Embedded Systems

Microcontroller Work

H. Cassé

2021

Planning

Hourly volume:

- 6H course
- 14H exercises
- 10H labwork

Organization:

- *CLB* $\rightarrow$ *SECIL SE*
- planned for 2022 $\rightarrow$ sabattical year of former teacher
- course/exercises: Thu. 10:00-12:00
- labwork: Thu. 10:00-12:00 (after)

Bare Metal

2 approaches to embedded programming:

- Linux – requires lot of hardware resources
- bare metal +/- OS as a library

What is the difference with usual programming?

- heterogeneous hardware
- set of physical requirements
- types of sensors and actuators
- types of applications

Table of Contents

Example 1: a quadcopter

Application:

- autopilot Paparazzi
- OS – Linux

Sensors: GPS, IMU (Inertial Memory Unit), Sonar, Camera

Actuators: 4 brushless motors/ controllers

Hardware:

- BeagleBone – ARM (32-bit)
- controller – AVR (8/16-bit)

Requirements:

- low power
- fast enough (real-time)
- relativily cheap

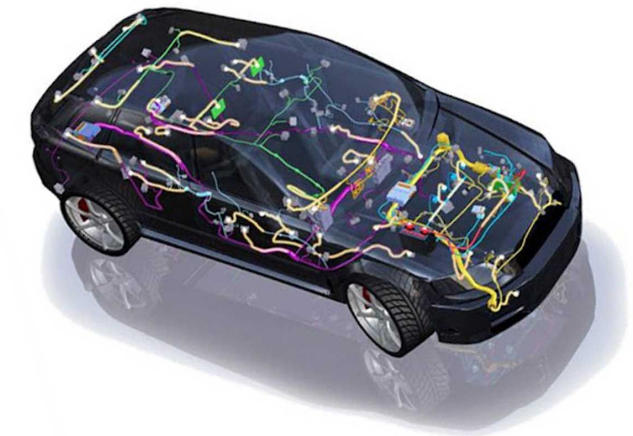
Example 2: a car

Application:

- powertrain
- light, breaks
- infotainment
- proximity sensors, drive assistance, etc

Hardware:

- lots of sensor/actuators
- distance $\rightarrow$ busses
- distributed CU (8-32-bit)
- $1 - n$ MCU (Tricore, PowerPC – 32-bit)



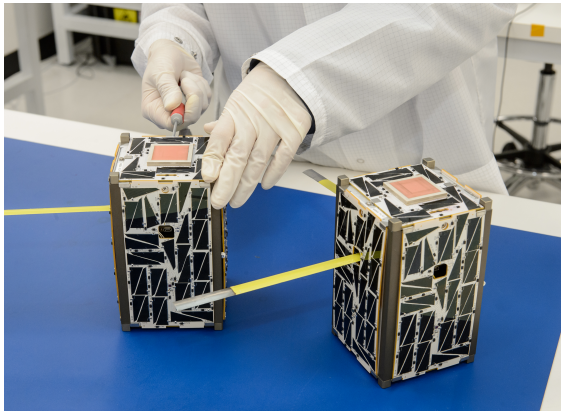
Requirements:

- cheap
- robust (heat, cold, mechanics)
- real-time
- safety

Example 3: a cubesat

Application:

- Attitude Control and Determination System (ACDS)
- payload application
- thruster, solar panel
- GPS, IMU, etc
- ACDS CPU (Leon / Sparc)
- payload CPU



Requirements:

- cheap
- very robust (SEE - *Single Event Effect*, temperature)
- safe (fixup mostly impossible)
- low power

Requirements

Server/desktop/laptop:

- safety is better
- expensive to very expensive
- lowly robust

Embedded system:

- safety (often stringent requirement)
- often cheap
- power consumption
- often very robust

Hardware & Software

Server/desktop/laptop:

- AMD/Intel 64b μP (2-3GHz) x86/ARM
- 2-4 Gb DRAM
- powerful OS (Linux, MacOSX, Windows)
- Ethernet / wifi
- keyboard, mouse, screen

Embedded system:

- 8/16/32-bit μP (16MHz-1GHz) ARM, TriCore, Leon, AVR, etc
- 20Kb-100Mb SRAM, 1Mb-16MB flash
- bare-metal / OS as a library
- Ethernet, wifi, wirless, Xbee, etc, none
- LED, button, specific sensors and actuators

Microcontroller

Cheap grouping:

- execution core(s)
- memory
- driver for devices

SOC (*System On Chip*):

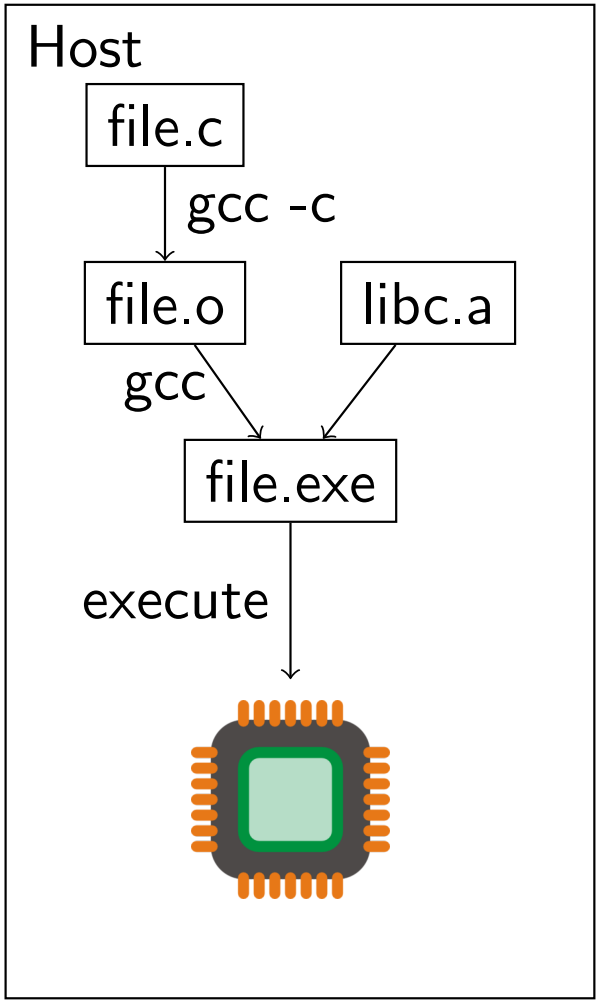
- several cores
- several memories
- interconnect network

Example: STM32F407 (STMicro)

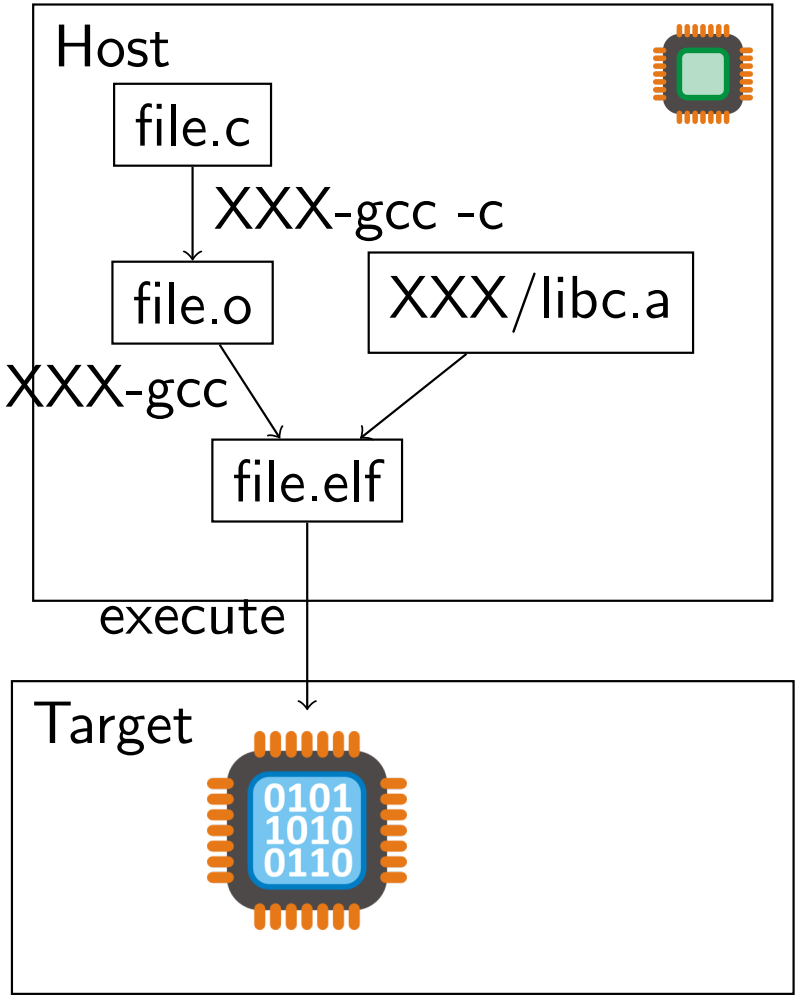
- ARM Cortex M4+FPU (monocore)
- SRAM (main 112Kb, 16Kb)
- flash (1Mb)
- network: Ethernet, USB, I2C, USART, SPI
- peripherals: GPIO, timers, ADC, DAC

Programming

Server/desktop/laptop:

