# Operating Systems

# Hardware-AVR Intro

**Giorgio Grisetti**
grisetti@diag.uniroma1.it

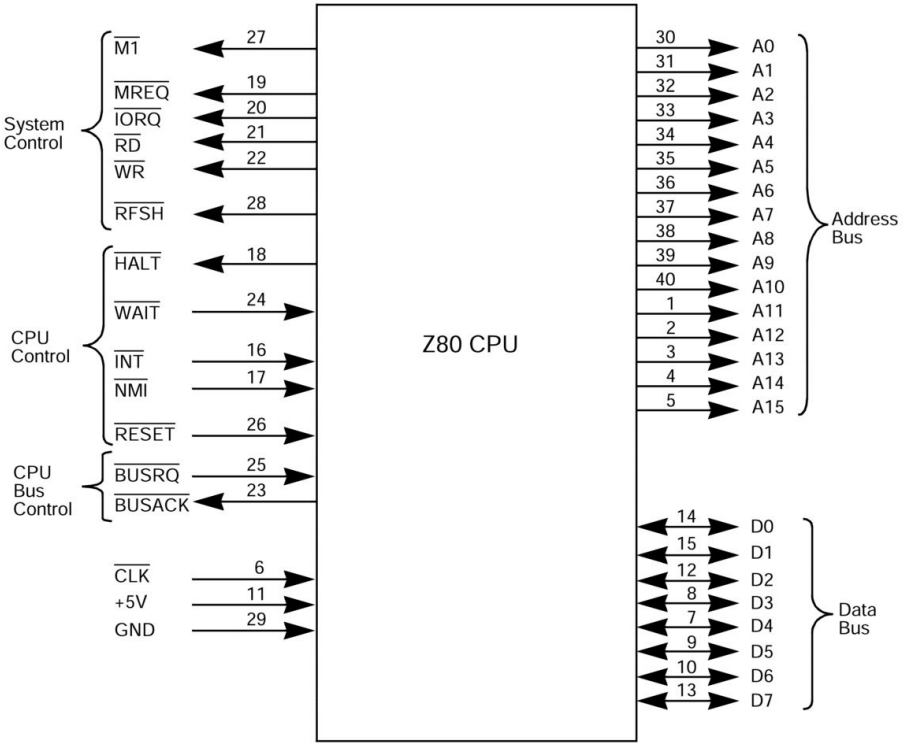
Department of Computer Control and Management Engineering
Sapienza University of Rome

# Outside CPU

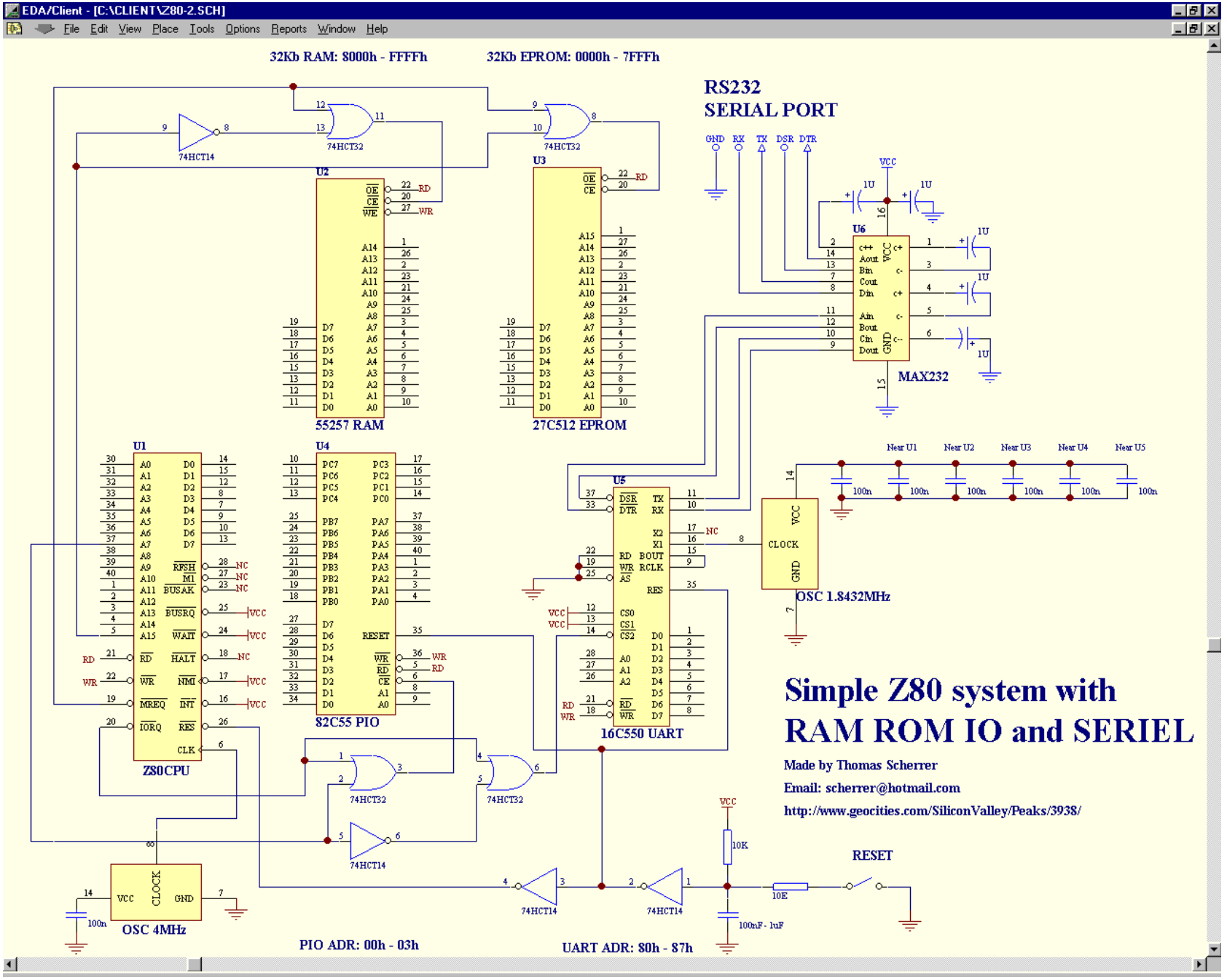
What are all those pins?

