DE MONTFORT UNIVERSITY LEICESTER

江西理工大學 信息工程学院
JIANGXI UNIVERSITY OF SCIENCE AND TECHNOLOGY SCHOOL OF INFORMATION ENGINEERING



# Instrumentation and Sensor Networks:

## ENGT5105

## ARDUINO VS EMBEDDED SYSTEMS IN INSTRUMENTATION WORLD



**Dr Ata Jahangir Moshayedi**

**Prof Associate ,**
**School of information engineering Jiangxi university of science and technology, China**


**EMAIL: ajm@jxust.edu.cn**


**Autumn _2021**

# Instrumentation and Sensor Networks:

# ENGT5105

## LECTURE 02: ARDUINO VS EMBEDDED SYSTEMS IN INSTRUMENTATION WORLD (2)

# Agenda

18 OCT 2021

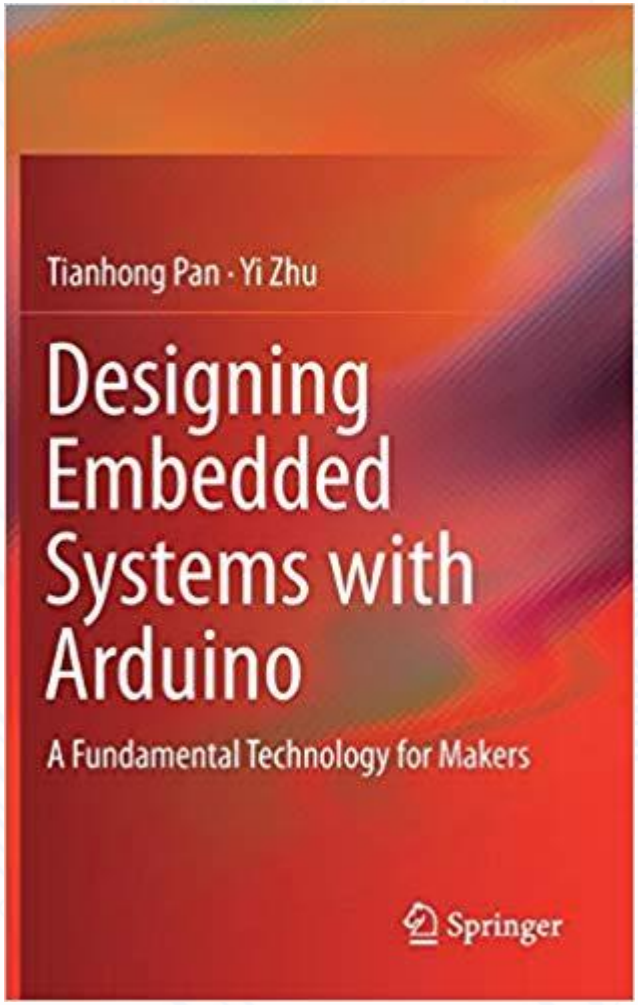
- Introduction to Arduino  and  Arduino types
- Fast review On C language and Arduino programming
- Which Arduino do you need for your project?
- Arduino - I/O Functions
- Introduction to tinker cad as Arduino simulator and some examples

# Reference book

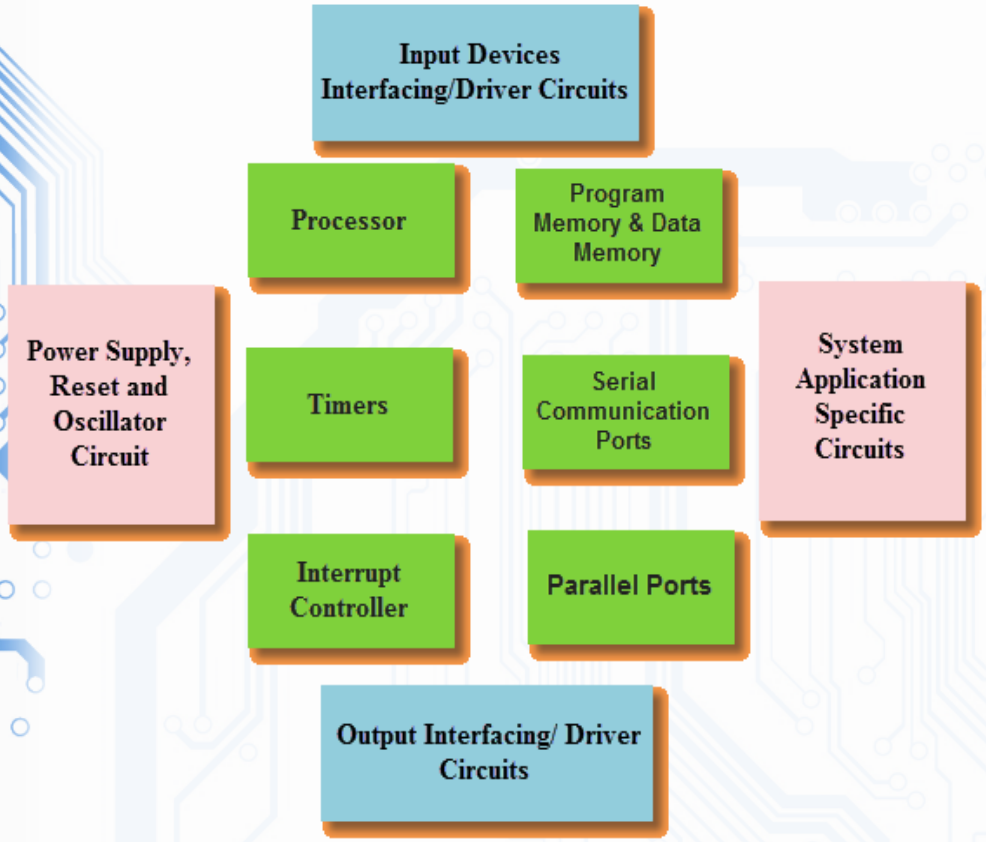
**Designing Embedded Systems with Arduino: A Fundamental Technology for Makers**

**1st ed. 2018 Edition**

**by Tianhong Pan , Yi Zhu**



# Review .....




Embedded system consists of computer hardware and software, and physical parts, designed to perform a specific task.

# Internet of Things and Arduino

- Nowadays **Internet of Things** (*IoT*) :is a pervasive technology that is changing the way we live and how we interact with devices.
- **In IoT project, all the physical objects (things) are connected together using internet infrastructure.**
- **Arduino** board is one of the most important device that enables us to prototype projects. In this post, we will explore how to integrate **Android with Arduino** making the first step in **IoT**.

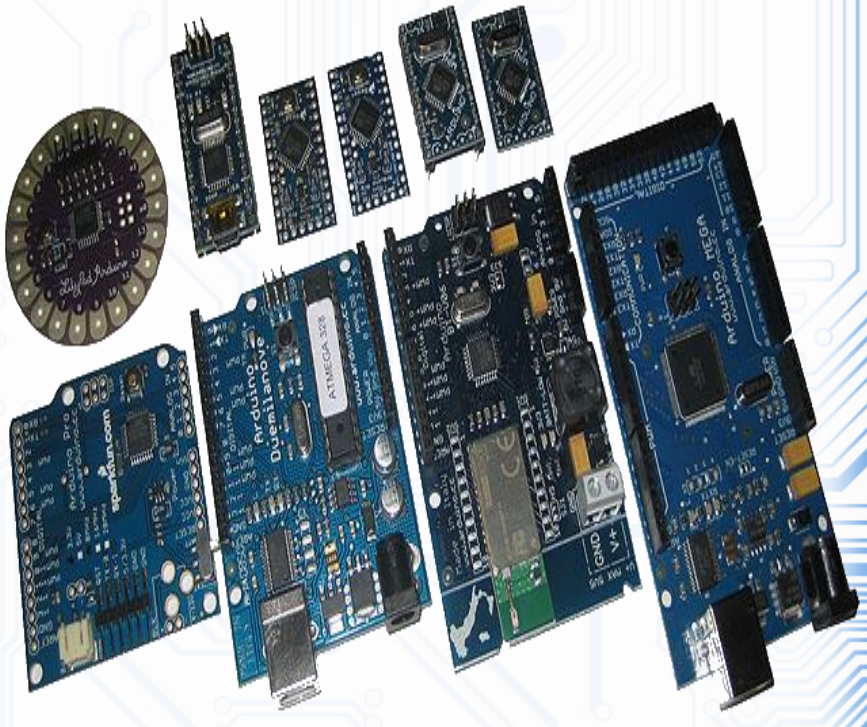
main objects involved in the Arduino IoT project:


DE MONTFORT UNIVERSITY LEICESTER

JIANGXI UNIVERSITY OF SCIENCE AND TECHNOLOGY

# Arduino Types

- Many different versions
  - Number of input/output channels
  - Form factor
  - Processor
- Leonardo
- Due
- Micro
- LilyPad
- Esplora
- Uno

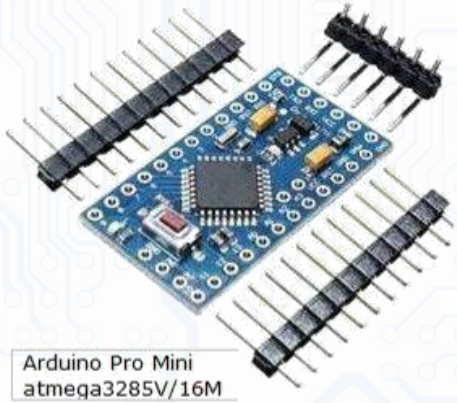


Arduino provides many alternative boards.
• These boards vary in the following aspects:
• the microcontroller type,
• The microcontroller speed (frequency)
• the physical size
• the number of input and output pins
• the memory space for programs, and
• the board price.

# Arduino Types

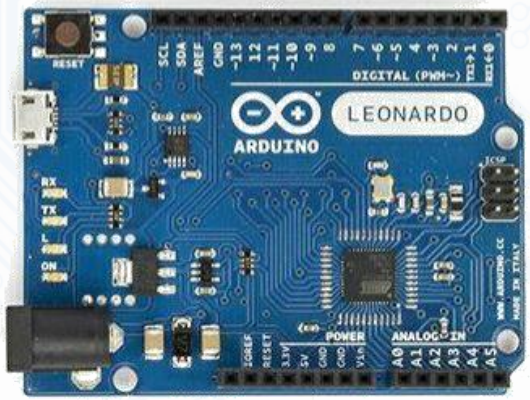

Arduino UNO R3
atmega328pwith USB cable

Arduino UNO R3
mega328pwith USB cable

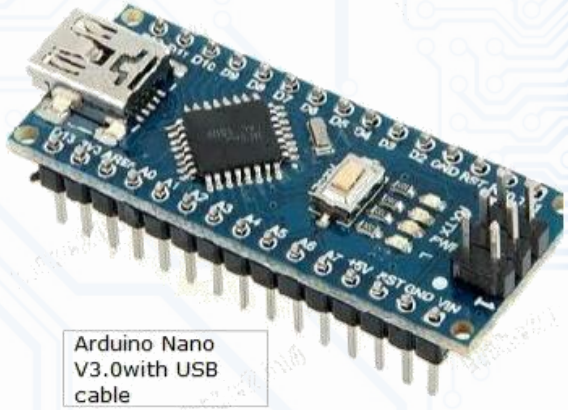


Arduino Pro Mini
atmega3285V/16M


Arduino MEGA2560 R3
withUSB cable(With logo)




Arduino Leonardo
R3ATMEGA32U4
with USB cable



Arduino Nano
V3.0with USB
cable

# Arduino Types

# Arduino Types

Arduino RS232 (male pins)

Arduino Diecimila

Arduino Duemilanove (rev 2009b)

Arduino Uno R2

Arduino Uno SMD R3

Arduino Leonardo

Arduino micro(AtMega 32U4)

Arduino pro micro (AtMega32U4)

Arduino Pro (No USB)

Arduino Mega

Arduino Nano (DIP-30 footprint)

Arduino LilyPad 00_ (rev 2007) (No USB)

Arduino Robot

Arduino Esplora

Arduino Ethernet (AVR + W5100)

Arduino Yun (AVR + AR9331)

Arduino Due (ARM Cortex-M3 core)

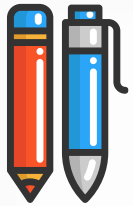
# 8-Bit vs. 32-Bit War Continues which one we should select?

- **8-Bit vs. 32-Bit War Continues**

Unlike early video game consoles, picking a processor is not as simple as the bit number. In general, 8-bit processors offer basic capabilities while consuming lower power. The simpler architectures mean directly programming registers tend to be relatively easy.

32-bit processors offer higher clock speeds along with more RAM, ROM, and serial peripherals. Their architectures could make programming more complicated. Fortunately, frameworks like the Arduino library and Circuit Python hide much of that complexity.

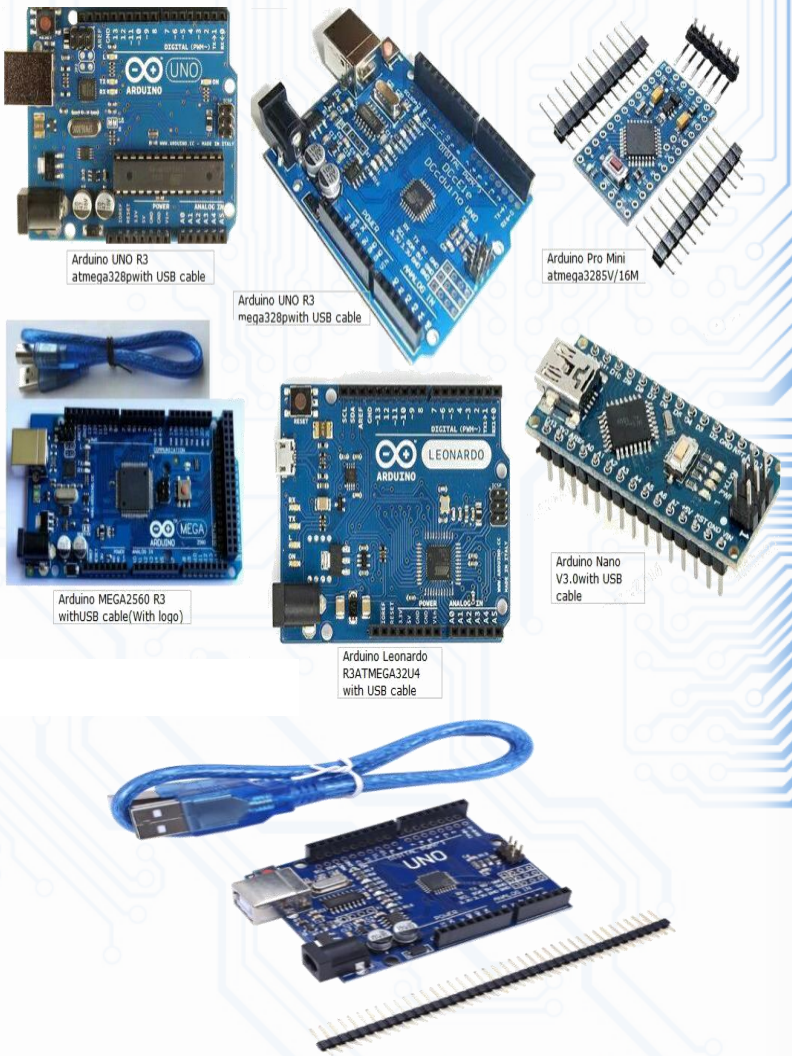
- Choosing a microprocessor just because it is 8-bit or 32-bit can be short-sighted. So it is essential to think about how you plan to use it.



- Arduino board based on BIT(8 or 32)
  - **8-Bit: Uno, Nano, and Mega**
  - **USB with 32U4**

# 8-Bit: Uno, Nano, and Mega

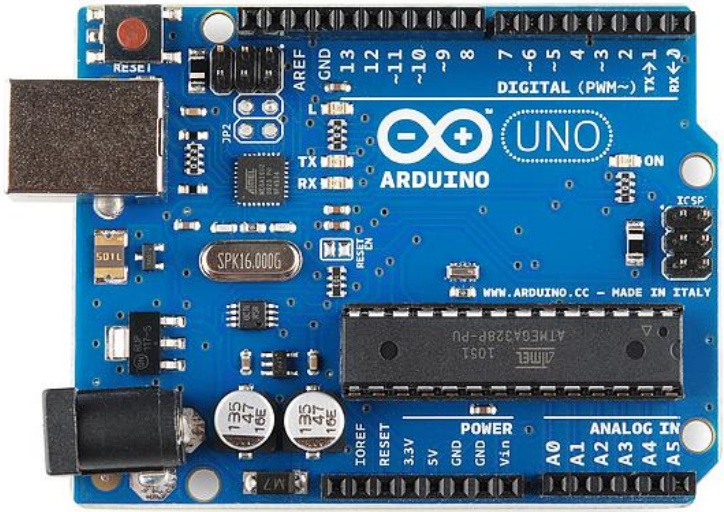
- The Uno is the favored starting point for Arduino projects. It has a distinctive shape with a pseudo-standard header pinout.
- Its CPU is Microchip's ATmega328P.
- This processor's most overlooked specification is the 2,048 bytes of RAM.
- The Uno is a lousy choice if you are thinking about sending, receiving, or processing strings.
- Outside of strings, you would be surprised what you can do with so little, especially given the number of GPIO available.



# UNO

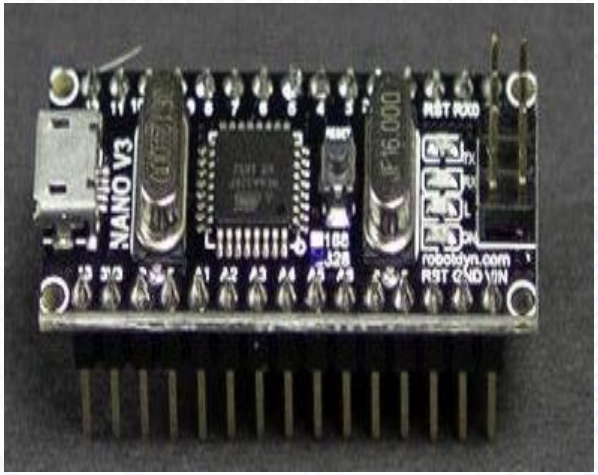
- The Uno is a great choice for your first Arduino. It's got everything you need to get started, and nothing you don't. It has 14 digital input/output pins (of which 6 can be used as PWM outputs), 6 analog inputs, a USB connection, a power jack, a reset button and more. It contains everything needed to support the microcontroller; simply connect it to a computer with a USB cable or power it with a AC-to-DC adapter or battery to get started.