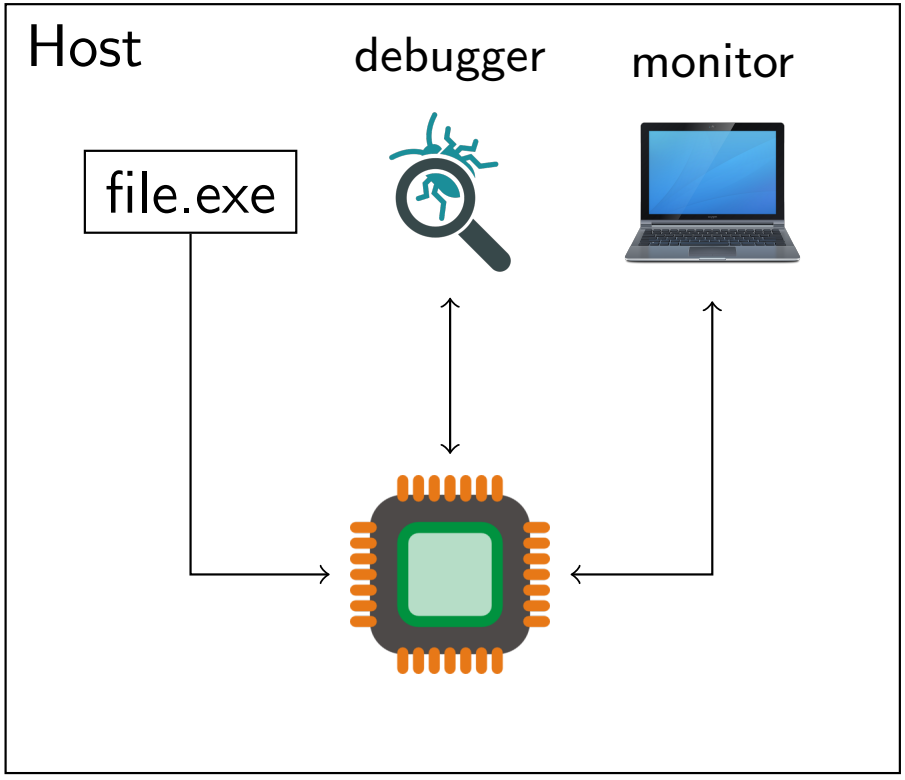


Embedded System:

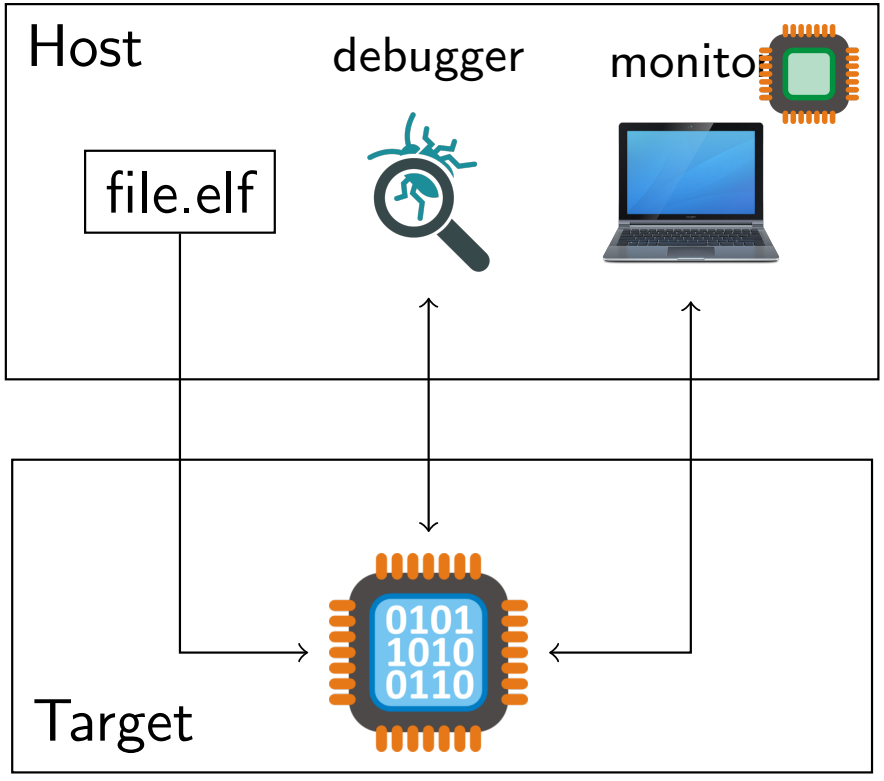


Debugging & Testing

Server/desktop/laptop:



Embedded System:



Host/Target links: USART, USB, JTag

Table of Contents

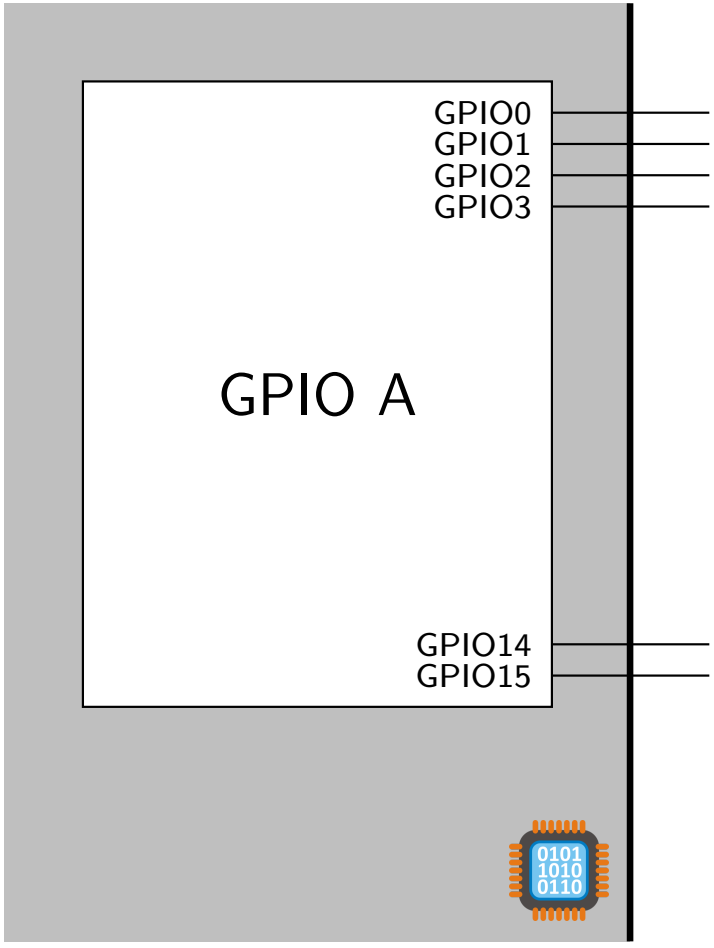
Example: GPIO

Generic Parallel Input/Output

- electric lines (*pin*)
- bidirectional
- output 0V or 5V
- input 0V ou 5V
- any sensor producing pulses (width, frequence)
- any actuator supporting *all-or-none* signal
- in fact: very used

STM32F407

- GPIO A-K (11 GPIO)
- 16 pins/GPIO



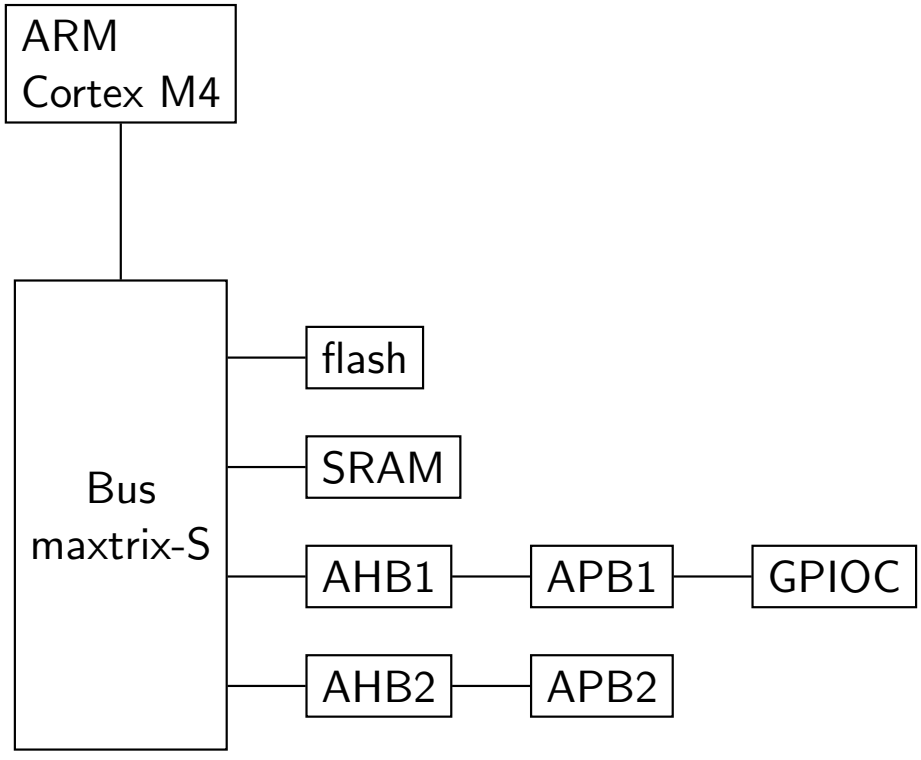
How to program?

GPIO drivers controlled by register mapped in memory:

- `GPIO_MODER` – configure intput/output
- `GPIO_IDR` – value as input
- `GPIO_ODR` – value as output
- `GPIO_BSRR` – set/reset output

address = driver address + offset

- `GPIO_IDR` offset = 0x10
- GPIO C base = 0x40020800
- address = 0x40020810



AHB = Amba High-speed Bus
APB = Amba Peripheral Bus

Finding the base addresses?

From the microcontroller reference manual:

RM0090 - Reference manual
STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced Arm ® -based 32-bit MCUs

In memory map section:

0x4002 3800 - 0x4002 3BFF	RCC	AHB1	Section 7.3.24: RCC register map on page 265
0x4002 3000 - 0x4002 33FF	CRC		Section 4.4.4: CRC register map on page 115
0x4002 2800 - 0x4002 2BFF	GPIOK		Section 8.4.11: GPIO register map on page 287
0x4002 2400 - 0x4002 27FF	GPIOJ		
0x4002 2000 - 0x4002 23FF	GPIOI		
0x4002 1C00 - 0x4002 1FFF	GPIOH		
0x4002 1800 - 0x4002 1BFF	GPIOG		
0x4002 1400 - 0x4002 17FF	GPIOF		
0x4002 1000 - 0x4002 13FF	GPIOE		Section 8.4.11: GPIO register map on page 287

In C

Use of defines:

```
#define GPIO_IDR     0x10
#define GPIO_ODR     0x14
...
#define GPIOC_BASE   0x40020800
...
```

Define to access IDR of GPIOC:

```
#define GPIOC_IDR    \
    *(volatile uint32_t *)(GPIOC_BASE + GPIO_IDR)
```

`volatile` $\Longrightarrow$ prevent CC optimizations

GPIOx_ODR

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
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

16 pins $\iff$ 1 bit / pin

- bit $= 0 \implies$ pin at 0V
- bit $= 1 \implies$ pint at 5V

How to change it?

