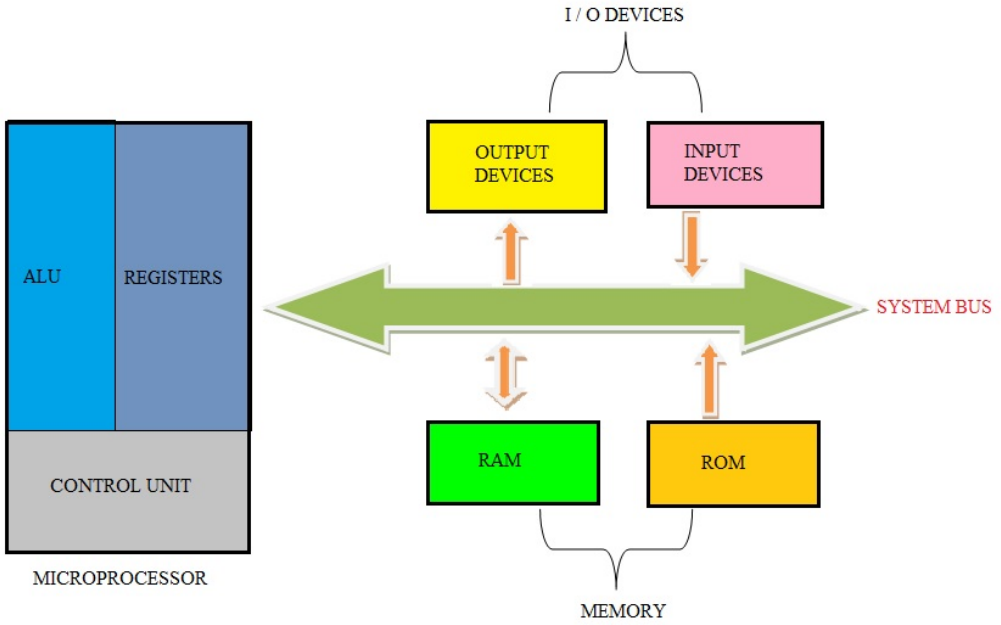
Arduino

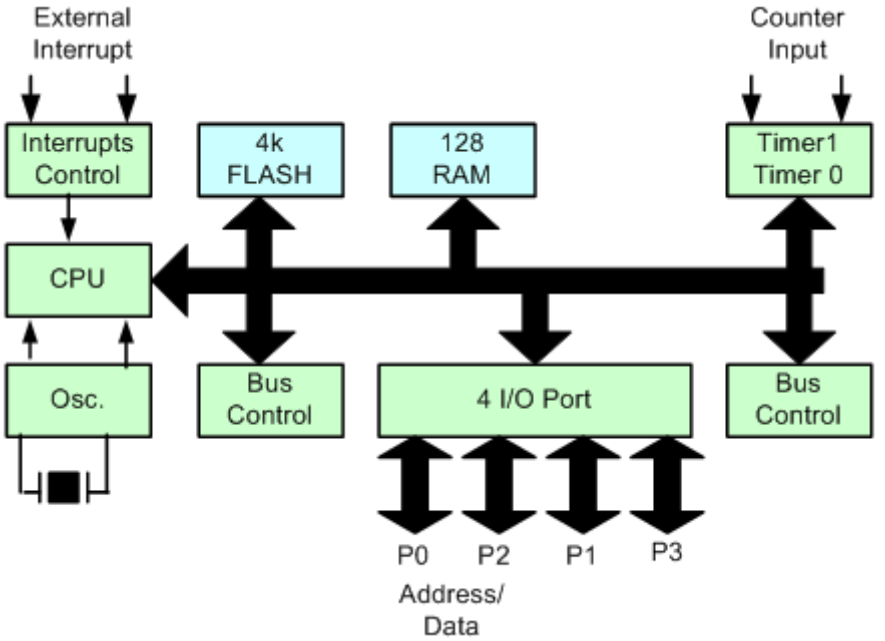
https://www.arduino.cc

Microprocessor *vs* Microcontroller

Comparison	Microprocessor	Microcontroller
Content	A CPU made up of a single silicon chip comprising an ALU, CU and registers	Consist of microprocessor, memory (RAM, ROM), I/O ports, counters, interrupt control unit, etc.
Characteristic	Dependent unit	Self-contained unit
System bus	External	Internal
I/O Ports	Does not contain built-in I/O port	Built-in I/O ports are present
Type of operation performed	General purpose in design and operation	Application oriented or domain specific
Targeted for	High end market	Embedded market
Clock frequency	Very high (GHz)	Low to medium (kHz to MHz)
Interrupt latency	Instruction throughput is given higher priority than interrupt latency	Optimized by design
Use in real time systems	Generally not used as they depend on other components	Handle real time tasks as they are single programmed and self sufficient
Power consumption	Provides less power saving options	Includes more power saving options

µproc *vs* µctrl Architecture





What is Arduino?

- Arduino is an open-source electronics platform based on easy-to-use hardware and software
- Arduino boards are able to read inputs (e.g., light on a sensor, a finger on a button, or a Twitter message) and turn it into an output (e.g., activating a motor, turning on an LED, publishing something online)

How to use it?

- You can tell your board what to do by sending a set of instructions to the microcontroller on the board
- To do so you use the Arduino programming langage
  - based on Wiring: http://wiring.org.co/