- **The pins are in three groups:**
  - **Invented in 2010**
  - **14 digital pins**
  - **6 analog pins**
  - **power**



# 8-Bit: Uno, Nano, and Mega

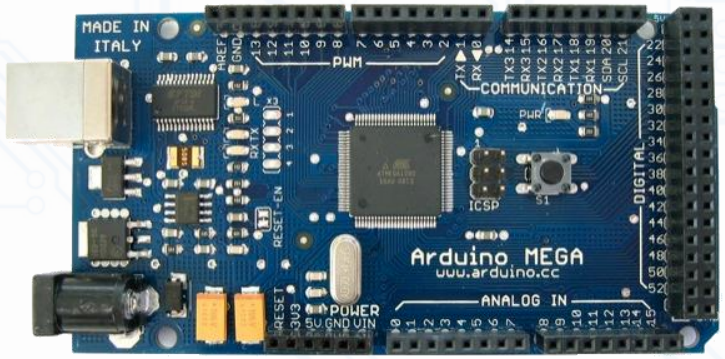
- **Nano**
  - An Arduino Nano clone
  - If the Uno is too large, consider the Nano. It's the most copied, or cloned, Arduino variant.
  - It is the same processor as the Uno, but the board is a slimmed down form factor.
  - The direct software capability means you can prototype with an Uno and install a Nano in your final project.



# 8-Bit: Uno, Nano, and Mega

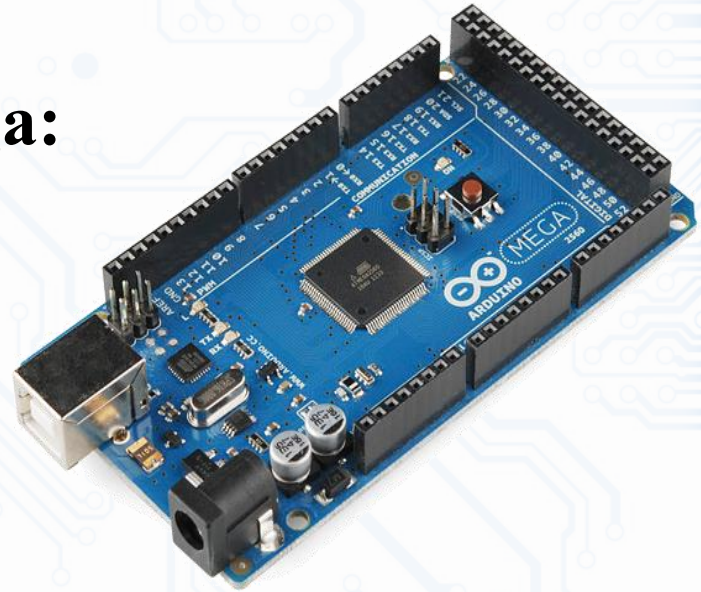
**Mega**

- If the Uno (or Nano) do not offer enough I/O or RAM, boards based on the ATmega2560 are an option.
- The Mega-boards are exceptionally popular in motor control applications, like 3D printers.
- While their cores are common, compared to the ATmega328p, the ATmega2560 has more timers, a second ADC, additional hardware UARTs, and a ton more I/O pins.
- However, it is still an 8-bit processor like the Uno.



# Mega

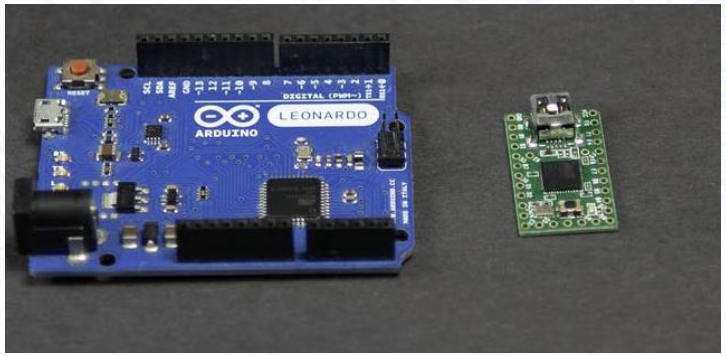
The Arduino Mega is like the UNO's big brother. It has lots (*54!*) of digital input/output pins (14 can be used as PWM outputs), 16 analog inputs, a USB connection, a power jack, and a reset button. It contains everything needed to support the microcontroller; simply connect it to a computer with a USB cable or power it with a AC-to-DC adapter or battery to get started. The large number of pins make this board very handy for projects that require a bunch of digital inputs or outputs (like lots of LEDs or buttons).




- **Compared to the Uno, the Mega:**
  - **Many more communication pins**
  - **More memory**
  - **Some interface hardware doesn't work**

# USB with 32U4

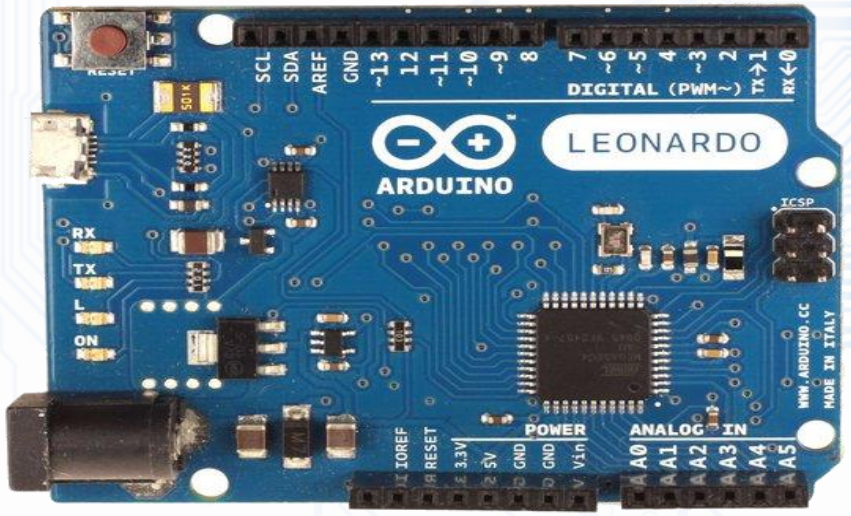
- Another variation of the Uno is the Leonardo or Micro. These boards use the ATmega32U4 chip. Unlike the other 8-bit boards mentioned here, the processor has a built-in USB interface.
- This feature makes it very simple to create USB keyboards, mice, and joysticks. A popular compatible board is the Teensy LC from PRJC.
- It is the same 32U4 but in a teensy-sized form factor.



# Leonardo

The Leonardo is Arduino's first development board to use one microcontroller with built-in USB. This means that it can be cheaper and simpler. Also, because the board is handling USB directly, code libraries are available which allow the board to emulate a computer keyboard, mouse, and more!

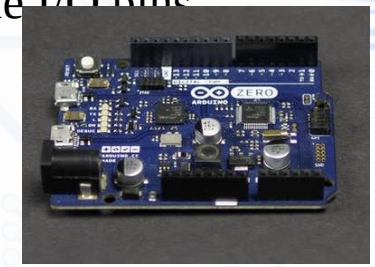
- **Compared to the Uno, a slight upgrade.**
- **Built in USB compatibility**
- **Bugs?**




- Presents to PC as a mouse or keyboard

# 32-Bit

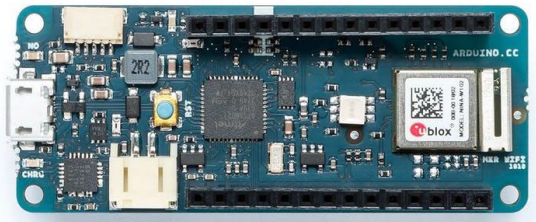
- **Arduino Zero**
- The Arduino Zero contains a 32-bit Microchip SAM D21, which is built around an Arm Cortex-M0+.
- The board shares the same pinout and form factor as the Uno, yet the processor is entirely different.
- In general, the 8-bit boards are based on a 5 volt rail while the 32-bit boards are based on a 3.3 volt rail. It is important to know that most 3.3 volt processors cannot tolerate 5 volt signals. So you may need to use level-shifters when interfacing between the two voltage levels.
- The most striking feature of the M0+ boards is the incredibly flexible serial interfaces. While the boards define I²C and SPI pins, the chip itself is highly configurable. It supports multiple types of serial interfaces on multiple I/O pins.
- Like its 8-bit cousin, the Zero comes in smaller form factors.



# 32-Bit_MKR Series

The **Arduino MKR series** includes boards such as the MKR ZERO, MKR GSM 1400, MKR FOX 1200, and the MKR WiFi 1010. The form factor of the boards in the MKR series is similar. Their I/O pins run down the sides, the form factor is slimmed down, and they all contain a LiPo battery connector with charger circuit. The **MKR Zero** includes the same processor as the Zero. In addition to the slimmer form factor and LiPo charger, the MKR Zero adds a MicroSD card slot

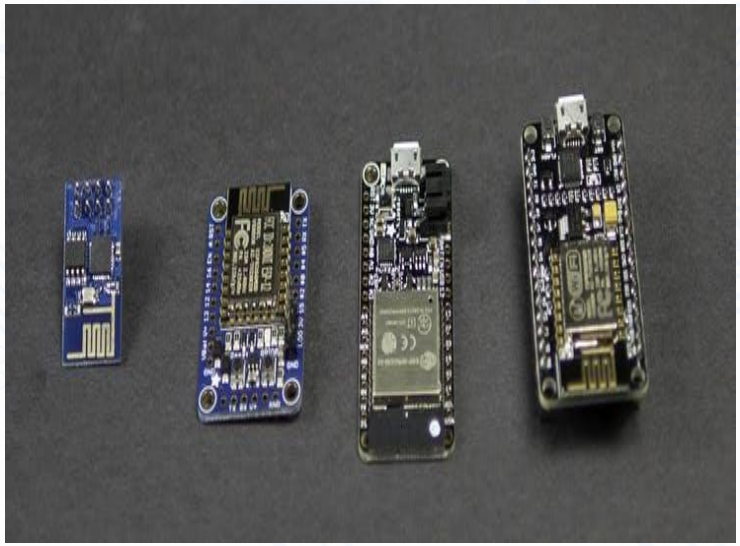
One other board to mention is the **MKR WiFi 1010.** This board is relatively new and contains a chip dedicated to crypto communication. The other exciting feature of the MKR1010 is its processor(s) choice. Onboard there is the same SAMD21 found on the Zero. However, the WiFi module from u-blox includes an ESP32. It is two processors in one. Lastly, there are shields designed for the MKR form factor. They add serial busses like RS-485 and CAN, among other features.

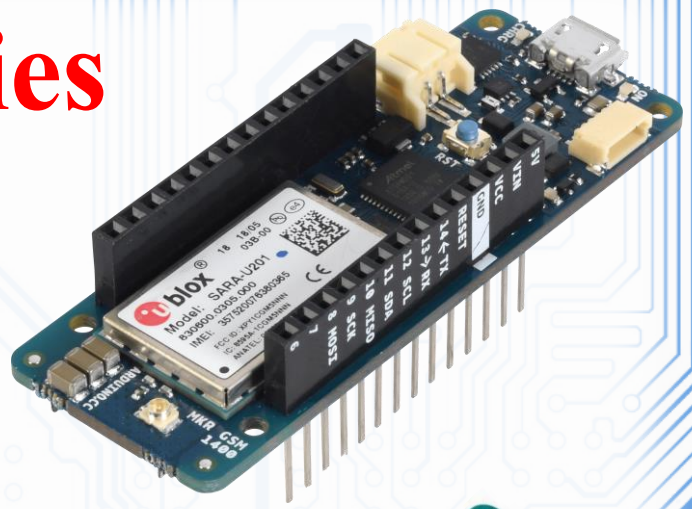
# 32-Bit_MKR Series

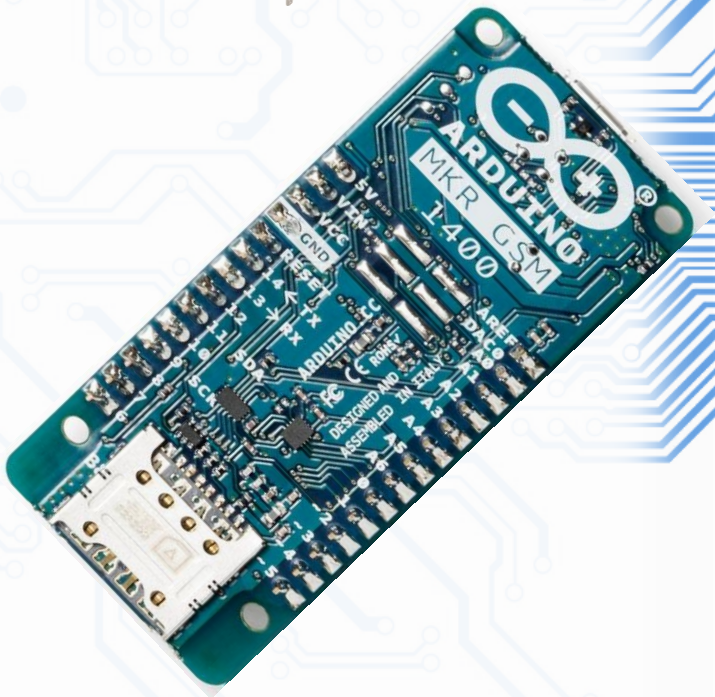
- **ESP8266 and ESP32**When the ESP8266 hit the market, it changed how projects incorporate WiFi. This system on a chip is a 32-bit microcontroller running at 80 MHz with a chip dedicated to WiFi operation. Which means, it runs the full TCP/IP stack separate from the microcontroller running your code.
- ESP modules contain a full microcontroller in their SoC package. The core Arduino library has been ported to the ESP12 and ESP32, meaning in some cases code for an Uno compiles for the ESP12 without any changes!

- The picture shows 4 different ESP options.
- A barebones ESP8266,
- Adafruit's Huzzah ESP8266, an Adafruit Feather ESP32, and a ESP8266 NodeMCU.
- The barebones board was popular because of cost, but it requires extra components to be useful. The Huzzah ESP8266 adds some of those parts and breaks out more of the I/O pins. However, it still requires a serial-to-USB adapter.



# 32-Bit_MKR Series

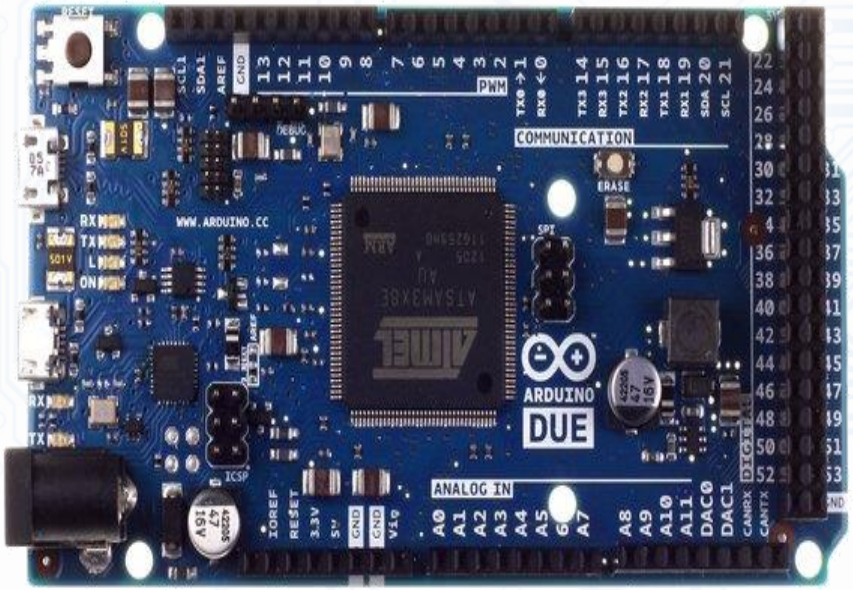



- Adafruit's Feather form factor adds serial-to-USB and a LiPo charger to the ESP8266 or ESP32. It makes working with the ESP very easy.
- You might notice the Feather looks like the MKR. Sadly, while they are visually similar, they are different sizes and do not have a common pinout.
- Lastly is the NodeMCU form factor which is not a form factor! NodeMCU is a firmware that runs a Lua script interpreter on the ESP8266.
- It can be replaced with an Arduino bootloader. Once replaced, you can program the board with the Arduino IDE.
- When you need to add WiFi to a project, ESP-based boards are an excellent starting point.

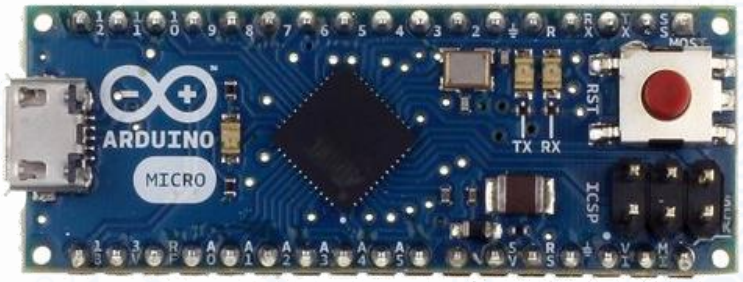
# Due

- Much faster processor, many more pins
- Operates on 3.3 volts
- Similar to the Mega



# Micro

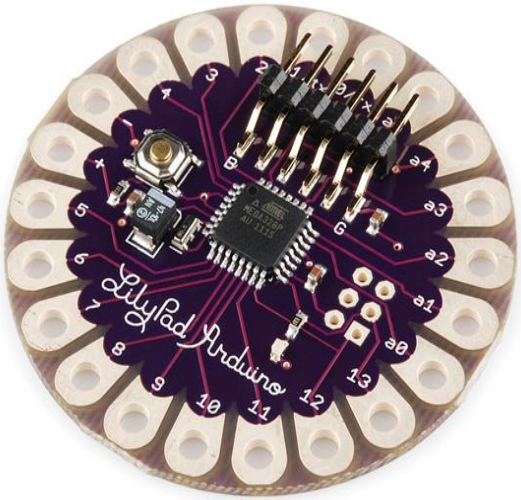
- When size matters: Micro, Nano, Mini
- Includes all functionality of the Leonardo
- Easily usable on a breadboard

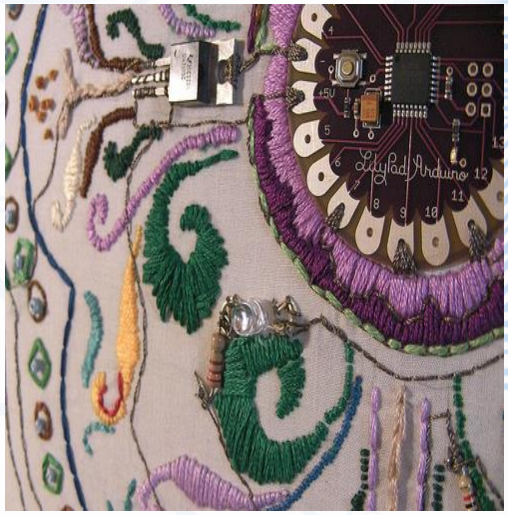


# LilyPad

This is LilyPad Arduino main board! LilyPad is a wearable e-textile technology developed by Leah Buechley and cooperatively designed by Leah and SparkFun. Each LilyPad was creatively designed with large connecting pads and a flat back to allow them to be sewn into clothing with conductive thread. The LilyPad also has its own family of input, output, power, and sensor boards that are also built specifically for e-textiles. They're even washable!

