Computer Systems / Rekenaarstelsels 245 - 2020

Lecture 7

Interrupts and Exceptions: Part 1
Onderbrekings en Uitsonderings: Deel 1

Dr Rensu Theart & Dr Lourens Visagie

Lecture Overview

- Normal program execution flow
- Interrupts and exceptions
- Interrupt handlers (ISR)
- External interrupt/event controller (EXTI)
- Interrupt example
- C's volatile keyword

Program Execution Flow / Program uitvoerings vloei

- Program instruction execution is normally sequential. CPU executes one instruction after another
  - Program Counter (PC) increments to point to next instruction
- Program can cause a change in the sequential execution by changing the program counter

`MOV PC, #0`	Directly manipulate the program counter. Set it to a fixed address, to execute instructions from there
`B <label> / BL <label>`	Change the program counter to point to an instruction identified by a label
`BX <register>`	Change the program counter to point to an instruction at address held in a register
`POP {PC}, or` `LDR PC, [<register>]`	Load a value from memory and place the result in the program counter

Address	Instruction
0x00	`Start:` `MOV r0, #1`
0x04	`MOV r0, #2`
0x08	`MOV r0, #3`
0x0c	`MOV r0, #4`
0x10	`B Start`

Program Execution Flow / Program uitvoerings vloei

- How will this code 'flow'? (Where are all the changes to the PC?)
  - The function has to find the maximum absolute value of all array elements.

```
    STMDB r13!, {r4, lr}
    MOV r4, #0              ;r4 = result
loop
    CMP r1, #0
    MOVEQ r0, r4            ; return result in r0
    LDMIAEQ r13!, {r4, pc}  ;exit
    LDRSB r2, [r0]
    CMP r2, #0              ;set condition flags
    RSBLT r3, r3, #0        ; if (r3 < 0) r3 = 0 - r3
    CMP r4, r3
    MOVGT r4, r3
    ADD r0, r0, #1          ;advance to next sample
    SUB r1, r1, #1          ;r1 = r1-1
    B loop
```

```c
uint8_t findMaxAbs(int8_t* data, uint32_t len)
{
    uint8_t maxAbs = 0;
    for(int i = 0; i < len; i++)
    {
        if(abs(data[i]) > maxAbs)
            maxAbs = abs(data[i]);
    }
    return maxAbs;
}
```

Program Execution Flow / Program uitvoerings vloei

How will this code 'flow'? (Where are all the changes to the PC?)

```
    STMDB r13!, {r4, lr}
    MOV r4, #0           ;r4 = result
loop
    CMP r1, #0
    MOVEQ r0, r4           ; return result in r0
    LDMIAEQ r13!, {r4, pc}   ;exit
    LDRSB r2, [r0]
    CMP r2, #0            ;set condition flags
    RSBLT r3, r3, #0      ; if (r3 < 0) r3 = 0 - r3
    CMP r4, r3
    MOVGT r4, r3
    ADD r0, r0, #1        ;advance to next sample
    SUB r1, r1, #1        ;r1 = r1-1
    B loop
```

All of this is still 'normal' program flow

Interrupts and Exceptions / Onderbrekings en Uitsonderings

- **Interrupts** and **Exceptions** are events that breaks the normal (programmed) execution flow of a microcontroller
  - It is like an unscheduled function call that branches to a new address.
  - They are used to respond as quickly as possible to an event.
  - They can be caused by hardware or software.

1. **Interrupts** – for external events

One of the peripherals of the microcontroller, trying to signal the processor that something happened that must be responded to.

- Button was pressed
- Data arrived on a serial communications link
- A timer expired

2. **Exceptions (error conditions)** – for internal events

- Instruction that has invalid bit pattern was loaded by the processor (undefined instruction)
- Attempt to access memory outside of allowed bounds (data and prefetch abort)

Interrupt considerations / Onderbreking oorwegings

- How does the processor know which function to execute (and where to find it in memory)?
- How do we ensure that the interrupted program does not get confused (i.e. register and status flag values unchanged)?
- Which interrupt will get preference if more than one occur at the same time?
- How do we ensure that no interrupt can occur during critical parts of the program?

