

# Embedded System Course 2021-22

---

**C4µC**

**C/C++ Programming for Microcontrollers**

---

Lecturers: Ing. Marco Santic

[marco.santic@univaq.it](mailto:marco.santic@univaq.it)

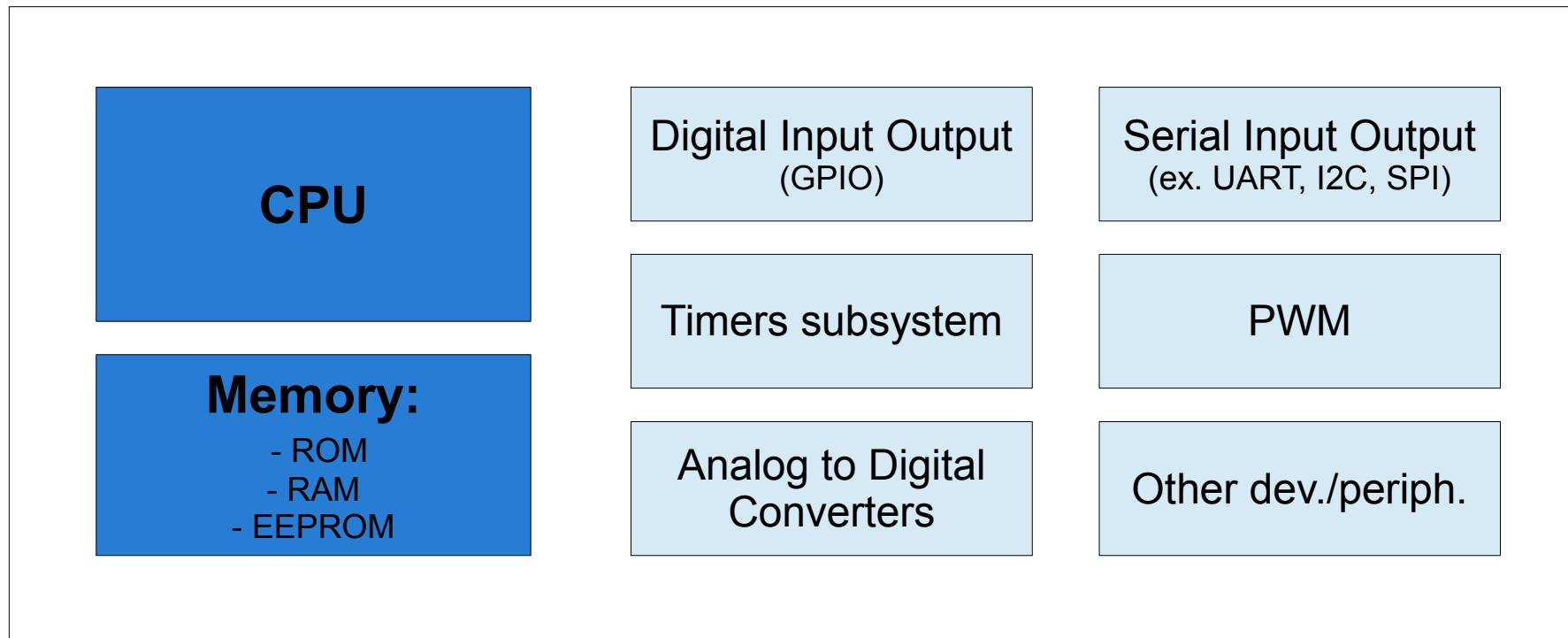
# C4μC



- Introduction
  - Atmega328P
  - Toolchain
  - IDE
  - Examples
  - C language recap
- Basic functionalities
  - GPIO
  - Timers
  - Interrupt
  - Examples
- Comm. Interfaces
  - I2C
  - UART
  - SPI
  - Examples
- Other peripherals
  - ADC
  - GPIO bitbanging
  - Examples

# C4μC - Basic functionalities

- μController block diagram:



- Devices/peripherals implement some basic functionalities
- CPU needs to communicate with (interface to) those devices/peripherals

What is the difference between a device and a peripheral??

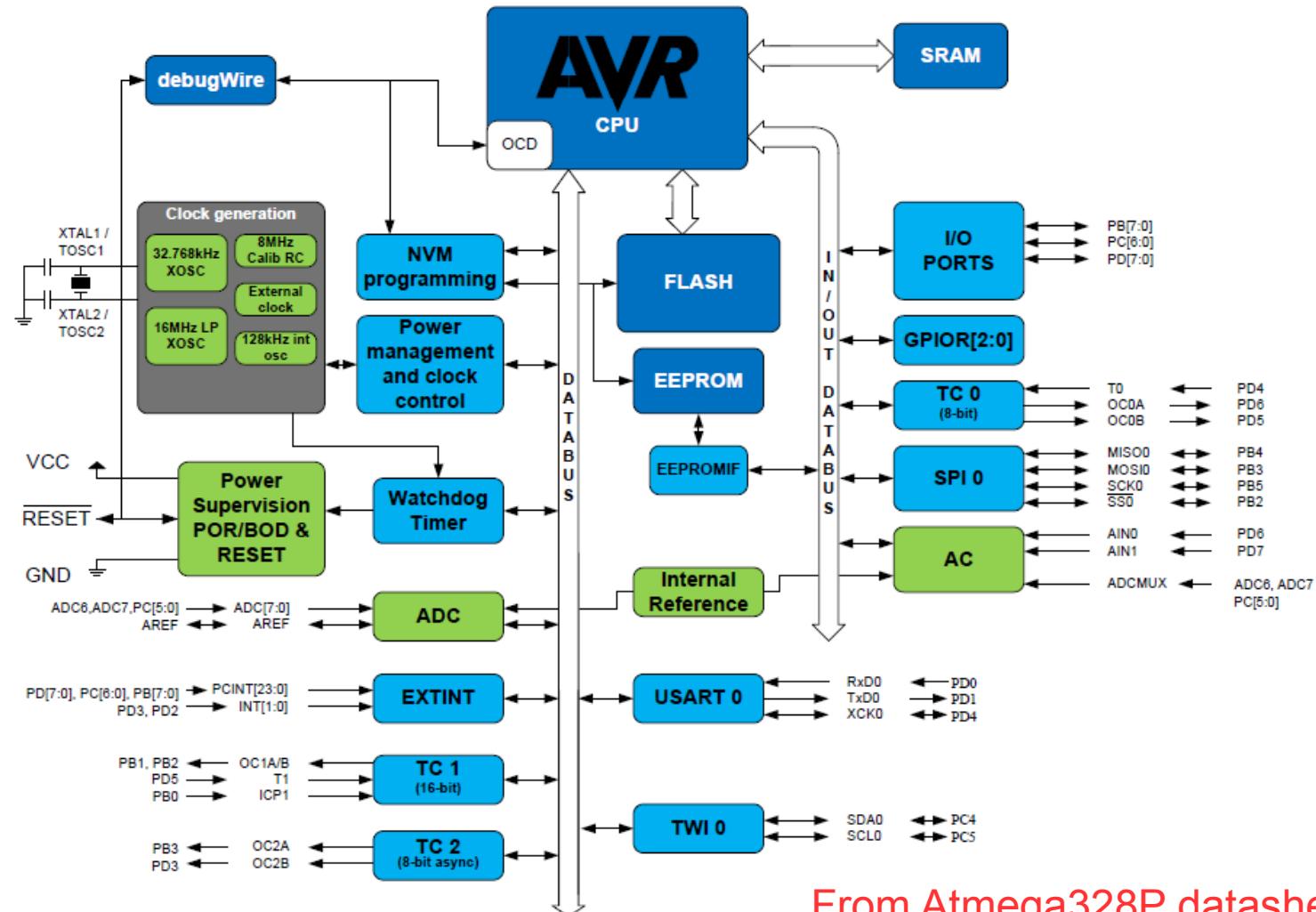
# C4μC - Basic functionalities

- To interface with devices, you have seen:
  - Interfacing:
    - Port-based I/O
    - Bus-based I/O (addr., data, ctrl. Registers)
      - Memory Mapped I/O (accessible using common instructions)
      - Standard I/O (+1 pin, access via special instr. IN, OUT)
  - And to be aware of "some new data":
    - Polling
    - Interrupt
  - Let's move to our reference μC...