- **LilyPad is popular for clothing-based projects.**



# HISTORY OF ARDUINO

2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 2014 2015 2016 2017

Arduino was born out of the need for a low-cost microcontroller platform for Massimo Banzi's students at the **Interaction Design Institute Ivrea.**

It's named after a local pub: **Bar di Re Arduino.**

The Arduino IDE (Integrated Development Environment) is built upon **Wiring** - a software project written by one of Banzi's students (**Hernando Barragán**). It provides easy-to-use libraries which hide some of the raw C++ going on behind the scenes.

Arduino Begins

IDE revision 0001 released

Atmega8 used for the first boards

Atmega168 doubles the flash memory

Atmega328 again doubles the memory

Adafruit estimate 300,000 official boards produced

create.arduino.cc web based IDE is launched

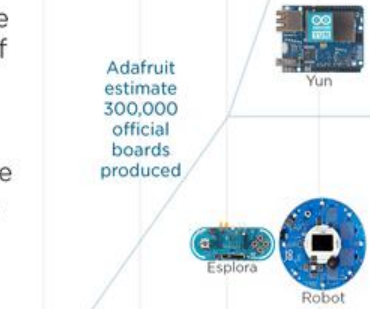
First ever Arduino day 29/03/14

First 32-bit Arduino

IDE 1.8 released

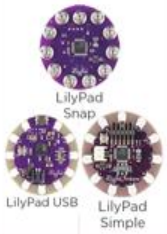
Arduino splits: arduino.cc (Genuino outside USA) and arduino.org

Arduino reunites under Arduino Foundation







## ARDUINO TODAY

### Industrial

- Yun/Yun Mini
- Zero
- M0/M0 Pro
- Tian
- 101/Industrial 101

Powerful, smart technology

Rapid prototyping

Easily integrated with other devices

### Educational

- Esplora
- Robot

Classroom friendly

Modern, STEM learning

Hands-on and intuitive

### IoT

- MKR1000
- MKRZero
- MKRFOX1200
- Uno Wi-Fi
- Ethernet
- Primo

Connectivity and communication

Low power consumption

Easy to prototype with

### Wearables

- LilyPad
- LilyPad Simple
- LilyPad Snap
- LilyPad USB
- Primo Core

Thin, compact form factor

Battery powered

Easy to use with condutive material

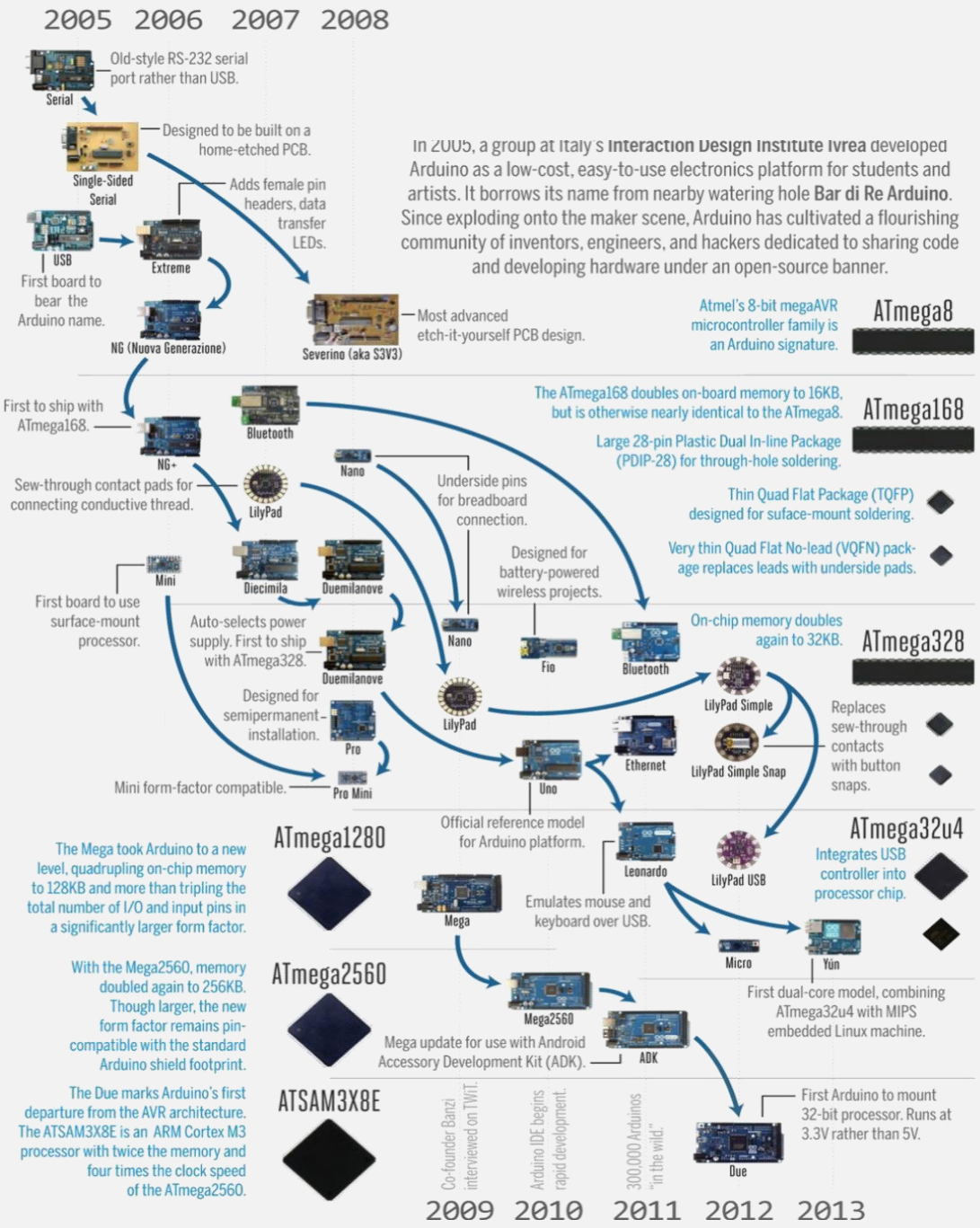
### Maker

- Uno
- Leonardo
- Mini/Pro Mini
- Nano/Micro
- Mega2560/ADK
- Primo
- Due

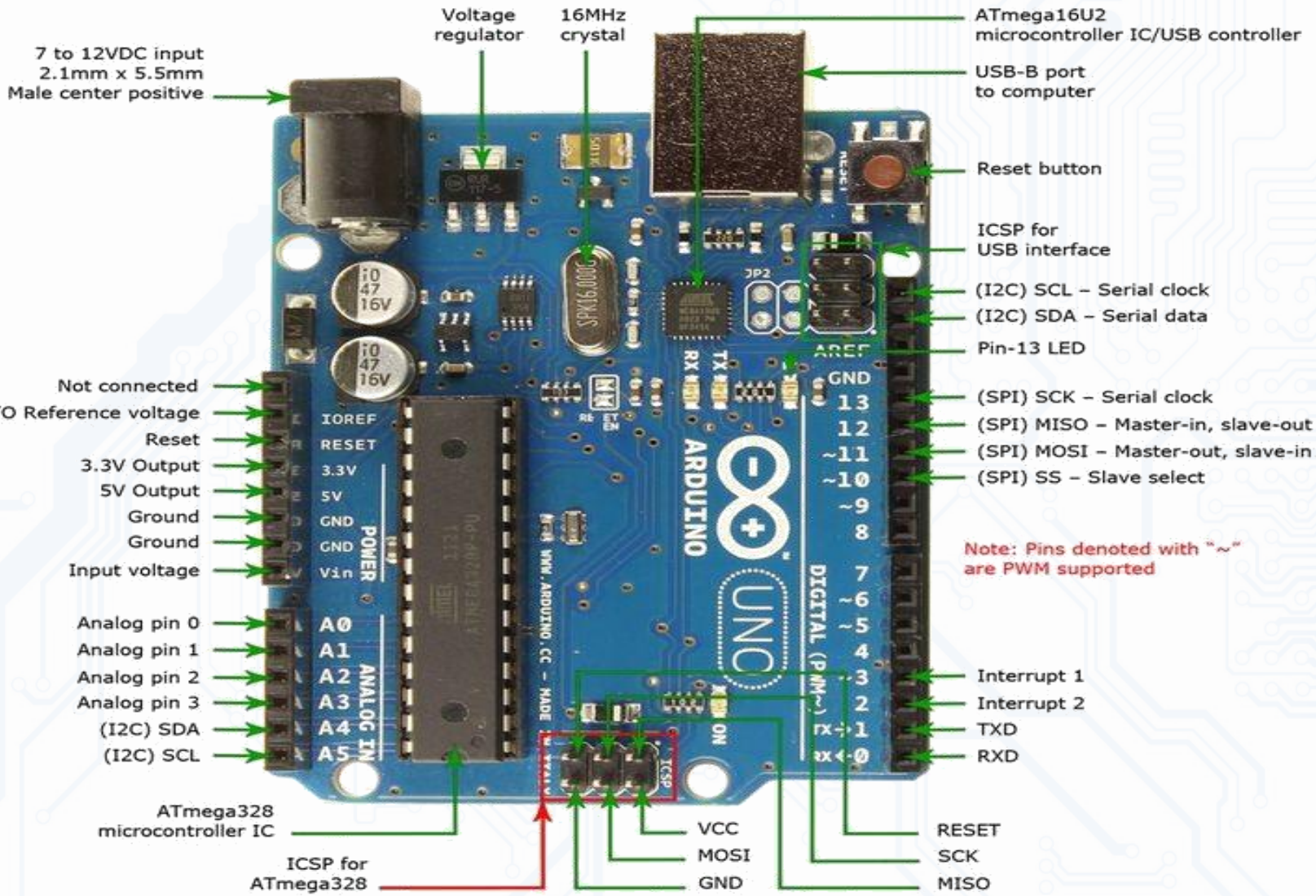
Affordable

Community driven

Modular and adaptable


2005 2006 2007 2008
Old-style RS-232 serial port rather than USB.
Serial
Designed to be built on a home-etched PCB.
Single-Sided Serial
Adds female pin headers, data transfer LEDs.
USB
First board to bear the Arduino name.
Extreme
NG (Nuova Generazione)
Severino (aka S3V3)
Most advanced etch-it-yourself PCB design.
In 2005, a group at Italy's Interaction Design Institute Ivrea developed Arduino as a low-cost, easy-to-use electronics platform for students and artists. It borrows its name from nearby watering hole Bar di Re Arduino. Since exploding onto the maker scene, Arduino has cultivated a flourishing community of inventors, engineers, and hackers dedicated to sharing code and developing hardware under an open-source banner.
Atmel's 8-bit megaAVR microcontroller family is an Arduino signature.
ATmega8
First to ship with ATmega168.
NG+
Bluetooth
The ATmega168 doubles on-board memory to 16KB, but is otherwise nearly identical to the ATmega8.
ATmega168
Large 28-pin Plastic Dual In-line Package (PDIP-28) for through-hole soldering.
Sew-through contact pads for connecting conductive thread.
LilyPad
Nano
Underside pins for breadboard connection.
Thin Quad Flat Package (TQFP) designed for suface-mount soldering.
Very thin Quad Flat No-lead (VQFN) package replaces leads with underside pads.
Mini
First board to use surface-mount processor.
Diecimila
Duemilanove
Designed for battery-powered wireless projects.
Auto-selects power supply. First to ship with ATmega328.
Duemilanove
Nano
Fio
Bluetooth
On-chip memory doubles again to 32KB.
ATmega328
Designed for semipermanent installation.
Pro
LilyPad
LilyPad Simple
Replaces sew-through contacts with button snaps.
Mini form-factor compatible.
Pro Mini
Uno
Ethernet
LilyPad Simple Snap
Official reference model for Arduino platform.
ATmega1280
The Mega took Arduino to a new level, quadrupling on-chip memory to 128KB and more than tripling the total number of I/O and input pins in a significantly larger form factor.
Leonardo
LilyPad USB
ATmega32u4
Integrates USB controller into processor chip.
Mega
Emulates mouse and keyboard over USB.
Micro
Yún
First dual-core model, combining ATmega32u4 with MIPS embedded Linux machine.
ATmega2560
With the Mega2560, memory doubled again to 256KB. Though larger, the new form factor remains pin-compatible with the standard Arduino shield footprint.
Mega2560
ADK
Mega update for use with Android Accessory Development Kit (ADK).
ATSAM3X8E
The Due marks Arduino's first departure from the AVR architecture. The ATSAM3X8E is an ARM Cortex M3 processor with twice the memory and four times the clock speed of the ATmega2560.
Due
First Arduino to mount 32-bit processor. Runs at 3.3V rather than 5V.
Co-founder Banzi interviewed on TWiT.
Arduino IDE begins rapid development.
300,000 Arduinos "in the wild."
2009 2010 2011 2012 2013

