Index

In this manual, all the libraries are in its **version (V0.0)**.

1. Debounce.h

Description: Debouncing & filtering digital/analog singnals at the input Arduino pins.

Classes:

- **R_TRIG:** Returns “true” when detects a rising edge of its variable.
  - No constructor arguments
  - **Functions:**
    - `void compute(bool IN)`: You enter the variable that you want to detect the rising edge. It stores the value in a private variable named “Q”.
    - `bool get_Q()`: Get the value computed.
  - **Example code:**

```
#include <Debounce.h>

void setup() {
  Serial.begin(9600);  // Initialize the serial to see the results
}

void loop() {
  bool variable_to_read = false;
  static R_TRIG my_r_trig;  // create an static R_TRIG object

  my_r_trig.compute(variable_to_read); // computing
  if (my_r_trig.get_Q()) Serial.println("Rising edge"); // print "Rising
edge" if there is a rising edge of "variable_to_read"
}
```

- **F_TRIG:** Returns “true” when detects a falling edge of its variable. Is the same structure as R_TRIG.

    - **Example code:**

```
#include <Debounce.h>

void setup() {
  Serial.begin(9600);  // Initialize the serial to see the results
}

void loop() {
  bool variable_to_read = false;
  static F_TRIG my_f_trig;  // create an static R_TRIG object

  my_f_trig.compute(variable_to_read); // computing
  if (my_f_trig.get_Q()) Serial.println("Falling edge"); // print
"Falling edge" if there is a rising edge of "variable_to_read"
}
```

- **PhysicButton:** Creates an object that debounces the assigned digital input. It incorporates a min_time activated to return “true” value of the button and a min_time to return a “false”
    - Constructor arguments:
        - Obligatory: `PhysicButton(int pin, int mode)`
            - `int pin:` Digital pin to read
            - `int mode: (INPUT or INPUT_PULLUP).`
        - Optional: `PhysicButton(int pin, int mode, unsigned debounce_time = 100, unsigned debounce_time_off = NULL)`
    - **Functions:**
        - `int Read():` This function is “int” because it can return not only the state of the digital input, it can return “RISING” or “FALLING” value. These values are macros for the preprocessor and they are saved by default in the Arduino IDE.
        - `unsigned long getTimeOn():` This function returns how much time the button has been “true”.

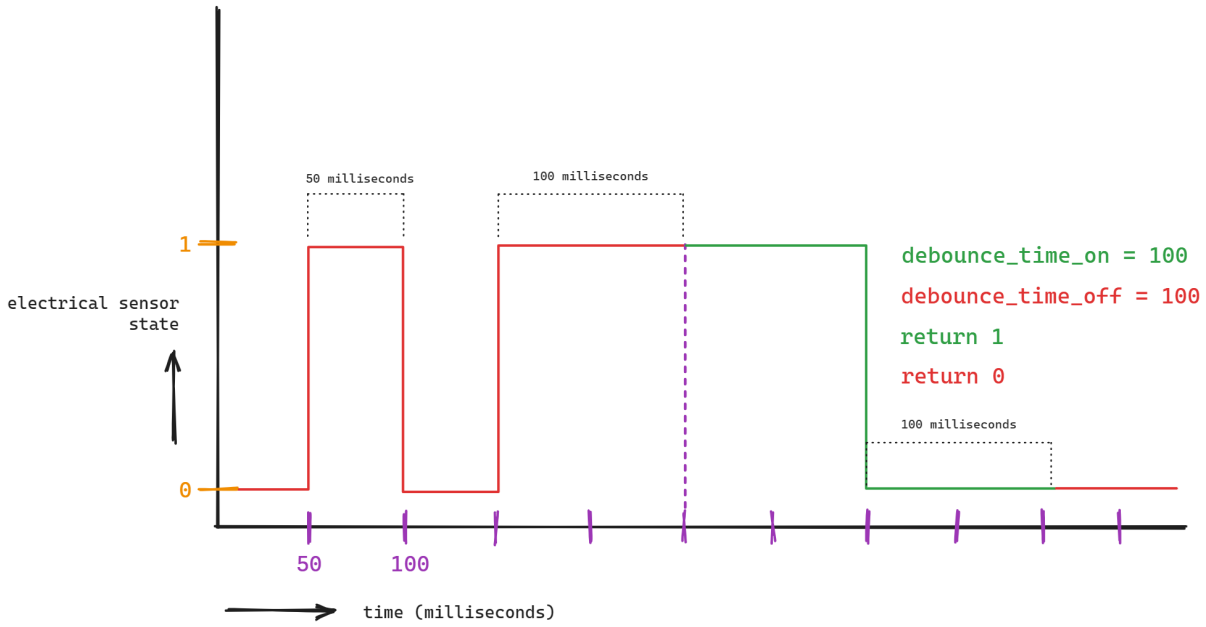
        - `unsigned long getTimeOff()`: This function returns how much time the button has been “false”.