# C4μC - Basic functionalities

- μC AVR Atmega328 block diagram



From Atmega328P datasheet (p.13)

# C4μC - Basic functionalities



- Datasheet reports all information about:
  - General architecture
  - Devices' characteristics
  - Power management
  - Pin description and configuration
  - ... (a lot more)

Datasheet  
is a friend  
of yours!!!

## Page 27 (about Status Register):

*When addressing I/O Registers as data space using LD and ST instructions, the provided offset must be used.*

*When using the I/O specific commands IN and OUT,...*

## Page 35 (about SRAM Data Memory):

*... device is a complex microcontroller with more peripheral units than can be supported within the 64 locations reserved in the Opcode for the IN and OUT instructions. For the Extended I/O space,...*

## Page 155 (about Timer/Counter):

*The 16-bit counter is mapped into two 8-bit I/O memory locations: Counter High (TCNT1H) containing the upper eight bits of the counter, and Counter Low (TCNT1L) containing the lower eight bits...*

# C4μC - Basic functionalities

- Hereafter a top-down approach will be used to explore the microcontroller:
  - See an example in the IDE
  - Inspect the code
  - Run through the source files
  - Find something "nice"
  - Correlate it with datasheet information
  - (Sometimes explore the ASM)



# C4μC - Basic functionalities

- Hereafter a top-down approach will be used to explore the microcontroller:
  - See an example in the IDE
  - Inspect the code
  - Run through the source files
  - Find something "nice"
  - Correlate it with datasheet information
  - (Sometimes explore the ASM)



# C4μC - GPIO

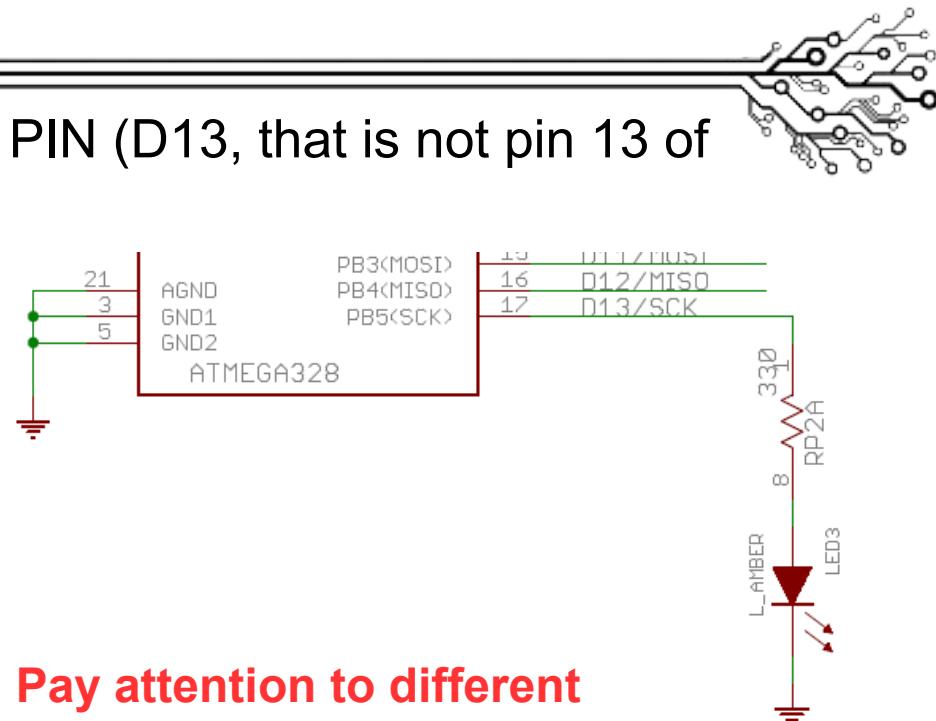
- Open the Arduino IDE
- Set the board and the communication port
  - Menu Tools --> Board --> (Uno, Nano, ...)
  - Menu Tools --> Port --> (COM\*, /dev/ttyACM\*, ...)
- Load Blink example
  - Menu File --> Examples --> Basics --> Blink

```
// the setup function runs once when you press reset or power the board
void setup() {
    // initialize digital pin 13 as an output.
    pinMode(13, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
    digitalWrite(13, HIGH);      // turn the LED on (HIGH is the voltage level)
    delay(1000);                // wait for a second
    digitalWrite(13, LOW);       // turn the LED off by making the voltage LOW
    delay(1000);                // wait for a second
}
```

# C4μC - GPIO

- To drive the blinkin' LED we are using a PIN (D13, that is not pin 13 of Arduino board)
- If we have a look at the schematic...
  - It is called **D13**
  - It is connected to pin 16 of the Arduino board
  - It corresponds to pin 17 of μC...
  - ...and its (I/O) name is **PB5**
- It is a GPIO pin (on PORT B)
  - In the code it is initialized as an OUTPUT
  - `pinMode(..., ...)` is a function of Arduino (IDE) core library



# C4μC - GPIO

- Go to: <your IDE path>\hardware\arduino\avr\cores\arduino\Arduino.h
- In that header file, it is defined (on line 125, or a bit after...):

```
void pinMode(uint8_t, uint8_t);
```
- Go to: <your IDE path>\hardware\arduino\avr\cores\arduino\wiring\_digital.c
- In that source file, it is implemented (on lines 31-61):

```
void pinMode(uint8_t pin, uint8_t mode)
{
    uint8_t bit = digitalPinToBitMask(pin);      ← Arduino.h (170)
    uint8_t port = digitalPinToPort(pin);         ← Arduino.h (169)
    volatile uint8_t *reg, *out;
    ...
    reg = portModeRegister(port);                ← Arduino.h (175)
    out = portOutputRegister(port);              ← Arduino.h (173)
    ...
} else { //mode = OUTPUT
    uint8_t oldSREG = SREG;
        cli();   ← ... we will see later...
    *reg |= bit;
    SREG = oldSREG;
}
```

# C4μC - GPIO

- In: <your IDE path>\...\Arduino.h (lines 169-175)

```
#define digitalPinToPort(P) ( pgm_read_byte( digital_pin_to_port_PGM + (P) ) )
#define digitalPinToBitMask(P) ( pgm_read_byte( digital_pin_to_bit_mask_PGM +
(P) ) )
#define digitalPinToTimer(P) ( pgm_read_byte( digital_pin_to_timer_PGM + (P) ) )
#define analogInPinToBit(P) (P)
#define portOutputRegister(P) ( (volatile uint8_t *)
(pgm_read_word( port_to_output_PGM + (P)))) )
#define portInputRegister(P) ( (volatile uint8_t *)
(pgm_read_word( port_to_input_PGM + (P)))) )
#define portModeRegister(P) ( (volatile uint8_t *)
(pgm_read_word( port_to_mode_PGM + (P)))) )
```

- A lot of other functions (or define)...