Exception handling / Uitsondering hantering

- When an exception is encountered the processor will automatically load a new value into the Program Counter; (there will not be an explicit “branch” instruction)
  - This value is looked up from the **vector table**.
  - This will cause the processor to execute a special "function" - **the exception handler** (this is code we write)
  - At the end of the exception handler, the program must return to the next instruction before exception occurred. So that program resumes as if the exception was just a “glitch”.
- The **vector table** is a table of addresses of special handler functions
  - There are different types of exception, and a handler function for each type of exception
  - Placed at the very beginning of the program (addresses 0x000 – 0x3FC)
  - On reset, the Cortex M4 will read two values from vector table (since the program starts at memory address 0x0000). The first one (at address 0x0000) is the initial Stack Pointer. The second (at address 0x0004) is the value that is loaded into PC – the address of the reset exception handler – this is where the CPU first loads instructions from when it starts up.

Figure 11. Vector table

Exception number	IRQ number	Offset	Vector
255	239	0x03FC	IRQ239
. . .		. . . 0x004C	. . .
18	2	0x0048	IRQ2
17	1	0x0044	IRQ1
16	0	0x0040	IRQ0
15	-1	0x003C	Systick
14	-2	0x0038	PendSV
13			Reserved
12			Reserved for Debug
11	-5	0x002C	SVCall
10			Reserved
9			
8			
7			
6	-10	0x0018	Usage fault
5	-11	0x0014	Bus fault
4	-12	0x0010	Memory management fault
3	-13	0x000C	Hard fault
2	-14	0x0008	NMI
1		0x0004	Reset
		0x0000	Initial SP value

MS30018V1

IRQ = Interrupt Request

Interrupt handlers / Onderbreking hanteerders

- Interrupt handlers are used to handle events from peripherals.
  - E.g. a button was pressed, serial data received from communications link, etc.
- Interrupt handler functions will typically be short.
  - Sometimes as simple as increment a counter (SysTick 1ms timer) or setting a boolean flag to 'true' (so that the main loop can further handle the event)
- An interrupt handler is sometimes also called an **ISR (Interrupt Service Routine)**
  - The interrupt requests (IRQx) are handle by ISRs.