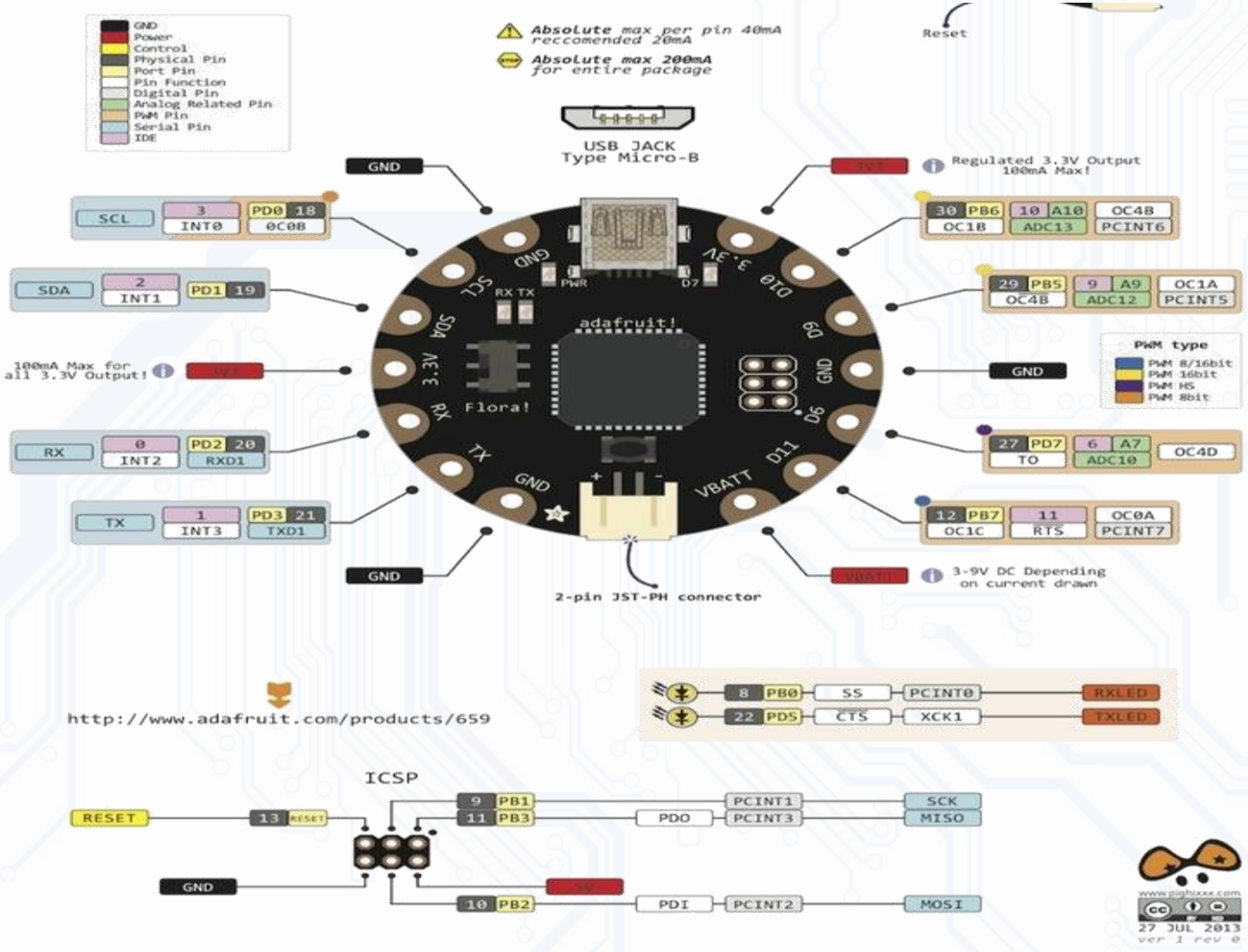
# UNO



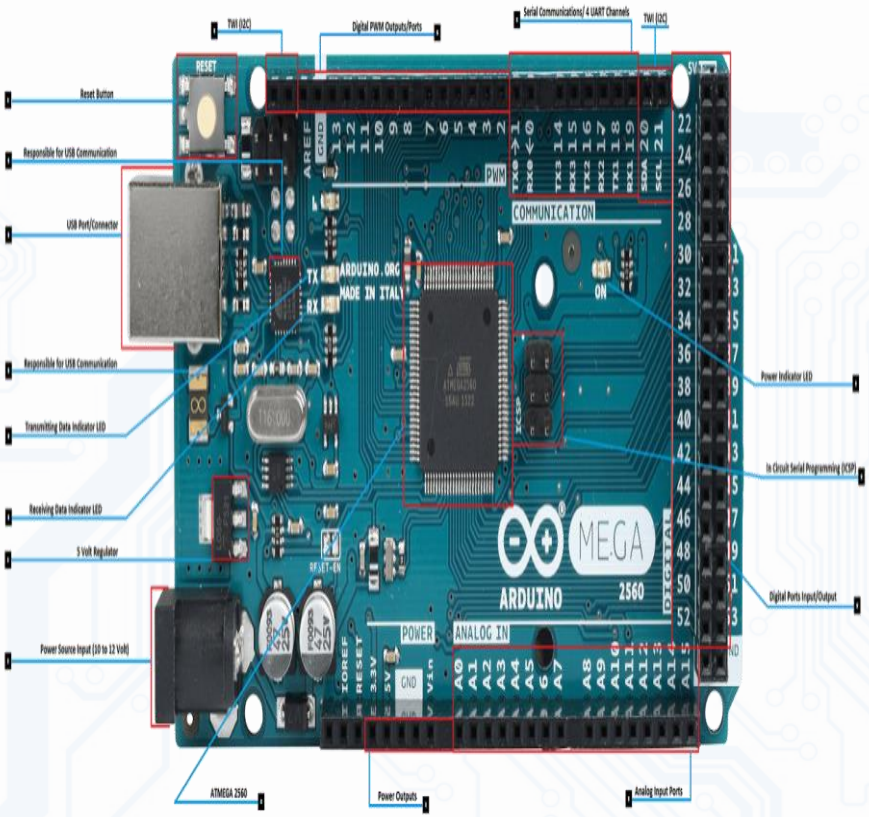

DE MONTFORT UNIVERSITY LEICESTER

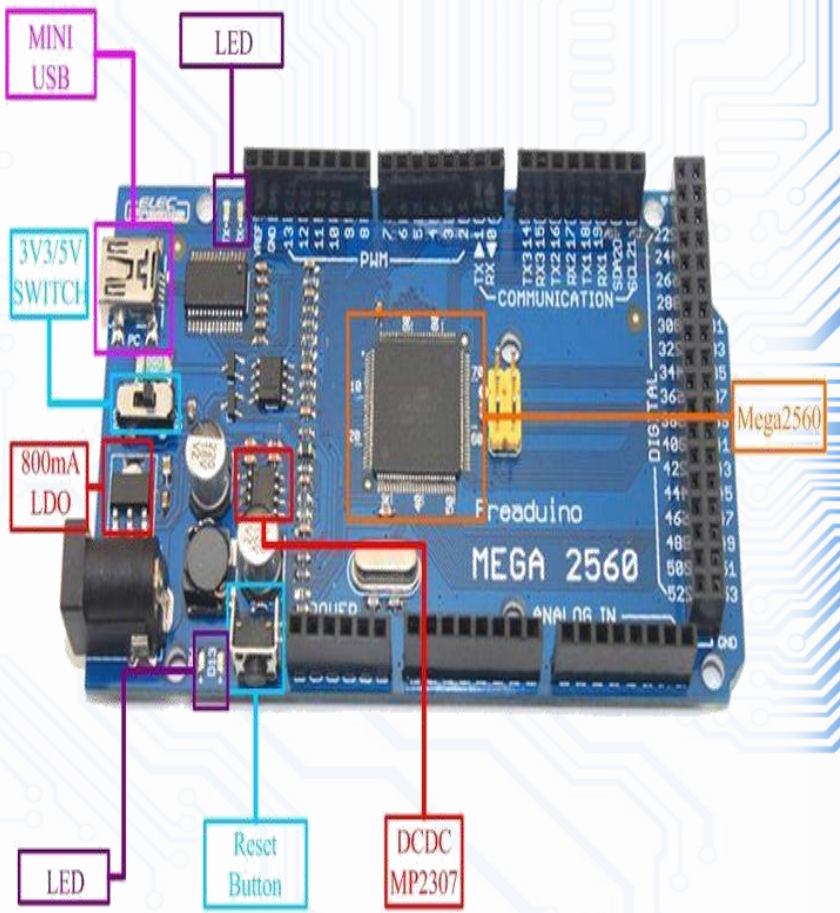
江西理工大學 信息工程学院
JIANGXI UNIVERSITY OF SCIENCE AND TECHNOLOGY SCHOOL OF INFORMATION ENGINEERING

# Arduino LilyPad



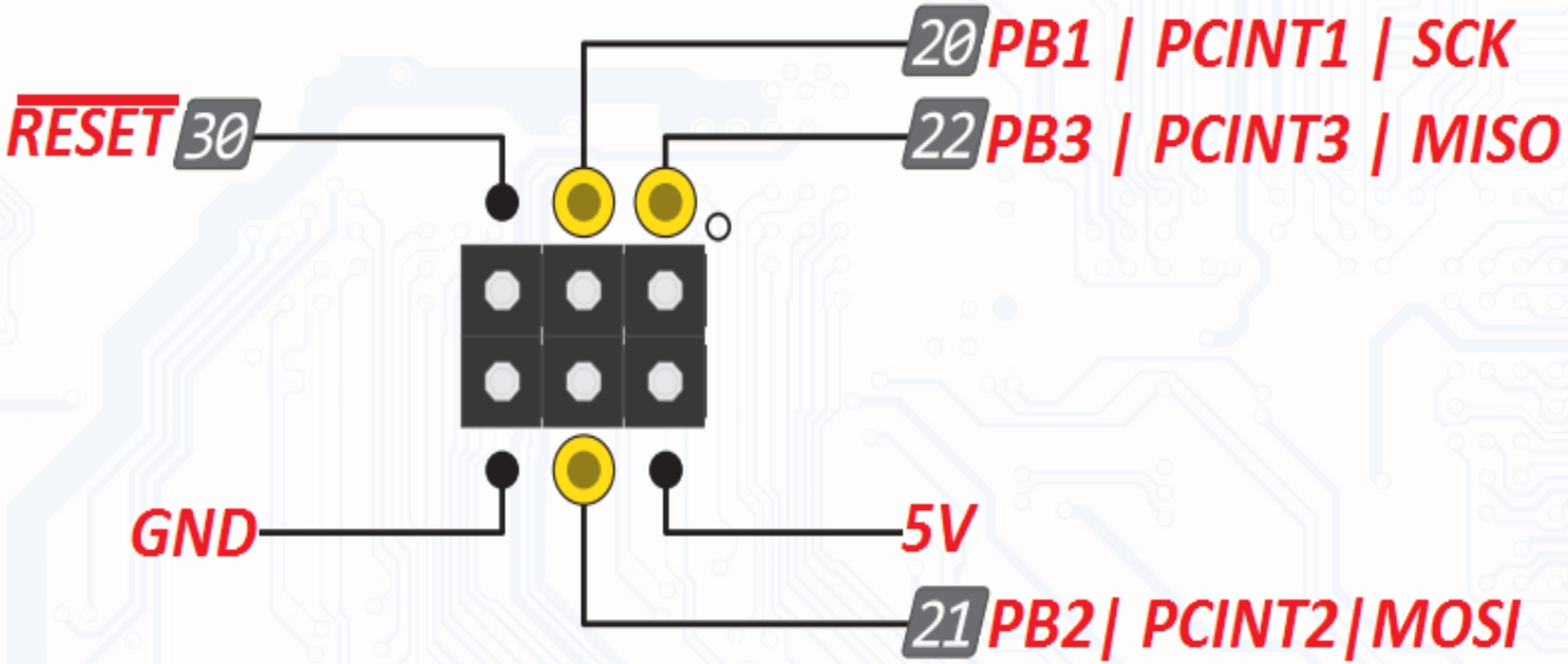
# Arduino Mega Pinout




DE MONTFORT UNIVERSITY LEICESTER

江西理工大學 信息工程学院
JIANGXI UNIVERSITY OF SCIENCE AND TECHNOLOGY SCHOOL OF INFORMATION ENGINEERING



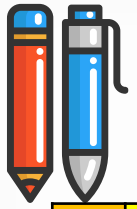

In Circuit Serial Programming Pinout (ICSP)

www.CircuitsToday.com

DE MONTFORT UNIVERSITY LEICESTER

江西理工大學 信息工程学院

JIANGXI UNIVERSITY OF SCIENCE AND TECHNOLOGY SCHOOL OF INFORMATION ENGINEERING



# Arduino Types

| NO | Board name | PROCESSOR | Voltage | Crystal |
|---|---|---|---|---|
| 1 | Arduino Uno | ATmega328 | 5 V/7-12 V | 16MHz |
| 2 | Due | AT91SAM3X8E | 3.3 V/7-12 V | 84 MHz |
| 3 | Leonardo | ATmega32u4 | 5 V/7-12 V | 16MHz |
| 4 | Mega 2560 | ATmega2560 | 5 V/7-12 V | 16MHz |
| 5 | Mega ADK | ATmega2560 | 5 V/7-12 V | 16MHz |
| 6 | Micro | ATmega32u4 | 5 V/7-12 V | 16MHz |
| 7 | Mini | ATmega328 | 5 V/7-9 V | 16MHz |
| 8 | Nano | ATmega168 ATmega328 | 5 V/7-9 V | 16MHz |
| 9 | Ethernet | ATmega328 | 5 V/7-12 V | 16MHz |
| 10 | Esplora | ATmega32u4 | 5 V/7-12 V | 16MHz |
| 11 | ArduinoBT | ATmega328 | 5 V/2.5-12 V | 16MHz |
| 12 | Fio | ATmega328P | 3.3 V/3.7-7 V | 8MHz |
| 13 | Pro (168) | ATmega168 | 3.3 V/3.35-12 V | 8MHz |
| 14 | Pro (328) | ATmega328 | 5 V/7-12 V | 16MHz |
| 15 | Pro Mini | ATmega168 | 3.3 V/3.35-12 V 5 V/5-12 V | 8MHz 16MHz |
| 16 | LilyPad | ATmega168V ATmega328V | 2.7-5.5 V/2.7-5.5 V | 8MHz |
| 17 | LilyPad USB | ATmega32u4 | 3.3 V/3.8-5V | 8MHz |
| 18 | LilyPad Simple | ATmega328 | 2.7-5.5 V/2.7-5.5 V | 8MHz |
| 19 | LilyPad SimpleSnap | ATmega328 | 2.7-5.5 V/2.7-5.5 V | 8MHz |
| 20 | Yun | ATmega32u4 | 5 V | 16MHz |

# Arduino Programming Cheat Sheet

Primary source: Arduino Language Reference
http://arduino.cc/en/Reference/

## Structure & Flow

### Basic Program Structure

```
void setup() {
  // runs once when sketch starts
}
void loop() {
  // runs repeatedly
}
```

### Control Structures

```
if (x < 5) { ... } else { ... }
while (x < 5) { ... }
do { ... } while ( x < 5);
for (int i = 0; i < 10; i++) { ... }
break;  // exit a loop immediately
continue;  // go to next iteration
switch (myVar) {
   case 1:
     ...
     break;
   case 2:
     ...
     break;
   default:
     ...
}
return x; // just return; for voids
```

## Operators

### General Operators

```
=  (assignment operator)
+  (add)          -  (subtract)
*  (multiply)     /  (divide)
%  (modulo)
== (equal to)     != (not equal to)
<  (less than)    >  (greater than)
<= (less than or equal to)
>= (greater than or equal to)
&& (and)     || (or)     ! (not)
```

### Compound Operators

```
++ (increment)
-- (decrement)
+= (compound addition)
-= (compound substraction)
*= (compound multiplication)
/= (compound division)
&= (compound bitwise and)
|= (compound bitwise or)
```

### Bitwise Operators

```
&  (bitwise and)    |  (bitwise or)
^  (bitwise xor)    ~  (bitwise not)
<< (shift left)     >> (shift right)
```

## Variables, Arrays, and Data

### Data types

```
void
boolean   (0, 1, true, false)
char      (e.g. 'a'  -128 to 127)
int       (-32768 to 32767)
long      (-2147483648 to 2147483647)
unsigned char  (0 to 255)
byte           (0 to 255)
unsigned int   (0 to 65535)
word           (0 to 65535)
unsigned long  (0 to 4294967295)
float     (-3.4028e+38 to 3.4028e+38)
double    (currently same as float)
```

### Qualifiers

```
static    (persists between calls)
volatile  (in RAM (nice for ISR))
const     (make read only)
PROGMEM   (in flash)
```

### Arrays

```
int myInts[6]; // array of 6 ints
int myPins[]={2, 4, 8, 3, 6};
int mySensVals[6]={2, 4, -8, 3, 2};
myInts[0]=42;  // assigning first
               // index of myInts
myInts[6]=12;  // ERROR! Indexes
               // are 0 though 5
```

### Constants

```
HIGH  | LOW
INPUT | OUTPUT
true  | false
143         (Decimal)
0173        (Octal - base 8)
0b11011111  (Binary)
0x7B        (Hexadecimal - base 16)
7U          (force unsigned)
10L         (force long)
15UL        (force long unsigned)
10.0        (force floating point)
2.4e5       (2.4*10^5 = 240000)
```

### Pointer Access

```
&  (reference: get a pointer)
*  (dereference: follow a pointer)
```

### Strings

```
char S1[8] =
 {'A','r','d','u','i','n','o'};
 // unterminated string; may crash
char S2[8] =
 {'A','r','d','u','i','n','o','\0'};
 // includes \0 null termination
char S3[]="Arduino";
char S4[8]="Arduino";
```

## Built-in Functions

### Pin Input/Output

```
Digital I/O (pins: 0-13 A0-A5)
  pinMode(pin,[INPUT, OUTPUT])
  int digitalread(pin)
  digitalWrite(pin, value)
    // Write HIGH to an input to
    // enable pull-up resistors
Analog In (pins: 0-5)
  int analogRead(pin)
  analogReference(
    [DEFAULT, INTERNAL, EXTERNAL])
PWM Out (pins: 3 5 6 9 10 11)
  analogWrite(pin, value)
```

### Advanced I/O

```
tone(pin, freqhz)
tone(pin, freqhz, duration_ms)
noTone(pin)
shiftOut(dataPin, clockPin,
  [MSBFIRST,LSBFIRST], value)
unsigned long pulseIn(pin,
  [HIGH,LOW])
```

### Time

```
unsigned long millis()
  // overflows at 50 days
unsigned long micros()
  // overflows at 70 minutes
delay(msec)
delayMicroseconds(usec)
```

### Math

```
min(x, y)   max(x, y)   abs(x)
sin(rad)    cos(rad)    tan(rad)
sqrt(x)     pow(base, exponent)
constrain(x, minval, maxval)
map(val, fromL, fromH, toL, toH)
```

### Random Numbers

```
randomSeed(seed) // long or int
long random(max)
long random(min, max)
```

### Bits and Bytes

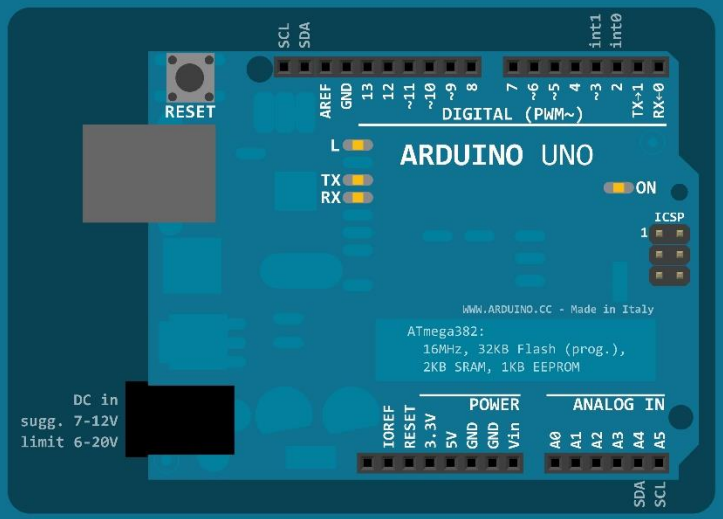
```
lowByte(x)    highByte(x)
bitRead(x, bitn)
bitWrite(x, bitn, bit)
bitSet(x, bitn)
bitClear(x, bitn)
bit(bitn)  // bitn: 0=LSB 7=MSB
```

### Type Conversions

```
char()        byte()
int()         word()
long()        float()
```

### External Interrupts

```
attachInterrupt(interrupt, func,
 [LOW, CHANGE, RISING, FALLING])
detachInterrupt(interrupt)
interrupts()
noInterrupts()
```




## Libraries

### Serial (communicate with PC or via RX/TX)

```
begin(long Speed) // up to 115200
end()
int available() // #bytes available
byte read() // -1 if none available
byte peek()
flush()
print(myData)
println(myData)
write(myBytes)
SerialEvent() // called if data rdy
```

### SoftwareSerial (serial comm. on any pins)

```
  (#include <softwareSerial.h>)
SoftwareSerial(rxPin, txPin)
begin(long Speed) // up to 115200
listen()      // Only 1 can listen
isListening() // at a time.
read, peek, print, println, write
  // all like in Serial library
```

### EEPROM (#include <EEPROM.h>)

```
byte read(intAddr)
write(intAddr, myByte)
```

### Servo (#include <Servo.h>)

```
attach(pin, [min_uS, max_uS])
write(angle)  // 0 to 180
writeMicroseconds(uS)
   // 1000-2000; 1500 is midpoint
int read()    // 0 to 180
bool attached()
detach()
```

### Wire (I²C comm.) (#include <Wire.h>)

```
begin()      // join a master
begin(addr) // join a slave @ addr
requestFrom(address, count)
beginTransmission(addr) // Step 1
send(myByte)            // Step 2
send(char * mystring)
send(byte * data, size)
endTransmission()       // Step 3
int available() // #bytes available
byte receive()  // get next byte
onReceive(handler)
onRequest(handler)
```



by Mark Liffiton

Adapted from:
- Original by Gavin Smith
- SVG version by Frederic Dufourg
- Arduino board drawing original by Fritzing.org

# ARDUINO CHEAT SHEET

Content for this Cheat Sheet provided by Gavin from Robots and Dinosaurs.
For more information visit: http://arduino.cc/en/Reference/Extended

sparkfun ELECTRONICS

## Structure
void **setup**() void **loop**()

## Control Structures
**if** (x<5){ } **else** { }
**switch** (myvar) {
  **case** 1:
  **break**;
  **case** 2:
  **break**;
  **default**:
}
**for** (int i=0; i <= 255; i++){ }
**while** (x<5){ }
**do** { } **while** (x<5);
**continue**; //Go to next in do/for/while loop
**return** x; // Or 'return;' for voids.
**goto** // considered harmful :-)

## Further Syntax
// (single line comment)
/* (multi-line comment) */
**#define** DOZEN 12 //Not baker's!
**#include** <avr/pgmspace.h>

## General Operators
= (assignment operator)
+ (addition) - (subtraction)
* (multiplication) / (division)
% (modulo)
== (equal to) != (not equal to)
< (less than) > (greater than)
<= (less than or equal to)
>= (greater than or equal to)
&& (and) || (or) ! (not)