- `pgm_read_byte` ( <your IDE path>\hardware\tools\avr\avr\include\avr\pgmspace.h , line 1046 )  

```
#define pgm_read_byte(address_short) pgm_read_byte_near(address_short)
```
- `digital_pin_to_port_PGM` ( <your IDE path>\...\Arduino.h , line 159 )  

```
extern const uint8_t PROGMEM digital_pin_to_port_PGM[];
```
- `digital_pin_to_bit_mask_PGM` ( <your IDE path>\...\Arduino.h , line 161 )  

```
extern const uint8_t PROGMEM digital_pin_to_bit_mask_PGM[];
```

# C4μC - GPIO

- Have a look at:

<your IDE path>\hardware\arduino\avr\variants\standard\pins\_arduino.h  
(lines 134-155)

```
const uint8_t PROGMEM digital_pin_to_port_PGM[] = {
    PD, /* 0 */
    PD,
    PD,
    PD,
    PD,
    PD,
    PD,
    PD,
    PB, /* 8 */
    PB,
    PB,
    PB,
    PB,
    PB,
    PB, ← we are using D13... at index 13 there is PB
    PC, /* 14 */
    PC,
    PC,
    PC,
    PC,
    PC,
    PC,
};

};
```



# C4μC - GPIO

- Have a look at:

<your IDE path>\hardware\arduino\avr\variants\standard\pins\_arduino.h  
(lines 157-178)

```
const uint8_t PROGMEM digital_pin_to_bit_mask_PGM[] = {
    _BV(0), /* 0, port D */
    _BV(1),
    _BV(2),
    _BV(3),
    _BV(4),
    _BV(5),
    _BV(6),
    _BV(7),
    _BV(0), /* 8, port B */
    _BV(1),
    _BV(2),
    _BV(3),
    _BV(4),
    _BV(5),      ← we are using D13... at index 13 there is _BV(5)
    _BV(0), /* 14, port C */
    _BV(1),
    _BV(2),
    _BV(3),
    _BV(4),
    _BV(5),
};
```



# C4μC - GPIO

- We could explore more, but... stop!  
Some considerations:
  - We have seen where the core library is
  - Without a real IDE with source navigation, it is hard and time-consuming
  - We have understood how to search and how the inclusion of different header files (for different platforms) works
  - We just had a confirmation of D13 <--> PB5
- Go back to see which registers are used...  
... with a little trick

# C4μC - GPIO

- Let's use a...
- Serial interface for output

```
void setup() {  
    Serial.begin(115200); while (!Serial);  
  
    uint8_t our_pin = 13;  
  
    uint8_t _bit = digitalPinToBitMask(our_pin);  
    uint8_t _port = digitalPinToPort(our_pin);  
    volatile uint8_t *reg, *out;  
  
    reg = portModeRegister(_port);  
    out = portOutputRegister(_port);  
  
    Serial.print("bit = "); Serial.print((uint16_t)_bit); Serial.print("\n");  
    Serial.print("port = "); Serial.print((uint16_t)_port); Serial.print("\n");  
    Serial.print("reg = "); Serial.print((uint16_t)reg); Serial.print("\n");  
    Serial.print("out = "); Serial.print((uint16_t)out); Serial.print("\n");  
}  
  
void loop() {  
    // put your main code here, to run repeatedly:  
}
```

The output:  
bit = 32  
port = 2  
reg = 36  
out = 37

# C4μC - GPIO

- In: <your IDE path>\...\wiring\_digital.c (on lines 31-61):

```
void pinMode(uint8_t pin, uint8_t mode)
{
    uint8_t bit = digitalPinToBitMask(pin);
    uint8_t port = digitalPinToPort(pin);
    volatile uint8_t *reg, *out;
    ...
    reg = portModeRegister(port);
    ...
} else { //mode = OUTPUT
    uint8_t oldSREG = SREG;
    cli();
    *reg |= bit;
    SREG = oldSREG;
}
}
```

**bit = 32  
port = 2  
reg = 36  
out = 37**

- So, `*reg |= bit;` //at address 36 (0x24) set the 5<sup>th</sup> bit (32 = 2<sup>5</sup>)

## 35. Register Summary

Offset	Name	Bit Pos.	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0
0x23	PINB	7:0								
0x24	DDRB	7:0	DDRB7	DDRB6	DDRB5	DDRB4	DDRB3	DDRB2	DDRB1	DDRB0
0x25	PORTB	7:0	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0

Datasheet,  
page 428

# C4μC - GPIO

- Register at SRAM address 0x24 is DDRB
- You can read datasheet:
  - I/O-Ports overview (p.97)
  - Configuring the Pin (p.98)

*Each port pin consists of three register bits: DD $xn$ , PORT $xn$ , and PIN $xn$ . As shown in the Register Description, the DD $xn$  bits are accessed at the DDRx I/O address, the PORT $xn$  bits at the PORTx I/O address, and the PIN $xn$  bits at the PINx I/O address.*

*The DD $xn$  bit in the DDRx Register selects the direction of this pin. If DD $xn$  is written to '1', Px $n$  is configured as an output pin. If DD $xn$  is written to '0', Px $n$  is configured as an input pin.*

...

- Note the example code in the following pages, in the assembly you can see the use of **IN** and **OUT** opcodes to access the register
- Port B Data Direction Register (p.117)

What do you expect from digitalWrite() function???

# C4μC - GPIO

- Blink code

```
// the loop function runs over and over again forever
void loop() {
    digitalWrite(13, HIGH);      // turn the LED on (HIGH is the voltage level)
    delay(1000);                // wait for a second
    digitalWrite(13, LOW);       // turn the LED off by making the voltage LOW
    delay(1000);                // wait for a second
}
```

- In Arduino.h and wiring\_digital.c, you have prototype and implementation of digitalWrite(), accessing the register PORTB (0x25)

```
void digitalWrite(uint8_t pin, uint8_t val){
    ...
    uint8_t bit = digitalPinToBitMask(pin);
    uint8_t port = digitalPinToPort(pin);
    volatile uint8_t *out;
    ...
    out = portOutputRegister(port);
    ...
    if (val == LOW) {
        *out &= ~bit; //at address 37 (0x25) clear the 5th bit (32 = 25)
    } else {
        *out |= bit; //at address 37 (0x25) set the 5th bit (32 = 25)
    }
    ...
}
```

As before:  
bit = 32  
port = 2  
out = 37

# C4μC - GPIO

- We can have the same result rewriting a "purged" code:

```
// include only avr libraries
#include <avr/io.h>
#include <util/delay.h>

#define BLINK_DELAY_MS 1000

int main(void) {
    /* set pin 5 of PORTB for output */
    DDRB |= _BV(DDB5);

    while(1) {
        /* set pin 5 high to turn led on */
        PORTB |= _BV(PORTB5);
        _delay_ms(BLINK_DELAY_MS);

        /* set pin 5 low to turn led off */
        PORTB &= ~_BV(PORTB5);
        _delay_ms(BLINK_DELAY_MS);
    }
    return 0;
}
```

# C4μC - GPIO

- Note that we have done a slight different access to the same registers:

- In including <avr/io.h>, we have included **avr/iom328p.h**

```
// from <avr/iom328p.h>

#define DDRB _SFR_IO8(0x04)
...
#define DDB5 5
...
#define PORTB _SFR_IO8(0x05)
...
#define PORTB5 5
```

- The explanation is again in the datasheet (p.35), where:  
*"The five different addressing modes for the data memory cover:..."*

- In <avr/srf\_defs.h>, we have:

```
#define __SFR_OFFSET 0x20
#define _SFR_IO8(io_addr) ((io_addr) + __SFR_OFFSET)
```

Which addressing mode is direct and which one indirect??

# C4μC - GPIO

- A GPIO can be used, of course, as an INPUT

```
// Modified: Examples -> Digital -> Button
const int buttonPin = 2;          // the number of the pushbutton pin
const int ledPin = 13;           // the number of the LED pin
int buttonState = 0;            // variable for reading the pushbutton status

void setup() {
    pinMode(ledPin, OUTPUT);      // init the LED pin as an output
    pinMode(buttonPin, INPUT_PULLUP); // init the pushbutton pin as an inputpulled-up
                                    // pin HIGH if button is not pressed
}

void loop() {
    buttonState = digitalRead(buttonPin); // read the state of the pushbutton value

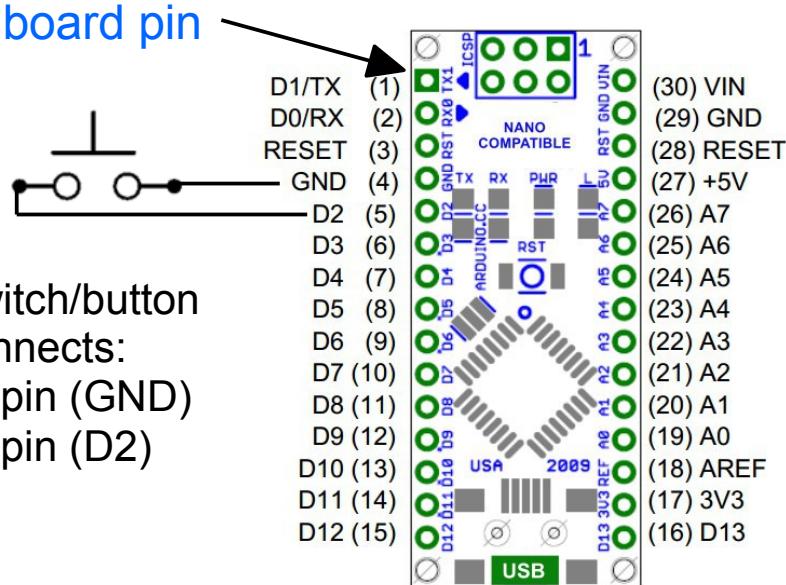
    // check if the pushbutton is pressed.
    if (buttonState == LOW) {           // If it is, the buttonState is LOW
        digitalWrite(ledPin, HIGH);    // turn LED on
    }
    else {
        digitalWrite(ledPin, LOW);     // turn LED off
    }
}
```

