Arduino UNO board
Arduino UNO Features

- 14 digital I/O pins, of which 6 can be analog (PWM) output pins, and 6 analog input pins
- Multiple serial ports (at the expense of digital I/O pins)
- 2 external interrupt pins
- 1KB EEPROM, easily accessible from code
- A very useful debugging LED
- USB-to-serial connection
- Power from USB or external power supply
## Uploading Code to the Arduino

1. Write a program in the Arduino programming environment
Uploading Code to the Arduino

1. Write a program in the Arduino programming environment
2. Connect the board to your computer via USB
Uploading Code to the Arduino

1. Write a program in the Arduino programming environment
2. Connect the board to your computer via USB
3. Click the 'upload' button to send your program to the board

After uploading, the code runs automatically!

If power comes from an external power supply (like a 9V battery), you can disconnect the USB cable and the board keeps running your program.

Even after all power was switched off, the board retains your program in memory. When you reconnect the power, it starts running automatically.
Uploading Code to the Arduino

1. Write a program in the Arduino programming environment
2. connect the board to your computer via USB
3. click the ‘upload’ button to send your program to the board
4. after uploading, the code runs automatically!
Uploading Code to the Arduino

1. Write a program in the Arduino programming environment
2. connect the board to your computer via USB
3. click the 'upload' button to send your program to the board
4. after uploading, the code runs automatically!
5. If power comes from an external power supply (like a 9V battery), you can disconnect the USB cable and the board keeps running your program.
Uploading Code to the Arduino

1. Write a program in the Arduino programming environment
2. connect the board to your computer via USB
3. click the ’upload’ button to send your program to the board
4. after uploading, the code runs automatically!
5. If power comes from an external power supply (like a 9V battery), you can disconnect the USB cable and the board keeps running your program.
6. Even after all power was switched off, the board retains your program in memory. When you reconnect the power, it starts running automatically.
Uploading Code to the Arduino

1. Write a program in the Arduino programming environment.
2. Connect the board to your computer via USB.
3. Click the 'upload' button to send your program to the board.
4. After uploading, the code runs automatically!
5. If power comes from an external power supply (like a 9V battery), you can disconnect the USB cable and the board keeps running your program.
6. Even after all power was switched off, the board retains your program in memory. When you reconnect the power, it starts running automatically.
Check out the URL http://arduino.cc

Our board is the UNO model.
First steps

Check out the URL http://arduino.cc

Our board is the UNO model.

Download the IDE by clicking the Download button.
Check out the URL http://arduino.cc

Our board is the UNO model.

Download the IDE by clicking the **Download** button.

Read the page corresponding to the **Learning** button.
First steps

Check out the URL http://arduino.cc

Our board is the UNO model.

Download the IDE by clicking the Download button.

Read the page corresponding to the Learning button.

The language definition is available by clicking the Reference button.
First steps

Check out the URL http://arduino.cc

Our board is the UNO model.

Download the IDE by clicking the **Download** button.

Read the page corresponding to the **Learning** button.

The language definition is available by clicking the **Reference** button.
IDE – Integrated Development Environment

- compile (verify program)
- stop execution
- new
- open
- save
- send program to board
- show serial
- code area
- status area
Code Basics

All code is written around two important functions: `setup()` and `loop()`. When you place code inside these functions, it executes as follows:

- `setup()` is executed once, on Arduino power-up and reset, after this,
- `loop()` is repeatedly executed *ad infinitum*.

Example:

```cpp
int n;

void setup() {
  n = 28;
}

void loop() {
  n++;
  n = n % 256;
}
```
Digital Output – LEDs

LEDs can be connected in any of the 14 digital I/O pins.

When in OUTPUT mode, digital pins can supply either 0V (LOW) or 5V (HIGH).