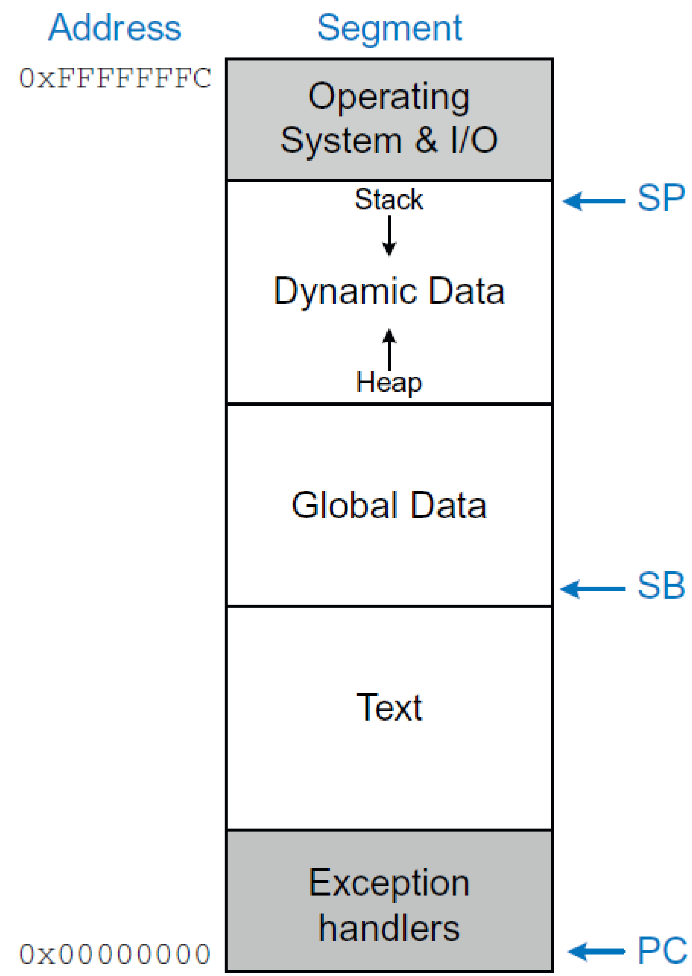


Figure 6.30 **Example ARM memory map**

Interrupts example / Onderbrekings voorbeeld

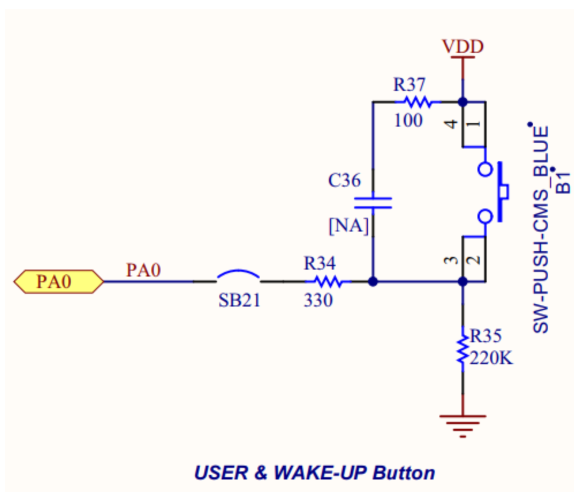
Problem: Turn on LED when button is pressed

Solution 1: Continuously check input port pin state (read from memory at specific address), and if it changes, turn on LED.

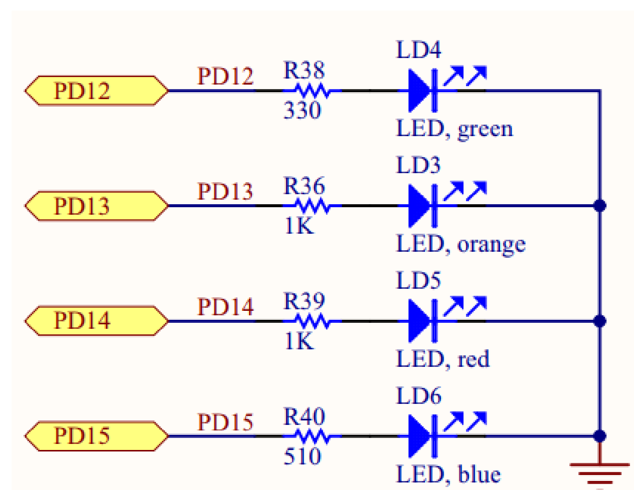
Solution 2: Enable interrupt on button press. In interrupt handler, switch on LED

⇒ Solution 2 is better if you want to handle a button press while the CPU does other things as well.

From [STM32F411 Discovery - User Manual]()



45

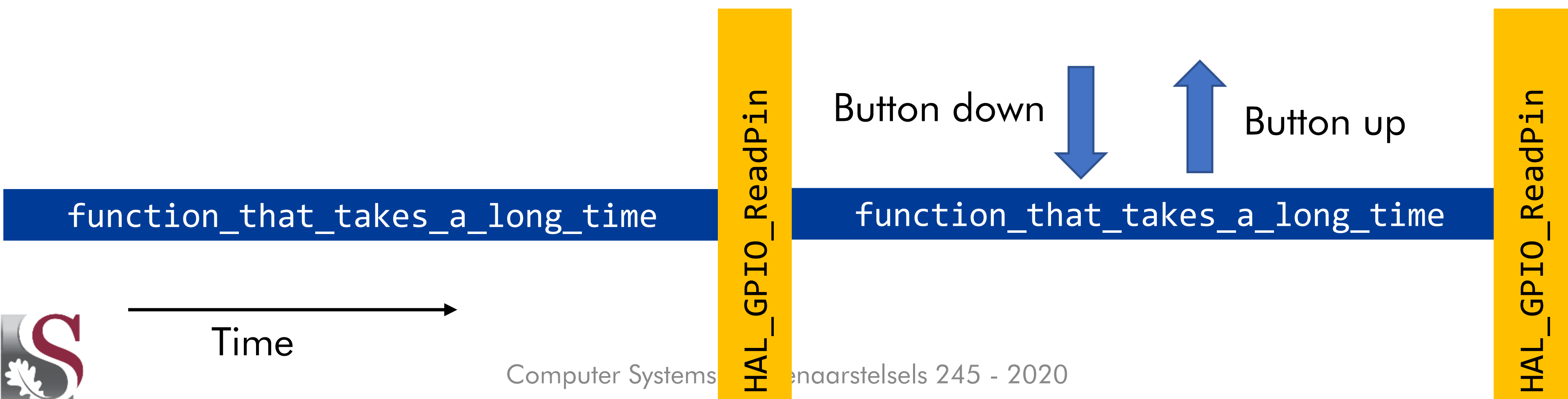


Interrupts example / Onderbrekings voorbeeld

Button press may not be registered if the main loop is busy with something else.