Exercise:  
do the same using  
DDRB, PORTB and ...?

# C4μC - GPIO

- Variation from the example:
- The pin D2 is not just an INPUT, but a pulled-up input (INPUT\_PULLUP)
  - If not shorten to GND by the button press, it is read as HIGH
- The logic of turning on the LED is adapted

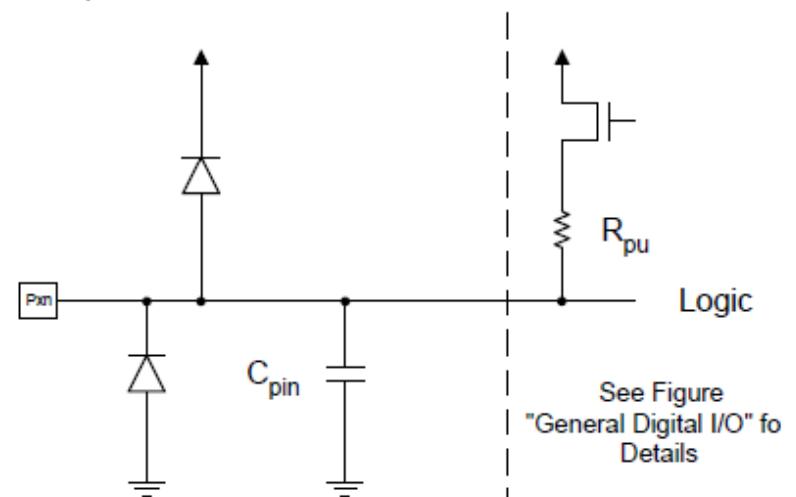
1<sup>st</sup> board pin



Switch/button  
connects:  
4<sup>th</sup> pin (GND)  
5<sup>th</sup> pin (D2)

USB side

Figure 18-1. I/O Pin Equivalent Schematic



See Figure  
"General Digital I/O" fo  
Details

Datasheet p.97

# C4μC - GPIO

- The previous application turns the LED on when the button is pressed
- But it does only that!!! Let's add some job...

```
void loop() {  
  
    delay(5000);    // Suppose here the micro is working hard  
  
    // check if the pushbutton is pressed.  
    if (digitalRead(buttonPin) == LOW) {  
        digitalWrite(ledPin, HIGH);    // turn LED on  
        delay(2000);  
    }  
    digitalWrite(ledPin, LOW);    // turn LED off  
  
    delay(5000);    // Still working hard  
}
```

- It's not easy to catch the moment when we can press the button
  - Can we wait for the input??



# C4μC - GPIO

- A variation with BLOCKING INPUT (our code is blocking)

```
void loop() {  
  
    delay(5000);    // Suppose here the micro is working hard  
  
    digitalWrite(ledPin, HIGH); // a short blink  
    delay(200);        // to know app  
    digitalWrite(ledPin, LOW); // is waiting...  
  
    // wait until the pushbutton is pressed.  
    while (digitalRead(buttonPin) == HIGH) {}  
  
    digitalWrite(ledPin, HIGH); // turn LED on  
    delay(2000);  
    digitalWrite(ledPin, LOW); // turn LED off  
  
    delay(5000);    // Still working hard  
}
```

- This is an example of BLOCKING instruction/device

- Sometime it is needed
  - Sometime it is unwanted

It depends on the logic  
of your application!!!

# C4μC - Timers

- We have used the function *delay(int millis)*
- What this function is built upon?
- The Atmega328 has **3 timers**: 2 x 8bit (TC0, TC2) and 1 x 16bit (TC1) (they differ in max value before overflowing: 255 or 65535)
- Other than prescaler config., control and counter registers there are also a set of output compare and input capture registers
- A big part of the datasheet explains the functionalities (Ch.19-22)  
*You can read the datasheet when you need a low-level control...*
- Or trust some info found online:  
*In the Arduino world timer0 is been used for the timer functions, like delay(), millis() and micros(). If you change timer0 registers, this may influence the Arduino timer function. So you should know what you are doing.*

# C4μC - Timers

- A schematic of timer 8bit and timer 16bit



Figure 19-1. 8-bit Timer/Counter Block Diagram

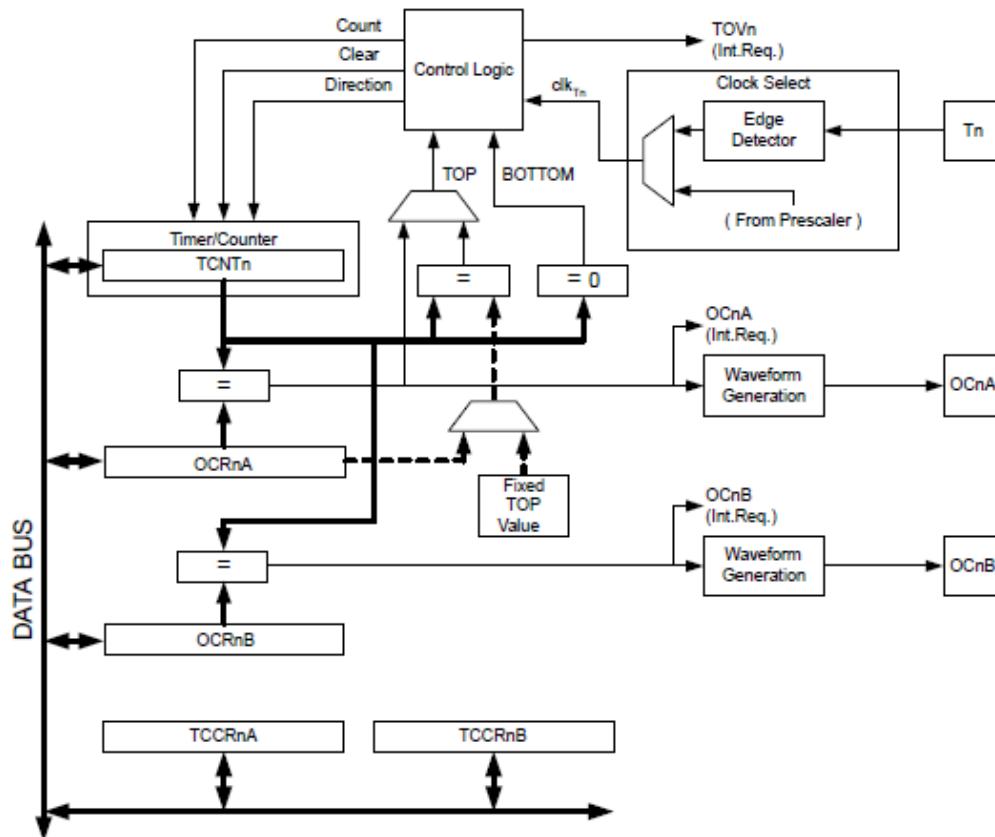
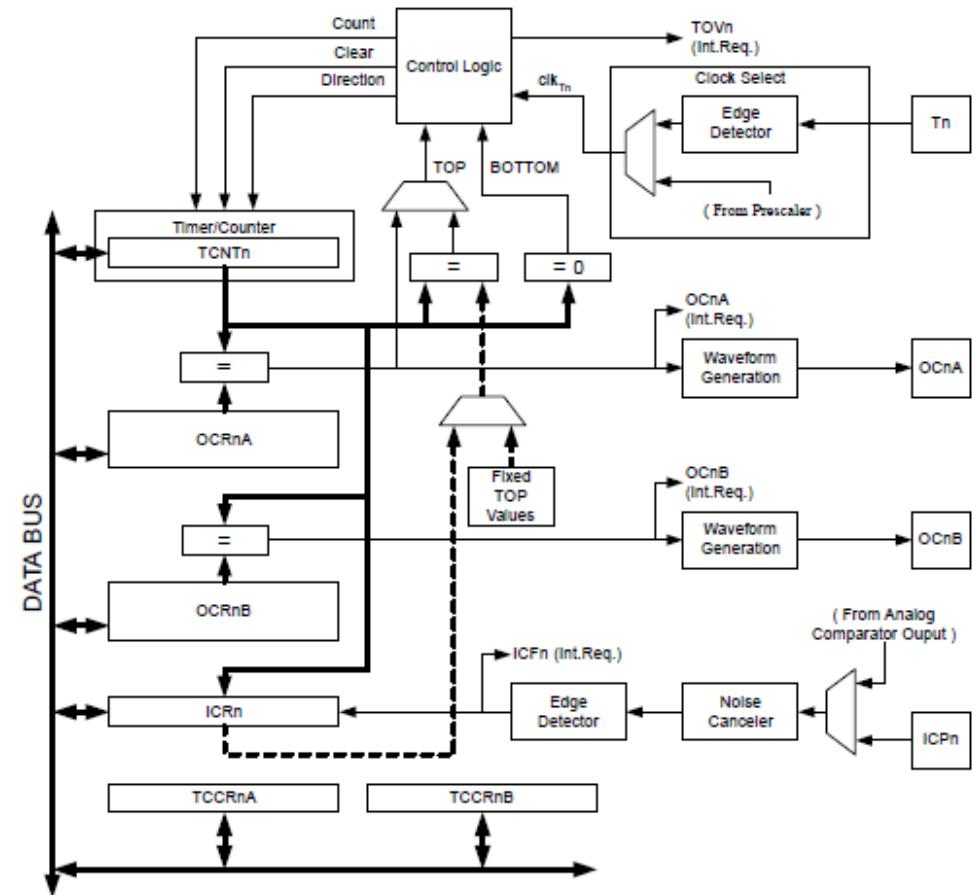