## Pointer Access
& reference operator
* dereference operator

## Bitwise Operators
& (bitwise and) | (bitwise or)
^ (bitwise xor) ~ (bitwise not)
<< (bitshift left) >> (bitshift right)

## Compound Operators
++ (increment) -- (decrement)
+= (compound addition)
-= (compound subtraction)
*= (compound multiplication)
/= (compound division)
&= (compound bitwise and)
|= (compound bitwise or)

## Constants
HIGH | LOW
INPUT | OUTPUT
true | false
143 // **Decimal** number
0173 // **Octal** number
**0b**11011111 //**Binary**
0x7B // **Hex** number
7**U** // Force unsigned
10**L** // Force long
15**UL** // Force long unsigned
10.0 // Forces floating point
2.4e5 // 240000

## Data Types
**void**
**boolean** (0, 1, false, true)
**char** (e.g. 'a' -128 to 127)
**unsigned char** (0 to 255)
**byte** (0 to 255)
**int** (-32,768 to 32,767)
**unsigned int** (0 to 65535)
**word** (0 to 655word (0 to 65535)
**long** (-2,147,483,648 to 2,147,483,647)
**unsigned long** (0 to 4,294,967,295)
**float** (-3.4028235E+38 to 3.4028235E+38)
**double** (currently same as float)
**sizeof**(myint) // returns 2 bytes

## Strings
char S1[15];
char S2[8]={'a','r','d','u','i','n','o'};
char S3[8]={'a','r','d','u','i','n','o','\0'};
//Included \0 null termination
char S4[ ] = "arduino";
char S5[8] = "arduino";
char S6[15] = "arduino";

## Arrays
int myInts[6];
int myPins[] = {2, 4, 8, 3, 6};
int mySensVals[6] = {2, 4, -8, 3, 2};

## Conversion
**char**() **byte**()
**int**() **word**()
**long**() **float**()

## Qualifiers
**static** // persists between calls
**volatile** // use RAM (nice for ISR)
**const** // make read-only
**PROGMEM** // use flash

## Digital I/O
**pinMode**(pin, [INPUT,OUTPUT])
**digitalWrite**(pin, value)
int **digitalRead**(pin)
//Write High to inputs to use pull-up res

## Analog I/O
**analogReference**([DEFAULT, INTERNAL,EXTERNAL])
int **analogRead**(pin) //Call twice if switching pins from high Z source.
**analogWrite**(pin, value) // PWM

## Advanced I/O
**tone**(pin, freqhz)
**tone**(pin, freqhz ,duration_ms)
**noTone**(pin)
**shiftOut**(dataPin, clockPin, [MSBFIRST,LSBFIRST], value)
unsigned long **pulseIn**(pin,[HIGH,LOW])

## Time
unsigned long **millis**() // 50 days overflow.
unsigned long **micros**() // 70 min overflow
**delay**(ms)
**delayMicroseconds**(us)

## Math
**min**(x, y) **max**(x, y) **abs**(x)
**constrain**(x, minval, maxval)
**map**(val, fromL, fromH, toL, toH)
**pow**(base, exponent) **sqrt**(x)
**sin**(rad) **cos**(rad) **tan**(rad)

## Random Numbers
**randomSeed**(seed) // Long or int
long **random**(max)
long **random**(min, max)

## Bits and Bytes
**lowByte**()
**highByte**()
**bitRead**(x,bitn)
**bitWrite**(x,bitn,bit)
**bitSet**(x,bitn)
**bitClear**(x,bitn)
**bit**(bitn) //bitn: 0-LSB 7-MSB

## External Interrupts
**attachInterrupt**(interrupt, function, [LOW,CHANGE,RISING,FALLING])
**detachInterrupt**(interrupt)
**interrupts**()
**noInterrupts**()

## Libraries:

**Serial.**
  **begin**([300, 1200, 2400, 4800, 9600,14400, 19200, 28800, 38400, 57600,115200])
  **end**()
  int **available**()
  int **read**()
  **flush**()
  **print**()
  **println**()
  **write**()

**EEPROM** (#include <EEPROM.h>)
byte **read**(intAddr)
  **write**(intAddr,myByte)

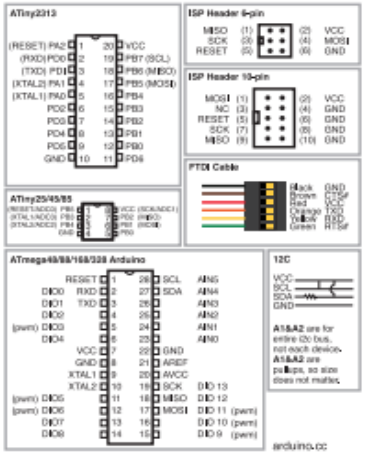
**Servo** (#include <Servo.h>)
  **attach**(pin , [min_uS, max_uS])
  **write**(angle) // 0-180
  **writeMicroseconds**(uS) //1000-2000,1500 is midpoint
  **read**() // 0-180
  **attached**() //Returns boolean
  **detach**()

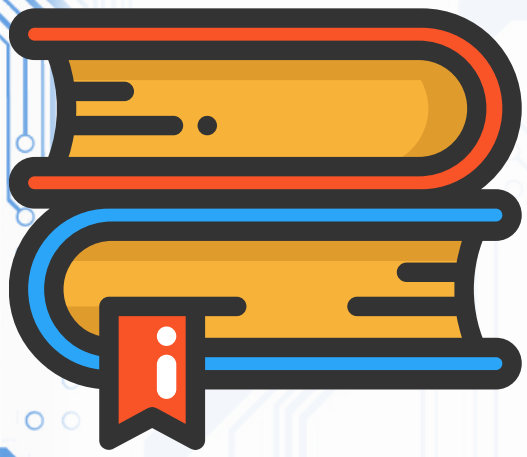
**SoftwareSerial**(RxPin,TxPin)
  // #include<**SoftwareSerial.**h>
  **begin**(longSpeed) // up to 9600
  char **read**() // blocks till data
  **print**(myData) or **println**(myData)

**Wire** (#include <Wire.h>) // For **I2C**
  **begin**() // Join as master
  **begin**(addr) // Join as slave @ addr
  **requestFrom**(address, count)
  **beginTransmission**(addr) // Step 1
  **send**(mybyte) // Step 2
  **send**(char * mystring)
  **send**(byte * data, size)
  **endTransmission**() // Step 3
  byte **available**() // Num of bytes
  byte **receive**() //Return next byte
  **onReceive**(handler)
  **onRequest**(handler)

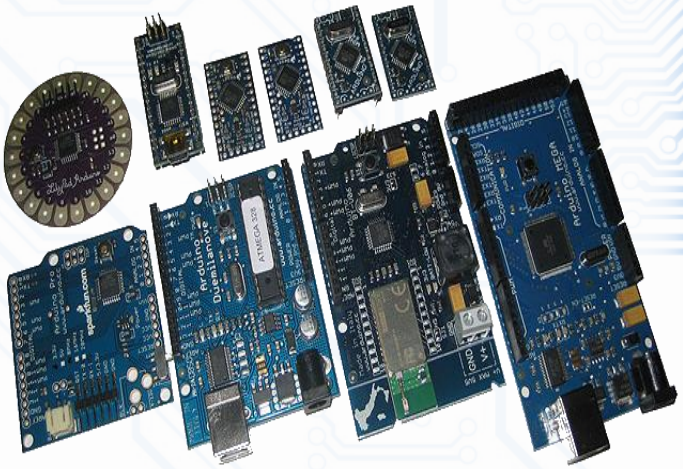
| | ATMega168 | ATMega328 | ATMega1280 |
|---|---|---|---|
| Flash (2k for bootloader) | 16kB | 32kB | 128kB |
| SRAM | 1kB | 2kB | 8kB |
| EEPROM | 512B | 1kB | 4kB |

| | Duemilanove/ Nano/ Pro/ ProMini | Mega |
|---|---|---|
| # of IO | 14 + 6 analog (Nano has 14 + 8) | 54 + 16 analog |
| Serial Pins | 0 - RX<br>1 - TX | 0 - RX1 1 - TX1<br>19 - RX2 18 - TX2<br>17 - RX3 16 - TX3<br>15 - RX4 14 - TX4 |
| Ext Interrupts | 2 - (Int 0)<br>1 - (Int 1) | 2,3,21,20,19,18<br>(IRQ0 - IFQ5) |
| PWM Pins | 5,6 - Timer 0<br>9,10 - Timer 1<br>3,11 - Timer 2 | 0 - 13 |
| SPI | 10 - SS<br>11 - MOSI<br>12 - MISO<br>13 - SCK | 53 - SS<br>51 - MOSI<br>50 - MISO<br>52 - SCK |
| I2C | Analog4 - SDA<br>Analog5 - SCK | 20 - SDA<br>21 - SCL |





# Fast Review On C Language



江西理工大學信息工程学院
JIANGXI UNIVERSITY OF SCIENCE AND TECHNOLOGY SCHOOL OF INFORMATION ENGINEERING
DE MONTFORT UNIVERSITY LEICESTER

江西理工大學信息工程学院
JIANGXI UNIVERSITY OF SCIENCE AND TECHNOLOGY SCHOOL OF INFORMATION ENGINEERING

# Arduino Code Basics

- **Arduino programs run on two basic sections:**

## Setup

```
void setup() {
        //setup motors, sensors etc
}
void loop() {
        // get information from sensors
        // send commands to motors
}
```

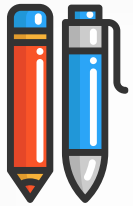
```
void setup() {

        pinMode(9, OUTPUT);

}
```

port #

Input or Output

## SETUP

- **The setup section is used for assigning input and outputs (Examples: motors, LED's, sensors etc) to ports on the Arduino**
- **It also specifies whether the device is OUTPUT or INPUT**
- **To do this we use the command "pinMode"**



# Arduino Code Basics

```
void setup() {
        //setup motors, sensors etc
}
void loop() {
        // get information from sensors
        // send commands to motors
}
```

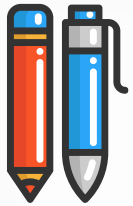
LOOP

```
void loop() {

    digitalWrite(9, HIGH);
    delay(1000);
    digitalWrite(9, LOW);
    delay(1000);

}
```

Port # from setup

Turn the LED on or off

Wait for 1 second or 1000 milliseconds



# Variables in C

- An important aspect of the C language is how it stores data.
- Here we will discuss:
  - Data types
  - Declarations
  - Assignments
  - Data type ranges
  - Type conversions

# Variables in C

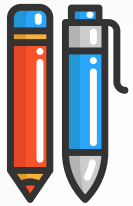
- The following table shows the meaning of the basic data types and type modifiers:

| Type | Meaning | Modifier |
|---|---|---|
| Character | Character data | char |
| Integer | Signed whole numbers | int |
| Float | Floating point numbers | float |
| Double | Double precision floating point numbers | double |
| Signed | Positive and negative numbers | signed |
| Unsigned | Positive only numbers | unsigned |
| Long | Double the length of a number | long |
| Short | Halves the length of a number | short |

C represents all negative numbers in the 2's complement format. For example to convert the signed number 29 into 2's complement:

```
00011101=   29
11100010    invert all bits
      1     add 1
11100011=   -29
```

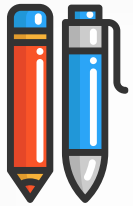
```
2          0000 0010
           1111 1101  +
                   1
           ----------
-2== FE    1111 1110
```



# Data types

| Type | Bit width | Description | Range |
|---|---|---|---|
| bit | 1 | To define a bit of data | 0 - 1 |
| Char | 8 | Character or small integer. | signed: -128 to 127<br>unsigned: 0 to 255 |
| Short Int | 16 | Used to define integer numbers | signed: -32768 to 32767<br>unsigned: 0 to 65535 |
| Long int | 32 | Used to define integer numbers | signed: -2147483648 to 2147483647<br>unsigned: 0 to 4294967295 |
| Float | 32 | Define floating point number | 1.175e-38 to 3.40e+38 |
| Double | 32 | Used to define largest integer numbers | 1.175e-38 to 3.40e+38<br>**Note: in other C ref 64 bit** |
| bool | 8 | Boolean value. | true or false |
| * (pointer) | width of memory | Use for addressing memory | Range of memory |

**For more details refer to "C Language Compiler User Guide"**



# Variable Declaration

- Variables can be declared inside a function or outside of all functions.
- Variables are declared as following:

- Where type is one of C's valid data types and variable_name is the name of the variable.

```
type  variable_name;
```

**Ex: float price;**

- Global variables are declared outside functions and are visible from the end of declaration to the end of the file.
- Local variables are declared inside a function and is visible only from the end of declaration to the end of the function.

# Variable Assignment

- Assignment of values to variables is simple:

  variable_name = value ;

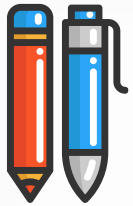
- Since a variable assignment is a statement, we have to include the semicolon at the end.

- Example:
  - count =100;
  - Int x=10;
  - float PI=3.14;

```
typedef signed char    smallint;

main()
    {
        smalint i;
        /* any data */
    }
```

  typedef old_name  new_name

- typedef:  used to create a new name for an existing type.



# Variable Assignment

- typedef are typically used for two reasons:
  - To create portable programs. (to make program works on Different μP data width bus)

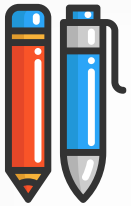
```
typedef   short   int   myint;          // int 8-bit for 8-bit processor

typedef    int    myint;               // int 16-bit for 16-bit processor
```

  - help to document your code.

**(If your code contains many variables used to hold a count of some sort)**

```
typedef   int   counter;
```

  - Someone reading your code would recognize that any variable declared as counter is used as a counter in the program.



# Type conversions (casting/parsing)

- **To convert from long to int the programmer has to manually type cast the value**

(type) value

- To do type casting, the value to cast has to be preceded by the target type enclosed in parentheses

```
int    Val16bit;                  // short integer (16 bits)
long Val32bit;                    // long integer (32 bits)
Val16bit = (int) Val32bit;        // type casting
                                  // the higher 16 bits are lost
```

- Information may be lost by type casting
- The range of a value should be checked before doing manual type casting

# Variable Storage Class

- **auto**
  - The auto specifier indicates that the memory location of a variable is **temporary**.
  - auto is the default for function/block variables
    - auto int a is the same as int a because it is the **default**, it is almost never used
- **extern**
  - Declares a variable that is defined somewhere else.
  - Useful when splitting software in multiple files.

In *declare.c*:

```
int farvar;
```

In *use.c*:

```
{
        extern int farvar;
        int a;
        a = farvar * 2;
}
```

```
{   char c;
    int a, b, e;
}
```

is the same as

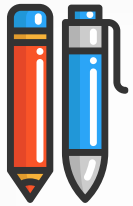
```
{   autocharc;
    autoint a, b, e;
}
```

# Variable Storage Class

- **static**
  - Variable stored in static memory.
  - **It retain their previous value on re-entry to a block of code. So it eat up your memory.**
  - In the example the variable count is initialized once and thereafter increment every time the function test is called.
- **register**
  - used to define local variables that should be stored in a register instead of RAM (variable should be stored in a processor register).
  - Register should only be used for variables that require quick access

The storage class is optional – if not specified the compiler uses a default storage class.

```
void test()
{
    charx,y,z;
    static int  count = 0;
    printf("count = %d\n",++count);
}
```



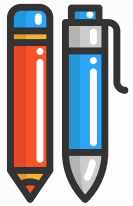
# Operators

- An operator is a symbol that tells the compiler to perform specific mathematical or logical functions. C language is rich in built-in operators and provides the following types of operators
  - Arithmetic Operators
  - Comparison Operators
  - Boolean Operators
  - Bitwise Operators
  - Compound Operators

# Operators

| Operator name | Operator simple | Description | Example |
|---|---|---|---|
| assignment operator | = | Stores the value to the right of the equal sign in the variable to the left of the equal sign. | A = B |
| addition | + | Adds two operands | A + B will give 30 |
| subtraction | - | Subtracts second operand from the first | A - B will give -10 |
| multiplication | * | Multiply both operands | A * B will give 200 |
| division | / | Divide numerator by denominator | B / A will give 2 |
| modulo | % | Modulus Operator and remainder of after an integer division | B % A will give 0 |

```
void loop () {
        int a = 9,b = 4,c;
       c = a + b;
       c = a - b;
        c = a * b;
       c = a / b;
        c = a % b; }
```



# Operators

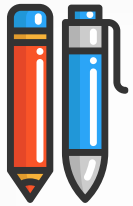
- **Arithmetic Operators:**
  - Assume variable A holds 10 and variable B holds 20 then:

| Operator | Description | Example |
|---|---|---|
| + | Adds two operands | A + B will give 30 |
| - | Subtracts second operand from the first | A - B will give -10 |
| * | Multiply both operands | A * B will give 200 |
| / | Divide numerator by denumerator | B / A will give 2 |
| % | Modulus Operator and remainder of after an integer division | B % A will give 0 |
| ++ | Increment operator, increases integer value by one | A++ will give 11 |
| -- | Decrement operator, decreases integer value by one | A-- will give 9 |

# Arduino - Comparison Operators

| Operator name | Operator or simple | Description | Example |
|---|---|---|---|
| equal to | = = | Checks if the value of two operands is equal or not, if yes then condition becomes true. | (A == B) is not true |
| not equal to | ! = | Checks if the value of two operands is equal or not, if values are not equal then condition becomes true. | (A != B) is true |
| less than | < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | (A < B) is true |
| greater than | > | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | (A > B) is not true |
| less than or equal to | < = | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | (A <= B) is true |
| greater than or equal to | > = | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (A >= B) is not true |

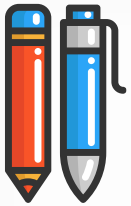
```
void loop () {
   int a = 9,b = 4
   bool c = false;
   if(a == b)
      c = true;
   else
      c = false;

   if(a != b)
      c = true;
   else
      c = false;

   if(a < b)
      c = true;
   else
      c = false;

   if(a > b)
      c = true;
   else
      c = false;

   if(a <= b)
      c = true;
   else
      c = false;

   if(a >= b)
      c = true;
   else
      c = false;
}
```



# Arduino - Boolean Operators

| Operator name | Operator simple | Description | Example |
|---|---|---|---|
| and | && | Called Logical AND operator. If both the operands are non-zero then then condition becomes true. | (A && B) is true |
| or | \|\| | Called Logical OR Operator. If any of the two operands is non-zero then then condition becomes true. | (A \|\| B) is true |
| not | ! | Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false. | !(A && B) is false |