main.c

```
while (1)
{
    function_that_takes_a_long_time();
    if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0))
    {
        do_stuff();
    }
}
```



Interrupts example / Onderbrekings voorbeeld

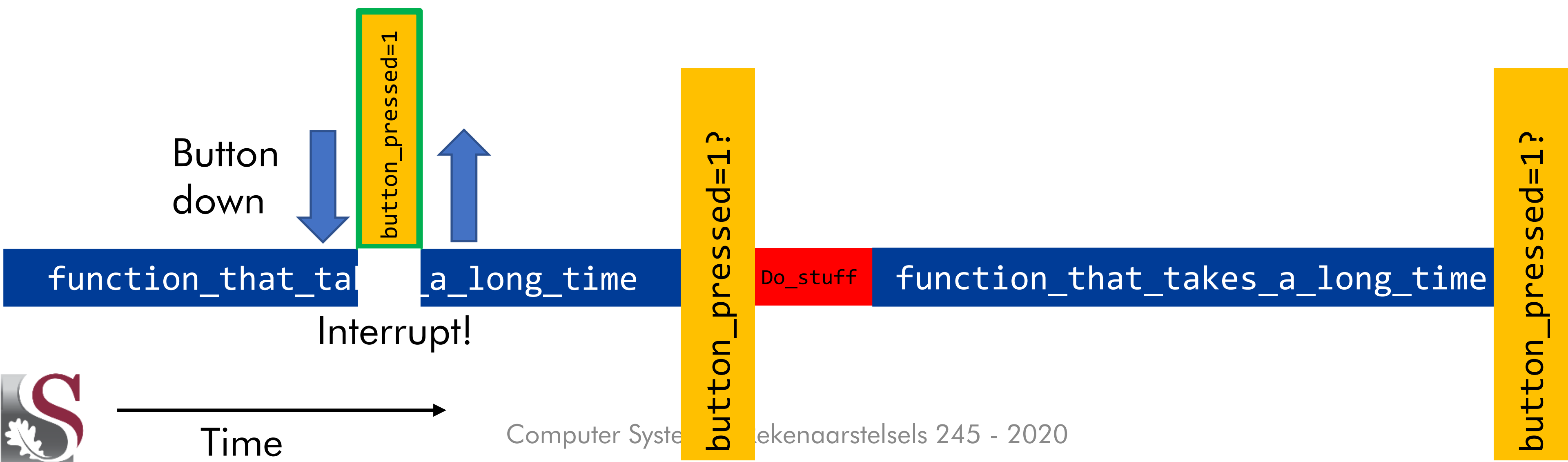
Solution – Use an interrupt

main.c

```c
while (1)
{
    function_that_takes_a_long_time();
    if (button_pressed)
    {
        do_stuff();
    }
}
```

Interrupt Handler

```c
void EXTI0_IRQHandler(void)
{
    button_pressed = 1;
}
```

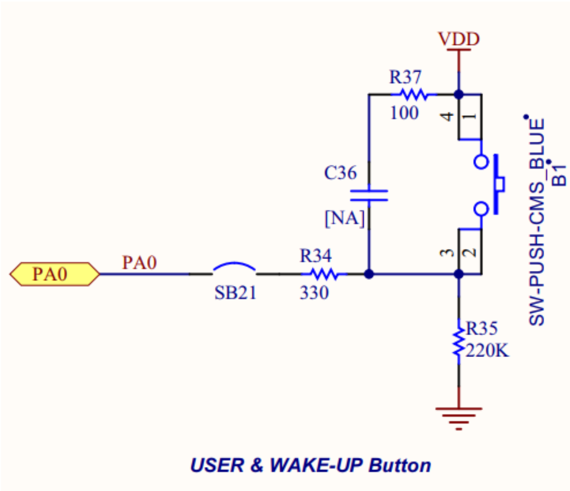


External Interrupt/event Controller (EXTI)

- How can we get our button to trigger an interrupt?
- The STM32F411 provides an external event input signal generated by the **External Interrupt/event Controller (EXTI)** on **asynchronous** event detection.
  - It consists of up to 23 edge detectors for generating event/interrupt requests.
  - Each input line can be independently configured to select the type (interrupt or event) and the corresponding trigger event (rising or falling or both).