How to configure pin 7 as output? pinMode(7, OUTPUT);

How to assign it a Low? digitalWrite(7, LOW);

How to assign it a High? digitalWrite(7, HIGH);
Example program to blink LEDs

```c
int ledPin = 8; // LED connected to pin 8

void setup()
{
    pinMode(ledPin, OUTPUT); // ledPin pin as output
}

void loop()
{
    digitalWrite(ledPin, HIGH); // set the LED on
    delay(1000); // wait for a second
    digitalWrite(ledPin, LOW); // set the LED off
    delay(1000); // wait for a second
}
```
How to connect a LED

1. Connect the LED to a digital output pin.
2. Add a resistor (200-470 ohm) in series with the LED to the ground.

**NOTE:** The LED short leg goes to ground.
Digital Input

A digital input signal can be either 0V (LOW) or 5V (HIGH).

In order for a pin to be read, it must be configured as an INPUT.
Example – read a switch

```cpp
int switchPin = 8;  // digital pin to attach the switch
int LED = 7;

void setup()
{
    pinMode(switchPin, INPUT);  // pin 8 as input
    pinMode(LED, OUTPUT);  // pin 7 as output
}

void loop()
{
    if (digitalRead(switchPin) == HIGH)  // if the switch is pressed
        digitalWrite(LED, HIGH);  // turns the LED ON
    else  // if the switch is not pressed
        digitalWrite(LED, LOW);  // turns the LED OFF
}
```
How to connect a switch

[Diagram of switch connection]

António Abreu (ESTSetúbal)
Analog Input

Analog input and output signals can have many values, not just 0V (LOW) and 5V (HIGH).

Voltage levels between 0 - 5V can be read via the analog input pins (A0 through A5 in the UNO board).

How to read an analogue voltage? int v = analogRead(pin);

The value of v will be between 0 and 1023 because the analog to digital converter inside the microcontroller has an output of 10 bits.
How to connect to an analogue source

António Abreu (ESTSetúbal)
Analog output pins can generate voltage levels between 0 - 5V, using a method called Pulse Width Modulation.

PWM is basically a way of faking different voltages by very quickly alternating between 0V and 5V – the longer the 5V spikes take, the higher the output voltage appears. So PWM does not generate pure analog waves.

How to generate a PWM: `analogWrite(pin, value);`

After a call to `analogWrite()`, the pin (`pin`) will generate a steady square wave of the specified duty cycle (`value`) until the next call to `analogWrite()`. The duty cycle value varies from 0 (always off, or 0%) to 255 (always on, or 100%). The frequency of the PWM signal is approximately 490 Hz.
Illustration of PWM

Pulse Width Modulation

0% Duty Cycle – analogWrite(0)

25% Duty Cycle – analogWrite(64)

50% Duty Cycle – analogWrite(127)

75% Duty Cycle – analogWrite(191)

100% Duty Cycle – analogWrite(255)
On most Arduino boards (those with the ATmega168 or ATmega328, the case of UNO), this function works on pins 3, 5, 6, 9, 10, and 11.

You do not need to call pinMode() to set the pin as an output before calling analogWrite().

The analogWrite function has nothing to do with the analog pins or the analogRead function.
You may be interested in consulting the `analogReference(type)` function, which sets the reference voltage used for analog input (i.e. the value used as the top of the input range).
Serial Communication

The board can communicate with other devices via the RS232 protocol, e.g., a computer.

Serial communication can act in two directions simultaneously – duplex or full-duplex.

RS232 is emulated over USB.

Most computers don’t have a RS232 port. So to use this type of communication, you will need an USB-to-serial adapter.
Communication is performed through pins 0 (RX) and 1 (TX), thus they cannot be used for digital input or output.

The Arduino IDE has a built-in serial monitor to communicate with an Arduino board. Click the serial monitor button in the toolbar and select the same baud rate used in the call to `begin()`.

To communicate with an external TTL serial device, connect the TX pin to your device’s RX pin, the RX to your device’s TX pin, and connect the grounds of both devices.

Don’t connect these pins directly to an RS232 serial port; they operate at +/- 12V and can damage the Arduino board.
void setup() {
    Serial.begin(9600);
}

void loop() {
    if (Serial.available() > 0) {
        int p = Serial.read();
        Serial.write(p);
    }
}
print() 

Prints data to the serial port as human-readable ASCII text.

Numbers are printed using an ASCII character for each digit.