Figure 20-1. 16-bit Timer/Counter Block Diagram



# C4μC - Timers

- Timer Registers

- You can change the Timer behaviour through the timer register. The most important timer registers are:
  - TCCR<sub>x</sub> - Timer/Counter Control Register. The prescaler can be configured here.
  - TCNT<sub>x</sub> - Timer/Counter Register. The actual timer value is stored here.
  - OCR<sub>x</sub> - Output Compare Register
  - ICR<sub>x</sub> - Input Capture Register (only for 16bit timer)
  - TIMSK<sub>x</sub> - Timer/Counter Interrupt Mask Register. To enable/disable timer interrupts.
  - TIFR<sub>x</sub> - Timer/Counter Interrupt Flag Register. Indicates a pending timer interrupt.



# C4μC - Timers

- There exists 3<sup>rd</sup> party libraries for timers...  
see: <http://playground.arduino.cc/Code/Timer1>
- Generic example: clock select and timer frequency

Different clock sources can be selected for each timer independently.  
To calculate the timer frequency (for example 2Hz using timer1) you will need:

- CPU frequency for Arduino: 16Mhz
- maximum timer counter value (255 for 8bit, 65535 for 16bit timer)
- Divide CPU frequency through the choosen prescaler  
 $(16000000 / 256 = 62500)$
- Divide result through the desired frequency ( $62500 / 2\text{Hz} = 31250$ )
- Verify the result against the maximum timer counter value  
( $31250 \leq 65535$ : success). If fail, choose bigger prescaler.

# C4μC - Interrupts



- If you need to monitor some changes or to verify if some data are ready:
- Polling: repeatedly read some status
- Interrupts: a mechanism that alerts when something is just happened
  - You must have seen (recap part-7 slide 13):
    - Fixed interrupt
    - Vectored interrupt
    - Interrupt address table
  - On Atmega328: Interrupt address table (see datasheet p.82)

Vector No	Program Address <sup>(2)</sup>	Source	Interrupts definition
1	0x0000 <sup>(1)</sup>	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x0002	INT0	External Interrupt Request 0
3	0x0004	INT1	External Interrupt Request 0
4	0x0006	PCINT0	Pin Change Interrupt Request 0
5	0x0008	PCINT1	Pin Change Interrupt Request 1
6	0x000A	PCINT2	Pin Change Interrupt Request 2
...	...	...	...

# C4μC - Interrupts

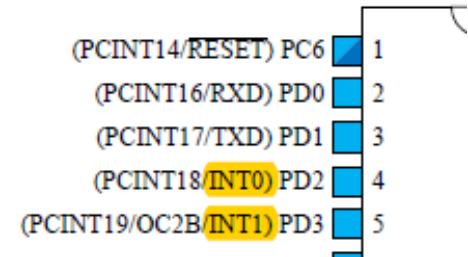
- A total of 24 interrupt:

7	0x000C	WDT	Watchdog Time-out Interrupt
8	0x000E	TIMER2_COMPA	Timer/Counter2 Compare Match A
9	0x0010	TIMER2_COMPB	Timer/Counter2 Compare Match B
10	0x0012	TIMER2_OVF	Timer/Counter2 Overflow
11	0x0014	TIMER1_CAPT	Timer/Counter1 Capture Event
12	0x0016	TIMER1_COMPA	Timer/Counter1 Compare Match A
13	0x0018	TIMER1_COMPB	Timer/Counter1 Compare Match B
14	0x001A	TIMER1_OVF	Timer/Counter1 Overflow
15	0x001C	TIMER0_COMPA	Timer/Counter0 Compare Match A
16	0x001E	TIMER0_COMPB	Timer/Counter0 Compare Match B
17	0x0020	TIMER0_OVF	Timer/Counter0 Overflow
18	0x0022	SPI STC	SPI Serial Transfer Complete
19	0x0024	USART_RX	USART Rx Complete
20	0x0026	USART_UDRE	USART Data Register Empty
21	0x0028	USART_TX	USART Tx Complete
22	0x002A	ADC	ADC Conversion Complete
23	0x002C	EE READY	EEPROM Ready
24	0x002E	ANALOG COMP	Analog Comparator

# C4μC - Interrupts



- Let's analyse some interrupts
- And let's start from the EXTernal INTerrupt (p.87)  
*" The External Interrupts are triggered by the INT pins or any of the PCINT "*
- The INT pins are on Port D (PD2 and PD3)  
(don't be confused by the package pin numbering,  
look at the Port name and number)
- They, by the schematic, correspond to D2 and D3
- We have already used D2 for the push button
  - Only reading the value, even in blocking mode
  - Let's use the interrupt INT0 (on D2) to manage the button



# C4μC - Interrupts

- Arduino API style (see <https://www.arduino.cc/en/Reference/Interrupts>):

```
const int buttonPin = 2;          // the number of the pushbutton pin
const int ledPin = 13;            // the number of the LED pin
volatile int buttonState = 0;    // variable for reading the pushbutton event

void setup() {
    pinMode(ledPin, OUTPUT);
    pinMode(buttonPin, INPUT_PULLUP);
    attachInterrupt(digitalPinToInterrupt(buttonPin), serve_pressed, FALLING);

}

void serve_pressed(){
    buttonState = 1;
}
```

- The function attachInterrupt takes:
  - The interrupt to register to (by the use of internal pin-int f.)
  - The **service function** (serve\_pressed, in our case)
  - The trigger type (LOW, CHANGE, RISING, FALLING)

# C4μC - Interrupts



- In: <your IDE path>\hardware\arduino\avr\cores\arduino\WInterrupts.c, we have (lines 69-71) the **attachInterrupt** implementation:

```
void attachInterrupt(uint8_t interruptNum, void (*userFunc)(void), int mode) {  
    if(interruptNum < EXTERNAL_NUM_INTERRUPTS) {  
        intFunc[interruptNum] = userFunc;)  
    ...  
}
```

- Note: the service function passed as parameter is:
  - A function pointer!
  - With parameter and return value **void**
- In: <your IDE path>\hardware\arduino\avr\cores\arduino\Arduino.h, we have (lines 60-62) the definitions for *mode*:

```
#define CHANGE 1  
#define FALLING 2  
#define RISING 3  
...
```

- See *External Interrupt Control Register A* (datasheet, p.89)  
"The External Interrupt Control Register A contains control bits for interrupt sense control."