External Interrupt/event Controller (EXTI)

- Since we want to use the button connected to PA0 for an interrupt, we must check which EXTI interrupt to use.
- Looking at the SYSCFG register, we see that PA0 is connected to EXTI0.



7.2.3 SYSCFG external interrupt configuration register 1 (SYSCFG_EXTICR1)

Address offset: 0x08

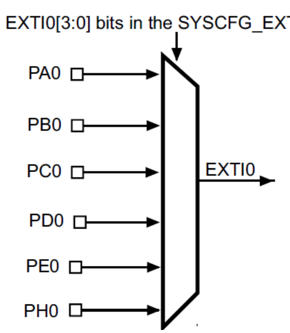
Reset value: 0x0000 0000

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
EXTI3[3:0]				EXTI2[3:0]				EXTI1[3:0]				EXTI0[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **EXTIx[3:0]**: EXTI x configuration (x = 0 to 3)
These bits are written by software to select the source input for the EXTIx external interrupt.
0000: PA[x] pin
0001: PB[x] pin
0010: PC[x] pin
0011: PD[x] pin
0100: PE[x] pin
0101: Reserved
0110: Reserved
0111: PH[x] pin

Figure 30. External interrupt/event GPIO mapping



From the STM32F411 Reference Manual

External Interrupt/event Controller (EXTI)

- The entry point for the interrupt handler (ISR) is stored at address 0x0000 0058 in the vector table.
- We must therefore ensure that our ISR's address is actually set up there.

Table 37. Vector table for STM32F411xC/E (continued)

Position	Priority	Type of priority	Acronym	Description	Address
4	11	settable	FLASH	Flash global interrupt	0x0000 0050
5	12	settable	RCC	RCC global interrupt	0x0000 0054
6	13	settable	EXTI0	EXTI Line0 interrupt	0x0000 0058
7	14	settable	EXTI1	EXTI Line1 interrupt	0x0000 005C
8	15	settable	EXTI2	EXTI Line2 interrupt	0x0000 0060
9	16	settable	EXTI3	EXTI Line3 interrupt	0x0000 0064
10	17	settable	EXTI4	EXTI Line4 interrupt	0x0000 0068
11	18	settable			0x0000 006C

From the STM32F411 Reference Manual

External Interrupt/event Controller (EXTI)

- Furthermore, to use EXTI, we must set up some other registers as well according to the reference manual:

Hardware interrupt selection

To configure the 23 lines as interrupt sources, use the following procedure:

- Configure the mask bits of the 23 interrupt lines (EXTI_IMR)
- Configure the Trigger selection bits of the interrupt lines (EXTI_RTSR and EXTI_FTSR)
- Configure the enable and mask bits that control the NVIC IRQ channel mapped to the external interrupt controller (EXTI) so that an interrupt coming from one of the 23 lines can be correctly acknowledged.

- Look at the reference and programming manual to see how to set these. Basically set:
  - EXTI_IMR bit 0 to 1 (make interrupt not masked, i.e. enabled)
  - EXTI_RTSR bit 0 to 1 (rising trigger enabled)
  - EXTI_FTSR bit 0 to 0 (falling trigger disabled)
  - NVIC_IPR0 bit 6 to 0 (Set priority of EXTI0 interrupt)
  - NVIC_ISER0 bit 6 to 1 (Enable interrupt for EXTI0)
- ⇒ Luckily using the HAL library this is much easier to set up.

How do I write an exception handler? / Hoe skryf ek 'n uitsondering hanterings funksie?