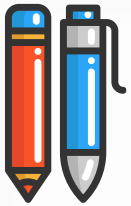
```
void loop () {
   int a = 9,b = 4
   bool c = false;
   if((a > b)&& (b < a))
      c = true;
   else
      c = false;

   if((a == b)|| (b < a))
      c = true;
   else
      c = false;

   if( !(a == b)&& (b < a))
      c = true;
   else
      c = false;
}
```



# Operators

- **Logical (or Relational) Operators:**
  - Assume variable A holds 10 and variable B holds 20 then

| Operator | Description | Example |
|---|---|---|
| == | Checks if the value of two operands is equal or not, if yes then condition becomes true. | (A == B) is not true. |
| != | Checks if the value of two operands is equal or not, if values are not equal then condition becomes true. | (A != B) is true. |
| > | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | (A > B) is not true. |
| < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | (A < B) is true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (A >= B) is not true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | (A <= B) is true. |
| && | Called Logical AND operator. If both the operands are non zero then then condition becomes true. | (A && B) is true. |
| \|\| | Called Logical OR Operator. If any of the two operands is non zero then then condition becomes true. | (A \|\| B) is true. |
| ! | Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false. | !(A && B) is false. |

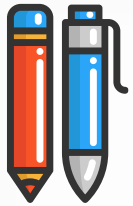


# Operators

- **Bitwise Operators:**
  - Bitwise Operators: Bitwise operator works on bits and perform bit by bit operation.
  - Assume if A = 60; and B = 13; Now in binary format they will be as follows:

    A = 0011 1100 , B = 0000 1101

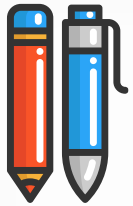
| Operator | Description | Example |
|---|---|---|
| & | Binary AND Operator copies a bit to the result if it exists in both operands. | (A & B) will give 12 which is 0000 1100 |
| \| | Binary OR Operator copies a bit if it exists in eather operand. | (A \| B) will give 61 which is 0011 1101 |
| ^ | Binary XOR Operator copies the bit if it is set in one operand but not both. | (A ^ B) will give 49 which is 0011 0001 |
| ~ | Binary Ones Complement Operator is unary and has the efect of 'flipping' bits. | (~A ) will give -60 which is 1100 0011 |
| << | Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand. | A << 2 will give 240 which is 1111 0000 |
| >> | Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. | A >> 2 will give 15 which is 0000 1111 |



# Arduino - Bitwise Operators

| Operator name | Operator simple | Description | Example |
|---|---|---|---|
| and | & | Binary AND Operator copies a bit to the result if it exists in both operands. | (A & B) will give 12 which is 0000 1100 |
| or | \| | Binary OR Operator copies a bit if it exists in either operand | (A \| B) will give 61 which is 0011 1101 |
| xor | ^ | Binary XOR Operator copies the bit if it is set in one operand but not both. | (A ^ B) will give 49 which is 0011 0001 |
| not | ~ | Binary Ones Complement Operator is unary and has the effect of 'flipping' bits. | (~A ) will give -60 which is 1100 0011 |
| shift left | << | Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand. | A << 2 will give 240 which is 1111 0000 |
| shift right | >> | Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. | A >> 2 will give 15 which is 0000 1111 |

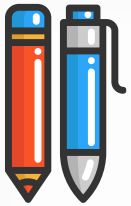
```
void loop () {
   int a = 10,b = 20
   int c = 0;
   c = a & b ;
   c = a | b ;
   c = a ^ b ;
   c = a ~ b ;
   c = a << b ;
   c = a >> b ;
}
```



# Arduino - Compound Operators

| Operator name | Operator simple | Description | Example |
|---|---|---|---|
| **increment** | ++ | Increment operator, increases integer value by one | A++ will give 11 |
| **decrement** | -- | Decrement operator, decreases integer value by one | A-- will give 9 |
| **compound addition** | += | Add AND assignment operator. It adds right operand to the left operand and assign the result to left operand | B += A is equivalent to B = B+ A |
| **compound subtraction** | -= | Subtract AND assignment operator. It subtracts right operand from the left operand and assign the result to left operand | B -= A is equivalent to B = B - A |
| **compound multiplication** | *= | Multiply AND assignment operator. It multiplies right operand with the left operand and assign the result to left operand | B*= A is equivalent to B = B* A |
| **compound division** | /= | Divide AND assignment operator. It divides left operand with the right operand and assign the result to left operand | B /= A is equivalent to B = B / A |
| **compound modulo** | %= | Modulus AND assignment operator. It takes modulus using two operands and assign the result to left operand | B %= A is equivalent to B = B % A |
| **compound bitwise or** | \|= | bitwise inclusive OR and assignment operator | A \|= 2 is same as A = A \| 2 |
| **compound bitwise and** | **&=** | **Bitwise AND assignment operator** | **A &= 2 is same as A = A & 2** |

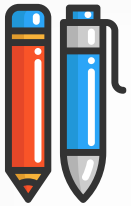
```
void loop () {
   int a = 10,b = 20
   int c = 0;

   a++;
   a--;
   b += a;
   b -= a;
   b *= a;
   b /= a;
   a %= b;
   a |= b;
   a &= b;
}
```



# Operators

- **Assignment Operators**

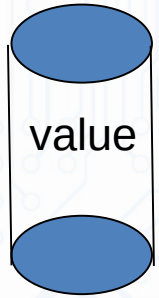
| Operator | Description | Example |
|---|---|---|
| = | Simple assignment operator, Assigns values from right side operands to left side operand | C = A + B will assigne value of A + B into C |
| += | Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand | C += A is equivalent to C = C + A |
| -= | Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand | C -= A is equivalent to C = C - A |
| *= | Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand | C *= A is equivalent to C = C * A |
| /= | Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand | C /= A is equivalent to C = C / A |
| %= | Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand | C %= A is equivalent to C = C % A |
| <<= | Left shift AND assignment operator | C <<= 2 is same as C = C << 2 |
| >>= | Right shift AND assignment operator | C >>= 2 is same as C = C >> 2 |
| &= | Bitwise AND assignment operator | C &= 2 is same as C = C & 2 |
| ^= | bitwise exclusive OR and assignment operator | C ^= 2 is same as C = C ^ 2 |
| \|= | bitwise inclusive OR and assignment operator | C \|= 2 is same as C = C \| 2 |



# Variables

DECLARING A VARIABLE

- **A variable is like "bucket"**
- **It holds numbers or other values temporarily**



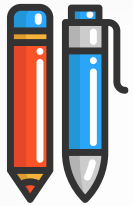

```
int val = 5;
```

Type

variable name

assignment "becomes"

value

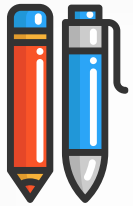


# USING VARIABLES

```
int delayTime = 2000;
int greenLED = 9;
void setup() {
        pinMode(greenLED, OUTPUT);
}

void loop() {
        digitalWrite(greenLED, HIGH);
        delay(delayTime);
        digitalWrite(greenLED, LOW);
        delay(delayTime);
}
```

Declare delay Time Variable

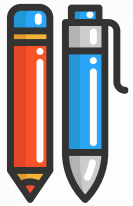
Use delay Time Variable



# Using Variables

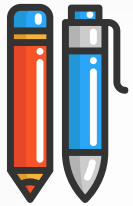
```
int delayTime = 2000;
int greenLED = 9;

void setup() {
        pinMode(greenLED, OUTPUT);
}
void loop() {
        digitalWrite(greenLED, HIGH);
        delay(delayTime);
        digitalWrite(greenLED, LOW);
 delayTime = delayTime - 100;
        delay(delayTime);
}
```

subtract 100 from delayTime to gradually increase LED's blinking speed



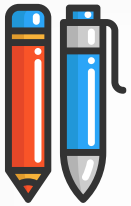
# VALUE COMPARISONS

| GREATER THAN | GREATER THAN OR EQUAL |
| --- | --- |
| a > b | a >= b |
| LESS | LESS THAN OR EQUAL |
| a < b | a <= b |
| EQUAL | NOT EQUAL |
| a == b | a != b |



# Flow of Control, Conditional Constructs, Loops

- This session will make you advance to control your code so control your world.
- The goal is to take you from the basic to learn more about Conditional Constructs and loops.



# C - Flow Control Statements

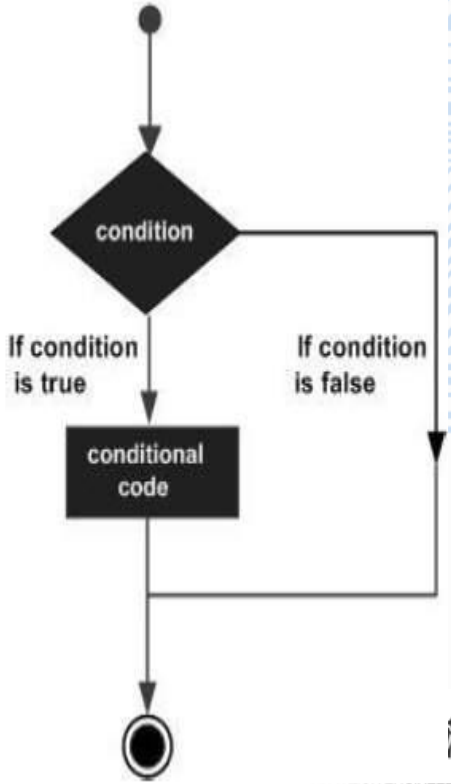
- Flow Control – Making the program behave in a particular manner depending on the input given to the program
- Why do we need flow control?
  - Not all program parts are executed all of the time( i.e. we want the program to intelligently choose what to do).
- Topics will discussed here are:
  - If
  - If-else
  - For
  - While
  - Do-while
  - Nesting loops
  - Switch

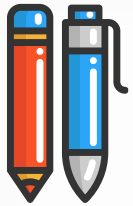
# Arduino - Control Statements

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program. It should be along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

- Following is the general form of a typical decision making structure found in most of the programming languages

| S.NO | Control Statement & Description |
|---|---|
| 1 | **If statement**<br>It takes an expression in parenthesis and a statement or block of statements. If the expression is true then the statement or block of statements gets executed otherwise these statements are skipped. |
| 2 | **If …else statement**<br>An if statement can be followed by an optional else statement, which executes when the expression is false. |
| 3 | **If…else if …else statement**<br>The if statement can be followed by an optional else if...else statement, which is very useful to test various conditions using single if...else if statement. |
| 4 | **switch case statement**<br>Similar to the if statements, switch...case controls the flow of programs by allowing the programmers to specify different codes that should be executed in various conditions. |
| 5 | **Conditional Operator ? :**<br>The conditional operator ? : is the only ternary operator in C. |





# Conditions

- To make decisions in Arduino code we use an ‘if’ statement
- ‘If’ statements are based on a TRUE or FALSE question

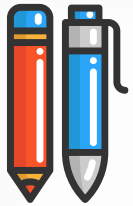
```
if(true)
{
    “perform some action”
}
```

# IF statement

- The **if else** statement decides on an action based on **if** being true
- The form of the statements is
  - IF (condition1)

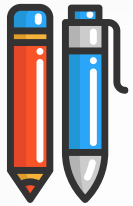
```
{
Statements1;
}
Else if (condition2)
{
Statements2;
}
Else
{
Statements3;
}
```

```
if (PRT2DR & 0x01) {
    PRT1DR &= ~0x02;
    PRT1DR |=  0x04;
}
else {
    PRT1DR &= ~0x04;
    PRT1DR |=  0x02;
}
```



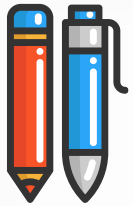
# IF Example

```
int counter = 0;

void setup() {
      Serial.begin(9600);
}
void loop() {

      if(counter < 10)
      {
            Serial.println(counter)
            ;
      }
      counter = counter + 1;
        }
```



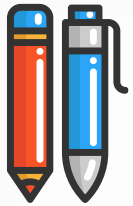
# IF - ELSE Condition

```
if( “answer is true”)
{
    “perform some action”
}
else
{
    “perform some other action”
}
```



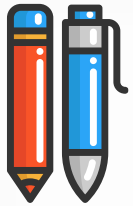
# IF - ELSE Example

```
int counter = 0;
void setup() {
      Serial.begin(9600);
      }
void loop() {
      if(counter < 10)
      {
            Serial.println("less than 10");
      }
      else
      {
            Serial.println("greater than or equal to 10");
            Serial.end();
      }
      counter = counter + 1;
}
```



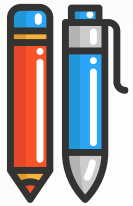
# IF - ELSE IF Condition

```
if( “answer is true”)
{
     “perform some action”
}
else if( “answer is true”)
{
     “perform some other action”
}
```



# IF - ELSE Example

```
int counter = 0;
void setup() {
      Serial.begin(9600);
}
void loop() {
   if(counter < 10)
      {
            Serial.println("less than 10");
      }
      else if (counter == 10)
      {
            Serial.println("equal to 10");
      }
      else
      {
            Serial.println("greater than 10");
            Serial.end();
      }
      counter = counter + 1;
}
```

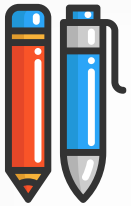


# BOOLEAN OPERATORS - AND

- If we want all of the conditions to be true we need to use ‘AND’ logic (AND gate)
- We use the symbols **&&**

  – Example

```
if ( val > 10 && val < 20)
```



# BOOLEAN OPERATORS - OR

- If we want either of the conditions to be true we need to use ‘OR’ logic (OR gate)
- We use the symbols ||

  – Example

```
if ( val < 10 || val > 20)
```

# SWITCH statement

- The **switch case** statement Compares a single variable to several possible constants
- The form of the statements is
  - **Switch (variable)**

**{**
**Case value1:**
**Statements1;**
**Break;**
**Case value2:**
**Statements2;**
**Break;**
**Default:**
**Statements3;**
**Break;**
**}**

```
switch (x) {
    case 1:
        PRT1DR |=  0x04;
    break;
    case 2:
        PRT1DR |=  0x02;
    break;
}
```

# WHILE statement

- The **while** statement tests a certain condition and repeats a set of statements until the condition is false
- The for of the statement
  - While(condition)

    {

    statements;

    }

```
while (1) {
    if (PRT2DR & 0x01) {
        PRT1DR &= ~0x02;
        PRT1DR |=  0x04;
    }
    else {
        PRT1DR &= ~0x04;
        PRT1DR |=  0x02;
    }
}
```

# DO statement

- The **Do while** statement is same as **while**, except the test runs after execution of a statement, not before
- The form of the statement
  - Do
    {
    statements
    }
    while(condition)

```
do {
PRT1DR |=  0x04;
} while (PRT2DR & 0x01);
```

# FOR statement

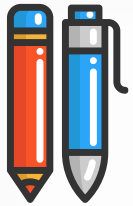
- The **For** statement Executes a limited loop.
- The form of the statement
  - For(initial value ; condition ; change)

    {

    Statements;

    }

```
for ( n=0; n < 100 ; n++ )
{
// whatever here...
}
```

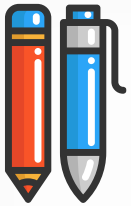


# Functions

- Using functions we can structure our programs in a more modular way.
- A function is a group of statements that is executed when it is called from some point of the program. The following is its format:
  - type name ( parameter1, parameter2, ...) { statements }

```
// function example
#include <iostream>
using namespace std;

int addition (int a, int b)
{
  int r;
  r=a+b;
  return (r);
}

int main ()
{
  int z;
  z = addition (5,3);
  cout << "The result is " << z;
  return 0;
}
```

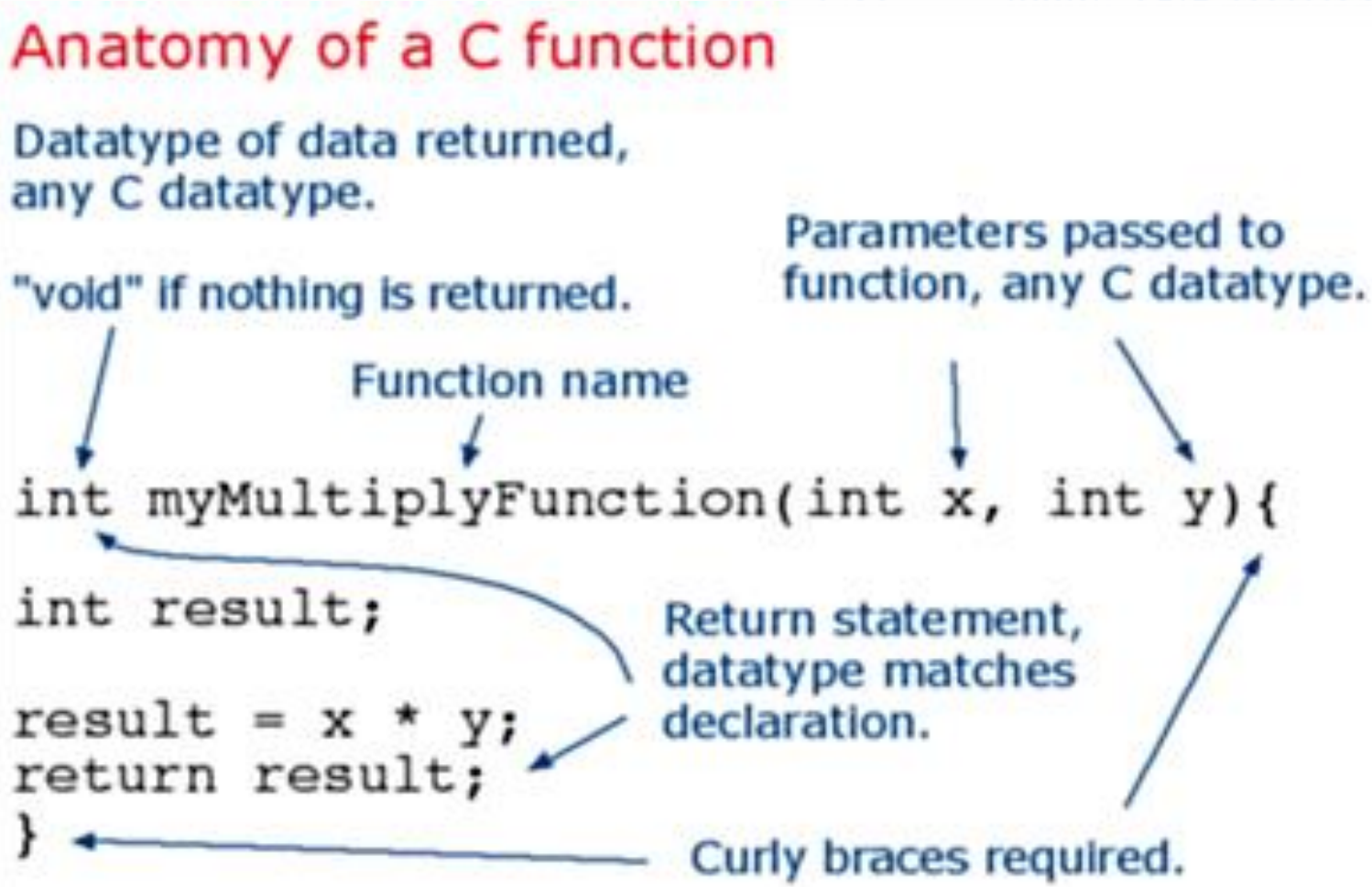
```
The result is 8
```

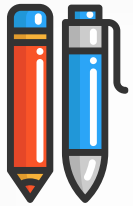


# Functions

- Segmenting code into functions allows a programmer to create modular pieces of code that perform a defined task and then return to the area of code from which the function was "called". The typical case for creating a function is when one needs to perform the same action multiple times in a program.
- For programmers accustomed to using BASIC, functions in Arduino provide (and extend) the utility of using subroutines (GOSUB in BASIC).
- Standardizing code fragments into functions has several advantages:
  - Functions help the programmer stay organized. Often this helps to conceptualize the program.
  - Functions codify one action in one place so that the function only has to be thought out and debugged once.
  - This also reduces chances for errors in modification, if the code needs to be changed.
  - Functions make the whole sketch smaller and more compact because sections of code are reused many times.
  - They make it easier to reuse code in other programs by making it more modular, and as a nice side effect, using functions also often makes the code more readable.
  - There are two required functions in an Arduino sketch, setup() and loop(). Other functions must be created outside the brackets of those two functions. As an example, we will create a simple function to multiply two numbers.