    - **Some `public` variables:**

        - `bool invert_input`: When you set the mode to `INPUT_PULLUP`, the value of your digital input is inverted electronically, to correct this inversion and program with the “logic” values of the real sensor, you can set this variable to `"true"`.

    - **Example code:**

```
#include <Debounce.h>

void setup() {
  Serial.begin(9600);  // Initialize the serial to see the results
}

void loop() {
  static PhysicButton my_button(5, INPUT_PULLUP); //creating a button at
the pin 5 with INPUT_PULLLUP = pinMode(5, INPUT_PULLUP)

  int my_button_stored = my_button.Read();
  // if you want to read the value in multiple lines in the same loop
you have to store the value in a variable,
  // this is because and edge is detected when the last value is
different than the next, this indirectly means that only lasts one scan
cycle

  if (my_button_stored == RISING) Serial.println("button rising");
  if (my_button_stored == FALLING) Serial.println("button falling");
  if (my_button_stored == HIGH) Serial.println("button high");
  if (my_button_stored == LOW) Serial.println("button low");

  if (my_button.getTimeOn() > 2000) Serial.println("Pressed more than
2000 millisseconds");
  if (my_button.getTimeOff() > 2000) Serial.println("Unpressed more than
2000 millisseconds");
}
```



- **VirtualButton:** This works by the same form than “PhysicButton” but it does not take a digital pin, you can read instead the variable you give to it. The only difference in the functions is in:
  - Constructor parameters:
    - Obligatory: None
    - Optional: `(unsigned debounce_time_ = 100, unsigned debounce_time_off_ = NULL).` The same as PhysicButton
  - **Functions:**
    - `int Read(bool state):` In PhysicButton that `(bool state)` was given internally by a `digitalRead()` but now we don’t read a pin, we read the variable we put inside `Read(bool)`.
    - `unsigned long getTimeOn(bool state):` The same happens in the other functions, we have to introduce some value to compute.
    - `unsigned long getTimeOff(bool state)`

- **PhysicPotenciometer:** The objective of this class is the same as PhysicButton, filter and debounce the input, but in this case will be an **analog pin input.**
    - Constructor arguments:
        - Obligatory: `(byte AI_pin_, int short value_tolerance_, unsigned short time_tolerance_)`
            - `byte AI_pin_:` Analog pin in
            - `int short value_torelance_:` Every time the analog value changes more than `"value_tolerance_"`, this will get updated. This tries to eliminate the electromagnetic noise and the false contacts of the pin.
            - `unsigned short time_tolerance_:` This is the minimum time between each update of the value, if we set this to 1000, either the value changes more than "value_tolerance_", this will not be updated until the time since the last actualization is more than 1000.
        - Optional: None
    - **Functions:**
        - `unsigned short getFiltered():` This function computes, updates and returns the value.
    - **Example Code:**

```
#include <Debounce.h>

void setup() {
  Serial.begin(9600);  // Initialize the serial to see the results
}

void loop() {
  //creating a PhysicPotenciometer at the pin A0, minimal change value = 25, minimal time passed = 10
  static PhysicPotenciometer my_potenciometer(0, 25, 10);

  if (my_potenciometer.getFiltered() > 512) Serial.println("Above 512");
  if (my_potenciometer.getFiltered() < 512) Serial.println("Below 512");
}
```