- In the STM32CubeIDE project file there is a .S file. It will set up the vector table.
- Function to handle exception should have a matching name, no arguments or return value.
- STM32CubeMX generated code already includes some exception handlers.

```
/*******************************************************************************
*
* The minimal vector table for a Cortex M3. Note that the proper constructs
* must be placed on this to ensure that it ends up at physical address
* 0x0000.0000.
*
*******************************************************************************/
  .section  .isr_vector,"a",%progbits
  .type  g_pfnVectors, %object
  .size  g_pfnVectors, .-g_pfnVectors

g_pfnVectors:
  .word  _estack
  .word  Reset_Handler
  .word  NMI_Handler
  .word  HardFault_Handler
  .word  MemManage_Handler
  .word  BusFault_Handler
  .word  UsageFault_Handler
  .word  0
  .word  0
  .word  0
  .word  0
  .word  SVC_Handler
  .word  DebugMon_Handler
  .word  0
  .word  PendSV_Handler
  .word  SysTick_Handler
```

Setup of vector table in file startup_stm32f411vetx.s

Handler function in assembly language

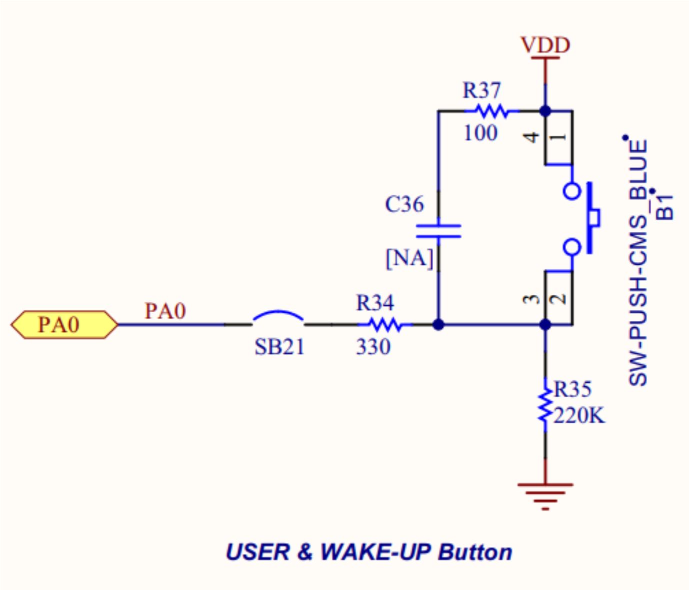
```
    .section  .text.Reset_Handler
  .weak  Reset_Handler
  .type  Reset_Handler, %function
Reset_Handler:
  ldr   sp, =_estack          /* set stack pointer */

/* Copy the data segment initializers from flash to SRAM */
  movs  r1, #0
  b  LoopCopyDataInit
```

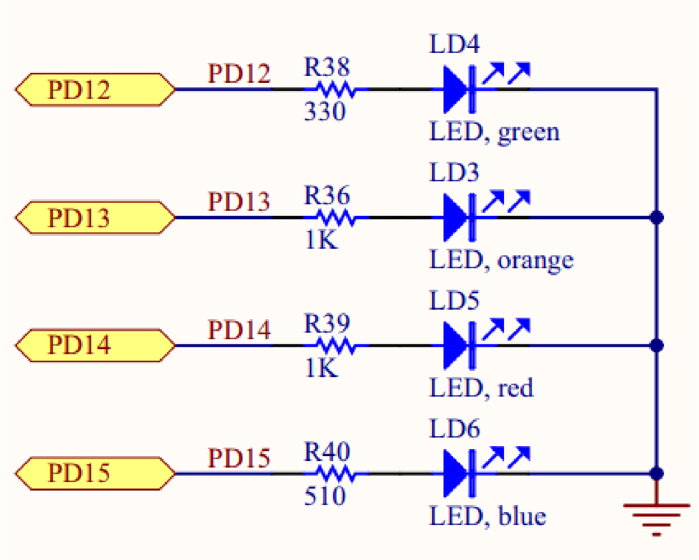
Handler function in C (stm32f4xx_it.c)

```
/**
  * @brief This function handles System tick timer.
  */
void SysTick_Handler(void)
{
  HAL_IncTick();
  HAL_SYSTICK_IRQHandler();
}
```

Interrupts example / Onderbrekings voorbeeld

- Ok, let's use an interrupt to turn on an LED when a button is pressed.
- External Interrupt 0 (EXTI0) triggered when user button is pressed.