A "bus" is a bunch of wires connecting multiple devices

- Data Bus
  - stores the data read from/written to memory
- Address Bus
  - stores the current address of memory being read/written
- Control Bus
  - other signals to handle read/write operation
  - interrupts
  - bus arbitration



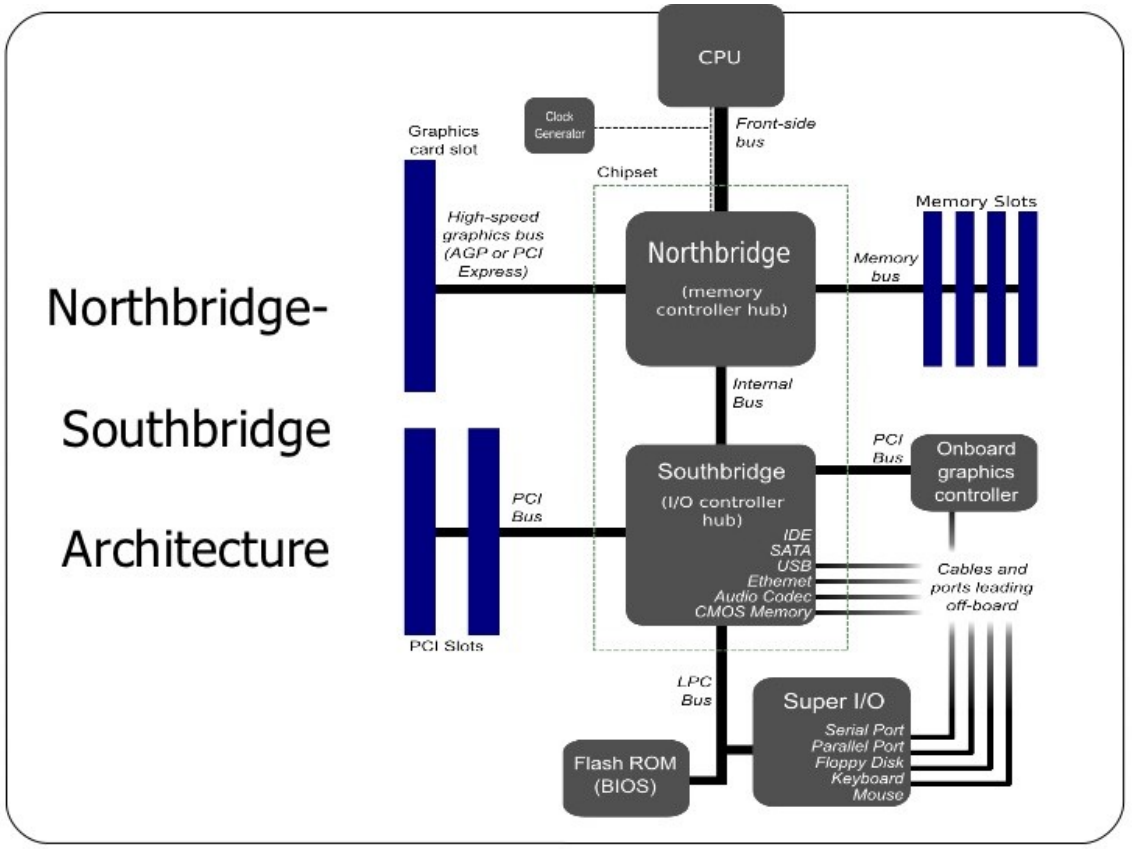
# Ancient (8 bit) architecture



# Slightly Modern Architecture

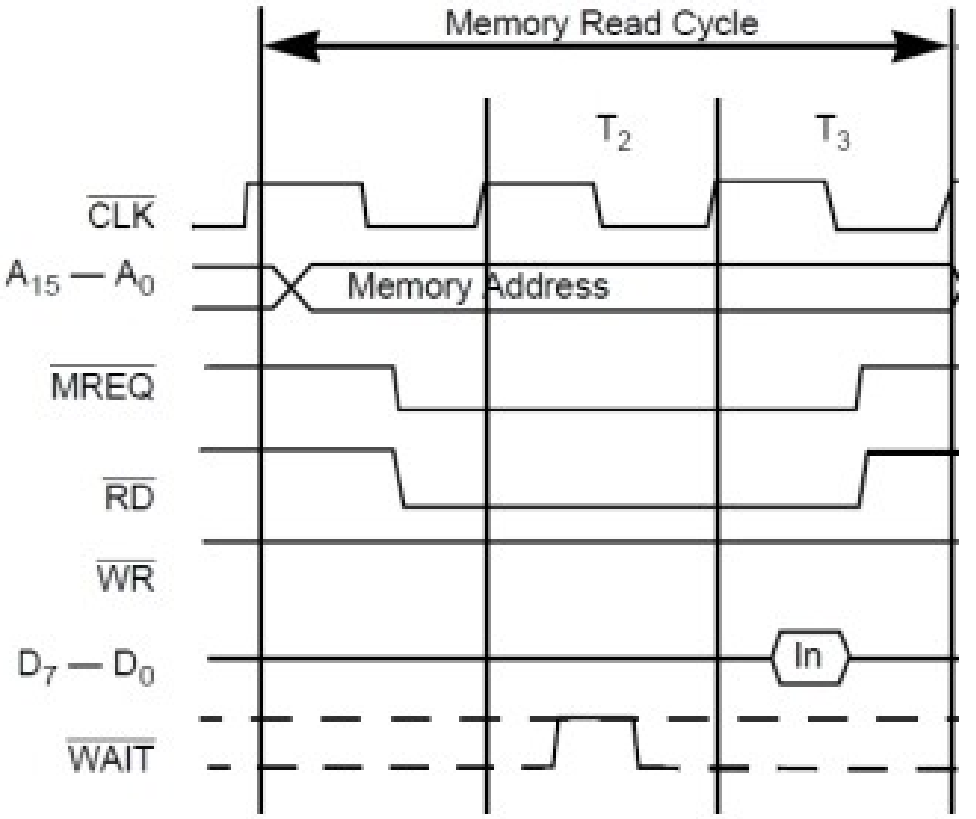
Albeit more complicated, at small scale there are still buses