  - Wiring is an open-source programming framework for microcontrollers
- and the Arduino Software (IDE)
  - based on Processing: https://processing.org/
  - Processing is a flexible software sketchbook and a language for learning how to code within the context of the visual arts

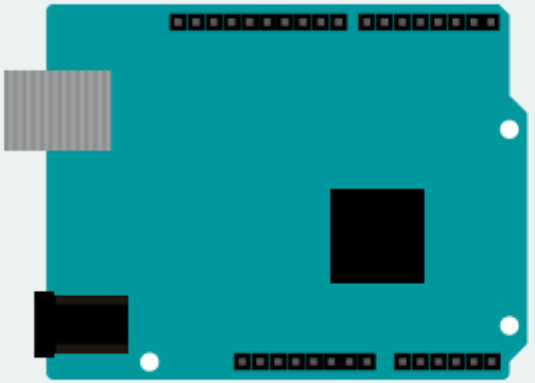
Benefits

- Inexpensive
  - Arduino boards are relatively inexpensive compared to other microcontroller platforms (<€50)
- Cross-platform
  - The Arduino Software (IDE) runs on Windows, Macintosh OSX, and Linux operating systems
- Simple, clear programming environment
  - The Arduino Software (IDE) is easy-to-use for beginners, yet flexible enough for advanced users to take advantage of as well. It's conveniently based on the Processing programming environment.
- Open source and extensible software
  - The Arduino software is published as open source tools, available for extension by programmers. The language can be expanded through C++ libraries, and people wanting to understand the technical details can make the leap from Arduino to the AVR C programming language on which it's based. Similarly, you can add AVR-C code directly into your Arduino programs if you want to.
- Open source and extensible hardware
  - The plans of the Arduino boards are published under a Creative Commons license, so circuit designers can make their own version of the module, extending it and improving it.

Programming Language

- Functions
  - for controlling the Arduino board and performing computations
- Variables
  - Arduino data types and constants
- Structure
  - the elements of Arduino (C++) code
- Libraries
  - https://www.arduino.cc/en/Reference/Libraries
- https://www.arduino.cc/reference/en/

