_Handler(void)`

- Control register bits:
  - 0: enable
  - 1: interrupt enable
  - 2: clock source
    - Free-running internal core clock (FCLK) - default
    - External clock (STCLK signal)
  - 16: rollover flag (set on count down from 1->0)

- CMSIS function startd timer, enabled interrupt, selectd clock source and sets reload value:
  
  ```
  #include "core_cm4.h"
  SysTick_Config (numberOfTicks);    //Ex. #ticks = SystemCoreClock/1000
  ```
STM32F4 external interrupt/event controller

23 edge detectors to trigger events and interrupts signaled by 240 GPIO pins and 7 internal events.
STM32F4xx external interrupt sources
(select in System Configuration Module – SYSCFG)

- 16 multiplexers select GPIO pins as external interrupts EXTI0..EXTI15
- Mux inputs selected via 4-bit fields of EXTICR[k] registers (k=0..3)
  - EXTIx = 0 selects PAx, 1 selects PBx, 2 selects PCx, etc.
  - EXTICR[0] selects EXTI3-EXTI0; EXTICR[1] selects EXTI7-EXTI4, etc

Example: Select pin PC2 as external interrupt EXTI2
SYSCFG->EXTICR[0] &= 0xF0FF;  //clear EXTI2 bit field
SYSCFG->EXTICR[0] |=  0x0200;  //set EXTI2 = 2 to select PC2
STM32F4 external interrupt sources

Seven “event” triggers:
EXTI16 = PVD output
EXTI17 = RTC Alarm event
EXTI18 = USB OTG FS Wakeup event
EXTI19 = Ethernet Wakeup event
EXTI20 = USB OTG HS Wakeup event
EXTI21 = RTC Tamper and TimeStamp events
EXTI22 = RTC Wakeup event
STM32F4 EXTI Registers

- 23 bits in each register - control 23 interrupts/events
- EXTI_IMR – interrupt mask register
  - 0 masks (disables) the interrupt
  - 1 unmasks (enables) the interrupt
- EXTI_RTSR/FTSR – rising/falling trigger selection register
  - 1 to enable rising/falling edge to trigger the interrupt/event
  - 0 to ignore the rising/falling edge
- EXTI_PR – interrupt/event pending register
  - read 1 if interrupt/event occurred
  - clear bit by writing 1 (writing 0 has no effect)
  - write 1 to this bit in the interrupt handler to clear the pending state of the interrupt
- EXTI_SWIER – software interrupt event register
  - 1 to set the pending bit in the PR register
  - Triggers interrupt if not masked
Signal flow/setup for External Interrupt EXTI\(x\), \(x = 0\ldots15\)

**SYSCFG module**

- Select source for EXTI\(x\)
- 4 bits each
- 0000 = PA\(x\)
- 0001 = PB\(x\)
- SYSCFG \(\rightarrow\) EXICR[0]

**GPIO module**

- Configure PA\(x\) as input (00)
- GPIOA \(\rightarrow\) MODER

**CPU**

- Enable
- __enable_irq();
- __disable_irq();
- 1 of 45, passed by NVIC to CPU

**NVIC**

- NVIC_EnableIRQ(n); interrupt mask (enable)
- Pending flag
- Set when EXTI\(x\) activates Clear when intr. handler exits
- NVIC_ClearPendingIRQ(n); clear pending flag
- NVIC_SetPriority(intr. #, value); value = priority of intr. #

**EXTI module**

- EXTI \(\rightarrow\) IMR
- EXTI \(\rightarrow\) FTSR
- EXTI \(\rightarrow\) RTSR
- EXTI \(\rightarrow\) PR
- edge
- select falling and/or rising edge
- For all EXTI\(x\) registers: Bit \(n\) controls EXTI\(n\)
Project setup for interrupt-driven applications

- Write the interrupt handler for each peripheral
  - Clear the flag that requested the interrupt (acknowledge the intr. request)
  - Perform the desired actions, communicating with other functions via shared global variables
  - Use function names from the vector table
    - Example: `void EXTI4_IRQHandler () { statements }`