- **VirtualPotenciometer:** The same as PhysicPotenciometer but now it doesn't compute an analog value from a pin, you can give the `int long` value what you want.
    - Constructor arguments:
        - Obligatory: `(byte AI_pin_, int short value_tolerance_, unsigned short time_tolerance_)`
        - Optional: None
    - **Functions:**
        - `int long getFiltered(int long value):`
    - **Example Code:**

```
#include <Debounce.h>

void setup() {
  Serial.begin(9600);  // Initialize the serial to see the results
}

void loop() {
  int long value_to_compute = 500;
  //creating a VirtualPotenciometer at the pin A0, minimal change value
= 25, minimal time passed = 10
  static VirtualPotenciometer my_potenciometer(25, 10);

  if (my_potenciometer.getFiltered(value_to_compute) > 512)
Serial.println("Above 512");
  if (my_potenciometer.getFiltered(value_to_compute) < 512)
Serial.println("Below 512");
}
```

2. Joystick.h

Description: Code for an Arduino Joystick template (2 axis joystick, convertible to a 3 simulated axis joystick)

This library takes advantage of the button incorporated into the most common joystick in Arduino kits.

You can convert this 2 axis to 3. When you press the button, the Y axis value goes to the Z axis (simulated), obviously you won't be able to used both a the same time.

At the time, this library just incorporates this class and depends from the **Debounce.h library.**

Classes:

- **R_TRIG:** Returns "true" when detects a rising edge of its variable.
  - Constructor arguments
    - Obligatory: `(byte x_pin, byte y_pin, byte sw_pin, int sw_mode, byte axis_num)`
      - `byte x_pin:` Analog pin for the axis X.
      - `byte y_pin:` Analog pin for the axis Y.
      - `byte sw_pin:` Potentiometer Switch pin.
      - `int sw_mode:` Potentiometer Switch pin mode.
      - `byte axis_num:` 2 axis (normal joystick), 3 axis (simulated 3rd axis).
    - Optional: None
  - **Functions:**
    - `unsigned short get_x();`
    - `unsigned short get_y();`
    - `unsigned short get_z();`
    - `int get_sw();`
  - **Example code:**

```
#include "string.h"
#include "Joystick.h"

// creating a joystick with axisX analog pin = 15 | axisY analog pin =
14 | button pin = 22 | button mode = INPUT_PULLUP | 3 axis mode
Joystick joystick(15, 14, 22, INPUT_PULLUP, 3);
```

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

void loop() {
  if (joystick.get_sw() == RISING) Serial.println("Sw");

  String str_print = "{joystick: {x:" + String(joystick.get_x()) + "},
{y:" + String(joystick.get_y()) + "}, {z:" + String(joystick.get_z()) +
"}}";
  Serial.println(str_print);
}
```

---

3. Cin.h

Description: Arduino "cin" functions. (tries to simulate the C++ cin function).

This library does not have any class, it works with a single function called “cin”.

The idea is that you stop the program in a while loop when you call the **“cin”,** waiting for a value entered through the **Serial port.** The value entered will be stored in the variable you indicate, using a pointer.

Its structure changes for different types of data (`bool or not bool`). That happens because the function in `not bool` values, has two arguments to constrain the value you entered, for a **min** and a **max** value, so in a bool variable, the possible values are just 0 and 1, so it’s not necessary to constrain.

- Functions:
  - `bool` variables: `void cin(String name, bool& p_var)`
    - `String name:` This is the name that will appear at the `Serial` port when you call the function, so for example:
      - String name = “LED_1”;
      - Serial output = Enter the LED_1 value.
    - `bool& p_var:` There you put the variable you want to assign to the value you enter.

- `not bool` variables: `void cin(String name, int min_val, int max_val, int& p_var)`
    - The same as the last function shown, but it has two more arguments; `min_val, max_val.` The value you enter will be constrained between these values.

- **Extra argument:** You can add a function (callback) to the cin call, so after you enter your value, this function will be called automatically
    - `bool` variables: `void cin(String name, bool& p_var, void (*p_void)())`
    - `void cin(String name, int min_val, int max_val, int& p_var, void (*p_void)())`

- Example Code:

```
#include "Cin.h"

int int_var;
bool bool_var;
float float_var;

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

void loop() {
  char c = Serial.read();

  while (!Serial.available()) {
    // your main program will be here
  }

  // we read the value entered by the Serial port whit "char c", so when
equals the char we have at the if(condition), the function is called
  if (c == 'i') cin("my int var", 0, 100, int_var);
  if (c == 'f') cin("my float var", 0, 100, float_var);
  if (c == 'b') cin("my bool var", bool_var);
  if (c == 'b') cin("my bool var", bool_var, callback_example);
}

void callback_example(){
  Serial.println("callback_test");
}
```

---

4. Blink.h

Description: It Incorporates classes that simulate types of time control functions. In automation there are some like "TON", "TP", "TOF", "TONR".

Classes:

- **Blink:** You set the time, and it works like a clock, each time you set it changes the output state
  - Constructor arguments
    - Obligatory: `Blink(unsigned cycle_time_)`
    - Optional: None
  - **Functions:**
    - `bool getState();`
- **TON:** It works exactly like the TON in **PLC programming**
  - No constructor arguments
  - **Functions:**
    - `void compute(bool IN, unsigned long PT);`
    - `bool get_Q();`
    - `unsigned long get_ET();`
- **TP:** It works exactly like the TP in **PLC programming**
  - No constructor arguments
  - **Functions:**
    - `void compute(bool IN, unsigned long PT);`
    - `bool get_Q();`
    - `unsigned long get_ET();`

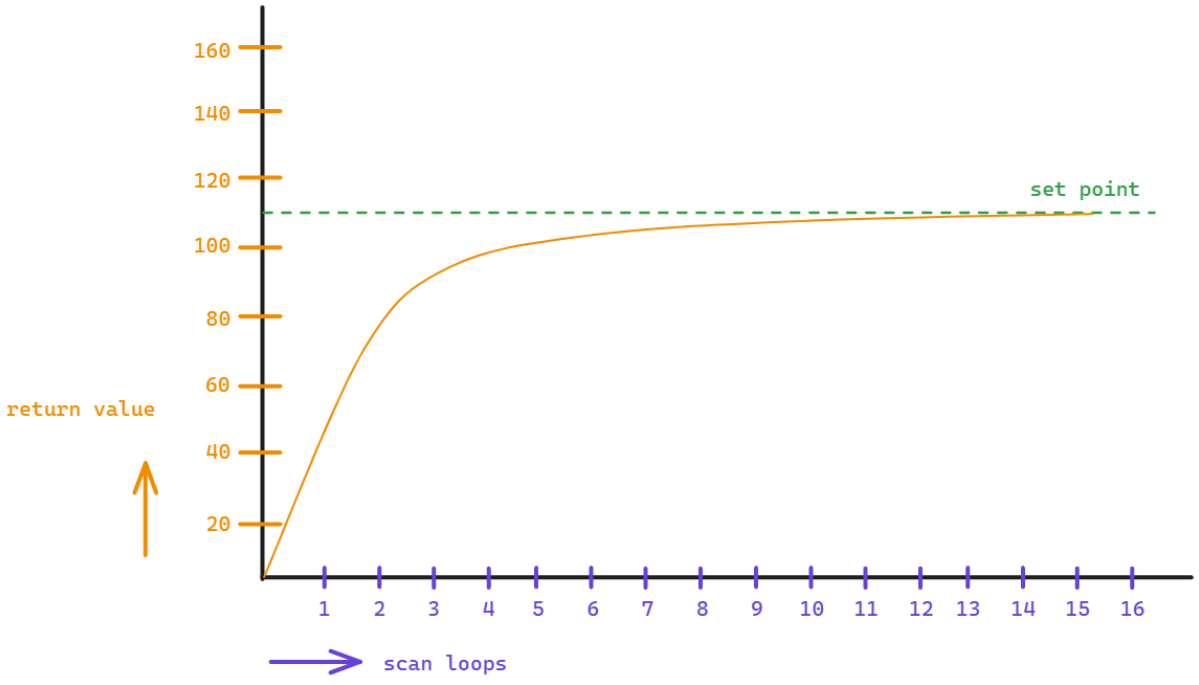
- **TOF: Coming soon**
  - No constructor arguments
  - **Functions:**
    - `void compute(bool IN, unsigned long PT);`
    - `bool get_Q();`
    - `unsigned long get_ET();`
- **TONR: Coming soon**
  - No constructor arguments
  - **Functions:**
    - `void compute(bool IN, unsigned long PT);`
    - `bool get_Q();`
    - `unsigned long get_ET();`

5. Math_aux.h

Description: It incorporates some math functions like, "bezier curves", "smooth control of variables", "equation_solver", "acceleration_maxspeed_deceleration control for a 3 dimensions space"

Classes:

- **Smooth:** This class is most commonly used to control servos. When you set the value to a servo it goes abruptly to that point.



By this way it goes slowly to the point.

- Constructor arguments
  - Obligatory: `Smooth(float percent_value_, unsigned cycle_time_, double first_value);`
    - `float percent_value_:` (0-1) values. As higher it is, slower it will get the set point
    - `unsigned cycle_time_:` This indicates every how many milliseconds it actualizes its value
    - `double first_value:` This is necessary if you want to start with values different from 0, if you set this to 0, but you want that your variable starts with 30, it will have to reach 30 from 0 first.
  - Optional: None
- **Functions:**
  - `double getSmoothed(double in_value):` This updates and returns the value. At `double in_value` you set the value you want to reach.

- **EquationSolver:** Simply, you give f(x), and the value you want to equal, for example: f(x) = x + 1 and you want the x value that returns 5 at this function. By algebra: 5 = x + 1 → 5 - 1 = x + 1 - 1 → 4 = x. This class asks for a first range of values to search the returning value, but it's not necessary to be as close as possible, I most often search between a = -10 and b = 10, but now I will explain what these variables are.
  - No constructor arguments
  - **Functions:**

`double solve_equation (double (*f_)(double), double a_, double b_, double point_to_reach_):`

- `double (*f_)(double):` Here you give your f(x) function, it has to `return double` and one `double` argument, like that:

```
double my_fx(double x){
      return (x + 1);
}
```

- `double a_:` This value NEEDS TO BE LOWER THAN b_.
- `double b_:` This value NEEDS TO BE HIGHER THAN a_.

This two values tells the first range to search the solution of the equation, if you don't know anything about the function set `a_ = -100` and `b_ = 100`.

- `double point_to_reach_:` The value that you want to solve for f(x) → f(x) = `point_to_reach_`.

- Example Code:

```
#include "Math_aux.h"
#include "Cin.h"

double a = -10, b = 10;
double point_to_reach = 0;

EquationSolver eq_solve;

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

void loop() {
  char c = Serial.read();

  while (!Serial.available()) {
  }
  if (c == 'a') cin("a", -20, 20, a);
  if (c == 'b') cin("b", -20, 20, b);
  if (c == 'p') cin("point to reach", -5, 5, point_to_reach);
  if (c == 'c') {
    double result = eq_solve.solve_equation(F_X, a, b, point_to_reach);
    Serial.print("\nresult: ");
    Serial.println(result);
  }
}

double F_X(double x) {
  return x*2 + 5;
    // return exp(x)*(1+x);
}
```

- Other functions of the library, they just copy other c++ functions that are usable for int values, and I just modify those to be usable for `float` and `double` values.
    - `double abs_2(double in):` abs() but works with double.
    - `double map_double(double x, int in_min, int in_max, int out_min, int out_max)`: map() but works with double.
    - `double round_2(double in, short unsigned decimals)`: round() but you can tell what decimal you want to round, for example: I want this number 1.736 rounded to one decimal: `round_2(1.736, 1)` -> so it returns 1.7

6. Download link

Click here 

JAVI MEJIAS

Make It Touchable

Contact: mejiasrjavi@gmail.com