# C4μC - Interrupts

- Arduino API style:

```
void loop() {  
  
    delay(5000);    // Suppose here the micro is working hard  
  
    // check if the pushbutton has been pressed.  
    if (buttonState == 1) {  
        digitalWrite(ledPin, HIGH);    // turn LED on  
        delay(2000);  
        buttonState = 0;  
    }  
    digitalWrite(ledPin, LOW);    // turn LED off  
  
    delay(5000);    // Still working hard  
  
}
```

- When it's time to turn the LED on, only the variable buttonState is checked; the variable is set in the service function to 1.
- None a buttonPress is miss, even during hard work and without blocking



# C4μC - Interrupts

- Same application, μC style (relating the INT):

```
#include <avr/interrupt.h>

const int buttonPin = 2;          // the number of the pushbutton pin
const int ledPin = 13;           // the number of the LED pin
volatile int buttonState = 0;    // variable for reading the pushbutton event

void setup() {
  pinMode(ledPin, OUTPUT);
  pinMode(buttonPin, INPUT_PULLUP);
  sei();                      // Enable global interrupts
  EIMSK |= (1 << INT0);      // Enable external interrupt INT0
  EICRA |= (1 << ISC01);     // Trigger INT0 on falling edge
  EICRA = (EICRA & ~((1<<ISC01) | (1<<ISC00))) | (FALLING << ISC00);
}

// Interrupt Service Routine attached to INT0 vector
ISR(INT0_vect){
  buttonState = 1;
}

//void loop() { //UNCHANGED
```

- Defines of `sei()` and `ISR(<source>_vect)` in `avr/interrupt.h`  
see some info: [http://www.nongnu.org/avr-libc/user-manual/group\\_\\_avr\\_\\_interrupts.html](http://www.nongnu.org/avr-libc/user-manual/group__avr__interrupts.html)

# C4μC - Interrupts

- LED toggle application, μC style (relating the INT):

```
#include <avr/interrupt.h>

const int buttonPin = 2;          // the number of the pushbutton pin
const int ledPin = 13;           // the number of the LED pin
volatile int buttonState = 0;    // variable for reading the pushbutton event

void setup() {
  pinMode(ledPin, OUTPUT);
  pinMode(buttonPin, INPUT_PULLUP);
  sei();                      // Enable global interrupts
  EIMSK |= (1 << INT0);      // Enable external interrupt INT0
  EICRA |= (1 << ISC01);      // Trigger INT0 on falling edge
  EICRA = (EICRA & ~((1<<ISC01) | (1<<ISC00))) | (FALLING << ISC00);
}

// Interrupt Service Routine attached to INT0 vector
ISR(INT0_vect){
  digitalWrite(13, !digitalRead(13)); // Toggle LED on pin 13
}

void loop(){} // Do nothing
```

Does it work as  
expected???



# C4μC - Interrupts

- How many times the pushButton has been pressed?

```
#include <avr/interrupt.h>

const int buttonPin = 2;          // the number of the pushbutton pin
const int ledPin = 13;           // the number of the LED pin
volatile int buttonState = 0;    // variable for reading the pushbutton events

void setup() {
  Serial.begin(115200);
  pinMode(ledPin, OUTPUT);
  pinMode(buttonPin, INPUT_PULLUP);
  sei();                      // Enable global interrupts
  EIMSK |= (1 << INT0);      // Enable external interrupt INT0
  EICRA |= (1 << ISC01);      // Trigger INT0 on falling edge
  EICRA = (EICRA & ~((1<<ISC01) | (1<<ISC00))) | (FALLING << ISC00);
  while(!Serial);
  buttonState = 0;
}

// Interrupt Service Routine attached to INT0 vector
ISR(INT0_vect){
  buttonState += 1;        // Count the button pressed
}
```

- Now we count the button pressed events and want to use serial to tell us

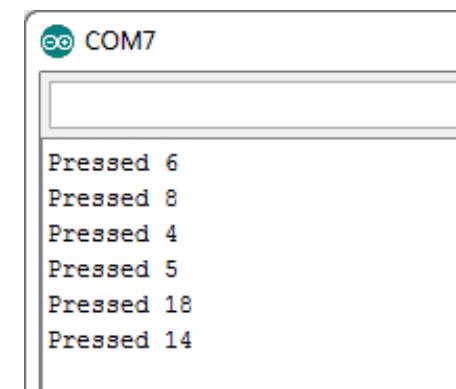
# C4μC - Interrupts

- How many times the pushButton has been pressed?

```
void loop() {  
  
    delay(2000); // Suppose here the micro is working hard  
  
    // check if the pushbutton has been pressed at least once  
    if (buttonState > 0) {  
        digitalWrite(ledPin, HIGH); // turn LED on  
        delay(2000);  
        Serial.print("Pressed "); Serial.print(buttonState); Serial.println();  
        buttonState = 0;  
    }  
    digitalWrite(ledPin, LOW); // turn LED off  
  
    delay(2000); // Still working hard  
}
```

- Open the Serial monitor and...
  - There were more than 1 interrupt

Does it work as expected? NO, bouncing!!!