1. read the word
2. changes the bits
3. write the word

How to change bits?

Use bit-to-bit operations of C:

- ~ – bit level NOT

$X =$	X_{31}	X_{30}	...	X_2	X_1	X_0
$\tilde{}X =$	$\overline{X_{31}}$	$\overline{X_{30}}$	...	$\overline{X_2}$	$\overline{X_1}$	$\overline{X_0}$

- & – bit level AND

$A =$	A_{31}	A_{30}	...	A_2	A_1	A_0
$B =$	B_{31}	B_{30}	...	B_2	B_1	B_0
$A\&B =$	$A_{31}.B_{31}$	$A_{30}.B_{30}$	...	$A_2.B_2$	$A_1.B_1$	$A_0.B_0$

- | – bit level OR

$A =$	A_{31}	A_{30}	...	A_2	A_1	A_0
$B =$	B_{31}	B_{30}	...	B_2	B_1	B_0
$A\|B =$	$A_{31} + B_{31}$	$A_{30} + B_{30}$	...	$A_2 + B_2$	$A_1 + B_1$	$A_0 + B_0$

Set a bit to 1

- 1 is absorbing for $+$: $A + 1 = 1$
- 0 is neutral for $+$: $A + 0 = A$

$X =$	X_{31}	X_{30}	...	X_i	...	X_1	X_0
$M =$	0	0	...	1	...	0	0
$X \vert M =$	X_{31}	X_{30}	...	1	...	X_1	X_0

How to get the mask M where only bit i is 1:

$$M = 1 << i$$

Set a bit to 0

- 1 is neutral for .: $A.1 = A$
- 0 is absorbing for .: $A.0 = 0$

$X =$	X_{31}	X_{30}	...	X_i	...	X_1	X_0
$M =$	1	1	...	0	...	1	1
$X \mid M =$	X_{31}	X_{30}	...	0	...	X_1	X_0

How to get the mask M where only bit i is 0 (and other are 1s):

$$M = \sim (1 << i)$$

Controlling a LED

Lighting on:

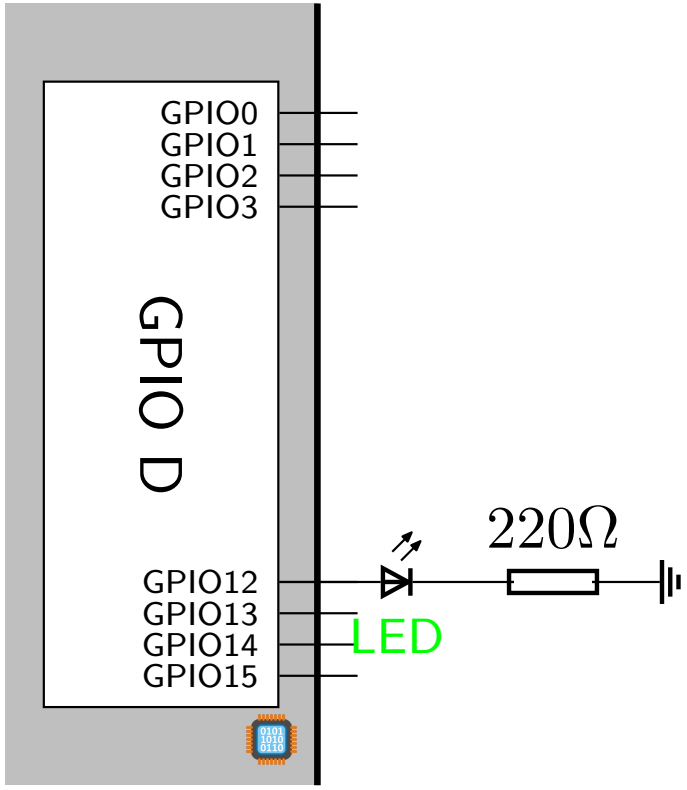
```
#define GREEN_LED    12

GPIOD_ODR = GPIOD_ODR | (1 << GREEN_LED);
```

Lighting off:

```
#define GREEN_LED    (12

GPIOD_ODR = GPIOD_ODR & ~(1 << GREEN_LED);
```

Exercise: write code to

- light on LED orange and red (GPIOD, pin 13 and 14)
- light off LED green and blue (GPIOD, pin 12 and 15)



Different ways to set the output of GPIO

Register `GPIOx_BSRR` (Bit Set/Regset)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

- only 1 have an effect!
- 1 in bit $i \in [15-0] \implies$ set 1 to output pin i
- 1 in bit $i \in [31-16] \implies$ set 0 to output pin $i-16$
- lighting on green LED: `GPIOD_BSRR = 1 << GREEN_LED;`
- lighting off green LED: `GPIOD_BSRR = 1 << (GREEN_LED + 16);`

Exercise: do the same as previous exercise with `GPIOx_BSSR`.

Blink program

Write a program that blinks the red LED with a period of 1s:

```
#include <stm32/gpio.h>
...
#define RED_LED      14

int main() {
    int HALF_PERIOD = 1000000;

    /* GPIO initialization */
    ...

    /* endless loop */
    while(1) {

        /* wait loop */
        for(int i = 0; i < HALF_PERIOD; i++);

        /* light on/off the LED */
        if((GPIOD_ODR & (1 << RED_LED)) == 0)
            GPIOD_BSRR = 1 << RED_LED;
        else
            GPIOD_BSRR = 1 << (RED_LED + 16);
    }
}
```

Setting blinking speed

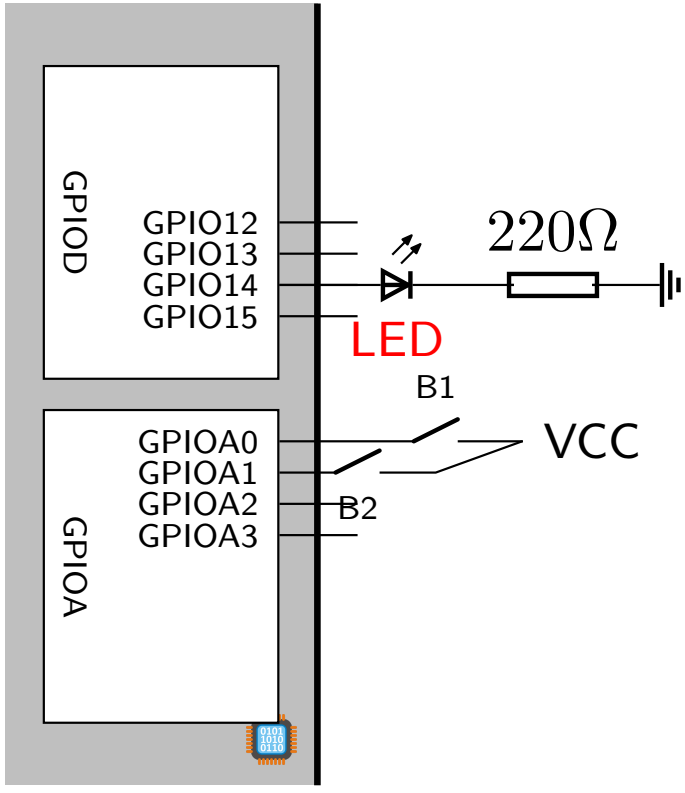
`GPIOx_IDR` (Input Data Register)

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
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

- bit *i* is 0 – input voltage 0V
- bit *i* is 1 – input voltage is 5V

Example

```
#define B1   0
#define B2   1
if((GPIOA_IDR & (1 << B1)) != 0)
    /* pin 0 = 1, B1 is pushed */
else
    /* pin 0 = 0, B1 is released */
```



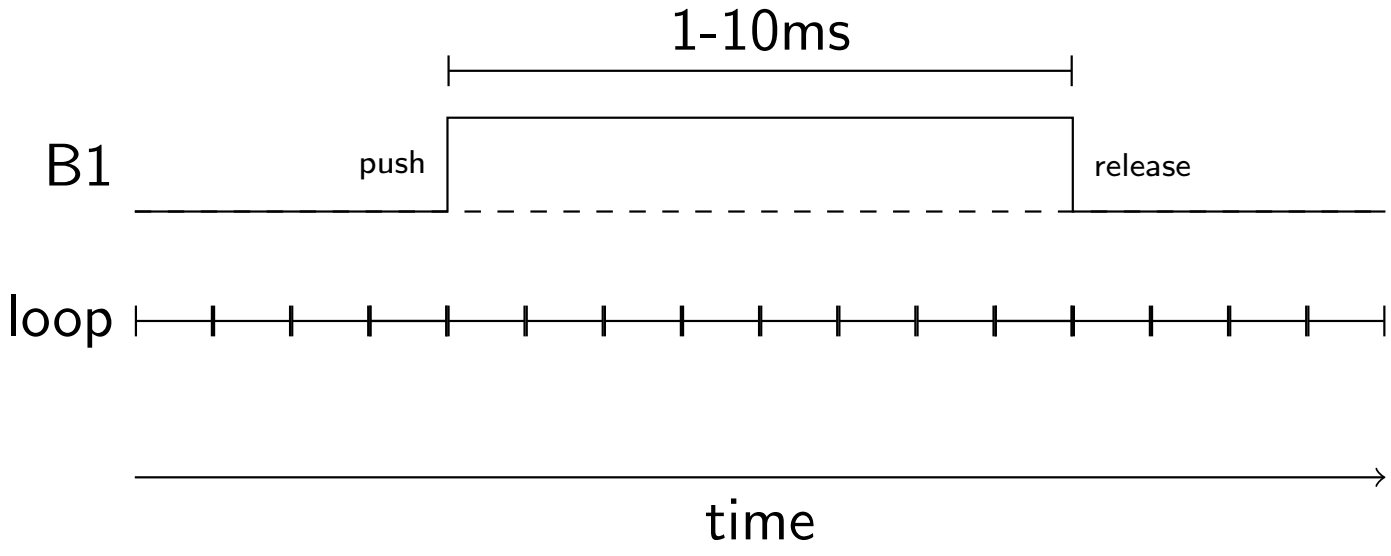
How to test a bit value?

Problem: inside a value x, we want to known if the bit i is 0 or 1?