# Functions

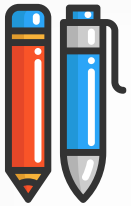

Anatomy of a C function
Datatype of data returned, any C datatype.
"void" if nothing is returned.
Function name
Parameters passed to function, any C datatype.
int myMultiplyFunction(int x, int y){
int result;
result = x * y;
return result;
}
Return statement, datatype matches declaration.
Curly braces required.



# Functions

- To "call" our simple multiply function, we pass it parameters of the datatype that it is expecting:

```
void loop()
{
        int i = 2;
        int j = 3;
        int k;
        k = myMultiplyFunction(i, j); // k now contains 6
                }
```



# Functions

- Our function needs to be *declared* outside any other function, so "myMultiplyFunction()" can go either above or below the "loop()" function.
- The entire sketch would then look like this:

```
void setup(){
  Serial.begin(9600);
}
void loop() {
  int i = 2;
  int j = 3;
  int k;
  k = myMultiplyFunction(i, j); // k now contains 6
  Serial.println(k);
  delay(500);
}
int myMultiplyFunction(int x, int y){
  int result;
  result = x * y;
  return result;
}
```

# Functions: **Another example**

- This function will read a sensor five times with analogRead() and calculate the average of five readings. It then scales the data to 8 bits (0-255), and inverts it, returning the inverted result.

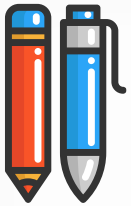
```
int ReadSens_and_Condition(){
  int i;
  int sval = 0;

  for (i = 0; i < 5; i++){
    sval = sval + analogRead(0);    // sensor on analog pin
0
  }

  sval = sval / 5;    // average
  sval = sval / 4;    // scale to 8 bits (0 - 255)
  sval = 255 - sval;  // invert output
  return sval;
}
```

# Functions: **Another example**

- To call our function we just assign it to a variable.

```
int sens;

sens = ReadSens_and_Condition();
```

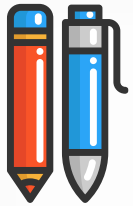
As you can see, even if a function does not have parameters and no returns is expected "(" and ")" brackets plus ";" must be given.



# Arrays

## Description

- An array is a collection of variables that are accessed with an index number.
- Arrays in the C++ programming language Arduino sketches are written in can be complicated, but using simple arrays is relatively straightforward.

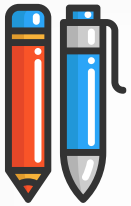


# Arrays

## Creating (Declaring) an Array

```
int myInts[6]; int myPins[] = {2, 4, 8, 3, 6};
 int mySensVals[6] = {2, 4, -8, 3, 2};
char message[6] = "hello";
```

- You can declare an array without initializing it as in myInts.
- In myPins we declare an array without explicitly choosing a size.
- The compiler counts the elements and creates an array of the appropriate size.
- Finally you can both initialize and size your array, as in mySensVals.
- Note that when declaring an array of type char, one more element than your initialization is required, to hold the required null character.



# Accessing an Array

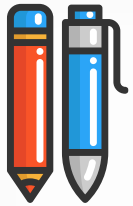
- Arrays are zero indexed, that is, referring to the array initialization above, the first element of the array is at index 0, hence

```
mySensVals[0] == 2, mySensVals[1] == 4, and so forth.
```

It also means that in an array with ten elements, index nine is the last element. Hence:

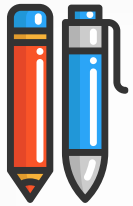
```
int myArray[10]={9, 3, 2, 4, 3, 2, 7, 8, 9, 11}; // myArray[9] contains 11

// myArray[10] is invalid and contains random information (other memory address)
```

For this reason you should be careful in accessing arrays. Accessing past the end of an array (using an index number greater than your declared array size - 1) is reading from memory that is in use for other purposes. Reading from these locations is probably not going to do much except yield invalid data.



# Accessing an Array

- Writing to random memory locations is definitely a bad idea and can often lead to unhappy results such as crashes or program malfunction. This can also be a difficult bug to track down.
- Unlike BASIC or JAVA, the C++ compiler does no checking to see if array access is within legal bounds of the array size that you have declared.



# Accessing an Array

**To assign a value to an array:** mySensVals[0] = 10;

**To retrieve a value from an array:** x = mySensVals[4];

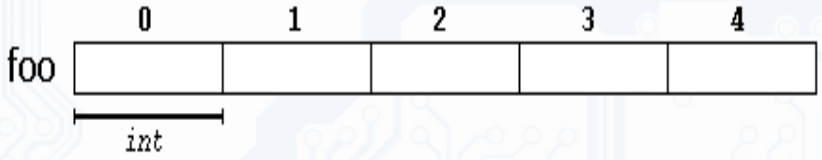
## Arrays and FOR Loops:

- Arrays are often manipulated inside for loops, where the loop counter is used as the index for each array element.
- For example, to print the elements of an array over the serial port, you could do something like this:

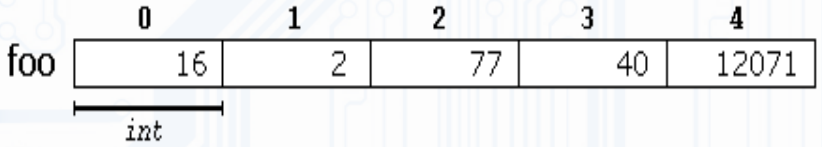
```
for (byte i = 0; i < 5; i = i + 1) {
        Serial.println(myPins[i]); }
```

# Arrays

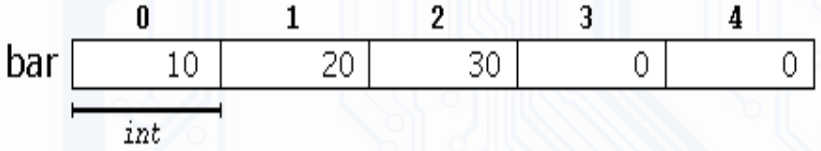
- An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.



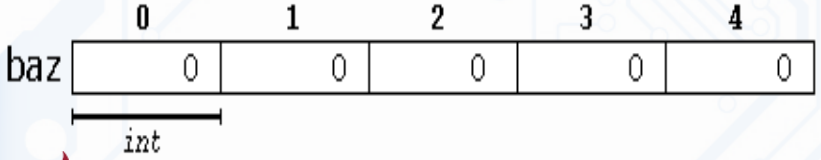

```
int foo [5] = { 16, 2, 77, 40, 12071 };
```




```
int bar [5] = { 10, 20, 30 };
```

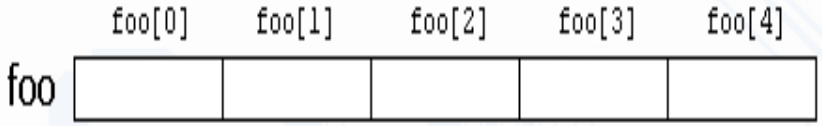



```
int baz [5] = {};
```



# Arrays

- **Accessing the values of an array**




| | |
|---|---|
| foo [2] = 75; | x = foo[2] |

```
1 int foo[5]; // declaration of a new array
2 foo[2] = 75; // access to an element of the array.
```

Some other valid operations with a

```
1 foo[0] = a;
2 foo[a] = 75;
3 b = foo [a+2];
4 foo[foo[a]] = foo[2] + 5;
```

For example:

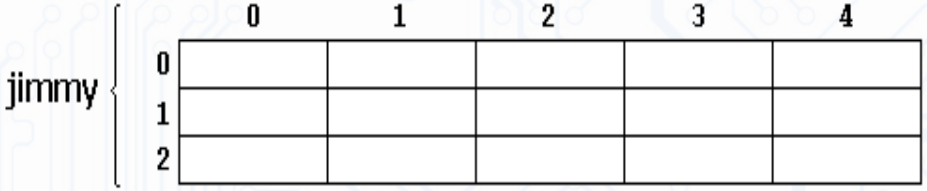
```
1  // arrays example
2  #include <iostream>
3  using namespace std;
4
5  int foo [] = {16, 2, 77, 40, 12071};
6  int n, result=0;
7
8  int main ()
9  {
10   for ( n=0 ; n<5 ; ++n )
11   {
12     result += foo[n];
13   }
14   cout << result;
15   return 0;
16 }
```

```
12206
```

# Multidimensional arrays

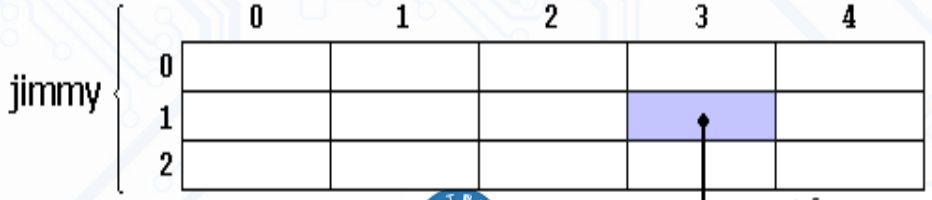
- Multidimensional arrays can be described as "arrays of arrays".
- Think of it as [row][col]
- Example:

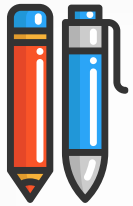
```
int jimmy [3][5];
```




The way to reference the second element vertically and fourth horizontally in an expression would be:

```
jimmy[1][3]
```





# Initializing Arrays

Can initialize an array just like a normal variable

Example:

- **String**

  **char szTemp[] = “Some string”;**

- **Values**

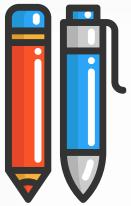
  **int nTemp[] = {5,15,20,25};**

- **Letters**

  **char szTemp[] = {‘A’,’B’,’C’,’D’};**

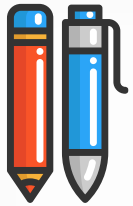
- **Double Dimensioned**

  **char szTemp[2][] = { {‘A’,’B’,’C’,’D’,’E’},**
  **{‘U’,’V’,’X’,’Y’,’Z’} };**



# Pointers

- Variables have been explained as locations in the memory which can be accessed by their identifier (their name).
  - This way, the program does not need to care about the physical address of the data in memory.
- The memory is like a succession of memory cells, each one byte in size, and each with a unique address.
  - These single-byte memory cells are ordered in a way that allows data representations larger than one byte to occupy memory cells that have consecutive addresses.
  - Each cell can be easily located in the memory by means of its unique address.
- When a variable is declared, the memory needed to store its value is assigned a specific location in memory (its memory address).
- Generally, C++ programs do not actively decide the exact memory addresses where its variables are stored.



# Pointers

- **Address-of operator (&)**
  - The address of a variable can be obtained by preceding the name of a variable with an ampersand sign (&), known as *address-of operator*. For example:

```
foo = &myvar;
```

  - This would assign the address of variable myvar to foo; by preceding the name of the variable myvar with the *address-of operator* (&), we are no longer assigning the content of the variable itself to foo, but its address.
  - The actual address of a variable in memory cannot be known before runtime.

# Pointers

- let's assume, in order to help clarify some concepts, that myvar is placed during runtime in the memory address 1776.
  In this case, consider the following code fragment:

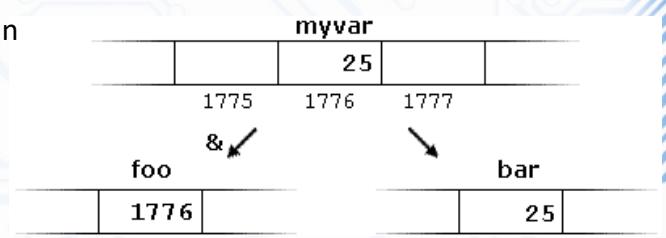
```
myvar = 25;
foo = &myvar;
bar = myvar;
```

- The values contained in each variable after the execution of this are shown in the following diagram:

First, we have assigned the value 25 to myvar (a variable whose address in memory we assumed to be 1776).

The second statement assigns foo the address of myvar, which we have assumed to be 1776.

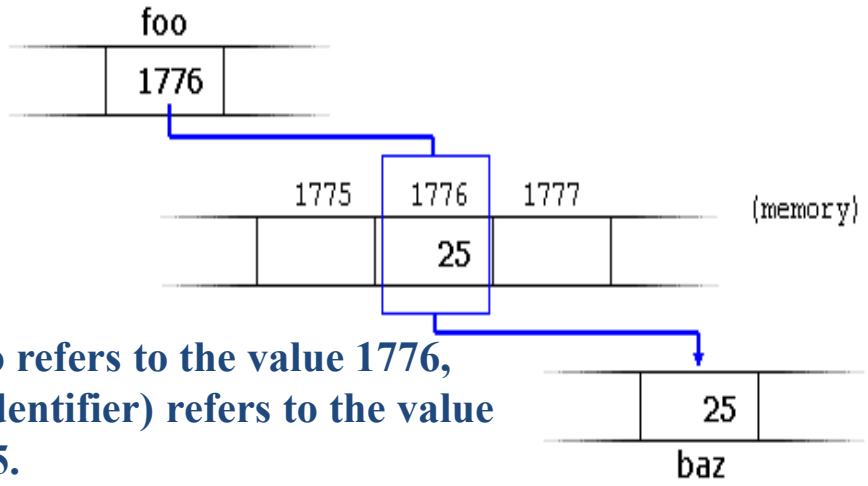
Finally, the third statement, assigns the value contained in myvar to bar.



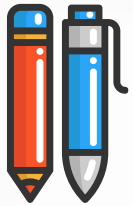
# Pointers

- **Dereference operator (*)**
  - As just seen, a variable which stores the address of another variable is called a *pointer*. Pointers are said to "point to" the variable whose address they store.
  - An interesting property of pointers is that they can be used to access the variable they point to directly. This is done by preceding the pointer name with the *dereference operator* (*).
  - **The operator itself can be read as "value pointed to by".**
  - **Therefore, following with the values of the previous example, the following statement:**
  - **This could be read as: "baz equal to value pointed to by foo", and the statement would actually assign the value 25 tobaz, since foo is 1776, and the value pointed to by 1776 (following the example above) would be 25.**

```
baz = *foo;
```




**It is important to clearly differentiate that foo refers to the value 1776, while *foo (with an asterisk * preceding the identifier) refers to the value stored at address 1776, which in this case is 25.**

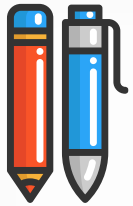


# Pointers

- Notice the difference of including or not including the *dereference operator*

```
baz = foo; // baz equal to foo (1776)
baz = *foo; // baz equal to value pointed to by foo (25)
```

- The reference and dereference operators are thus complementary:
  - & is the *address-of operator*, and can be read simply as "address of"
  - * is the *dereference operator*, and can be read as "value pointed to by"

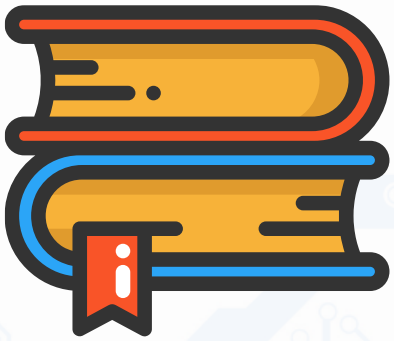


# Pointers

- **Declaring pointers**
  - Due to the ability of a pointer to directly refer to the value that it points to, a pointer has different properties when it points to a char than when it points to an int or a float.
  - Once dereferenced, the type needs to be known. And for that, the declaration of a pointer needs to include the data type the pointer is going to point to.
  - The declaration of pointers follows this syntax: type * name;

    where type is the data type pointed to by the pointer. This type is not the type of the pointer itself, but the type of the data the pointer points to. For example:

```
int * number;
char * character;
double * decimals;
```