- inside cpu
- outside cpu



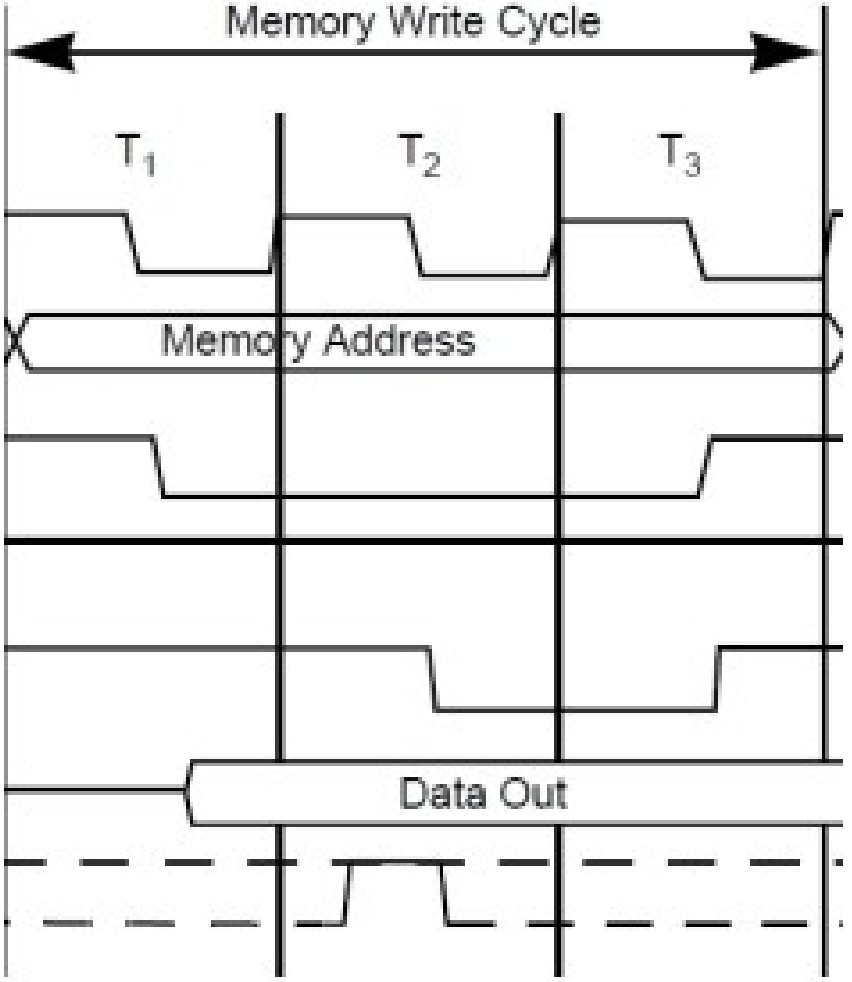
# Reading from Memory

- The cpu puts the desired address on address bus
- When address is electrically stabilized, it lowers (or raises) the RD pin
- After a while, the memory places on the data bus the value at the requested address
  - Optionally the memory informs the CPU it is ready toggling a control signal (WAIT)
- The CPU samples the value from the data bus
- The read cycle ends



# Writing to Memory

- The cpu puts
  - the desired address on address bus
  - the data to write on the data bus
- When address and data are electrically stabilized, it lowers (or raises) the WR pin
- The memory reacts to the trigger of WR by storing the data on the address bus at the desired location
  - Optionally the memory informs the CPU it is ready toggling a control signal (WAIT)
- The write cycle ends



# I/O

- Io mapped to memory occurs as a normal memory write
- Some CPU have separate buses for IO address
- I/O operation can be slower than memory operation

# Interrupt

- An hardware interrupt is an external event that once notified to the CPU by raising a pin, leads to the execution of a special routine (interrupt handler)
- Interrupts are triggered by changing the voltage applied to a pin
- The microcode of the CPU checks for interrupts at each machine cycle (instruction)
- If an interrupt is detected, the device generating the interrupt is queried and identified
- The program counter and the flags are saved on the stack and the flow continues from the interrupt handler
- Once terminated the handler, the execution continues from where it was interrupted

# Interrupt

- An interrupt handler is a special function that terminates with an **`reti`** (return from interrupt) instruction
- This is different from the **`ret`**, used when returning from function calls, since it restores also the flags
- The compiler can be instructed that a function becomes an interrupt routine, by using compiler extension such as
  - **`__attribute__((interrupt))`**
  - **`ISR(<event>)`**
- An interrupt handler, typically does not have parameters
- **`<event>`** is a macro specifying the event that will trigger the interrupt. E.g. **`Timer0`**, or **`INT5`**

# DMA

- Some peripherials might operate in parallel to the CPU on main memory
  - Video Cards
  - Ethernet Cards
  - Smart controllers
  - etc...
- This requires additional signals to handle the bus arbitration. These peripherials "share" the bus with the CPU by requesting/releasing the bus through appropriate signals.
- When a peripherial is writing/reading to memory the CPU is prevented from accessing the bus.
- The CPU typically can keep on working as long as the cache is vaild

# Microcontrollers

Microcontrollers are system on a chip, integrating

- a cpu
- RAM
- peripherials (serial, camera, USB, DigIO, ADC...)
- flash memory to store a program

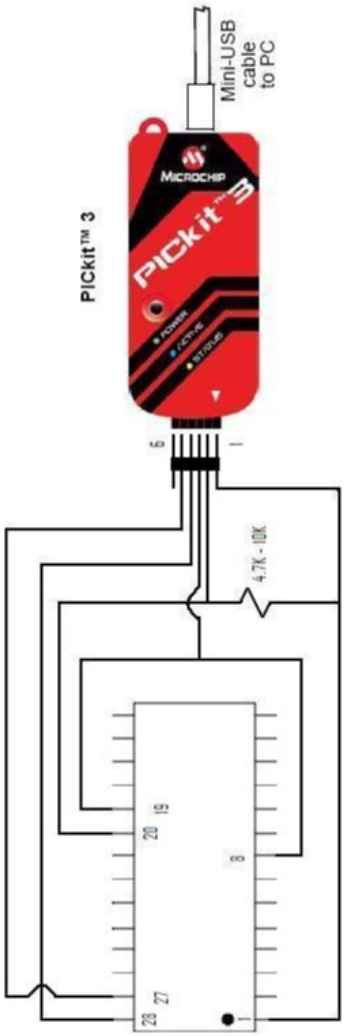
Very useful in embedded applications

Tight programming: no OS, little RAM, cross compilation, hard debugging

# Programming uControllers

One needs

- a cross compilation environment capable of producing machine code for the target machine
- a device to upload the generated binary on the microcontroller flash (programmer)
- a special utility on the host PC that drives the programmer