	31						0	
x	X_{31}	X_{30}	...	X_i	...	X_1	X_0	
M	0	0	...	1	...	0	0	
$x\&M$	0	0	...	X_i	...	0	0	$= R$

- if $R = 00...X_i...00 = 0$ then $X_i = 0$,
- if $R = 00...X_i...00 \neq 0$ then $X_i = 1$.

What is a click?

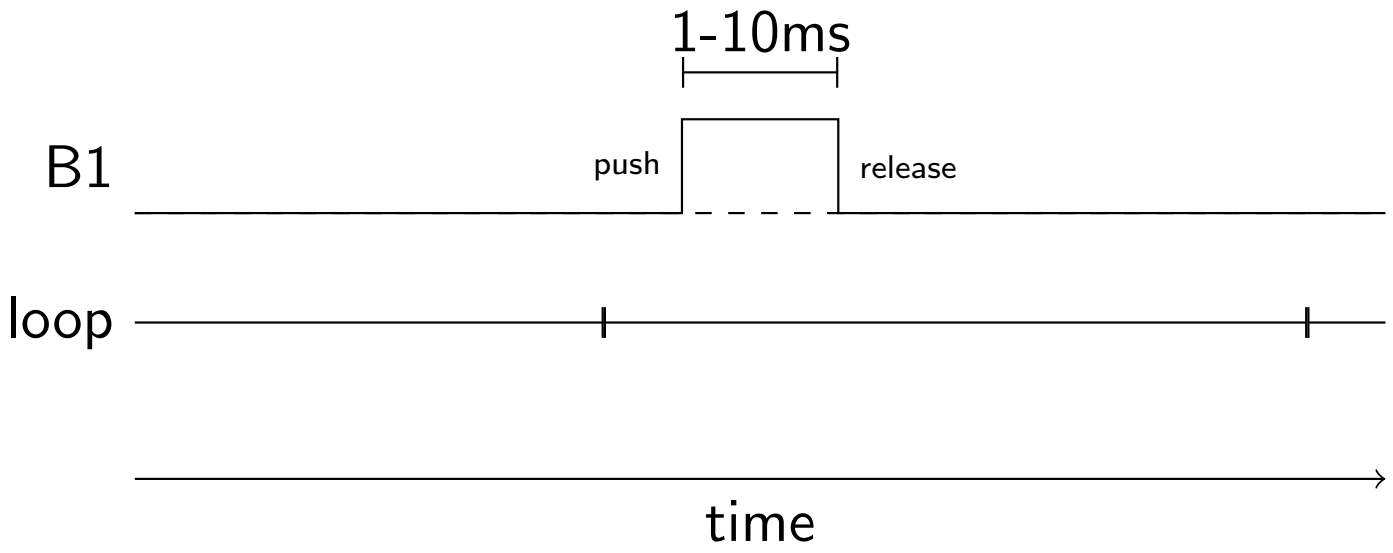



```
int b1_state = 0;
if((GPIOA_IDR & (1 << B1)) != 0)
    b1_state = 1;
else if(b1_state == 1) {
    b1_state = 0;
    /* action here */
}
```

Blinker with speed up/down buttons

```
int b1_state = 0, b2_state = 0;
while(1) {
    for(int i = 0; i < HALF_PERIOD; i++);

    /* invert LED */
    if((GPIOD_ODR & (1 << RED_LED)) == 0)
        GPIOD_BSRR = 1 << RED_LED;
    else
        GPIOD_BSRR = 1 << (RED_LED + 16);

    /* poll buttons */
    if((GPIOD_IDR & (1 << B1)) == 0)
        b1_state = 1;
    else if(b1_state) {
        b1_state = 0;
        HALF_PERIOD += 1000;
        }
    if((GPIOD_IDR & (1 << B2)) == 0)
        b2_state = 1;
    else if(b2_state) {
        b2_state = 0;
        HALF_PERIOD += 1000;
    }
}
```

What will happen?



$\Longrightarrow$

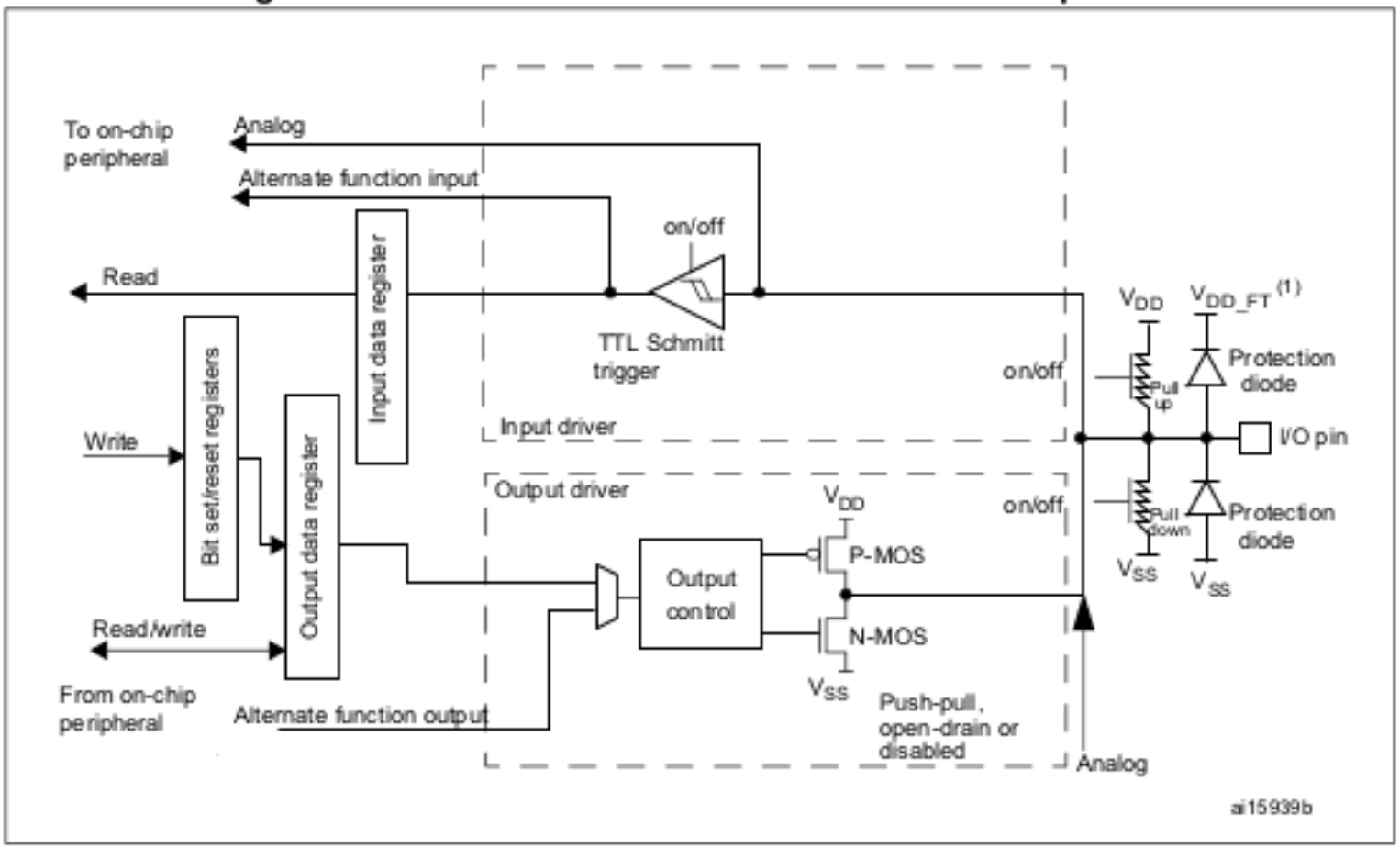
- button polling must be put in waiting loop
- additional processing in wait loop $\Longrightarrow$ count reduction

Better blinker version

```
int HALF_PERIOD = 200000;
int b1_state = 0, b2_state = 0;
while(1) {

    /* waiting loop and button test */
    for(int i = 0; i < HALF_PERIOD; i++) {
        if((GPIOD_IDR & (1 << B1)) == 0)
            b1_state = 1;
        else if(b1_state) {
            b1_state = 0;
            HALF_PERIOD += 200;
        }
        if((GPIOD_IDR & (1 << B2)) == 0)
            b2_state = 1;
        else if(b2_state) {
            b2_state = 0;
            HALF_PERIOD += 200;
        }
    }

    /* invert LED */
    if((GPIOD_ODR & (1 << RED_LED)) == 0)
        GPIOD_BSRR = 1 << RED_LED;
    else
        GPIOD_BSRR = 1 << (RED_LED << 16);

}
```

About initialization: GPIO circuit

Figure 25. Basic structure of a five-volt tolerant I/O port bit



About pull-up/down

To force state of a pin to 0 or 1 (not floating):