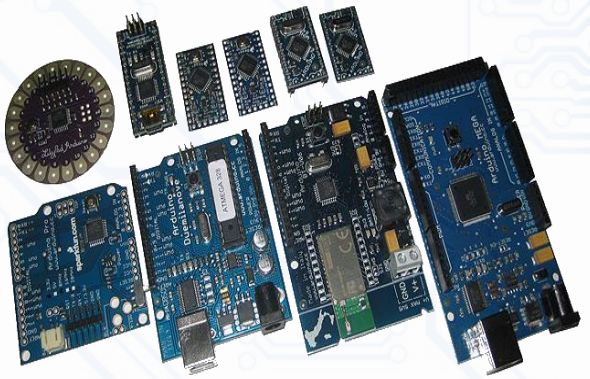


# 嵌入式系统

# EMBEDDED SYSTEMS

**LECTURE 08:** Introduction To Arduino I/O

**Arduino - I/O Functions**

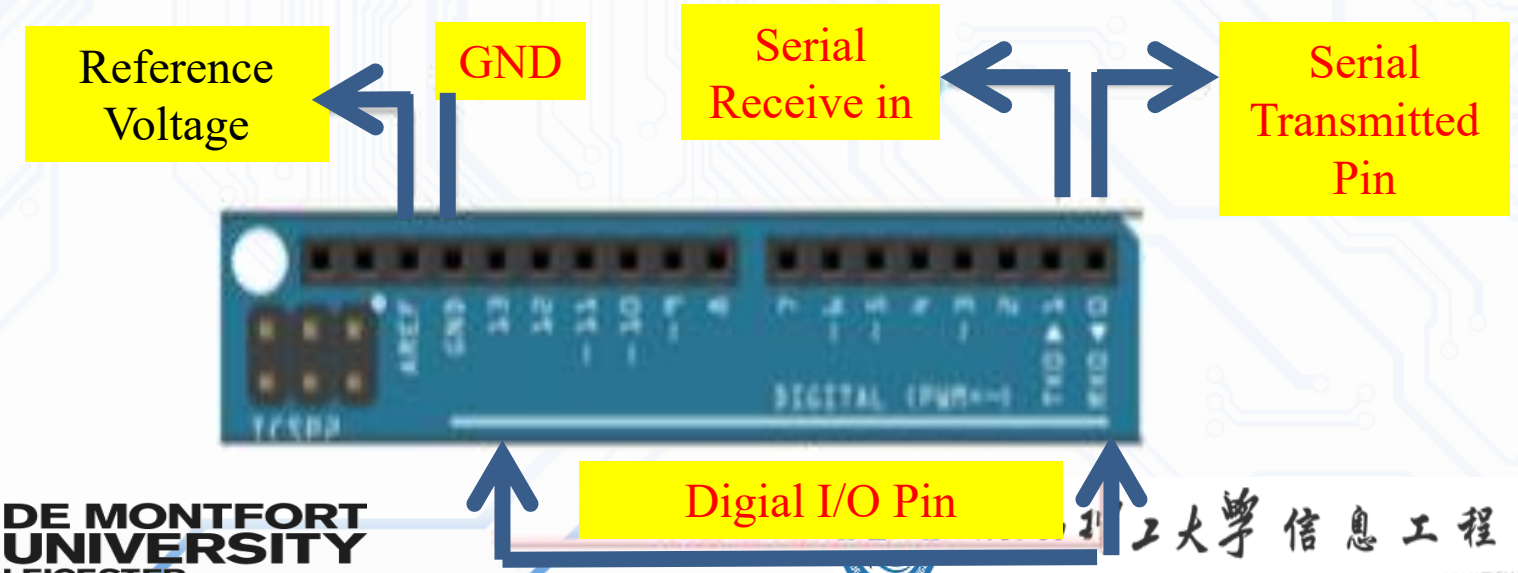


DE MONTFORT UNIVERSITY LEICESTER

江西理工大學 信息工程学院
JIANGXI UNIVERSITY OF SCIENCE AND TECHNOLOGY SCHOOL OF INFORMATION ENGINEERING

# Arduino - I/O Functions

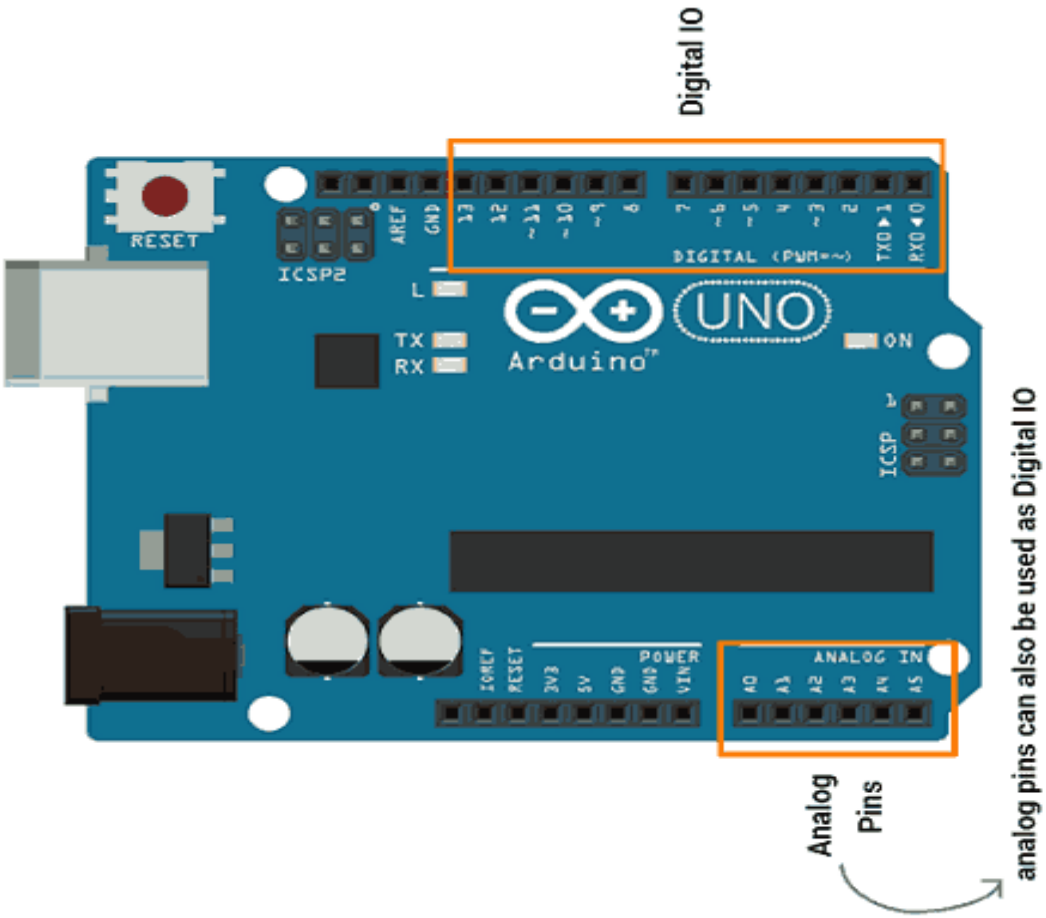
- **G**eneral-**P**urpose **I**nput **O**utput (GPIO) is a digital pin of an IC. It can be used as input or output for interfacing devices.
- If we want to read switch's state, sensor data, etc then we need to configure it as input. And if we want to control the LED brightness, motor rotation, show text on display, etc then we need to configure it as output.
- The pins on the Arduino board can be configured as either inputs or outputs.
- It is important to note that a majority of Arduino analog pins, may be configured, and used, in exactly the same manner as digital pins.

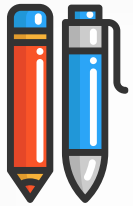


# Pins Configured

Pin defines the Arduino pin number used.

- There are three types of modes that can be assigned to pins of Arduino, which are:
  - OUTPUT
  - INPUT
  - INPUT_PULLUP





# Pins Configured

**pinMode(pin no, Mode)**
This function is used to configure GPIO pin as input or output.
pin no number of pin whose mode we want to set.

- Mode INPUT, OUTPUT or INPUT_PULLUP

**E.g. pinMode (3, OUTPUT) //set pin 3 as output**

These Arduino (ATmega) pins can source or sink current up to 40 mA which is sufficient to drive led, LCD display but not sufficient for motors, relays, etc.

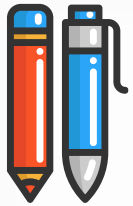
**Note:** While connecting devices to Arduino output pins use resistor. If any connected device to Arduino withdraw current more than 40 mA from the Arduino then it will damage the Arduino pin or IC.

# Pins Configured as INPUT

Pins configured as pinMode(pin, INPUT) with nothing connected to them, or with wires connected to them that are not connected to other circuits, report seemingly random changes in pin state, picking up electrical noise from the environment, or capacitively coupling the state of a nearby pin.

**Example**

pinMode(3,INPUT) ; // set pin to input without using built in pull up resistor



# Pull-up Resistors

- Pull-up resistors are often useful to steer an input pin to a known state if no input is present. This can be done by adding a pull-up resistor (to +5V), or a pull-down resistor (resistor to ground) on the input. A 10K resistor is a good value for a pull-up or pull-down resistor.

- Using Built-in Pull-up Resistor with Pins Configured as Input.There are 20,000 pull-up resistors built into the Atmega chip that can be accessed from software. These built-in pull-up 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.

# Pull-up Resistors

- The value of this pull-up 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.

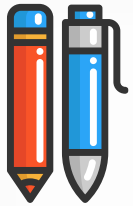
- When connecting a sensor to a pin configured with INPUT_PULLUP, the other end should be connected to the ground. In case of a simple switch, this causes the pin to read HIGH when the switch is open and LOW when the switch is pressed. The pull-up resistors provide enough current to light an LED dimly connected to a pin configured as an input.

- If LEDs in a project seem to be working, but very dimly, this is likely what is going on.
- Same registers (internal chip memory locations) that control whether a pin is HIGH or LOW control the pull-up resistors.

- Consequently, a pin that is configured to have pull-up resistors turned on when the pin is in INPUTmode, will have the pin configured as HIGH if the pin is then switched to an OUTPUT mode with pinMode().
- This works in the other direction as well, and an output pin that is left in a HIGH state will have the pull-up resistor set if switched to an input with pinMode().

**Example**

```
pinMode(3,INPUT) ; // set pin to input without using built in pull up resistor
pinMode(5,INPUT_PULLUP) ; // set pin to input using built in pull up resistor
```



# Pins Configured as OUTPUT

- Pins configured as OUTPUT with pinMode() are said to be in a low-impedance state.

**pinMode(3, OUTPUT ) ; // set pin to OUTPUT**

This means that 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 (milliamps) of current to other devices/circuits. This is enough current to brightly light up an LED (do not forget the series resistor), or run many sensors but not enough current to run relays, solenoids, or motors.
- Attempting to run high current devices from the output pins, can damage or destroy the output transistors in the pin, or damage the entire Atmega chip. Often, this results in a "dead" pin in the microcontroller but the remaining chips still function adequately.
- For this reason, it is a good idea to connect the OUTPUT pins to other devices through 470Ω or 1k resistors, unless maximum current drawn from the pins is required for a particular application.

# digitalWrite() Function

**digitalWrite Arduino Command** is used to write the status of digital Pins, and can make them either HIGH or LOW. The Pin needs to be an OUTPUT Pin.

The **digitalWrite()** function is used to write a HIGH or a LOW value to a digital pin. If the pin has been configured as an OUTPUT with pinMode(), its voltage will be set to the corresponding value: 5V (or 3.3V on 3.3V boards) for HIGH, 0V (ground) for LOW. If the pin is configured as an INPUT, digitalWrite() will enable (HIGH) or disable (LOW) the internal pullup on the input pin.

It is recommended to set the pinMode() to INPUT_PULLUP to enable the internal pull-up resistor.

If you do not set the pinMode() to OUTPUT, and connect an LED to a pin, when calling digitalWrite(HIGH), the LED may appear dim. Without explicitly setting pinMode(), digitalWrite() will have enabled the internal pull-up resistor, which acts like a large current-limiting resistor.

## digitalWrite() Function Syntax

```
Void loop() {
  digitalWrite (pin ,value);
}
```

**pin** − the number of the pin whose mode you wish to set
**value** − HIGH, or LOW.

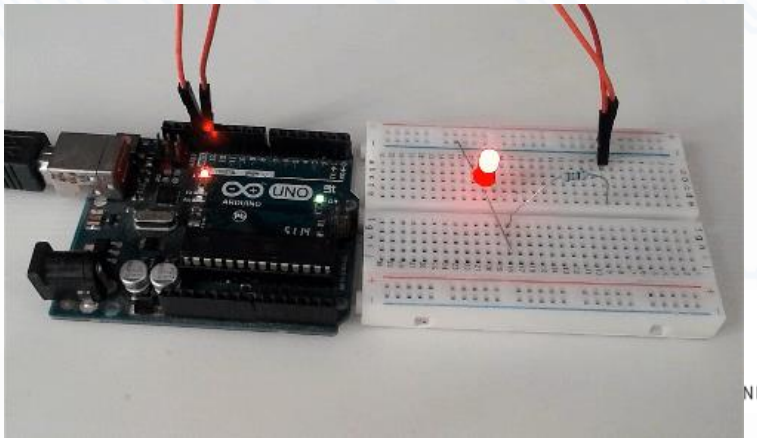
The “pin” defines the Arduino pin number used. It has to be an OUTPUT Pin. And “value” defines if the pin will be HIGH or LOW.

For example:
digitalWrite(8, HIGH);

# Example: Sketch for LED Blinking using Arduino

Let's write a program for led blinking using Arduino Uno. Here, we will blink on-board LED of Arduino Uno which is connected to digital pin 13.

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

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



# Thank You

# FOR LISTENING


## Dr. Ata Jahangir Moshayedi

- School of information engineering Jiangxi university of science and technology, China
- E-mail: ajm@jxust.edu.cn

# Where you can find all my lectures

Embedded Systems Lecture 01 | Introduction to ES | Dr AJM | JXUST

https://www.youtube.com/watch?v=bcCCPqDKr9Y&list=PLqipy5sqhIACnJodz3zB8gUuOosku6yYP


**Dr Ata Jahangir Moshayedi**

**Prof Associate ,**

**School of information engineering Jiangxi university of science and technology, China**

**EMAIL:**

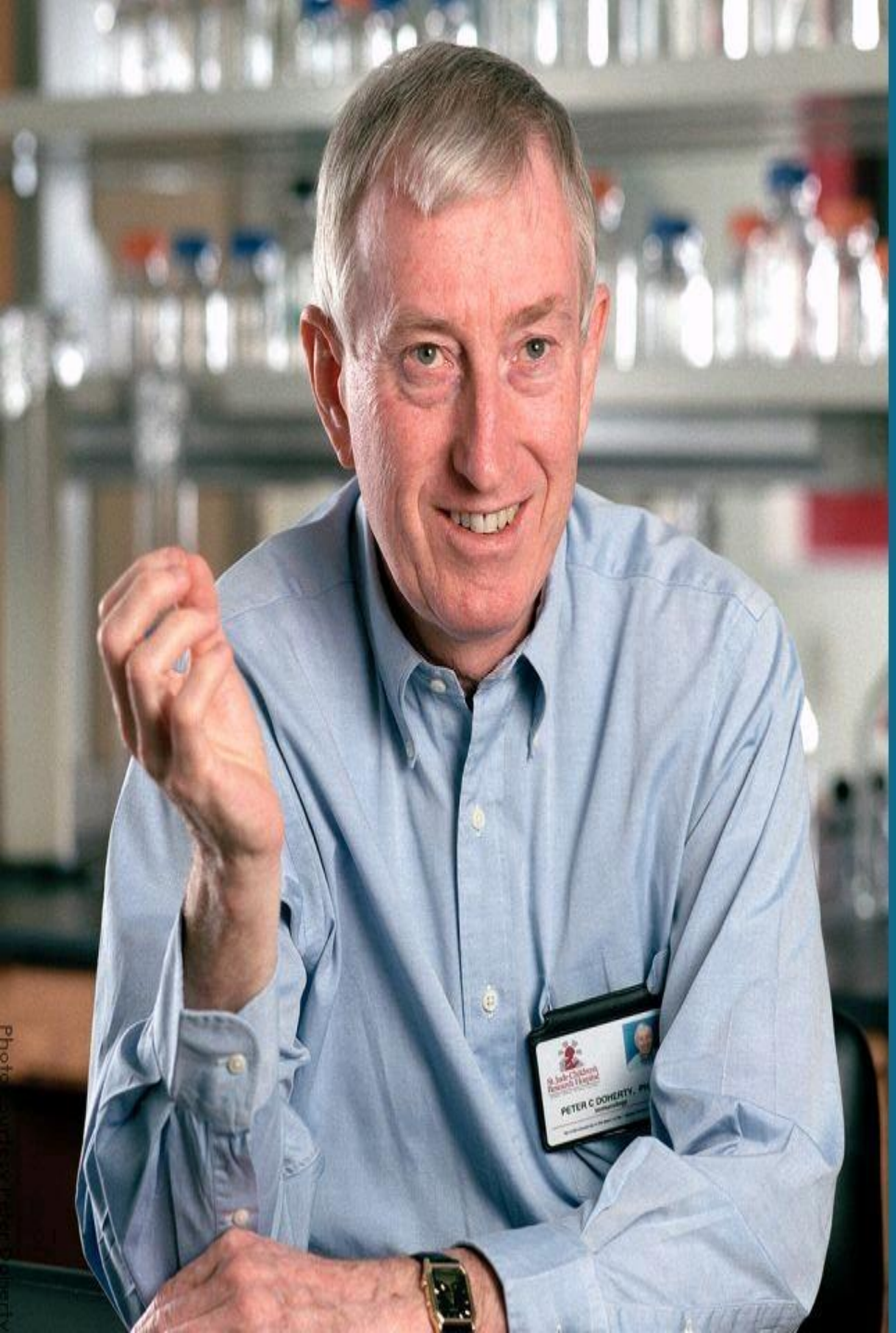
**ajm@jxust.edu.cn**

**moshaydi@gmail.com**


DE MONTFORT UNIVERSITY LEICESTER

江西理工大學 信息工程学院

JIANGXI UNIVERSITY OF SCIENCE AND TECHNOLOGY SCHOOL OF INFORMATION ENGINEERING




"Good scientists are perpetual adolescents. They never grow up."

PETER DOHERTY
Nobel Prize in Physiology or Medicine 1996

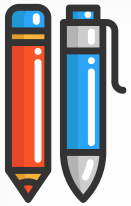
# Some of used References


- Designing Embedded Systems with Arduino A Fundamental Technology for Makers ,Tianhong Pan • Yi Zhu,Springer Nature Singapore Pte Ltd. 2018, ISBN 978-981-10-4417-5 ISBN 978-981-10-4418-2 (eBook)
- Enjoy the programming with Arduino board, Dr Ata Jahangir Moshayedi
- http://www.firmcodes.com/difference-uart-usart/
- https://www.edn.com/electronics-blogs/embedded-basics/4440395/USART-vs-UART--Know-the-difference
- https://www.watelectronics.com/classification-of-embedded-systems/
- https://www.theengineeringprojects.com/2018/09/how-to-use-digitalwrite-arduino-command.html
- https://www.tutorialspoint.com/arduino/arduino_io_functions.htm
- https://www.watelectronics.com/classification-of-embedded-systems/


江西理工大学
Jiangxi University of Science and Technology
信息工程学院
School of information engineering

# Digital Image Processing


## THANK YOU



"BE HUMBLE. BE HUNGRY. AND ALWAYS BE THE HARDEST WORKER IN THE ROOM."