Floats are printed as ASCII digits, defaulting to two decimal places.

Bytes are sent as a single character.

Characters and strings are sent as is.
Examples

Serial.print(78) gives “78”
Serial.print(1.23456) gives “1.23”
Serial.print(’N’) gives “N”
Serial.print("Hello world.") gives “Hello world.”
There is an optional second parameter, which specifies the base (format) to use. Examples:

Serial.print(78, BIN) gives “1001110”

Serial.print(78, OCT) gives “116”

Serial.print(78, DEC) gives “78”

Serial.print(78, HEX) gives “4E”

Serial.println(1.23456, 0) gives “1”

Serial.println(1.23456, 2) gives “1.23”

Serial.println(1.23456, 4) gives “1.2346”

To send a single byte, use Serial.write().
available()

Gets the number of bytes (characters) available for reading from the serial port.

The serial receive buffer holds a maximum of 64 bytes.

The function returns the number of bytes available to read.
EEPROM (Electrically Erasable Programmable Read-Only Memory) is memory that retains the values stored in it when power is disconnected.

When you upload code into the board, the values in the EEPROM are not changed.

Arduino typically has 512-1024 bytes of EEPROM memory available to store values in.
#include <EEPROM.h>

byte b;

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

void loop() {
    b = EEPROM.read(28); // read byte value from eeprom
    Serial.println(b, DEC);
    b = (b+1) % 256;
    EEPROM.write(28, b); // write new value to eeprom
    delay(1000);
}
Interrupts

`interrupts()` (re-)enables interrupts (after they’ve been disabled by `noInterrupts()`).

Interrupts allow certain important tasks to happen in the background and are enabled by default.

Some functions will not work while interrupts are disabled, and incoming communication may be ignored.

Interrupts can slightly disrupt the timing of code, however, and may be disabled for particularly critical sections of code.
```cpp
void setup()
{
}

void loop()
{
    noInterrupts();
    // critical, time-sensitive code here
    interrupts();
    // other code here
}
```
**External Interrupts**

`attachInterrupt()` specifies a function to call when an external interrupt occurs.

**Syntax:**
```
attachInterrupt(digitalPinToInterrupt(pin), ISR, mode);
```

- Where `pin` can be 2 or 3 (for UNO), `ISR` represents the function to call when the interrupt occurs (this function must take no parameters and return nothing), and `mode` represents the event that triggers the interrupt:
  - LOW to trigger the interrupt whenever the pin is low;
  - CHANGE to trigger the interrupt whenever the pin changes value;
  - RISING to trigger when the pin goes from low to high;
  - FALLING for when the pin goes from high to low.
Rising and falling edges of a digital signal

- Falling edge: from 1 to 0
- Rising edge: from 0 to 1
const byte ledPin = 13;
nvolatile byte state = LOW;

void setup() {
    pinMode(ledPin, OUTPUT);
    pinMode(2, INPUT_PULLUP);
    attachInterrupt(digitalPinToInterrupt(2), blink, CHANGE);
}

void loop() {
    digitalWrite(ledPin, state);
}

void blink() {
    state = !state;
}
Be aware that:

- Start by figuring out how interrupts apply to your board.
- An ISR does not receive parameters, and don’t return anything.
- ISRs should be as short and fast as possible.
- If your program uses multiple ISRs, only one can run at a time.
- `millis()` relies on interrupts to count, so it will never increment inside an ISR.
- `delay()` requires interrupts to work, it will not work if called inside an ISR.
- `micros()` works initially, but will start behaving erratically after 1-2 ms.
- `delayMicroseconds()` does not use any counter, so it will work as normal.
- Variables shared between ISR functions and normal functions should be declared `volatile`. 
Detaching interrupts

When an external interrupt is no longer needed, it must be detached.