# Bootloaders

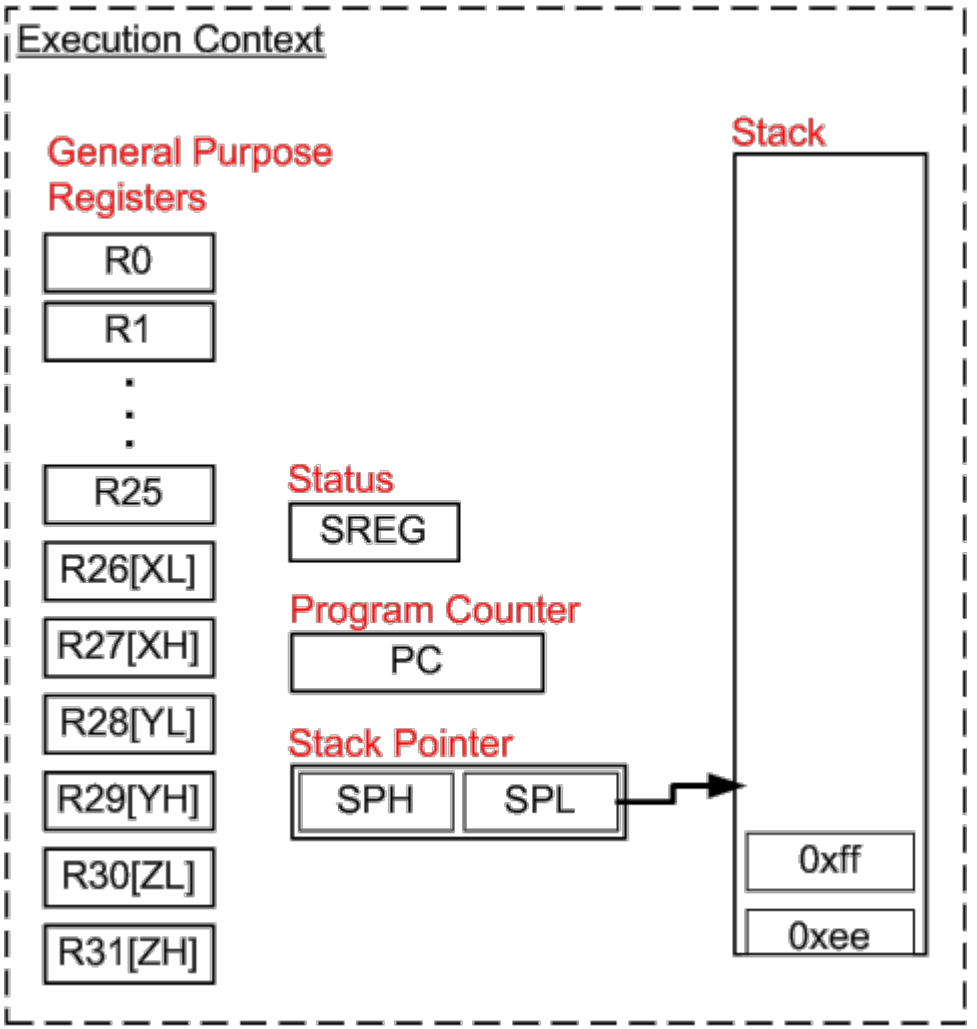
- To avoid the need of a programmer, some microcontroller are loaded with a special program, called the bootloader.
- The bootloader is activated through some special control sequence.
- When activated it offers to the host a protocol to read/write to the flash, and thus to upload the program
- ARDUINO MEGA 2560 has an additional chip that runs (still a microcontroller) the bootloader.

# AVR

- 8 Bit
- RISC architecture
  - (most ops 1istruction/cycle)
  - 32 general purpose registers
- Advanced models have plenty of peripherials integrated
- Designed to support compilers
  - Excellent toolchain (gcc/g++)
  - High code density
- Easily available boards (arduino)

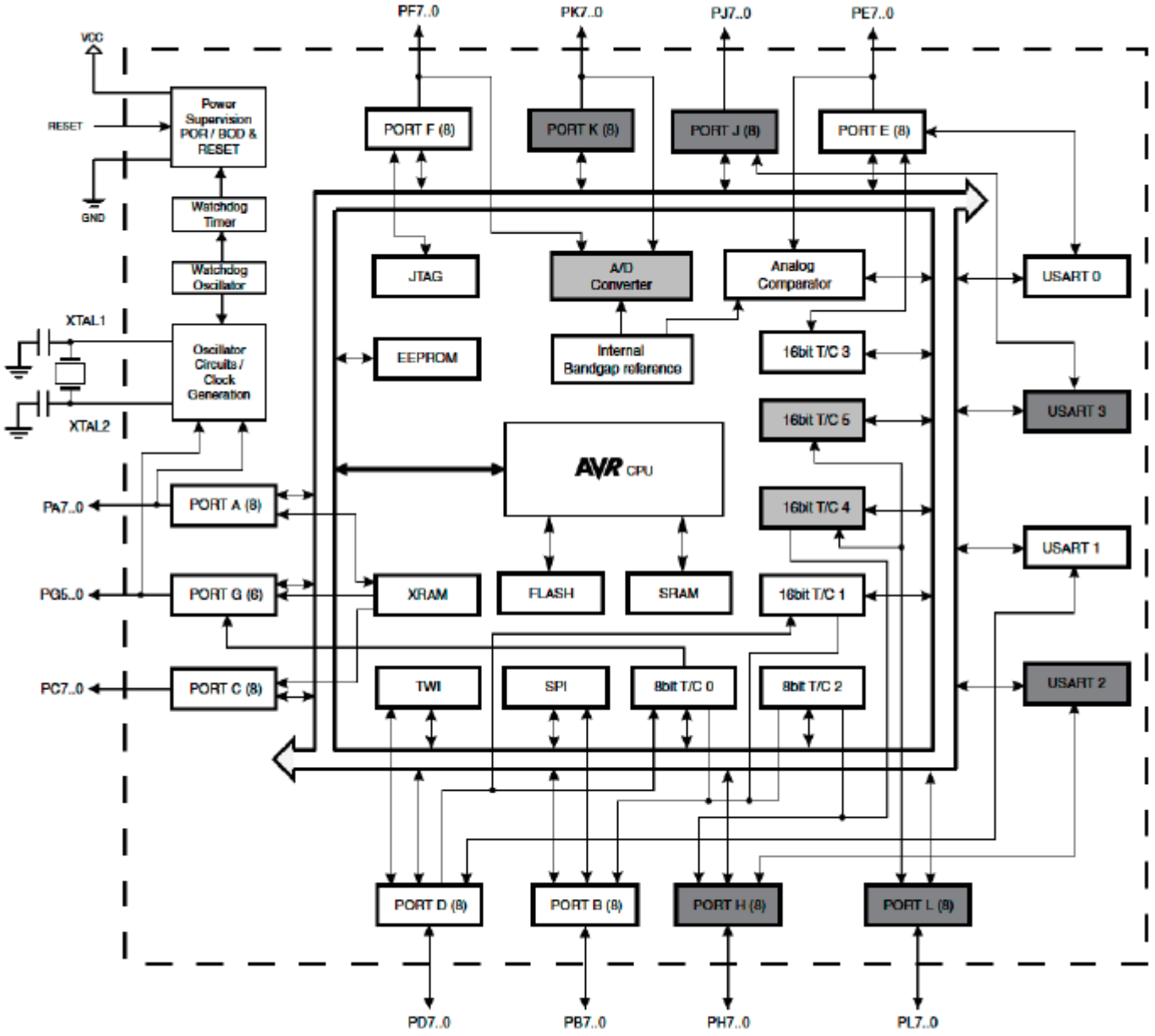
Datasheet for AVR Mega2560( click kere)