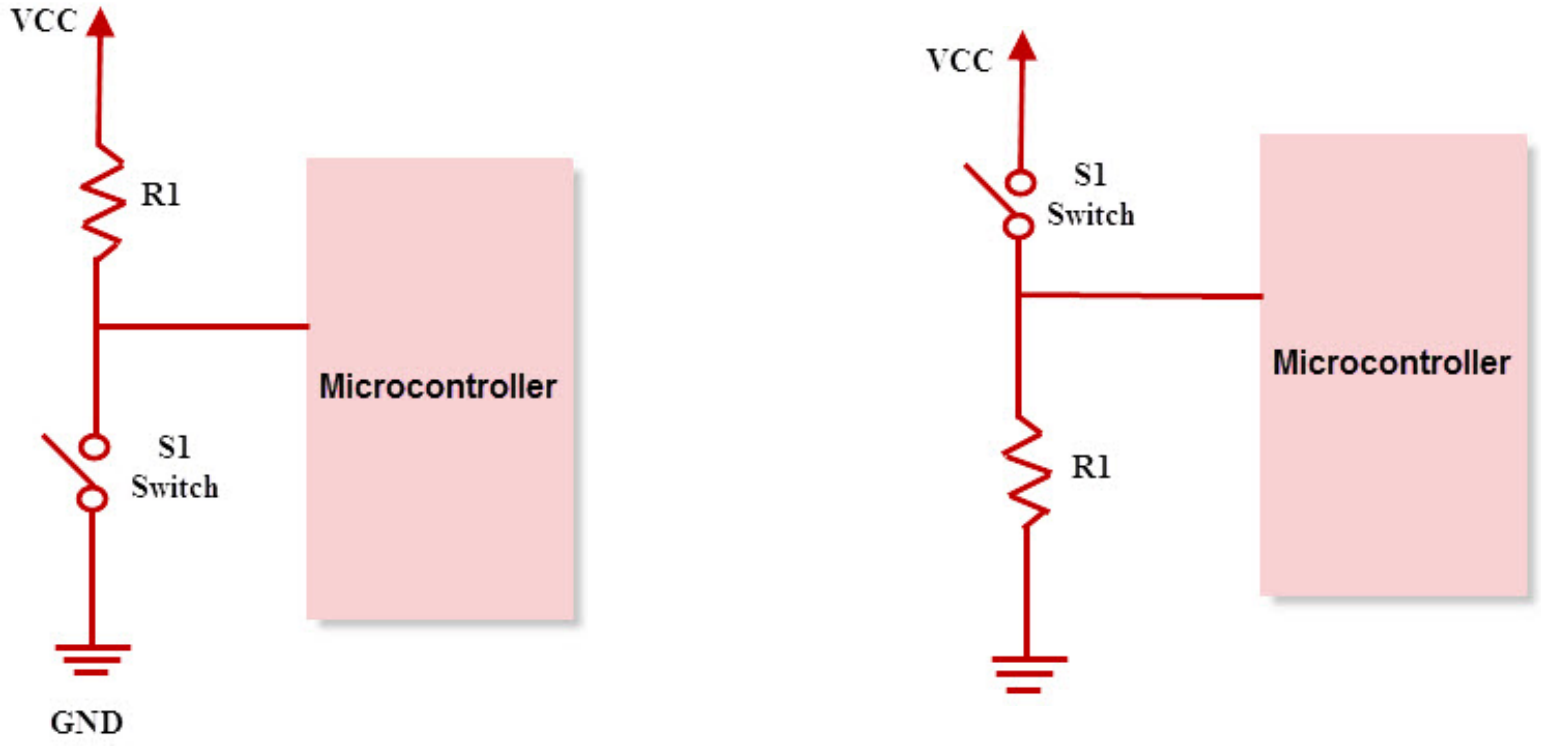
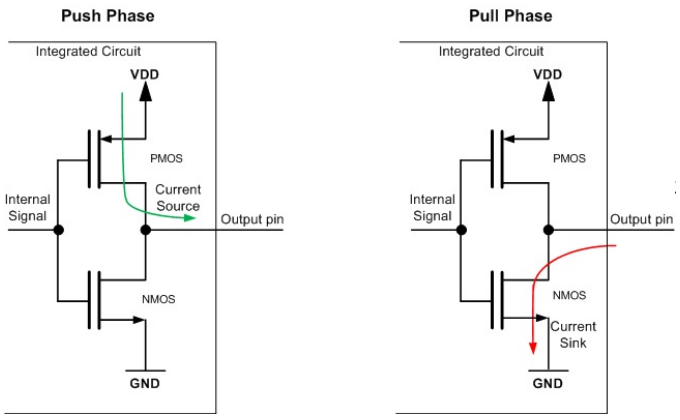
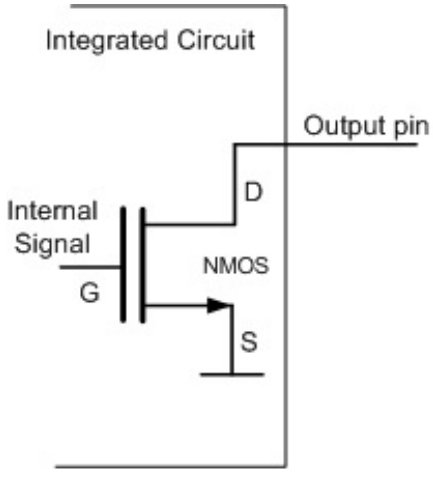


Figure from `https://www.elprocus.com/pull-up-and-pull-down-resistors-with-applications/`.

About open-drain/push pull



- never floating
- unidirectional output
- NPN transistor open $\Longrightarrow$ 1
- PNP transistor open $\Longrightarrow$ 0



- completed by pull-up resistor
- support bus connection
- all transistors closed $\Longrightarrow$ 0
- PNP transistor open $\Longrightarrow$ 1

GPIOx_MODER

MODE Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
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
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- 00 input mode (reset state)
- 01 output mode
- 10 alternate function mode (seen later)
- 11 analog mode (seen later)

GPIO_OTYPER

Output TYPE Register

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
OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

0 = push-pull (reset state)

1 = open-drain

GPIO_OSPEEDR

Output SPEED Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
OSPEEDR15 [1:0]		OSPEEDR14 [1:0]		OSPEEDR13 [1:0]		OSPEEDR12 [1:0]		OSPEEDR11 [1:0]		OSPEEDR10 [1:0]		OSPEEDR9 [1:0]		OSPEEDR8 [1:0]	
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
OSPEEDR7[1:0]		OSPEEDR6[1:0]		OSPEEDR5[1:0]		OSPEEDR4[1:0]		OSPEEDR3[1:0]		OSPEEDR2[1:0]		OSPEEDR1 [1:0]		OSPEEDR0 1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- 00 low speed (reset state)
- 01 medium speed
- 10 high speed
- 11 very high speed

GPIOx_PUPDR

Pull-Up/Pull-Dow Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PUPDR15[1:0]		PUPDR14[1:0]		PUPDR13[1:0]		PUPDR12[1:0]		PUPDR11[1:0]		PUPDR10[1:0]		PUPDR9[1:0]		PUPDR8[1:0]	
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
PUPDR7[1:0]		PUPDR6[1:0]		PUPDR5[1:0]		PUPDR4[1:0]		PUPDR3[1:0]		PUPDR2[1:0]		PUPDR1[1:0]		PUPDR0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- 00 no pull-up, no pull-down
- 01 pull-up
- 10 pull-down
- 11 reserved

How to set up a range of bits?

Problem: setting a value x starting at bit i in word y on l bits.

First try: `(y | (x << i)) & (x << i)`

Example: $y = 01101000$, $x = 01$, $i = 2$, $l = 2$, expected 01100100

y	0	1	1	0	1	0	0	0
x << i	0	0	0	0	0	1	0	0
y | (x << i)	0	1	1	0	1	1	0	0
(x << i)	0	0	0	0	0	1	0	0
... & (x << i)	0	0	0	0	0	1	0	0

Working solution: `(y & ~((1 << l) - 1) << i) | (x << i)`

y	0	1	1	0	1	0	0	0
(1 << l)-1	0	0	0	0	0	0	1	1
~((1 << l)-1) << i	1	1	1	1	0	0	1	1
y&...	0	1	1	0	0	0	0	0
(x << i)	0	0	0	0	0	1	0	0
... & (x << i)	0	1	1	0	0	1	0	0

```
#define SET_BITS(y, i, l, x) \{}
    (((y) & ~((1 << (l))-1) << (i))|((x) << (i)))
```

How get a range of bits?

Problem: how get bit range starting at i and expanding on l bits from x?

Solution: `(x >> i) & ((1 << l)-1)`

Example: $x = 01101000$, $i = 2$, $l = 2$, expected 00000010

`x`	0	1	1	0	1	0	0	0
`x >> i`	0	0	0	1	1	0	1	0
`(1 << l)-1`	0	0	0	0	0	0	1	1
`(x >> i) & ((1 << l)-1)`	0	0	0	0	0	0	1	0

```
#define MASK(l)          ((1 << (l))-1)
#define GET_BITS(x, i, l)  (((x) >> (i)) & MASK(l))
```

Remark: `(1 << l)-1` is often called a mask.

The configuration

Red LED (GPIOD14) programmed as output, push-pull:

```
GPIOD_MODER = SET_BITS(GPIOD_MODER, RED_LED*2, 2, 0b01);
GPIOD_TYPER = GPIOD_TYPER & ~(1 << RED_LED);
GPIOD_PUPDR = SET_BITS(GPIOD_PUPDR, RED_LED*2, 2, 0b01);
```

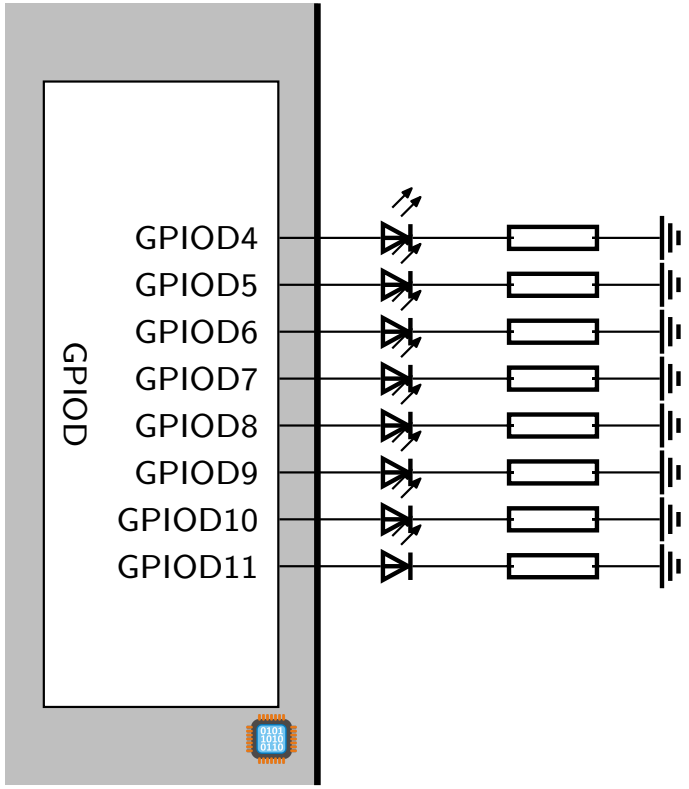
B1, B2 input, pull-down:

```
GPIOA_MODER = SET_BITS(GPIOA_MODER, B1*2, 2, 0b00);
GPIOA_OTYPER = GPIOA_OTYPER & ~(1 << B1);
GPIOA_OPUPDR = SET_BITS(GPIOA_OPUPDR, B1*2, 2, 0b10);
GPIOA_MODER = SET_BITS(GPIOA_MODER, B2*2, 2, 0b00);
GPIOA_OTYPER = GPIOA_OTYPER & ~(1 << B2);
GPIOA_OPUPDR = SET_BITS(GPIOA_OPUPDR, B2*2, 2, 0b10);
```

Exercise (1)

PIO pins 4 to 11 (8 pins) are connected to LED to display a pattern 0b11001010 that we want to shift to the right to achieve a visual effect.
To count time, an empty have to iterate 1000000 times.
The current state of the pattern is stored in global variable pattern.