Syntax: detachInterrupt(digitalPinToInterrupt(pin));
Example

```cpp
const byte ledPin = 13;
volatile byte state = LOW, count = 0;
void setup() {
  pinMode(ledPin, OUTPUT);
  pinMode(2, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(2), blink, FALLING);
  Serial.begin(9600);
}
void loop() {
  digitalWrite(ledPin, state);
  if(count==10) {
    detachInterrupt(digitalPinToInterrupt(2));
    Serial.println("count of 10 reached.\nNo more interrupts, please.");
    count++;
  }
}
void blink() {
  state = !state;
  count++;
}
```
String inputString = ""; // a string to hold incoming data
boolean stringComplete = false; // whether the string is complete

void setup() {
    Serial.begin(9600);
    inputString.reserve(200);
}

void loop() {
    // print the string when a newline arrives:
    if (stringComplete) {
        Serial.println(inputString);
        // clear the string:
        inputString = "";
        stringComplete = false;
    }
}
/* SerialEvent() occurs whenever a new data comes in the hardware serial RX. This routine is run between each time loop() runs, so using delay inside loop can delay response. */

void serialEvent() {
    while (Serial.available()) {
        // get the new byte:
        char inChar = (char)Serial.read();
        // add it to the inputString:
        inputString += inChar;
        // if the incoming character is a newline, set a flag
        // so the main loop can do something about it:
        if (inChar == '
') {
            stringComplete = true;
        }
    }
}
Time functions – `millis()` and `delay()`

`millis()` returns the number of milliseconds (unsigned long\(^1\)) since the Arduino board began running the program.

This number will overflow (go back to zero), after approximately 50 days.

`delay()` pauses the program for the amount of time (in milliseconds) specified as parameter.

\(^{1}\) 0 to 4,294,967,295 (\(2^{32} - 1\)).
Example

```c
unsigned long startime;
unsigned int secs = 0;

void setup() {
    Serial.begin(9600);
    startime = millis();
}

void loop() {
    int timepassed = millis()-startime;
    if ( timepassed>1000 ) {
        secs++;
        Serial.print(secs);
        Serial.println(" segs");
        startime = millis();
    }
}
```
Timer1 library

This library is a collection of routines for configuring the 16 bit Timer1 on the ATmega168/328.

Before use, you have to install the library (sketch–include library – install libraries; search for TimerOne and install).

```cpp
#include "TimerOne.h"

boolean timerFlag = 0;
int count = 0;

void setup() {
    Serial.begin(9600);
    pinMode(13, OUTPUT);
    Timer1.initialize(100000); // timer1's period is usecs
    Timer1.attachInterrupt(callback);
}

void callback() {
    if (++count == 10) {
        timerFlag = 1;
        count = 0;
    }
}
```
Example – part 2

```c
void loop() {
    static unsigned long secs=0;
    if(timerFlag) {
        timerFlag = 0;
        Serial.print(++secs);
        Serial.println(" secs");
        digitalWrite(13,HIGH);
        delay(50);
        digitalWrite(13,LOW);
    }
}
```

In this example, we set the period to be 100 msec (or 100000 µsec), after which function callback is called.
Configuration functions

Timer1.initialize(period)  This function must be called before using the timer. The period is in microseconds.

Timer1.setPeriod(period)  Use this function in case the period must be redefined.
Control functions

Timer1.start() Start a new period.
Timer1.stop() Stop the timer.
Timer1.restart() Restart the timer. A new period is begun.
Timer1.resume() Resume a stopped timer. A new period is not begun.
**PWM functions**

`Timer1.pwm(pin, duty)` Configure one of the timer’s PWM pins (9, 10 on the UNO). `duty` is from 0 to 1023.

`Timer1.setPwmDuty(pin, duty)` Set a new duty cycle, without reconfiguring `pin`.

`Timer1.disablePwm(pin)` Stop using PWM on `pin`. The pin reverts to being controlled by `digitalWrite()`.
Interrupt functions

**Timer1.attachInterrupt(function)** Run function each time the timer period finishes. Note that function is an interrupt service routine (ISR), so remember the special care this requires (see slide ??).

**Timer1.detachInterrupt()** Disable the interrupt, so function no longer runs.
Data memory is much shorter in size than program memory.

In case there are constant data it’s preferable to save them in program memory.