click-here



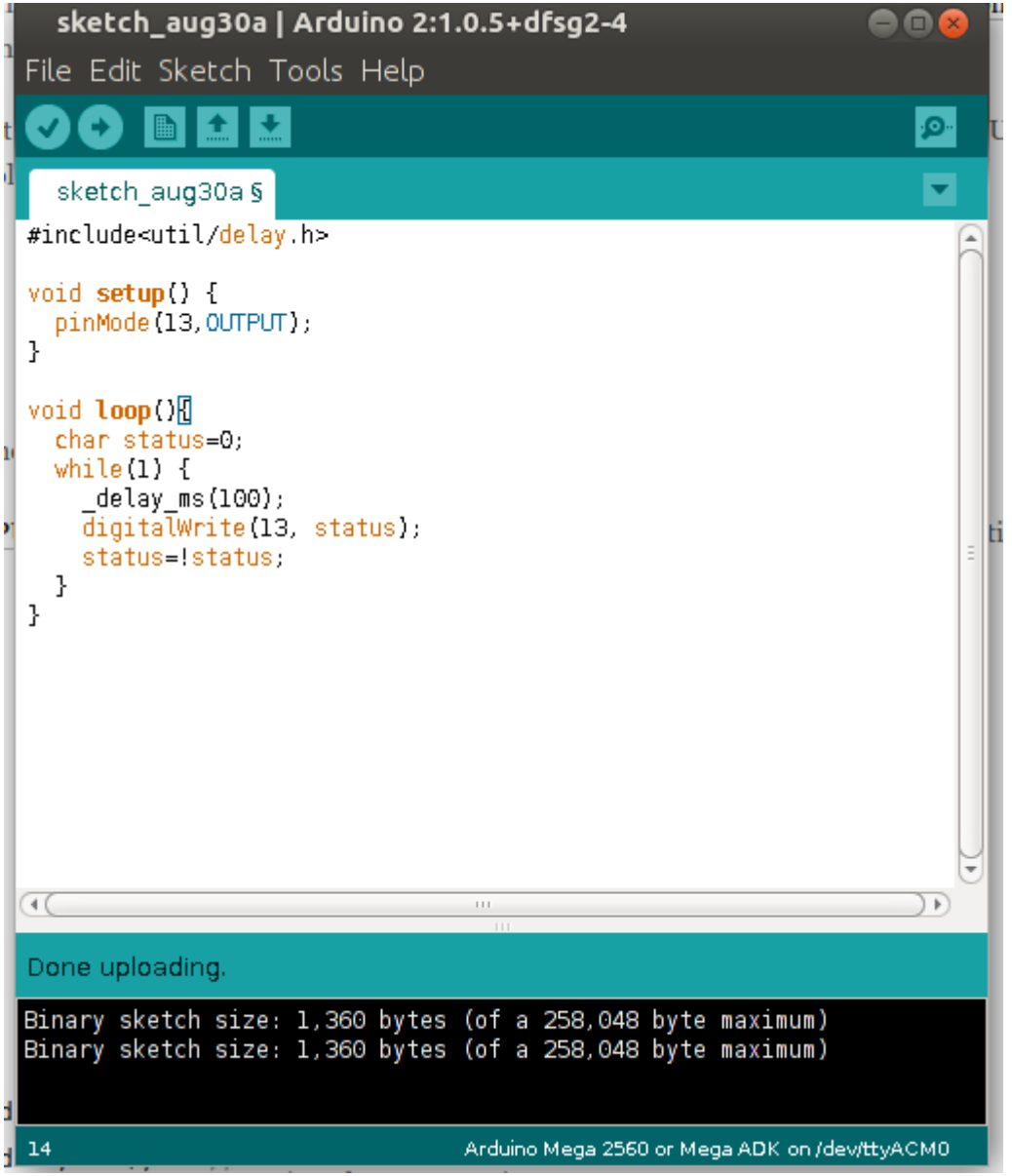
# AVR Peripherials

Snapshot of ATMega2560 (Arduino Mega)



# Arduino

- Is a board that integrates a usb-to-serial converter and a bootloader
- The designers of Arduino implemented an easy-to-use IDE on top of gcc to simplify the development
- They also provided a series of libraries wrapping the most common peripherials for ease of use
- **WE WILL NOT USE THE ARDUINO IDE**



# Mega 2560 Compilation

The programs can be compiled in AVR machine language by avr-gcc.

avr-gcc is a compiler installed on ubuntu when installing the arduino ide

**`sudo apt-get install arduino arduino-mk`**

The compiler and the binary utilities coming with the avr-gcc package allow to generate a program on the arduino

**`avr-gcc [options] -o filename.elf filename.c`**

here [options] encode the options for the compiler

- MCU type: **`-mmcu=atmega2560`**
- CLOCK frequency: **`-DF_CPU=16000000UL`**

plus the usual gcc options

- **`-Wall --std=gnu99`**

# Mega 2560 uploading

To upload a program through the bootloader we need to

- convert the binary .elf file to an .hex that can be understood by the programmer's software

```
avr-objcopy -O ihex -R .eeprom filename.elf filename.hex
```

- upload the .hex to the board, specifying the device it is connected to, using avrdude

```
avrdude -p m2560 -P /dev/ttyACM0 -c  -b 115200    -D -q -V -C
/usr/share/arduino/hardware/tools/avr/../avrdude.conf -c wiring
-U flash:w:filename.hex:i
```

- Here we have lots of options
  - **`-V -C`** specifies the bootloader software on arduino
  - **`-p`** specifies the MCU
  - **`-P`** specifies the serial device
  - **`-Uflash:w:filename.hex:i`** specifies the that we want to write in the flash memory the content of filename.hex

In your projects just replace filename.hex with the hex file you generated by avr-objcopy and the trick is done. Leave the other options unchanged

# Mega 2560 makefile

The procedure outlined in the previous slides is automated by a fancy makefile your teacher made for you

Using a Makefile (from the source directory of the application)

- to generate one .elf per application:

  **$> make**

- to upload a file (assuming the app is called main1)
  - **$> make main1.hex**

Creating a makefile:

- define all .c and .h files that constitute the project. one of these files will contain the main routine, let's call it  main.c

- in the makefile
  - add all .o files to a OBJS list, for each non-main .c file a .o file should be named

    **OBJS=file1.o file2.o file3.o ...** add all main  files, one for each application to the bin list

    **BINS=main1.o main2.o ...**
  - add all headers in the HEADERS list

    **HEADERS=file1.h file2.h file3.h ...**
  - include your teacher's magic

    **include ../avr_common/avr.mk**

  DONE

# Debugging

- Unless we are writing a program that accesses the serial, we can use the serial line for output
- I implemented for you a **`printf`** wrapper you can use as if it was a regular printf
- **`printf`** just writes on the serial line configured at 19600 baud
- See the output of your arduino you can use a terminal program (e.g cutecom)

# Our First AVR program

We just write a simple program on the AVR, that prints a string on the serial line

In these examples i provided you a simplified version of printf(...) that outputs to serial port.