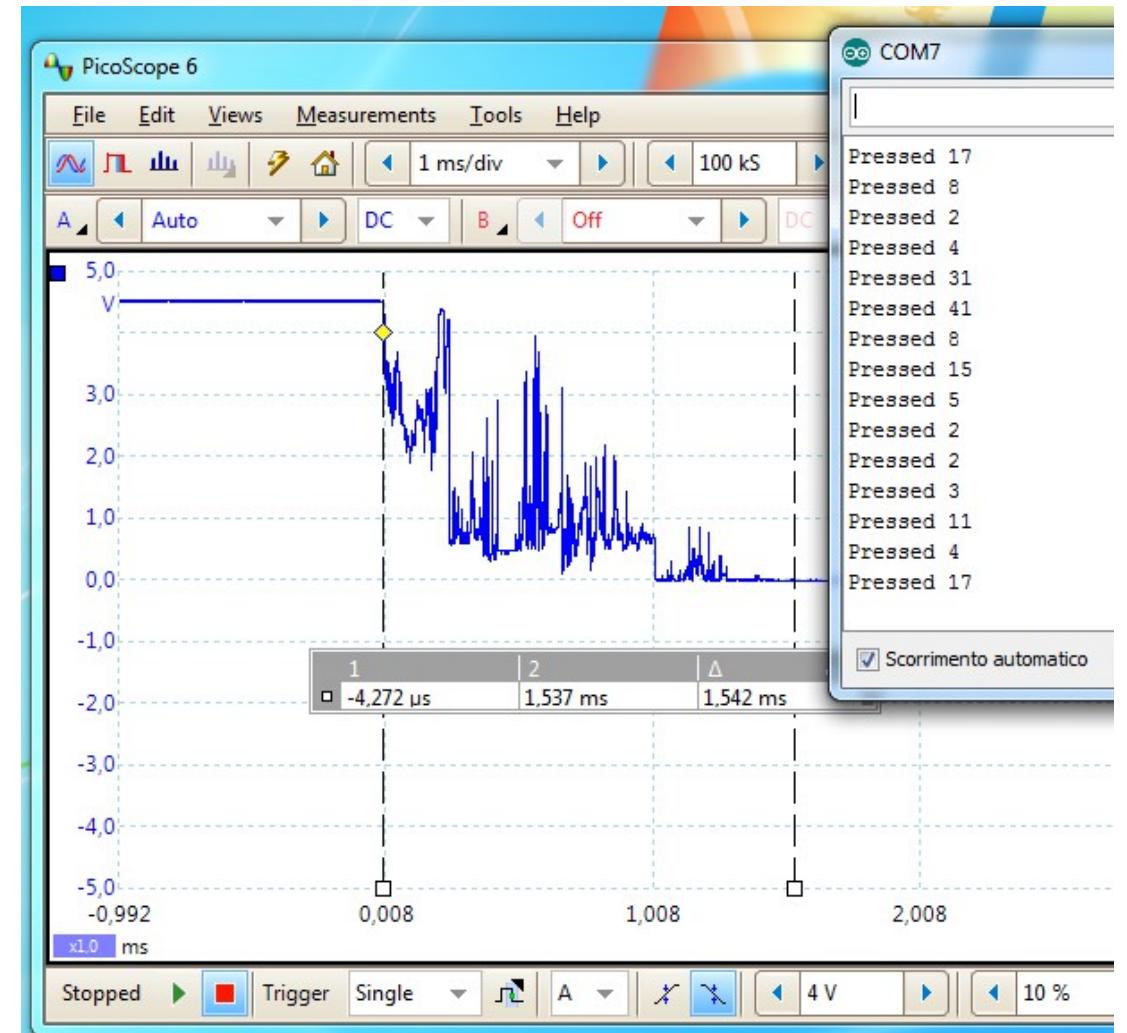


# C4μC - Interrupts



- Bouncing phenomenon
- The switch is not an ideal contact OPEN/CLOSED
- There are more or less fluctuations in the signal transition
  - That are captured by the interrupt
- In the image the signal appears stable after 1.5ms
- It depends on the switch

Need to debounce!!!



# C4μC - Interrupts



- Bouncing phenomenon
- The bouncing can be controlled by discrete extra components (R + C + Buffer or inverter logic) outside the μC
- The bouncing can be controlled via software:
  - by the introduction of a delay interval in which INT is disabled and GPIO re-read
  - By the introduction of state variables
    - As happened for our long delay (2 s) and buttonState = 0
- There are some examples in Arduino IDE
  - Menu File --> Examples --> Digital --> Debounce
  - Menu File --> Examples --> Digital --> StateChangeDetection

Have a look at other solutions online!!

# C4μC - Interrupts

- Internal interrupts, let's use the **timer** (Blink LED 2Hz)

```
// taken from: robotshop.com/letsmakerobots/arduino-101-timers-and-interrupts
#define ledPin 13

void setup() {
    pinMode(ledPin, OUTPUT);

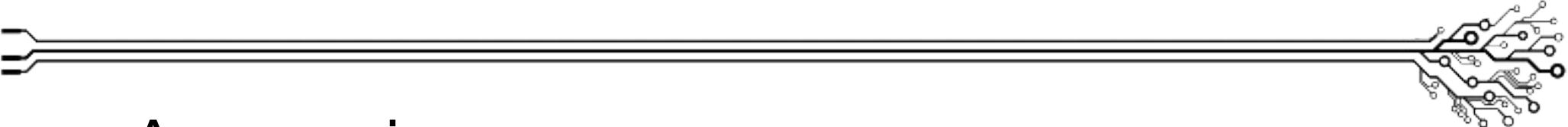
    // initialize timer1
    noInterrupts();           // disable all interrupts
    TCCR1A = 0;
    TCCR1B = 0;
    TCNT1  = 0;
    OCR1A = 31250;           // compare match register 16MHz/256/2Hz
    TCCR1B |= (1 << WGM12); // CTC mode = Clear Timer on Compare match
    TCCR1B |= (1 << CS12);  // 256 prescaler
    TIMSK1 |= (1 << OCIE1A); // enable timer compare interrupt
    interrupts();             // enable all interrupts
}

ISR(TIMER1_COMPA_vect){      // timer compare interrupt service routine
    digitalWrite(ledPin, digitalRead(ledPin) ^ 1); // toggle LED pin
}

void loop() { ; /* our program here here */ }
```

- The blink is controlled entirely by the timer and one of its ISR

# C4μC - Basic functionalities



- As exercise:
  - Try to modify the examples and combine them
  - Have a look at: <https://playground.arduino.cc/Main/AVR>
  - Try to use both the "Arduino API style" and the "μC style"
    - Classify your sketches, it will be useful

Some exercices in the exam can be a little variation of what you have seen today

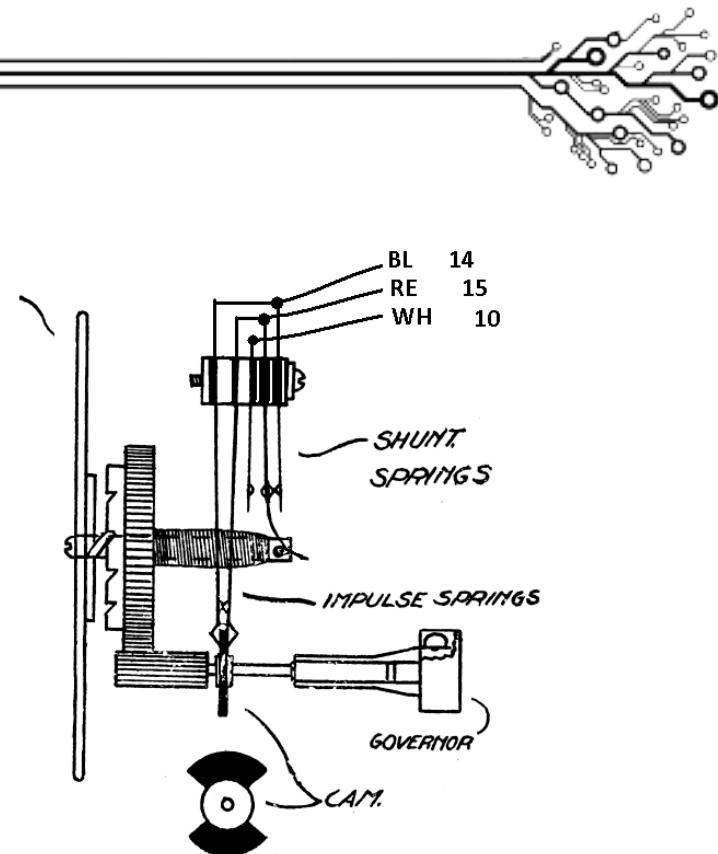
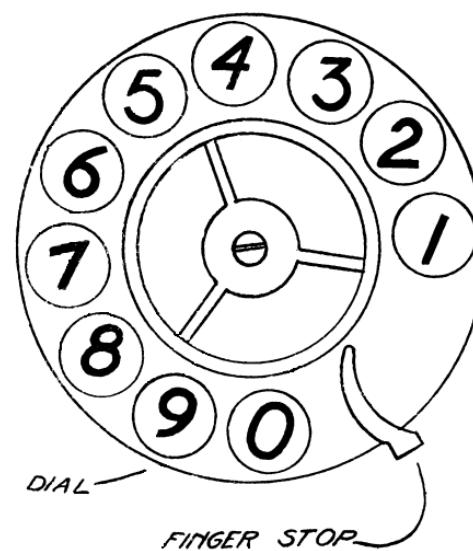
Any question before the last example?