- Perform all initialization for each peripheral device:
  - Initialize the device, “arm” its interrupt, and clear its “flag”
    - Example: External interrupt EXTI
      - Configure GPIO pin as a digital input
      - Select the pin as the EXTI source (in SYSCFG module)
      - Enable interrupt to be requested when a flag is set by the desired event (rising/falling edge)
      - Clear the pending flag (to ignore any previous events)
  - NVIC
    - Enable interrupt: `NVIC_EnableIRQ (IRQn);`
    - Set priority: `NVIC_SetPriority (IRQn, priority);`
    - Clear pending status: `NVIC_ClearPendingIRQ (IRQn);`

- Initialize counters, pointers, global variables, etc.
- Enable CPU Interrupts: `__enable_irq () ;`

(diagram on next slide)
Example: Enable EXTI0 as rising-edge triggered

;System Configuration Registers
SYSCFG EQU 0x40013800
EXTICR1 EQU 0x08

;External Interrupt Registers
EXTI EQU 0x40013C00
IMR EQU 0x00 ;Interrupt Mask Register
RTSR EQU 0x08 ;Rising Trigger Select
FTSR EQU 0x0C ;Falling Trigger Select
PR EQU 0x14 ;Pending Register

;select PC0 as EXTI0
ldr r1,=SYSCFG ;SYSCFG selects EXTI sources
ldrh r2,[r1,#EXTICR1] ;EXTICR1 = sources for EXTI0 - EXTI3
bic r2,#0x000f ;Clear EXTICR1[3-0] for EXTI0 source
orr r2,#0x0002 ;EXTICR1[3-0] = 2 to select PC0 as EXTI0 source
strh r2,[r1,#EXTICR1] ;Write to select PC0 as EXTI0

;configure EXTI0 as rising-edge triggered
ldr r1,=EXTI ;EXTI register block
mov r2,#1 ;bit #0 for EXTI0 in each of the following registers
str r2,[r1,#RTSR] ;Select rising-edge trigger for EXTI0
str r2,[r1,#PR] ;Clear any pending event on EXTI0
str r2,[r1,#IMR] ;Enable EXTI0
EXTI example – accessing registers directly (in C)

```c
#include "STM32F4xx.h"

/*------------------------------------------
   Initialize the GPIO and the external interrupt
   *------------------------------------------*/
void Init_Switch(void){

    //Enable the clock for GPIO
    RCC->AHB1ENR|=RCC_AHB1ENR_GPIOAEN;

    //Pull-up pin 0
    GPIOA->PUPDR|=GPIO_PUPDR_PUPDR0_1;

    //Connect the portA pin0 to external interrupt line0
    SYSCFG->EXTICR[0]&=SYSCFG_EXTICR1_EXTI0_PA;

    //Interrupt Mask
    EXTI->IMR |= (1<<0);

    //Falling trigger selection
    EXTI->FTSR|= (1<<0);

    //Enable interrupt
    __enable_irq();

    //Set the priority
    NVIC_SetPriority(EXTI0_IRQHandler,0);

    //Clear the EXTI pending bit
    NVIC_ClearPendingIRQ(EXTI0_IRQn);

    //Enable EXTI0
    NVIC_EnableIRQ(EXTI0_IRQn);
}

/*------------------------------------------
   Interrupt Handler – count button presses
   *------------------------------------------*/
void EXTI0_IRQHandler(void) {

    //Clear the EXTI pending bits
    NVIC_ClearPendingIRQ(EXTI3_IRQn);
    EXTI->PR|=(1<<0);

    //Make sure the Button is really pressed
    if (!(GPIOA->IDR & (1<<0)) )
    {
        count++;
    }
}
```
Interrupt Rituals