To upload the program, connect the arduino and from the program folder type

```
$>make hello_avr.hex
```

This will compile the program convert the elf in hex and upload the program to the AVR

The default serial port is /dev/ttyACM0. If your board maps to something different edit the Makefile.

To see the output, start "cutecom" and configure the serial port to 19200 baud, 8 bit parity, 1 bit stop

```
#include <util/delay.h>
#include <stdio.h>

//this includes the printf
#include "../avr_common/uart.h" //

int main(void){
  // this initializes the printf/uart thingies
  printf_init();

  int k=0;
  while(1){
    printf("hello %d\n",++k);
    _delay_ms(1000); // from delay.h, wait 1 sec
  }

}
```

# Peripherials

The AVR integrates plenty of peripherials

- serial ports (UARTS, I2C)
- analog to digital converters
- PWM generators
- timers
- digital ports
- eeprom

- These devices are memory mapped as a set of control registers (variables)
- Some can generate interrupts
- A device is configured by writing something in the corresponding registers
- Consult the datasheet for details

# Digital Ports

- Digital Ports A-L are memory mapped
- Each bit of a port corresponds to a specific pin (see datasheet)
- In C they are accessible as variables
- Three Registers
  - DDR<X>: direction port (write).
    - bit i=1, pin is set as output
    - bit i=0, bin is set as input
  - PORT<X>: output port (write).
    - bit i=1, if pin configured as output you can read a value of +5V. if the pin is set as input, the pull up resistor will be enabled.
    - bit i=0, if the port is output, you will read 0v on the pin.
  - PIN<X>: input port (read)
    - bit i=1: you read a logical 1 on the pin if the value externally applied to the pin is close to 5v
    - bit i=0: you read a logical 0 if the value applied to the pin is close to 0v

# Digital Ports: Led Blink

Pin 13 is connected to the LED

Looking at the port mapping for the arduino mega

- click here

we discover that pin 13 is connected to PORT B, pin 7

We configure the pin as output

In the main loop we toggle it each second

```
int main(void){
  printf_init();

  // the LED is connected to pin 13
  // that is the bit 7 of port b
  const uint8_t mask=(1<<7);
  // we configure the pin as output
  DDRB |= mask;
  int k=0;
  while(1){
    printf("led %d\n", (k&1));
    if (k&1)
      PORTB=mask;
    else
      PORTB=0;
    _delay_ms(1000); // from delay.h, wait 1 sec
    ++k;
  }

}
```

# Digital Ports: Input Value

We will use pin 12 to read a value from the external world

We need a wire.

- We connect one end of the wire to a GND pin
- With the other end, we "disturb" pin 12
- On cutecom we see the results

Pin 12 is mapped to PORTB, bit 6. From port mapping for the arduino mega

- click here

We set it as input (DDRB, bit 6=0)

We set a pull up resistor (PORTB,6=1)

- this ensures than when the pin is floating you will read a logical 1

We periodically read the value of the port, and select the corresponding pin.

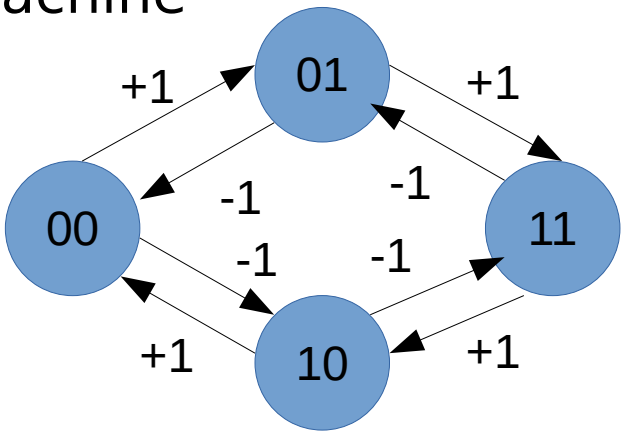
Output will be written on the serial line.

```
int main(void){
  // this initializes the printf/uart thingies
  printf_init();

  // we connect the switch to pin 12
  // that is the bit 6 of port b

  const uint8_t mask=(1<<6);
  // we configure the pin as input,
  //clearing the bit 6
  DDRB &= ~mask;

  // we enable pullup resistor on that pin
  PORTB |= mask;

  while(1){
     // we extract the bit value of the 6th bit
    int key=(PINB&mask)==0;

    printf("switch %02x, %d\n", (int) PORTB, key);
    _delay_ms(500); // from delay.h, wait 1 sec
  }

}
```

# Exercise: Shaft Encoder

A shaft encoder is a device used to determine the angular position of a wheel

- Its outputs are two digital signals (A, and B)
- The "angular position" of the wheel is incremented/decremented depending on the transitions of the two signals, according to the following state machine




What to do:

- Implement a program that "listens" to two pins and increments/decrements a counter accordingly
- On change, the program should output to the serial port the value of the counter. printf is allowed.
- Test the program using two wires between GND and pinA,B.

# Exercise: Keyboard Matrix

Write a program to decode a 4x4 keyboard matrix, connected on port A

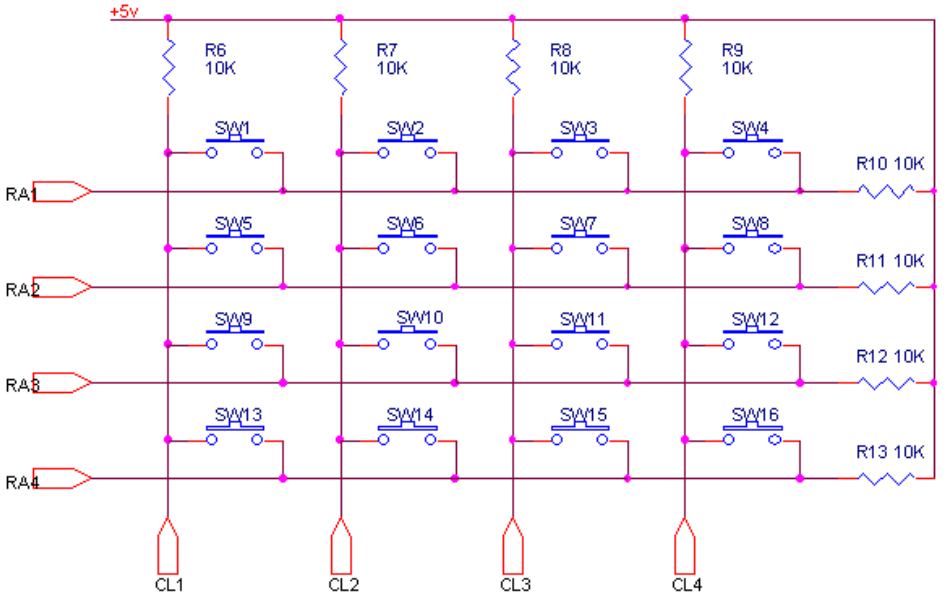
- first 4 bits are the output.
- last 4 bits are the input