Interrupts example / Onderbrekings voorbeeld

- In main program, check if variable button_pressed (stored in memory at address 0x1000 0000) changes from 0 to 1

Main program

Address	Value/ instruction
0000 1000	LDR r0, =0x1000 0000
0000 1004	MOV r1, #0
0000 1008	STR r1, [r0]
0000 100C	Loop: LDR r1, [r0]
0000 1010	CMP r1, #0
0000 1014	BEQ Loop
0000 1018	SetLed: ....
0000 101C	B Loop

Vector table

Address	Value/ instruction
0000 0000	
0000 0004	
0000 0008	
0000 000C	
…	
0000 0058	0000 1100
…	

Interrupt Handler

Address	Value/ instruction
0000 1100	LDR r0, =0x1000 0000
0000 1104	MOV r1, #1
0000 1108	STR r1, [r0]
0000 110C	BX LR

Interrupts example / Onderbrekings voorbeeld

- The button press event can occur at any point in the main program execution
- The exception that occurs is External Interrupt 0 (EXTI0). The entry for this type of exception is at address 0x0000 0058 in the vector table (for the STM32F4 microcontroller on our development board)
- The value at entry 0x0000 0058 in the vector table is 0x0000 1100. This is the value that is loaded into the PC – the address of our handler function.

Main program

Address	Value/ instruction
0000 1000	LDR r0, =0x1000 0000
0000 1004	MOV r1, #0
0000 1008	STR r1, [r0]
0000 100C	Loop: LDR r1, [r0]
0000 1010	CMP r1, #0
0000 1014	BEQ Loop
0000 1018	SetLed: ….
0000 101C	B Loop

Vector table

Address	Value/ instruction
0000 0000	
0000 0004	
0000 0008	
0000 000C	
…	
0000 0058	0000 1100
…	

Interrupt Handler

Address	Value/ instruction
0000 1100	LDR r0, =0x1000 0000
0000 1104	MOV r1, #1
0000 1108	STR r1, [r0]
0000 110C	BX LR

Interrupt!

Interrupts example / Onderbrekings voorbeeld

- The processor will automatically save the current PC (and other registers) onto the stack before changing to the Interrupt Handler
- The ISR sets the variable (at address 0x1000 0000) to a 1.
- The ISR ends by branching back to the main program – similar to a function.

Vector table

Address	Value/ instruction
0000 0000	
0000 0004	
0000 0008	
0000 000C	
…	
0000 0058	0000 1100
…	

Interrupt Handler

Address	Value/ instruction
0000 1100	LDR r0, =0x1000 0000
0000 1104	MOV r1, #1
0000 1108	STR r1, [r0]
0000 110C	BX LR

Main program

Address	Value/ instruction
0000 1000	LDR r0, =0x1000 0000
0000 1004	MOV r1, #0
0000 1008	STR r1, [r0]
0000 100C	Loop: LDR r1, [r0]
0000 1010	CMP r1, #0
0000 1014	BEQ Loop
0000 1018	SetLed: ….
0000 101C	B Loop

C `volatile` keyword

- The C compiler usually makes a attempts to optimize the code. This includes making decision of where variables are stored, and if a variable is never used it removes it completely from the machine code.
- This can cause problems when global variables are changed in interrupts, since the compile will ‘mispredict’ when a variable might change.
  - The problem is that the compiler has no idea that `button_pressed` can be changed within the ISR function, which doesn't appear to be ever called.
  - Therefore, any half decent optimizer will "break" the program.
- The solution is to declare the variable `button_pressed` to be **`volatile`**. After which, the program will work as you intended.

C `volatile` keyword

- C's `volatile` keyword is a qualifier that is applied to a variable when it is declared. It tells the compiler that the value of the variable may change at any time – without any action being taken by the code the compiler finds nearby.
- To declare a variable volatile, include the keyword volatile before or after the data type in the variable definition. (these two are equivalent)

```c
volatile uint16_t x;
uint16_t volatile y;
```

- Pointers to volatile variables are also common, especially with memory-mapped I/O registers. (these two are equivalent)

```c
volatile uint8_t * p_reg;
uint8_t volatile * p_reg;
```

C `volatile` keyword

A variable should be declared volatile whenever its value could change unexpectedly. The two most common cases are:

1. Memory-mapped peripheral registers
2. Global variables modified by an interrupt service routine