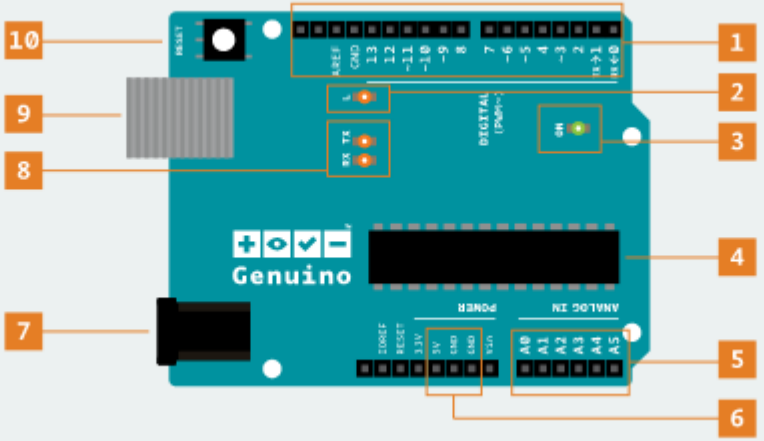
Principle



```
void setup() {
 }

void loop() {
 }
```

Board Anatomy



1. Digital pins : Use these pins with digitalRead(), digitalWrite(), and analogWrite(). analogWrite() works only on the pins with the PWM symbol.
2. Pin 13 LED : The only actuator built-in to your board. Handy target for the blink sketch and useful for debugging.
3. Power LED : Indicates that your board is receiving power. Useful for debugging.
4. ATmega microcontroller : The heart of your board.
5. Analog in : Use these pins with analogRead().
6. GND and 5V pins : Use these pins to provide +5V power and ground to your circuits.
7. Power connector : This is how you power the board when it’s not plugged into a USB port for power.
8. TX and RX LEDs : These LEDs indicate communication between the board and a computer. Useful for debugging.
9. USB port : Used for powering the board, uploading your sketches, and for communicating with your sketch (via Serial. println() etc.).
10. Reset button : Resets the microcontroller

Arduino Software (IDE)

- The Arduino Integrated Development Environment (IDE) contains a text editor for writing code, a message area, a text console, a toolbar with buttons for common functions and a series of menus
- It connects to the Arduino hardware to upload programs and communicate with them
- Programs written using Arduino Software (IDE) are called **sketches**
- These sketches are written in the text editor and are saved with the file extension **.ino**
- The message area gives feedback while saving and exporting and also displays errors
- The console displays text output by the Arduino Software (IDE), including complete error messages and other information
- The bottom righthand corner of the window displays the configured board and serial port
- The toolbar buttons allow you to verify and upload programs, create, open, and save sketches, and open the serial monitor

Sketchbook

- The Arduino Software (IDE) uses the concept of a **sketchbook**
  - a standard place to store your programs (or sketches)
- The sketches in your sketchbook can be opened from the **File > Sketchbook** menu
  - or from the Open button on the toolbar
- The first time you run the Arduino software, it will automatically create a directory for your sketchbook
- You can view or change the location of the sketchbook location from with the **Preferences** dialog

Tabs, Multiple Files, Compilation

- Allows you to manage sketches with more than one file (each of which appears in its own tab)
- These can be normal Arduino code files (no visible extension), C files (.c extension), C++ files (.cpp), or header files (.h)

Uploading

- Before uploading your sketch, you need to select the correct items from the **Tools > Board and Tools > Port** menus
- On Linux, the serial port should be **/dev/ttyACMx**, **/dev/ttyUSBx** or similar
- Once you've selected the correct serial port and board, press the upload button in the toolbar
  - or select the Upload item from the Sketch menu
- Current Arduino boards will reset automatically and begin the upload
  - With older boards (pre-Diecimila) that lack auto-reset, you'll need to press the reset button on the board just before starting the upload
- On most boards, you'll see the RX and TX LEDs blink as the sketch is uploaded
  - The Arduino Software (IDE) will display a message when the upload is complete, or show an error
    - When you upload a sketch, you're using the Arduino bootloader, a small program that has been loaded on to the microcontroller on your board. It allows you to upload code without using any additional hardware. The bootloader is active for a few seconds when the board resets; then it starts whichever sketch was most recently uploaded to the microcontroller. The bootloader will blink the on-board (pin 13) LED when it starts (i.e. when the board resets)