The program scans the keyboard one "column" at a time, setting to 1 all outputs but the one of the row to read

Waits a bit to let the signals settle(use _delay_us(100))

Checks which key in a row has changed its status with respect to the previous time the row was scanned

If a change is detected (press or release), it writes the event on the serial



# Timers

- A timer is a peripherial that can be used for
  - measuring time intervals
  - capturing the time at which an external event occurs
  - generating waveforms
  - generating interrupts at specified periods
- The Mega2560 has 5 timers, some have a 16 bit counter others have an 8 bit counter
- A timer updates a counter TCNT<X>{L/H} register, that is periodically incremented by the hardware
- The increments can occur at each clock (16 MhZ on arduino), or at a fraction of it that is a power of 2, through a PRESCALER circuit, that divides the clock
- A timer has multiple (3 in case of AVR) output compare registers (OCR<X>{L/H})
- When the counter value matches the value in OCR an event is generated
- Events can result in
  - Interrupts
  - Toggling a pin (used for waveform generation)
- The behavior to take at the occurrence of an event is controlled through a set of special register :TCCR<X>{L/H}. Setting the bits of these registers allows the user to control the behavior

# Timers: measure time

We will configure a prescaled timer, and we will periodically read the value of the counter.

```
int main(void){
  // this initializes the printf/uart thingies
  printf_init();

  // we will use timer 4
  // this disables all PWM and interrupts
  // connected with timer
  TCCR4A=0;

  // this sets the prescaler to divide
  // the clock by 1024
  TCCR4B=(1 << CS42)|(1<<CS40);

  while(1){
    // we read the timer value
    uint16_t timer_val=TCNT4;
    printf("count %u\n", timer_val);

    _delay_ms(1000); // from delay.h, wait 1 sec
  }

}
```

# Timers: PWM

Timers can be used to generate waveforms, or pseudo continuous value

We can tell the timer to set an output pin to 1(or 0) whenever the value of the counter is higher than the corresponding output compare register.

An output compare register is connected to a specific pin (see datasheet)

In a square wave, the duty cycle is the fraction of the period the signal is up

- a duty cycle of 50% results in a perfectly square wave
- a duty cycle of 25% results in a wave where the signal is high 25% of the time
- a duty cycle of 0 is a signal constantly at 0v, while a duty cycle of 100% is a signal that is always at +5v.

This mechanism can be used to simulate a "continuous" value between 0 and 5 v (the analogWrite of arduino )

# Timers: PWM

In this example we will dim the brightness of the arduino LED by controlling it with a PWM signal with varying duty cycle.

- we configure the timer, to get a proper frequency for the generated waveform
- we use OCR1 as output compare, since it is connected to PortB:7 (the LED pin)
- we configure the timer to generate a waveform, that is 0 when the counter is lower than OCR

```
// configuration bits for PWM
// fast PWM, 8 bit, non inverted
// output compare set low
#define TCCRA_MASK (1<<WGM10)|(1<<COM1C0)|(1<<COM1C1)
#define TCCRB_MASK ((1<<WGM12)|(1<<CS10))

int main(void){
  printf_init();

  // we will use timer 1
  TCCR1A=TCCRA_MASK;
  TCCR1B=TCCRB_MASK;
  // clear all higher bits of
  // output compare for timer
  OCR1CH=0;
  OCR1CL=0;

  const uint8_t mask=(1<<7);
  // we configure the pin as output
  DDRB |= mask;//mask;

  uint8_t intensity=0;
  while(1){
    // we write on the OCR a value
    // that will be proportional to the
    // opposite of the
    // duty_cycle
    OCR1CL=intensity;

    printf("v %u\n", (int) OCR1CL);
    _delay_ms(100); // from delay.h, wait 1 sec
    intensity+=8;
  }
}
```

# Timers: OCR Value

To generate an event each X ms we need to determine which value to write in the OCR

To this end we need to:

- determine how many "counts" are per time unit
  - e.g. 16Mhz, prescaler at 1024 -> 16e6/1024=15625 counts per second
- obtain the OCR value by multiplying the time interval by the value we found in the previous step:
  - e.g, if we want the event to be called each 0.01 s, the OCR value should be

    0.01*15625 = 156.25

Of course only integer values can be written in the registers, so you will have some "quantization" error, unless the OCR value has no decimal values

# Timers: Interrupts

We can use timers to generate an interrupt at specified time intervals

To this end, we need to:

- configure the frequency of the timer by setting the prescaler appropriately
- select an interrupt capable OCR register
- tell the timer what to do when the event occurs
  - (nothing, clear TCCR)
  - trigger an interrupt
- Configure the TIMSK<X> register by setting a bit corresponding to an interrupt
- Write an interrupt handler, that will be invoked when the event occurs

Interrupt handlers NEED TO BE SMALL AND EFFICIENT

# Timers: Interrupt

In this example we will use Timer5/OCR5A to generate an interrupt each 100 ms

The interrupt routine in response to an event is declared as

- ISR(<Event>){...}

This will ensure the compiler to

- write the appropriate footer for the function (iret)
- install the pointer to the function in the appropriate location of the interrupt vector

In this example our interrupt simply toggles a flag

The printf() (SLOW) is handled outside the ISR.