1. Initialize the PIO in `init()`.
2. Write a function `display()` that displays the pattern on the GPIO.
3. Write the `main()` function.



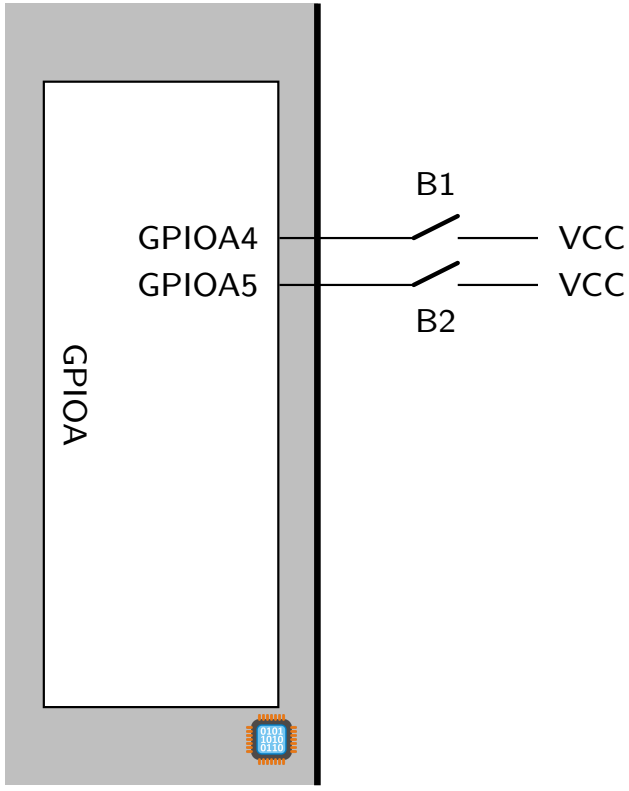

```
char pattern = 0b11001010;
void display();
void init();
```

Exercise (2)

We consider that pin *GPIOA*4 and *GPIO*5 are connected to push buttons named, respectively, *B*1 and *B*2:

- Pushing on *B*1 make the pattern to shift left.
- Pushing on *B*2 make the pattern to shift to right.

Change the endless loop in `main()` that manages the push buttons.



Better timing with Timer

TIM3: timer counter

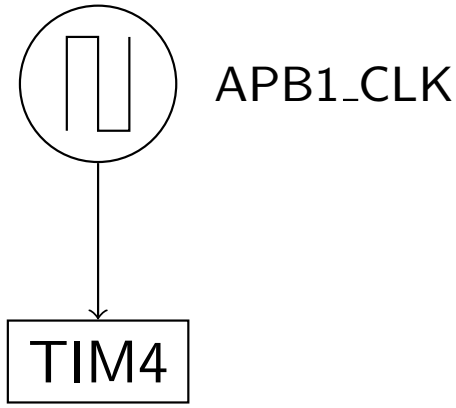
- counter events
- events from APB1_CLK (42MHz)

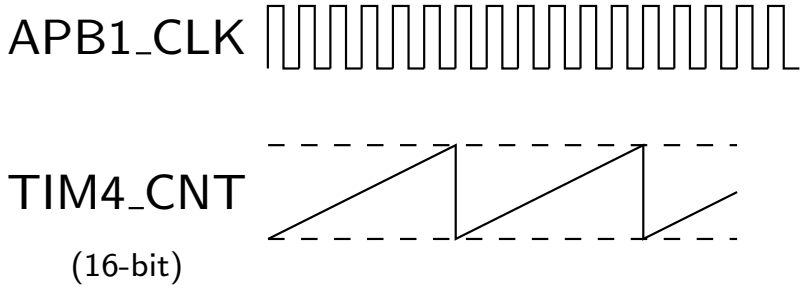
`TIM4_CNT` – event CouNTer register

`TIM4_ARR` – Auto-Reload Register

`TIM4_SR` – Status Register

`TIM4_PSC` – PreSCaler register





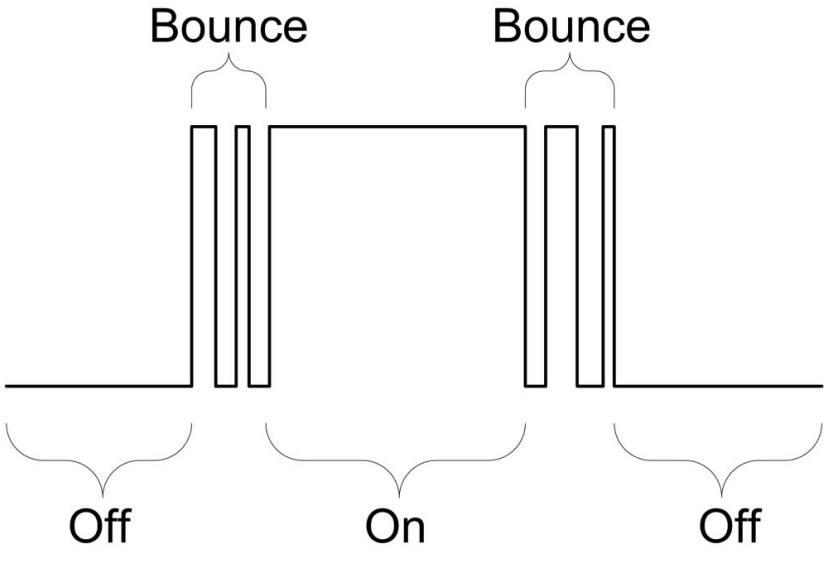
TIM4 Configuration

```
#define WAIT_PSC     1000
#define WAIT_DELAY   (APB1_CLK / PSC)
int HALF_PRIOD = WAIT_DELAY;

void init_TIM4() {

    /* stop the stimer: TIM_CEN set to 0 */
    TIM4_CR1 = 0;

    /* clock = APB1_CLK / 1000 */
    TIM4_PSC = WAIT_PSC;

    /* 0.5s = 42MHz / 2 / 1000 */
    TIM4_ARR = HALF_PERIOD;

    /* reset the counter */
    TIM4_EGR = TIM_UG;

    /* reset the status register */
    TIM4_SR = 0;

    /* Auto-Reload Preloaded Enable,
       Counter ENable */
    TIM4_CR1 = TIM_ARPE;
}
```

Blink with TIM4

```
TIM4_CR1 = TIM4_CR1 | TIM_CEN;
while(1) {

    /* waiting loop and button test */
    while((TIM4_SR & TIM_UIF) == 0) {
        if((GPIOD_IDR & (1 << B1)) != 0)
            b1_state = 1;
        else if(b1_state) {
            b1_state = 0;
            HALF_PERIOD += 200;
        }
        if((GPIOD_IDR & (1 << B2)) != 0)
            b2_state = 1;
        else if(b2_state) {
            b2_state = 0;
            HALF_PERIOD += 200;
        }
        TIM4_ARR = HALF_PERIOD;
        TIM4_SR = 0;
    }

    /* invert LED */
    if((GPIOD_ODR & (1 << RED_LED)) == 0)
        GPIOD_BSRR = 1 << RED_LED;
    else
        GPIOD_BSRR = 1 << (RED_LED << 16);

    // set wait time
    TIM4_ARR = HALF_PERIOD;
}
```

Principle of reality: bouncing



B1

Consider that a normal human user cannot push-release $< 50ms$.

Blink and debounce

```
int last_b1 = 0;
TIM4_EGR = TIM_UG;
while(1) {

    /* waiting loop and button test */
    while((TIM4_SR & TIM_UIF) == 0) {
        if((GPIOD_IDR & (1 << B1)) == 0) {
            b1_state = 1;
            last_b1 = TIM4_CNT;
        }
        else if(b1_state) {
            int now = TIM4_CNT;
            if(now <= last_b1)
                now += DELAY_50MS;
            if(now - last_b1 >= DELAY_50MS) {
                b1_state = 0;
                HALF_PERIOD += 200;
            }
        }
        ...
        TIM4_SR = 0;
    }

    /* invert LED */
    ...
}
```

Exercise (3)

Write a watchdog program:

- It waits 1s without any B1 click.
- If there is a click on B1, waiting restarts.
- If there is no click, it makes the red LED to blink.
- If B1 is then clicked, the red RED is switched off and the process restarts.
- If, after 1 more second of blinking, the button is not clicked, the program freezes and keep the red LED lighted on.

Hint: the behaviour of the program is very complex. Could you imagine a clever representation of tghis behaviour.

Table of Contents

Polling approach

```
while(1) {
    if(test_R1)
        action1();
    if(test_R2)
        action2();
    if(test_R3)
        action3();
    ...
}
```

Pros:

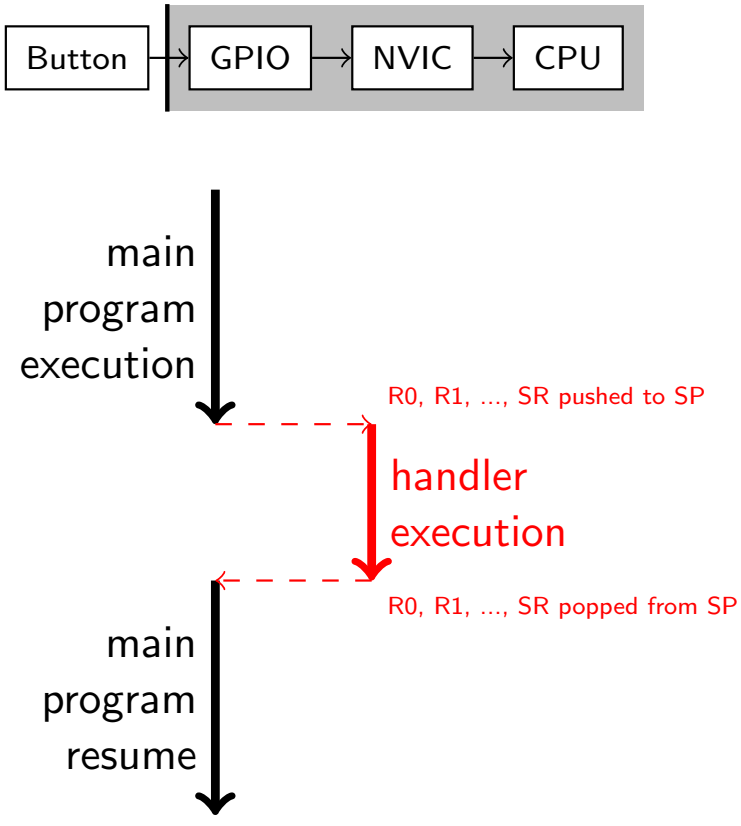
- easy to write
- easy to understand
- easy to debug