Serial Monitor

- The monitor displays serial sent from the Arduino board over USB or serial connector
- To send data to the board, enter text and click on the "send" button or press enter
  - Choose the baud rate from the drop-down menu that matches the rate passed to Serial.begin in your sketch
  - Note that on Windows, Mac or Linux the board will reset (it will rerun your sketch) when you connect with the serial monitor
- The Serial Monitor does not process control characters
  - if your sketch needs a complete management of the serial communication with control characters, you can use an external terminal program and connect it to the COM port assigned to your Arduino board

Boards

- Before uploading, you must select the proper board hardware
- The board selection has two effects
  - it sets the parameters (e.g. CPU speed and baud rate) used when compiling and uploading sketches
  - and sets the file and fuse settings used by the burn bootloader command
- Some of the board definitions differ only in the latter, so even if you've been uploading successfully with a particular selection you'll want to check it before burning the bootloader
- Arduino Software (IDE) includes the built-in support for the boards based on the **AVR Core**
  - The **Boards Manager** included in the standard installation allows to add support for the growing number of new boards based on different cores like Arduino Due, Arduino Zero, Edison, Galileo and so on

Boards Comparison

- https://www.arduino.cc/en/Products/Compare
- Most boards contain **AVR** MCUs (e.g., ATmega328) made by **Atmel** (now Microchip Technology)
  - https://en.wikipedia.org/wiki/AVR_microcontrollers
- Some boards (e.g., MKR1000) contain ARM-based MCUs
  - https://en.wikipedia.org/wiki/Atmel_ARM-based_processors
- The SAMD21 SoC contains an ARM Cortex-M0+ core (ARMv6-M architecture)
  - https://en.wikipedia.org/wiki/ARM_Cortex-M#Cortex-M0+

Memory

- There are three pools of memory in the microcontroller used on AVR-based Arduino boards
  - Flash memory (program space), is where the Arduino sketch is stored
  - SRAM (static random access memory) is where the sketch creates and manipulates variables when it runs
  - EEPROM is memory space that programmers can use to store long-term information
- Flash memory and EEPROM memory are non-volatile (the information persists after the power is turned off)
- SRAM is volatile and will be lost when the power is cycled

Running Out of RAM

- If you run out of SRAM, your program may fail in unexpected ways; it will appear to upload successfully, but not run, or run strangely
- To check if this is happening, try commenting out or shortening the strings or other data structures in your sketch (without changing the code). If it then runs successfully, you're probably running out of SRAM
- To address this problem
  - If your sketch talks to a program running on a (desktop/laptop) computer, you can try shifting data or calculations to the computer, reducing the load on the Arduino
  - If you have lookup tables or other large arrays, use the smallest data type necessary to store the values you need; for example, an int takes up two bytes, while a byte uses only one (but can store a smaller range of values)
  - If you don't need to modify the strings or data while your sketch is running, you can store them in flash (program) memory instead of SRAM, to do this, use the PROGMEM keyword
- To use the EEPROM, see the EEPROM library
  - http://www.arduino.cc/en/Reference/EEPROM

Digital Pins

- The pins on the Arduino can be configured as either **inputs** or **outputs**
- Arduino pins default to **inputs**, so they don't need to be explicitly declared as inputs with `pinMode()` when used as inputs
- Pins configured this way are in a **high-impedance** state
- Input pins make extremely small demands on the circuit that they are sampling
  - equivalent to a series resistor of 100MΩ in front of the pin
- Thus, it takes very little current to move the input pin from one state to another
  - this can make the pins useful for such tasks as implementing a **capacitive touch sensor** or reading an LED as a **photodiode**
- In order to steer an input pin to a known state if no input is present, add a pullup resistor (to +5V), or a pulldown resistor (resistor to ground) on the input
  - A 10kΩ resistor is a good value for a pullup or pulldown resistor