The interrupts can be controlled through the following intrinsic functions:

- cli(): clear interrupt flag
- sei(): set interrupt flag

```
volatile uint8_t interrupt_occurred=0;
volatile uint16_t int_count=0;
// our interrupt routine installed in
// interrupt vector position
// corresponding to output compare
// of timer 5
ISR(TIMER5_COMPA_vect) {
  interrupt_occurred=1;
  int_count++;
}

int main(void){
  printf_init();
  // configure timer
  // set the prescaler to 1024
  TCCR5A = 0;
  TCCR5B = (1 << WGM52) | (1 << CS50) | (1 << CS52);

  const int timer_duration_ms=100;
  uint16_t ocrval=(uint16_t)(15.625*timer_duration_ms);
  OCR5A = ocrval;

  // clear int
  cli();
  TIMSK5 |= (1 << OCIE5A);  // enable the timer int
  // enable int
  sei();
  while(1){
    while (! interrupt_occurred);
    // we reset the flag;
    interrupt_occurred=0;
    printf("int %u!\n", int_count);
  }
}
```

# External Interrupt

We can trigger an interrupt also through an external pin

We will see two types of these external interrupts:

- INT<X>: triggered at a specific pin. An ISR can handle a single pin.
- PCINT<X>: triggered on pin change. An ISR can handle multiple pins

When configuring INT<X> we need to specify if the interrupt is triggered on the falling edge, on the rising edge or on both.

# External Interrupt: INT<X>

In this example we will use a wire.

- one extrema is at GND
- The other extrema disturbs pin 21

Our ISR will simply count how many interrupts occurred so far, and will react to the rising edge of the pin change.

Pin 21 is mapped on PD1, or on INT0 (as seen in the documentation)

To trigger the interrupt we need to activate INT0, in the EIMSK control register, and to configure the trigger mode in the EICRA register.

```
volatile uint8_t interrupt_occurred=0;
uint16_t int_count=0;
ISR(INT0_vect) {
  interrupt_occurred=1;
  int_count++;
}

int main(void){
  printf_init();
  DDRD=0x0; // all pins on port b set as input
  PORTD=0x1; // pull_up on port b

  // enable interrupt 0
  EIMSK |=1<<INT0;

  // trigger int0 on rising edge
  EICRA= 1<<ISC01 | 1<<ISC00;
  sei();
  while(1){
    while (! interrupt_occurred);
    // we reset the flag;
    interrupt_occurred=0;
    printf("int %u!\n", int_count);
  }
}
```

# External Interrupt: PCINT<X>

In this example we will install an handler that reacts to a change in the value of 4 pins PB(1:4), corresponding to the pins 50:53.

Use a wire, and short circuit one of the pins to GND.

In this case we need to tweak

- PCMSK0 (that tells which bits of the port triggers the interrupt)
- PCICR (that tells what PCINT trigger when pins change)

```
#define PIN_MASK 0x0F
volatile uint8_t previous_pins;
volatile uint8_t current_pins;

volatile uint8_t int_occurred=0;
volatile uint16_t int_count=0;

// interrupt routine for position PCINT0
ISR(PCINT0_vect) {
  previous_pins=current_pins;
  current_pins=PINB&PIN_MASK;
  int_occurred=1;
  int_count=1;
}

int main(void){
  printf_init();
  DDRB &= ~PIN_MASK; //set PIN_MASK pins as input
  PORTB |= PIN_MASK; //enable pull up resistors

  // set interrupt on change, looking up PCMSK0
  PCICR |= (1 << PCIE0);

  // set PCINT0 to trigger on state change
  PCMSK0 |= PIN_MASK;
  sei();
  while(1){
    while (! int_occurred);
    // we reset the flag;
    int_occurred=0;
    printf("int %u, p:%x, c:%x!\n",
         int_count, previous_pins, current_pins);
  }
}
```

# UART

So far we used the serial port through printf.

Now we will learn to directly write to the serial port.

- A serial port is controlled by
  - UDR<X>: a data register, that contains the data being sent/received
  - UBRR<X>{H,L}: a baud-rate register that tells how fast the transmission should be
  - UCSR<X>{A,B,C}: a status register that allows to configure other aspects of the port and to poll if a the UART is ready to transmit data or has received some
- To write a byte on the serial port, just write a datum on UDR<X>
  - Using the flags in UCSR<X> you can check if the data are sent (UDRE<X> is the bit number).
- To read data you can
  - check is something has been received by polling the bit RXC<X> of UCSR<X>A
  - When the bit is 1 something has been received and you read it from UDR<X>. Reading clears the flag.

# UART: Atomics

These routines use UART0 (there are two others on the Mega) to communicate with the PC.

Three functions

- Uart_init()
- Uart_putChar(uint8_t)
- Uart_getChar(uint8_t)

All operations are blocking, since we poll on the status bits

```
#define BAUD 19600
#define MYUBRR (F_CPU/16/BAUD-1)

void UART_init(void){
  // Set baud rate
  UBRR0H = (uint8_t)(MYUBRR>>8);
  UBRR0L = (uint8_t)MYUBRR;

  /* 8-bit data */
  UCSR0C = (1<<UCSZ01) | (1<<UCSZ00);

  /* Enable RX and TX */
  UCSR0B = (1<<RXEN0) | (1<<TXEN0) | (1<<RXCIE0);
}

void UART_putChar(uint8_t c){
  // wait for transmission completed,
  // looping on status bit
  while ( !(UCSR0A & (1<<UDRE0)) );

  // Start transmission
  UDR0 = c;
}

uint8_t UART_getChar(void){
  // Wait for incoming data,
  // looping on status bit
  while ( !(UCSR0A & (1<<RXC0)) );

  // Return the data
  return UDR0;

}
```

# UART: Echo Server

Based on the above primitives we will implement an echo server that "repeats" through the serial line a string we enter through cutecom.

- We will build
  - getString()
  - putString()

functions on top of the primitives above and we will write a simple main that does the job.

```
// reads a string until the first newline or 0
// returns the size read
uint8_t UART_getString(uint8_t* buf){
  uint8_t* b0=buf; //beginning of buffer
  while(1){
    uint8_t c=UART_getChar();
    *buf=c; ++buf;
    // reading a 0 terminates the string
    if (c==0) return buf-b0;
    // reading a \n  or a \r return results
    // in forcedly terminating the string
    if(c=='\n'||c=='\r'){
      *buf=0; ++buf; return buf-b0;
    }
  }
}
void UART_putString(uint8_t* buf){
  while(*buf){
    UART_putChar(*buf); ++buf;
  }
}

#define MAX_BUF 256
int main(void){
  UART_init();
  UART_putString((uint8_t*)
     "write something, i'll repeat it\n");
  uint8_t buf[MAX_BUF];
  while(1) {
    UART_getString(buf);
    UART_putString((uint8_t*)"received\n");
    UART_putString(buf);
  }
}
```

# Exercises

- 1. resolve the exercise on shaft encoder using PCINT, for 4 encoders (in adjacent bits of a port)
- 2. extend the previous exercise to print on the serial the status of the encoder each 100 ms. Use (correctly) the timer to set a flag on interrupt, and do the slow job of printing on the serial in the main loop
- 3. extend exercise 2, using the UART directly (no printf allowed)