Cons:

- more and more tests $\Longrightarrow$ increase action latency

Interrupts: principles

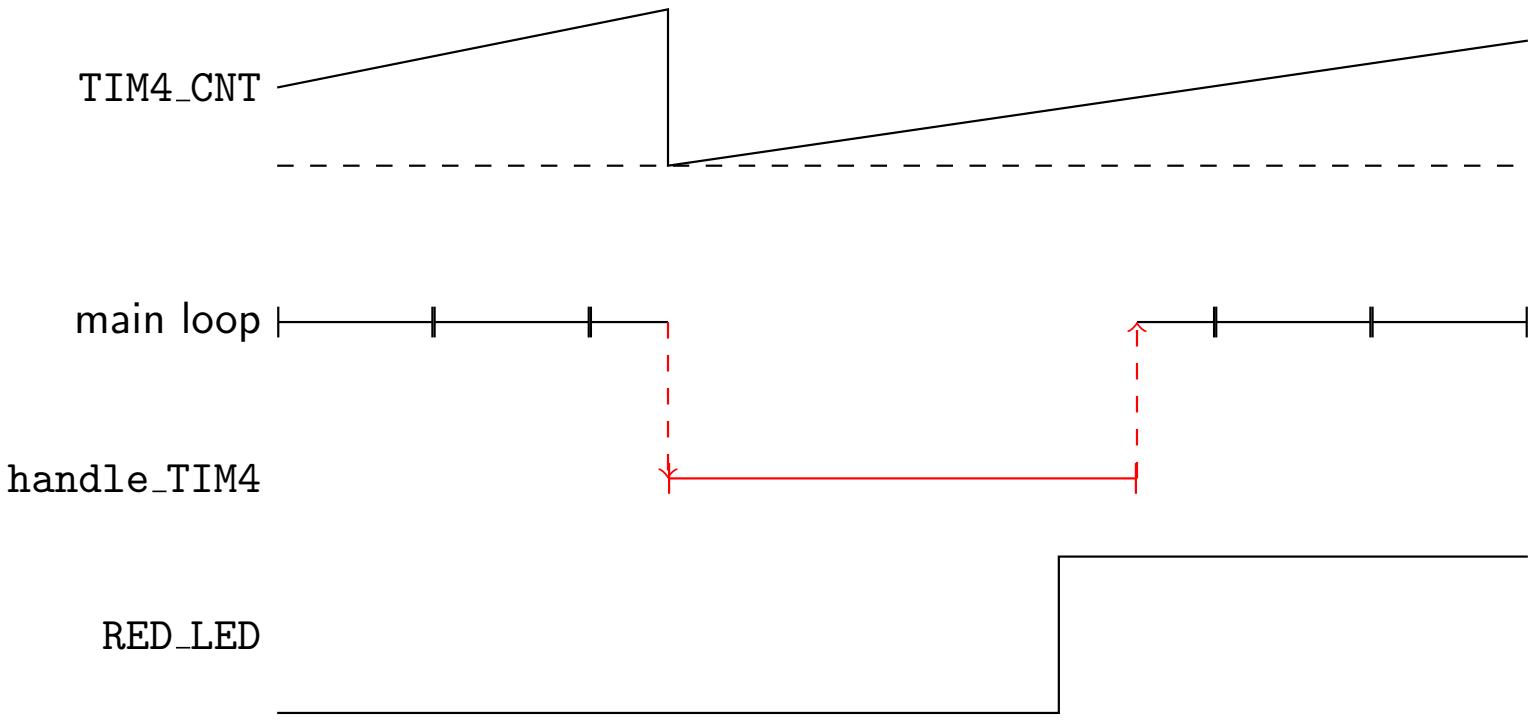
- main program executes (main loop)
- change in the input (GPIO button)
- IRQ signal (Interrupt ReQuest) from GPIO
- IRQ line passed to IRQ controller
- IRQ line passed to core
- execution stopped, context saved
- selection of IRQ handler
- execution of IRQ handler
- context restored, execution resumed



Blinking with Interrupt

```
void handle_TIM4() {

    /* set time */
    TIM4_ARR = HALF_PERIOD;

    /* invert LED */
    if((GPIOD_ODR & (1 << RED_LED)) == 0)
        GPIOD_BSRR = 1 << RED_LED;
    else
        GPIOD_BSRR = 1 << (RED_LED << 16);

    /* release TIM4 */
    TIM4_SR &= ~TIM_UIF;
}

int main() {

    // initialization
    ...

    // main loop
    while(1) {

        // B1 test
        ...

        // B2 test
        ...

    }
}
```

Interrupt Timetable



How to configure?

- before any change, disable the IRQ in the core/device
  $\Longrightarrow$ prevent spurious/uncontrolled IRQ
- clear any pending IRQ in the device driver
- provide the address of handler in IRQ vector table (function pointer)
- configure/enable IRQ in the device driver
- configure/enable IRQ in the NVIC
- enable IRQ in the core (disabled by default)

Cortex-M4 IRQ Vector Table

Basically mapped at address 0x0000 0000 at startup (can be moved after).

Address	Name	Description
0000 0000	SP_INIT	Initial stack address
0000 0004	Reset	Startup address
0000 0008	NMI	Non-Maskable Interrupt
0000 000C	Hard Fault	
0000 0010	MemManage	Memory management
0000 0014	BusFault	Memory access fault
0000 0018	UsageFault	Undefined instruction
0000 001C-20	Reserved	
0000 002C	SVCall	System Call
0000 0030	Debug Monitor	
0000 0034	Reserved	
0000 0038	PendSV	Pending System Call
0000 003C	SysTick	System tick timer
00000 0040-184	I/O IRQ	

I/O IRQ: number $N \implies$ address $= 0x40 + N \times 4$

Enabling IRQ in the Core

```
#define ENABLE_IRQS __asm("CPSIE I")

#define DISABLE_IRQS __asm("CPSIE I")
```

Hint: can be used to implement critical area.

IRQ Controller

NVIC (Nested Vectored Interrupt Controller)

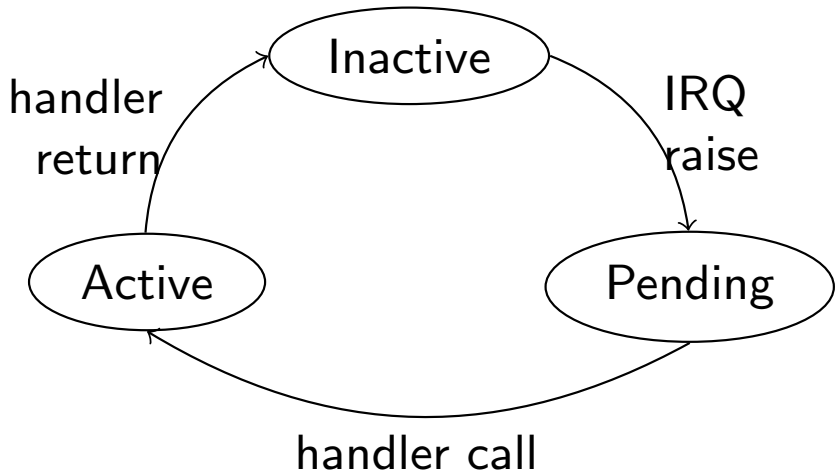
`NVIC_ISER0-7`	enable interrupts
`NVIC_ICER0-7`	disable interrupts
`NVIC_ISPR0-7`	set pending interrupts (software)
`NVIC_ICPR0-7`	clear pending interrupts
`NVIC_IABR0-7`	active interrupts
`NVIC_IPR0-59`	interrupt priorities

Accessing to the interrupt number N (32-bits registers):

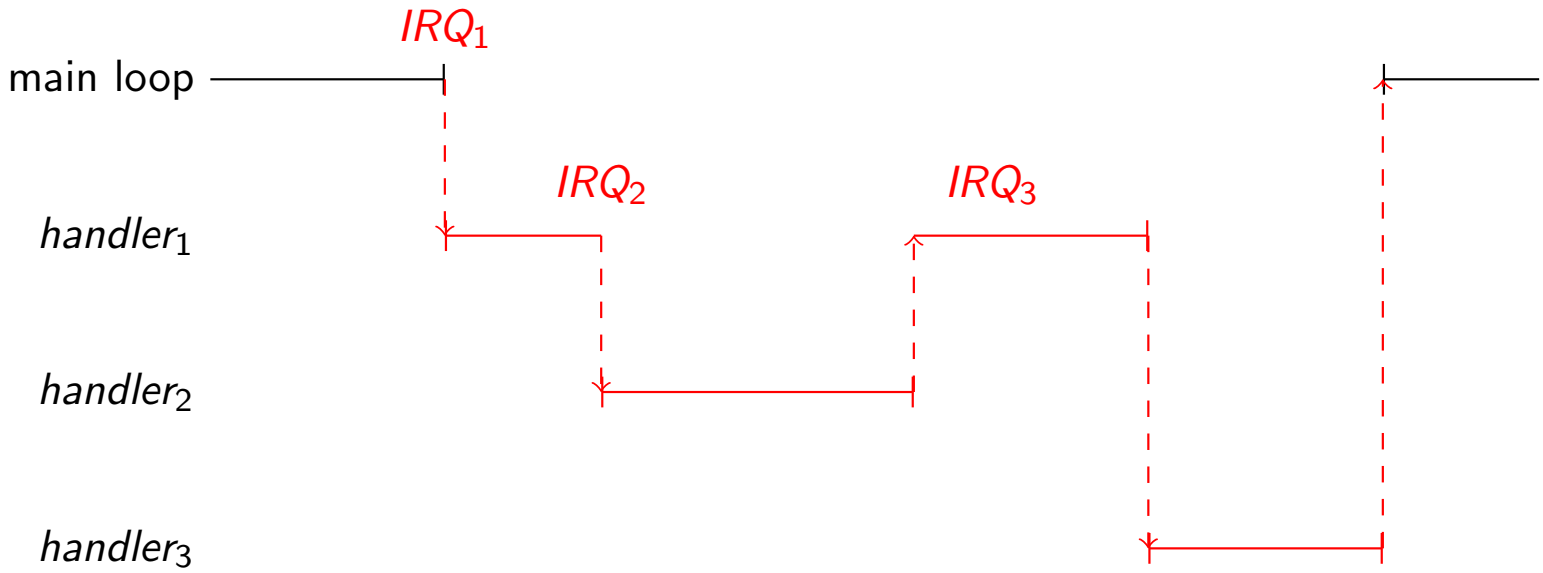
```
NVIC_XXX(N >> 5) = 1 << (N & 0x1f);
```