- Things you must do in every ritual
  - Initialize data structures (counters, pointers)
  - Arm interrupt in the peripheral device
    - Enable a flag to trigger an interrupt
    - Clear the flag (to ignore previous events)
  - Configure NVIC
    - Enable interrupt (NVIC_ISERx)
    - Set priority (NVIC_IPRx)
  - Enable CPU Interrupts
    - Assembly code  
      \[ \text{CPSIE I} \]
    - C code
      \[ \text{EnableInterrupts();} \]
Supervisor call instruction (SVC)

- Use to access system resources from the O/S or other “privileged operations”
- SVC_Handler() defined in the vector table
  - CMSIS NVIC functions IRQn for SVC is -5
    (if NVIC functions needed to change priority)
- Assembly language syntax:   SVC  #imm
  - #imm is the “SVC number” (0-255), which indicates a particular “service” to be performed by the handler
    - # is encoded into the instruction, but ignored by the CPU
  - Handler can retrieve #imm by using stacked PC to read the SVC instruction code
SVC in C programs

- ARM C compiler keyword extension `__svc` (double underscore) declares a Supervisor Call (SVC) function
  - May take up to four integer arguments
  - May return up to four results in a “value_in_regs” structure
- Syntax:

  ```
  __svc(int svc_num) return-type function-name(argument-list)
  ```

  - `svc_num` (8-bit constant) is the immediate value for the SVC instruction
  - “return-type function-name(argument-list)” is a basic C function prototype

ARM Compiler toolchain Compiler Reference: `__svc`
Example: SVC call from C code

/*-----------------------------------------------------------
   Set up SVC “calls” to “SVC_Handler”
   -----------------------------------------------------------*/
#define SVC_00 0x00
#define SVC_01 0x01

/* define “svc_zero” as SVC #0, passing pointer in R0 */
/* define “svc_one” as SVC #1, passing pointer in R0 */
void __svc(SVC_00) svc_zero(const char *string);
void __svc(SVC_01) svc_one(const char *string);

int call_system_func(void) {
    svc_zero("String to pass to SVC handler zero");       //Execute SVC #0
    svc_one("String to pass to a different OS function");   //Execute SVC #1
}

Reference: ARM Compiler toolchain Developing Software for
ARM Processors”: Supervisor Calls, Example 56
/* Stack contains:
* r0, r1, r2, r3, r12, r14, return address, xPSR
* First argument: svc_args[0] is (r0)
* Return address is svc_args[6]
*/

void SVC_Handler(unsigned int * svc_args) {
    unsigned int svc_number;

    //Read SVC# byte from SVC instruction code
    svc_number = ((char *)svc_args[6])[-2];

    //Execute code for each SVC #
    switch(svc_number) {
        case SVC_00:  /* Handle SVC 00 */
            break;
        case SVC_01:   /* Handle SVC 01 */
            break;
        default:            /* Unknown SVC */
            break;
    }
}

SVC_Handler with SVC # (example in MDK-ARM Help)
Determining the SVC immediate operand in assembly language

SVC_Handler:

TST    LR,#0x04 ;Test bit 2 of EXC_RETURN
ITE    EQ     ;Which stack pointer was used?
MRSEQ  R0,MSP ;Copy Main SP to R0
MRSNE  R0,PSP ;Copy Process SP to R0
LDR    R1,[R0,#24] ;Retrieve stacked PC from stack
LDRB   R0,[R1,#-2] ;Get #N from SVC instruction in program
ADR    R1,SVC_Table ;SVC Vector Table address
LDR    PC,[R1,R0,SLL #2] ;Branch to Nth routine

....