Pins Configured as INPUT_PULLUP

- There are 20K pullup resistors built into the Atmega chip that can be accessed from software. These built-in pullup resistors are accessed by setting the `pinMode()` as INPUT_PULLUP
  - This effectively inverts the behavior of the INPUT mode, where HIGH means the sensor is off, and LOW means the sensor is on.
- The value of this pullup depends on the microcontroller used
  - On most AVR-based boards, the value is guaranteed to be between 20kΩ and 50kΩ. On the Arduino Due, it is between 50kΩ and 150kΩ. For the exact value, consult the datasheet of the microcontroller on your board
- When connecting a sensor to a pin configured with INPUT_PULLUP, the other end should be connected to ground
  - In the case of a simple switch, this causes the pin to read HIGH when the switch is open, and LOW when the switch is pressed
- The pullup resistors provide enough current to dimly light an LED connected to a pin that has been configured as an input
  - If LEDs in a project seem to be working, but very dimly, this is likely what is going on.
- The pullup resistors are controlled by the same registers (internal chip memory locations) that control whether a pin is HIGH or LOW. Consequently, a pin that is configured to have pullup resistors turned on when the pin is an INPUT, will have the pin configured as HIGH if the pin is then switched to an OUTPUT with `pinMode()`
  - This works in the other direction as well, and an output pin that is left in a HIGH state will have the pullup resistors set if switched to an input with `pinMode()`

Pins Configured as OUTPUT

- Pins configured as OUTPUT with `pinMode()` are said to be in a low-impedance state
- They can provide a substantial amount of current to other circuits
  - Atmega pins can source (provide positive current) or sink (provide negative current) up to 40 mA of current to other devices/circuits
- This is enough current to brightly light up an LED (don't forget the series resistor), or run many sensors
  - but not enough current to run most relays, solenoids, or motors
- Short circuits on Arduino pins, or attempting to run high current devices from them, can damage or destroy the output transistors in the pin, or damage the entire Atmega chip
  - this will result in a **dead** pin in the microcontroller but the remaining chip will still function adequately
- For this reason always connect OUTPUT pins to other devices with 470Ω or 1k resistors
  - unless maximum current draw from the pins is required for a particular application