This is done by using (the AVR macro) `pgm_read_word_near`. There isn’t a `pgm_write_word_near` macro.
Example

```c
#include <avr/pgmspace.h>
const PROGMEM uint16_t constSet[] = {3.141592, 2.718281, 1.414213};
const char constName[] PROGMEM = {"PI, e, sqrt(2)"};
unsigned int d, k;
char c;
void setup() {
    Serial.begin(9600);
    while (!Serial);
    for (k = 0; k < 3; k++) {
        d = pgm_read_word_near(constSet + k);
        Serial.println(d);
    }
    int len = strlen_P(constName);
    for (k = 0; k < len; k++) {
        c = pgm_read_byte_near(constName + k);
        Serial.print(c);
    }
}
void loop() {}
```
State Machines

A state machine is a modelling strategy used to design both computer programs and sequential logic circuits.

A given problem/system is described by a set of states in a way that the transitions between states is a convenient representation of the behaviour of the system we try to model or the problem we try to solve.

For example, the state of the weather may be \{sunny, rainy, cloudy\}. The state of the gearbox of an automobile may be \{neutral, first gear, second gear, third gear, forth gear, fifth gear\}.

The set of states is finite.

The way the state machine makes transitions from state to state depends on the present state and the inputs to the system.
The behaviour of the state machine is deterministic, i.e., for a given state and a given input, the state machine always makes the transition to the same fixed state.

The state machine receives also a finite set of inputs, and produces a finite set of outputs.

States, inputs and outputs are all put together by a state diagram.
Example

We want to switch on a LED while a button is pressed, and switch it off otherwise.

states = \{S0, S1\}, or \{on, off\}.
inputs = \{pressed, released\}.
outputs = \{LED on, LED off\}.

The state diagram and the way the output depends on the state are depicted as follows.
The simplest way is to implement the behaviour of the state machine by a switch-case, as follows.

```c
state = S0;
while(true)
    switch(state) {
        case S0:
            if(digitalRead(button)==RELEASED)
                state = S0;
            else
                state = S1;
            break;
        case S1:
            if(digitalRead(button)==PRESSED)
                state = S1;
            else
                state = S0;
            break;
    }
```

António Abreu (ESTSetúbal)
Evaluating the output from the state is done by,

```c
switch(state) {
    case S0:
        digitalWrite(LED,LED_OFF);
        break;
    case S1:
        digitalWrite(LED,LED_ON);
        break;
}
```
Sleep modes in microcontrollers exist for saving electrical power. In fact, in many cases there is no need for the microcontroller to execute instructions; in such cases it is important to have some means to save power. This is typically achieved by disconnecting the oscillator from some parts of the microcontroller.

The ATmega328p has several sleep modes. Each mode allows different amounts of power to be saved. See details on the datasheet.

In order for the microcontroller to resume program execution, it must wake. There are also several ways to wake it: external interrupt, pin change interrupt, timer interrupt, watchdog and \( \text{I}^2\text{C} \) address match.
We want the arduino going to sleep mode after 5 secs. and be waken by the serial port.

The start bit (falling transition) is the event that triggers the external interrupt, thus awaking the arduino. This requires pin 0 (rx) to be externally connected to pin 2 (external interrupt). The LED is lit while the arduino is alive, and not lit while sleeping.
#include <avr/sleep.h>
volatile unsigned int count = 0;
void setup()
{
    Serial.begin(9600);
    pinMode(13, OUTPUT);
    digitalWrite(13, LOW);
    pinMode(2, INPUT);
    digitalWrite(2, HIGH);
}
void loop()
{
    delay(5000);
    sleepSetup();
}
void sleepSetup()
{
    sleep_enable();
    attachInterrupt(0,pinInterrupt,FALLING);
    set_sleep_mode(SLEEP_MODE_PWR_DOWN);
    digitalWrite(13,LOW);
    sleep_cpu();
    Serial.print(count);
    Serial.println("-th woke up");
    digitalWrite(13,HIGH);
}

void pinInterrupt()
{
    count++;
    sleep_disable();
    detachInterrupt(0);
}
Bibliography

http://mediatechnology.leiden.edu/openaccess/arduino
http://arduino.cc/
http://www.makecourse.com/