Setting the priority P (only bits 7-4 are used):

```
NVIC_IPR(N) = P;
```



Prioirities in NVIC



$$P_{handler_2} > P_{handler_1} \geq P_{handler_3}$$

Configuration of TIM4

```
DISABLE_IRQS;        // or just disable device

// configure NVIC
NVIC_ICER(TIM4_IRQ >> 5) |= 1 << (TIM4_IRQ & 0X1f);
NVIC_IRQ(TIM4_IRQ) = (uint32_t)handle_TIM4;
NVIC_IPR(TIM4_IRQ) = 0;

// purge pending IRQ
NVIC_ICPR(TIM4_IRQ >> 5) |= 1 << (TIM4_IRQ & 0X1f);
NVIC_ISER(TIM4_IRQ >> 5) |= 1 << (TIM4_IRQ & 0X1f);

// configure TIM4
TIM4_CR1 &= ~TIM_CEN;
TIM4_PSC = WAIT_PSC;
TIM4_ARR = WAIT_DELAY;
TIM4_EGR = TIM_UG;
TIM4_SR = 0;
TIM4_CR1 = TIM_ARPE;
TIM4_SR &= ~TIM_UIF;
TIM4_DIER = TIM_UIE;

// starting all
ENABLE_IRQS;
TIM4_CR1 |= TIM_CEN;
```

Writing Handlers

Interrupt handler should be short
$\Longrightarrow$ do not block other interrupts for too long!

When an interrupt handler is too big:

- transfer some processing in the main loop
- use global variable (boolean, etc) to trigger it from the IRQ handler

Warning: as interrupts preempts the main loop
$\Longrightarrow$ close issues as thread/thread communication

```
int TIM4_triggered = 0;

void handle_TIM4() {
    TIM4_ARR = HALF_PERIOD;
    TIM4_triggered = 1;
    TIM4_SR &= ~TIM_UIF;
}

int main() {
    ...
    // main loop
    while(1) {

        /* invert LED */
        if(TIM4_triggered) {
            TIM4_triggered = 0;
            if((GPIOD_ODR & (1 << RED_LED))
                GPIOD_BSRR = 1 << RED_LED;
            else
                GPIOD_BSRR =
                    1 << (RED_LED << 16);
        }


        // B1 test

        // B2 test

    }
}
```

Exercise

Goal: Morse code emitter.

- dot – 100ms
- dash – 300ms
- space – 100ms
- output to `GREEN_LED`

```
#define WAIT_DELAY   APB1_CLK/PSC/10

#define END      0
#define DOT      1
#define DASH     2

int message[] = {
    DOT, DOT, DOT,
    DASH, DASH, DASH,
    DOT, DOT, DOT
}    // SOS
```

Using the definitions on the left, write the code for:

1. configuring TIM4 with a period of 100ms,
2. writing TIM4 handler to emit the message,
3. configuring the TIM4 interrupt.

Debugging with Interrupts

- interrupts supports break-points
- but interrupts are caused by external devices (sensors)
- CPU is stopped (and possibly device drivers)
- but not outside world
- often: break, examine and restart

Interrupts are hard to debug: they are asynchronous!

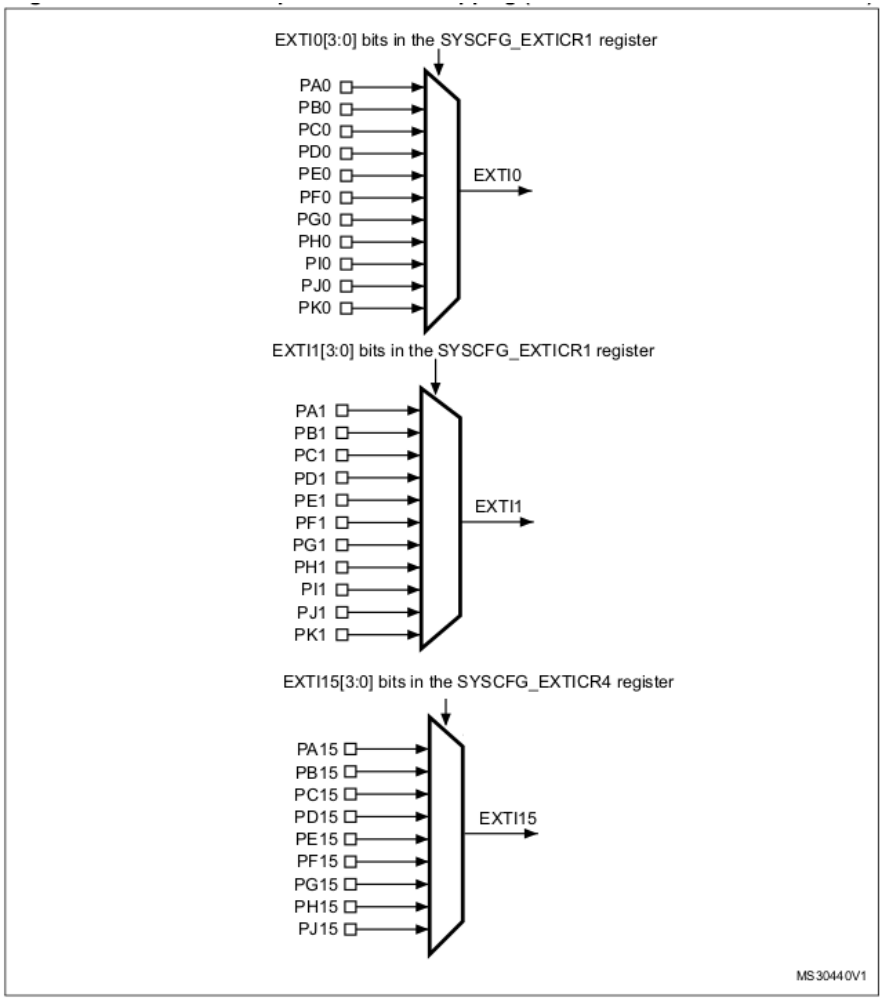
Hints to debug interrupts:

- IRQ is never raised
  check the activatio of IRQ: core level, NVIC level, device level
- after first IRQ, it is re-raised even it is not supposed to be
  in the handler, IRQ reset is not performed in the device
- you get BusFault, UsageFault
  there is a data/code error:
  break-point in the handler, step-by-step until exception raise

Final hint: the hardware registers can be displayed from the debugger!

Interrupts with GPIO

In STM32F4, no support for IRQ in GPIO $\Longrightarrow$ use of EXTI (EXTernal Interrupts):



At most, 16 IRQ can generated over $11 \times 16 = 176$!

Mapping to Interrupts

Address	Name	EXTI lines
0000 0058	EXTI0	0
0000 005C	EXTI1	1
0000 0060	EXTI2	2
0000 0064	EXTI3	3
0000 0068	EXTI4	4
0000 009C	EXTI9_5	9, ..., 5
0000 00E0	EXTI15_10	15, ..., 10

EXTI9_5 and EXTI15_10 require polling!

Mapping GPIO pins $\iff$ EXTI lines must be chosen (application dependent):

- B0 $\rightarrow$ EXTI0
- B1 $\rightarrow$ EXTI1

Configuring EXTI

`EXTI_IMR`	Interrupt Mask Register
`EXTI_RTSR`	Rising Trigger Selection Register
`EXTI_FTSR`	Falling Trigger Selection Register
`EXTI_SWIER`	SoftWare Interrupt Event Register
`EXTI_PR`	Pending Register (to reset after IRQ)
`SYSCFG_CR`i	EXTI Configuration Register

1 bit / EXTI line

`SYSCFG_CR`i: 4 bits/line, 4 line/register, $i \in [1, 4]$

Configuring the IRQ

```
DISABLE_IRQS;

// configure EXTI
SET_BITS(SYSCFG_CR1, 0, 4, 0);   // 0 = GPIOA0
EXTI_RTSR |= 1 << 0;
EXTI_FTSR |= 1 << 0;
EXTI_IMR |= 1 << 0;
EXTI_IPR |= 1 << 0;

// configure NVIC
NVIC_ICER(EXTI0_IRQ >> 5) |= 1 << (EXTI0_IRQ & 0X1f);
NVIC_IRQ(EXTI0_IRQ) = (uint32_t)handle_B0;
NVIC_IPR(EXTI0_IRQ) = 0;
NVIC_ICPR(EXTI0_IRQ >> 5) |= 1 << (EXTI0_IRQ & 0X1f);
NVIC_ISER(EXTI0_IRQ >> 5) |= 1 << (EXTI0_IRQ & 0X1f);

ENABLE_IRQS;
```

Writing the Handler

```
int last_b1 = 0;

void handle_B1() {

    // handle push/release button
    if((GPIOD_IDR & (1 << B1)) == 0) {
        b1_state = 1;
        last_b1 = TIM4_CNT;
    }
    else if(b1_state) {
        int now = TIM4_CNT;
        if(now <= last_b1)
            now += DELAY_50MS;
        if(now - last_b1 >= DELAY_50MS) {
            b1_state = 0;
            HALF_PERIOD += 200;
        }
    }

    // reset the IRQ
    EXTI_IPR |= 1 << 0;
}
```

Exercise

Table of Contents

Connclusion

In this lesson,

- how to program input/output in a microcontroller,
- basics to drive GPIO and timer,
- driving by polling,
- driving by interrupts,
- principle of interaction with real world.

In the next lessons,

- how to drive sensors and actuators,
- how to manage the memory (at compilation-time and at run-time),
- SysML – UML-like application model adapted to ES.