SVC_TABLE: ;Function address table

DCD  SVC0_Function
DCD  SVC1_Function
DCD  SVC2_Function
ARM (not Cortex) operating modes

- **User**: unprivileged mode under which most tasks run
- **FIQ**: entered when a high priority (fast) interrupt is raised
- **IRQ**: entered when a low priority (normal) interrupt is raised
- **Supervisor**: entered on reset and when a Software Interrupt instruction (SWI) is executed
- **Abort**: used to handle memory access violations
- **Undef**: used to handle undefined instructions
- **System**: privileged mode using the same registers as user mode
# ARM Operating Modes

<table>
<thead>
<tr>
<th>CPSR[4:0]</th>
<th>Mode</th>
<th>Use</th>
<th>Registers</th>
<th>Vector Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>User</td>
<td>Normal user code</td>
<td>User</td>
<td>0x00000000</td>
</tr>
<tr>
<td>10001</td>
<td>FIQ</td>
<td>Fast interrupt</td>
<td>__fiq</td>
<td>0x00000004</td>
</tr>
<tr>
<td>10010</td>
<td>IRQ</td>
<td>Standard interrupt</td>
<td>__irq</td>
<td>0x00000008</td>
</tr>
<tr>
<td>10011</td>
<td>SVC</td>
<td>Software interrupt</td>
<td>__svc</td>
<td>0x0000000C</td>
</tr>
<tr>
<td>10111</td>
<td>Abort</td>
<td>Memory faults</td>
<td>__abt</td>
<td>0x00000010</td>
</tr>
<tr>
<td>11011</td>
<td>Undef</td>
<td>Undefined instruction</td>
<td>__und</td>
<td>0x00000018</td>
</tr>
<tr>
<td>11111</td>
<td>System</td>
<td>Privileged O/S task</td>
<td>user</td>
<td>0x0000001C</td>
</tr>
</tbody>
</table>

CPSR = Current Processor Status Register
The ARM Register Set

Current Visible Registers

Abort Mode

Banked out Registers

User | FIQ | IRQ | SVC | Undef
--- | --- | --- | --- | ---
| r8 | r9 | r10 | r11 | r12
| r13 (sp) | r13 (sp) | r14 (lr) | r14 (lr) | r14 (lr)
| r13 (sp) | r13 (sp) | r14 (lr) | r14 (lr) | r14 (lr)
| r13 (sp) | r13 (sp) | r14 (lr) | r14 (lr) | r14 (lr)
| r13 (sp) | r13 (sp) | r14 (lr) | r14 (lr) | r14 (lr)
| cpsr | spsr | spsr | spsr | spsr
**ARM interrupts**

- ARM7 “core” supports two types of interrupts:
  - **Fast interrupt requests** (FIQs)
    - User registers R0-R7, R15/pc
    - Shadow registers R8_fiq – R14_fiq
  - **Interrupt requests** (IRQs)
    - User registers R0-R12, R15/pc
    - Shadow registers R13_irq, R14_irq

- Interrupt table starts at location 0 and contains subroutine calls to handlers
ARM interrupt vector table

Table 6-1. ARM7 Exception Table

<table>
<thead>
<tr>
<th>Vector Address</th>
<th>Interrupt Source</th>
<th>Interrupt Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000 0000</td>
<td>RESET</td>
<td>RESET</td>
</tr>
<tr>
<td>0x0000 0004</td>
<td>UNDEFINED INSTRUCTION</td>
<td>Undef’ Instr’</td>
</tr>
<tr>
<td>0x0000 0008</td>
<td>SWI</td>
<td>S/W Int</td>
</tr>
<tr>
<td>0x0000 000C</td>
<td>ABORT (Prefetch)</td>
<td>ABORT</td>
</tr>
<tr>
<td>0x0000 0010</td>
<td>ABORT (Data)</td>
<td>ABORT</td>
</tr>
<tr>
<td>0x0000 0018</td>
<td>IRQ</td>
<td>Normal Interrupt</td>
</tr>
<tr>
<td>0x0000 001C</td>
<td>FIQ</td>
<td>Fast Interrupt</td>
</tr>
</tbody>
</table>

• “Exception” condition changes operating mode
• ARM CPU defines only 7 vector addresses
• Any additional distinction of interrupt sources must be done in software, working with external hardware
LPC2292 – Vector table setup

AREA    RESET , CODE, READONLY

; Exception Vectors - Mapped to Address 0 (must use absolute addressing mode)