Analog Input Pins

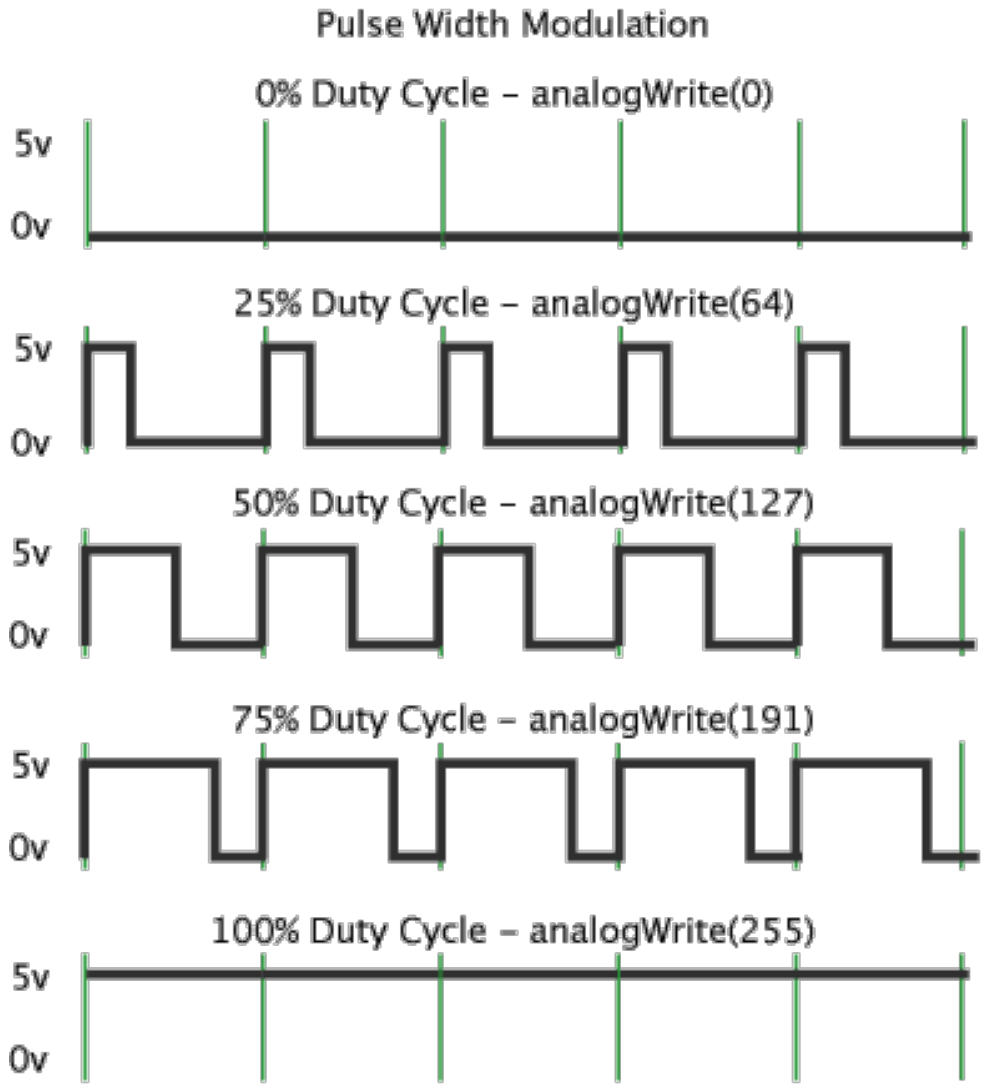
- The ATmega controllers used for the Arduino contain an onboard multi-channel analog-to-digital (A/D) converter. The converter has 10 bit resolution, returning integers from 0 to 1023. Analog pins are used to read analog sensors
  - analog pins also have all the functionality of general purpose input/output (GPIO) pins (the same as digital pins 0 - 13)
- Analog pins can be used identically to the digital pins, using the aliases A0 (for analog input 0), A1, etc
  - `pinMode(A0, OUTPUT); // set analog pin 0 to an output`
  - `digitalWrite(A0, HIGH); // set it to HIGH`
- Analog pins also have pull-up resistors, which work identically to pull-up resistors on the digital pins
  - `pinMode(A0, INPUT_PULLUP);  // set/enable pull-up on analog pin 0`
- Be aware however that turning on a pull-up will affect the values reported by analogRead().
- Details and Caveats
  - The analogRead command will not work correctly if a pin has been previously set to an output, so if this is the case, set it back to an input before using analogRead. Similarly if the pin has been set to HIGH as an output, the pull-up resistor will be set, when switched back to an input.
  - The ATmega datasheet also cautions against switching analog pins in close temporal proximity to making A/D readings (analogRead) on other analog pins. This can cause electrical noise and introduce jitter in the analog system. It may be desirable, after manipulating analog pins (in digital mode), to add a short delay before using analogRead() to read other analog pins

Pulse-Width Modulation

- Technique for getting analog signals with digital means
- Digital control is used to create a square wave, a signal switched between on and off
- This on-off pattern can simulate voltages in between full on (5V) and off (0V) by changing the portion of the time the signal spends on versus the time that the signal spends off
- The duration of **on time** is called the **pulse width**
- To get varying analog values, you change, or modulate, that pulse width
- If the on-off pattern is repeated fast enough, the result is as if the signal is a steady voltage between 0 and 5V

PWM Example

- Green lines represent a regular time period
- This period is the inverse of the PWM frequency
- `analogWrite()` is on a scale of 0 – 255
  - `analogWrite(255)` requests a 100% duty cycle (always on)
  - `analogWrite(127)` is a 50% duty cycle (on half the time)



Hacking Arduino

- Arduino Project Hub
  - https://create.arduino.cc/projecthub
- Arduino IoT Cloud
  - https://www.arduino.cc/en/IoT/HomePage
- Arduino forum
  - https://forum.arduino.cc/
- Extending and developing it
  - https://www.arduino.cc/en/Hacking/HomePage