# C4μC - Vintage example



# C4μC - Vintage example

- Dialer disk

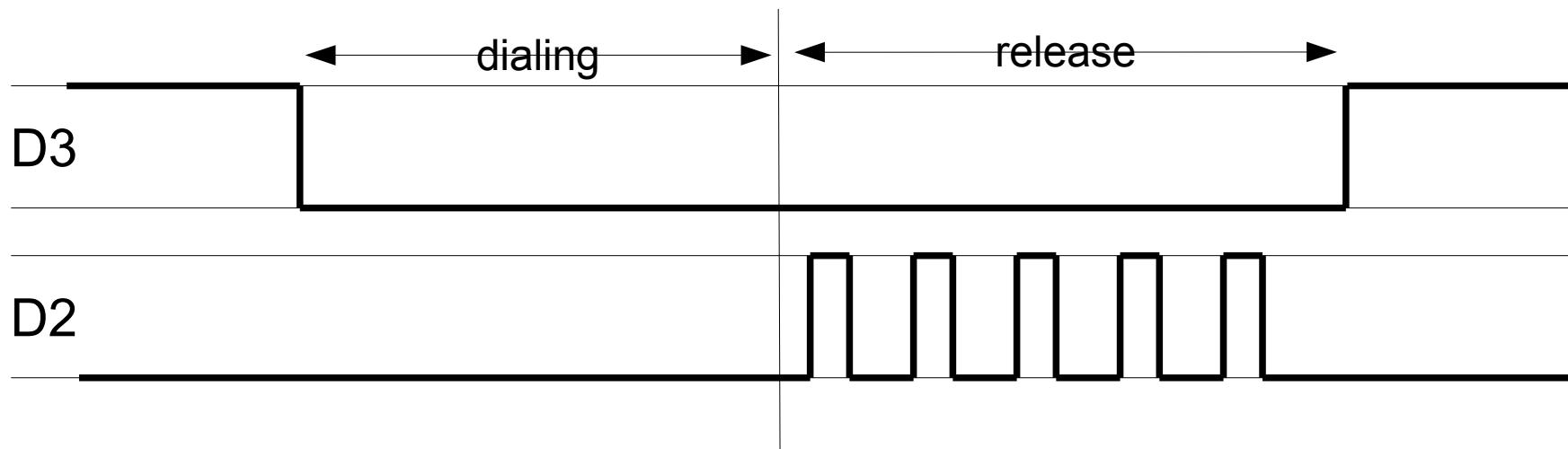


- This dialer has three wires:
  - Red – common
  - Blue – NO, CLOSED during dialing and release
  - White – NC, pulse OPENED during release

# C4μC - Vintage example

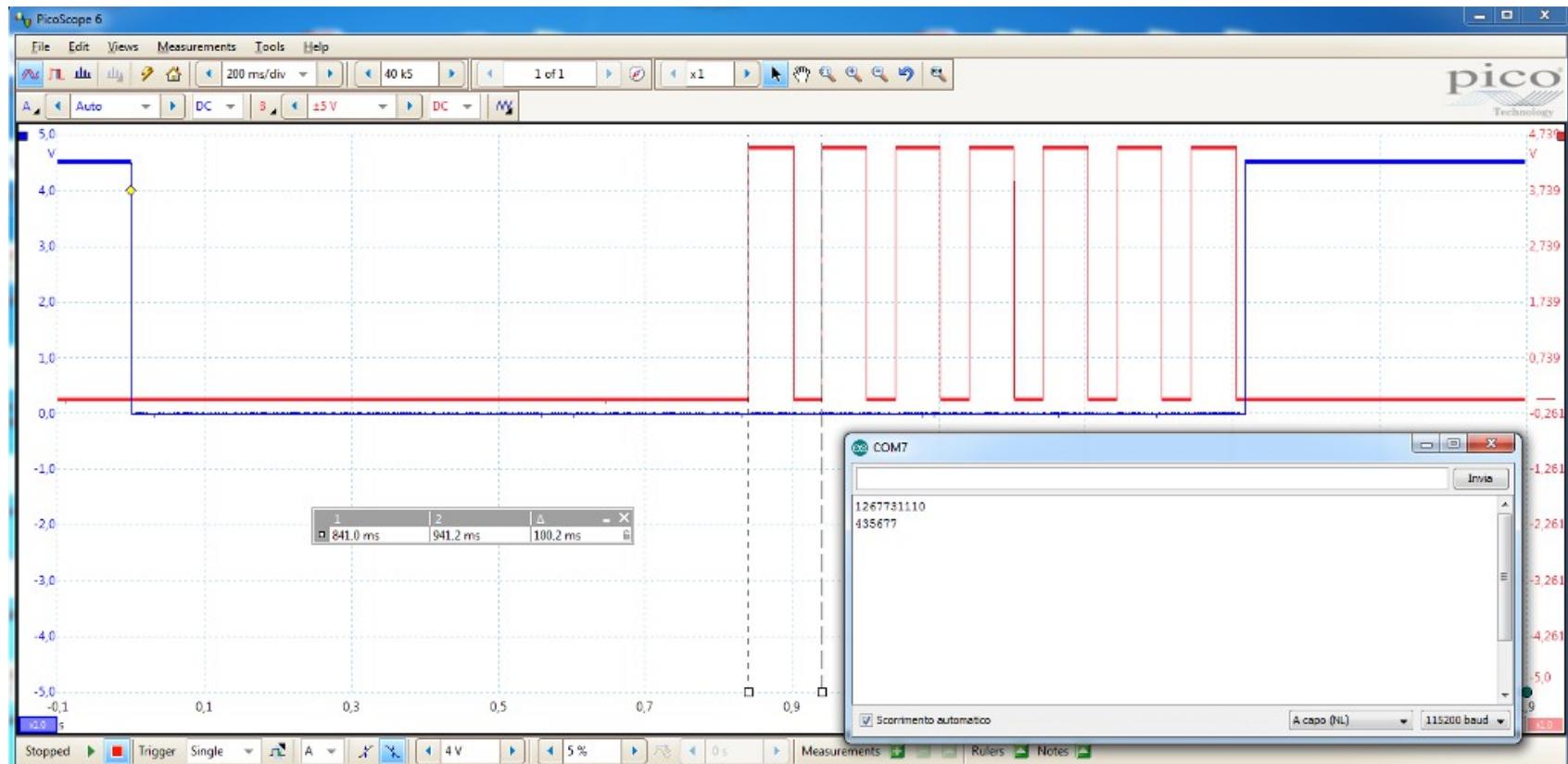


- This dialer has three wires:
  - Red – common GND
  - Blue – NO, CLOSED during dialing and release D3 - pull-up
  - White – NC, pulse OPENED during release D2 – pull-up
- The resulting signal should be something like this:



# C4μC - Vintage example

- The resulting measured signal is



- Pulse period 100ms (63ms HIGH, 37ms LOW)
- Interval from *last pulse falling* to *dialing ended* 12ms
- Timing diagram is useful to trim the app.logic (considerations...)

# C4μC - Vintage example

- The most dummy code - setup()

```
const int pulsePin = 2;           // the pulse signal
const int dialingPin = 3;         // the dialing signalization
const int ledPin = 13;            // the led pin

int dialedDigit = 0;
int numDigits = 0;

void setup() {
  Serial.begin(115200);
  while(!Serial);
  pinMode(ledPin, OUTPUT);
  pinMode(pulsePin, INPUT_PULLUP);
  pinMode(dialingPin, INPUT_PULLUP);
}
```

# C4μC - Vintage example

- The most dummy code - loop()

```
void loop() {  
    dialedDigit = 0;  
  
    while(digitalRead(dialingPin) == HIGH);  
    delay(50);      // here we are dialing  
  
    while(digitalRead(dialingPin) == LOW){  
        while(digitalRead(pulsePin) == LOW);  
        digitalWrite(ledPin, HIGH); // turn LED on  
        delay(10);  
        while(digitalRead(pulsePin) == HIGH);  
        digitalWrite(ledPin, LOW); // turn LED off  
        delay(20);  
        dialedDigit +=1;  
    }  
    if(dialedDigit>0){  
        dialedDigit = dialedDigit % 10;  
        numDigits = (numDigits + 1) % 10;  
        Serial.print(dialedDigit);  
    }  
    else dialedDigit = -1;  
    if(numDigits == 0) Serial.println();  
}
```

Test it!!

Exercise warning:  
possible variations