Vectors    LDR     PC, Reset_Addr
            LDR     PC, Undef_Addr
            LDR     PC, SWI_Addr
            LDR     PC, PAbt_Addr
            LDR     PC, DAbt_Addr
            NOP    ; Reserved Vector

;               LDR     PC, IRQ_Addr   ; If not using Vectored Interrupt Controller
            LDR     PC, [PC, #-0x0FF0]     ; Vector from VicVectAddr
            LDR     PC, FIQ_Addr

; List of handler addresses – loaded by LDR instructions above

Reset_Addr  DCD     Reset_Handler
Undef_Addr  DCD     Undef_Handler
SWI_Addr    DCD     SWI_Handler
PAbt_Addr   DCD     PAbt_Handler
DAbt_Addr   DCD     DAbt_Handler
            DCD     0                 ; Reserved Address
IRQ_Addr    DCD     IRQ_Handler
FIQ_Addr    DCD     FIQ_Handler
ARM7 Interrupt Exception Summary

<table>
<thead>
<tr>
<th>Register</th>
<th>FIQ</th>
<th>IRQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt Sample</td>
<td>Detect FIQ asserted</td>
<td>Detect FIQ negated, IRQ asserted</td>
</tr>
<tr>
<td>Link Register</td>
<td>R14_fiq = nextInst + 4</td>
<td>R14_irq = nextInst + 4</td>
</tr>
<tr>
<td>Saved Status Register</td>
<td>SPSR_fiq = CPSR</td>
<td>SPSR_irq = CPSR</td>
</tr>
<tr>
<td>Current Processor Status Register</td>
<td>(\text{CPSR[M]} = 5'b10001,)\n</td>
<td>(\text{CPSR[T]} = 1'b0,)\n</td>
</tr>
<tr>
<td>Stack Pointer</td>
<td>Activate R13_fiq</td>
<td>Activate R13_irq</td>
</tr>
<tr>
<td>Other Banked Registers</td>
<td>Activate R[8-12]_fiq</td>
<td>None</td>
</tr>
<tr>
<td>Program Counter</td>
<td>(\text{PC} = 0x0000_001C)</td>
<td>(\text{PC} = 0x0000_0018)</td>
</tr>
</tbody>
</table>

- **CPU actions:**
  - Save PC. Copy CPSR to SPSR (*Saved PSR*)
  - Activate shadow registers
  - Force bits in CPSR to record interrupt.
  - Force PC to vector.

- **Handler responsibilities:**
  - Restore proper PC.
  - Restore CPSR from SPSR.
  - Clear interrupt disable flags.
Supervisor mode

• Use supervisor mode to manage the various programs.
  • Provide protective barriers between programs.
  • Avoid memory corruption.
  • Control access to I/O
  • Provide operating system functions

• ARM: Use SWI (software interrupt) instruction to enter supervisor mode, similar to subroutine:
  ```
  SWI CODE_1
  ```
  • Sets PC to 0x08.
  • Argument to SWI is passed to supervisor mode code.
  • Saves CPSR in SPSR.
Using SWI in C programs

• Function prototype for calling from C
  
  ```c
  int __swi(0)  add (int i1, int i2);
  ```

  **Attribute** Function name to use in C program

• Call via:  
  ```c
  c = add(a,b);  //treat as normal function
  ```

• Function implementation
  
  ```c
  int __SWI_0 (int i1, int i2)
  {
    return (i1 + i2);
  }
  ```

• Vector table (in startup.s):
  ```c
  DCD  __SWI_0 ; SWI 0 Function Entry
  ```

SEE KEIL SWI EXAMPLE
Returning from interrupt

- R14 (link register) holds return information
- IRQ and FIQ: R14 = next instruction PC + 4
  - Return from IRQ/FIQ:  `subs pc,r14,#4`
- SWI: PC of next instruction in 14
  - Return from SWI:  `movs pc,r14`
- Data abort R14 saves next PC+8
  - Return:  `subs pc,r14,#8`
- Notes:
  - ‘s’ modifier allows CPSR to be restored
  - Different saved values relates to pipeline architecture