

## Intro

Introduction to Tweakly Page. 3

Installing Tweakly Page. 3-4

## Writing your first program with Tweakly

1.0 Blink – Basic Page. 4-5

1.1 Blink – Il Pad Page. 5-6

1.2 Blink – Il TickTimer Page.6-8

## Pad Management

2.0 Initialization Modes Page. 9

2.1 Initialization Page. 9-13

2.2 read and write (explicit methods) Page 13-14

2.3 on and off (explicit methods) Page 14

2.4 write (implicit method) Page 15

2.5 toggle Page. 15

2.6 pinNumber Page. 16

2.7 lock and unlock Page 16-17

2.8 Multiple Action Functions Page. 17-24

2.9 Debounce Buttons and Methods Page. 24-25

## TickTimer

3.0 Timing Page 26-28

3.1 Assigning Statements via Lambda Page. 28-29

## doList

4.0 Lists and Explicit Scrolling Method Page 29-31

4.1 Implicit Scrolling Method Page 31-32

## inputHunter

5.0 InputHunter Page 32-34

## Rotary encoders

6.0 Managing Rotary Encoders Page 35-36

## Usage Scenarios

7.0 Usage Scenarios Page. 36

## Introduction to Tweakly

Tweakly is a library for Arduino made specifically to simplify the drafting of a code and use advanced programming mechanisms on boards that do not have the possibility to use an RTOS.

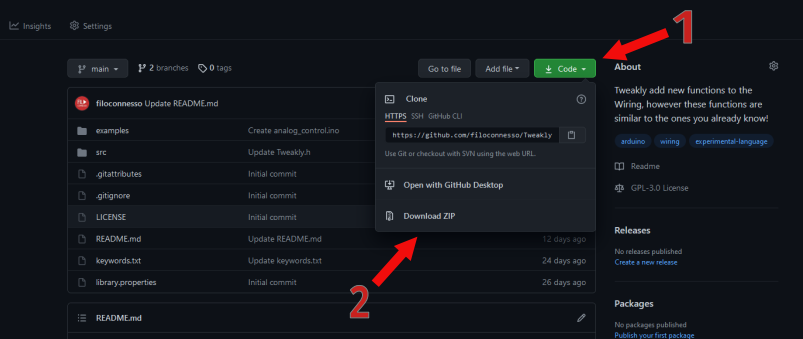
The Tweakly library is designed to be as "Memory Friendly" as possible, so much so that it allows you to be run, reducing memory usage as much as possible, on micro-controllers with medium-low technical characteristics, also working on Attiny85.

The code that makes Tweakly work is written in C++ and is executable on hardware with Core software based on Arduino.h. All of the features included in Tweakly do not require sub-libraries and install additional software, all you need is the Arduino development IDE and you're ready to begin!

## Installing Tweakly

You can install Tweakly in your development IDE by downloading the zip package from GitHub:

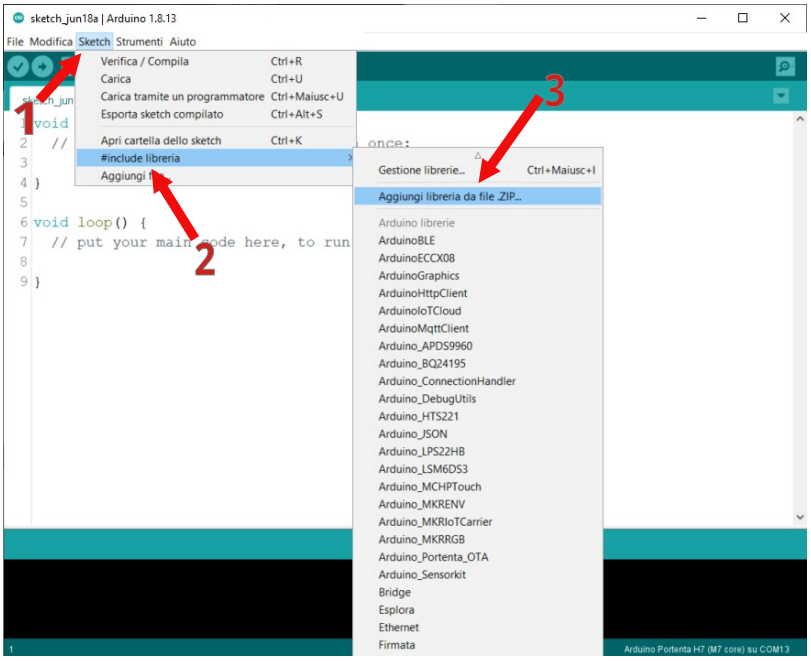
<https://github.com/filoconnesso/Tweakly>



1. Click on the "Code" button.

2. Download the .ZIP file by clicking on "Download ZIP".

Open the Arduino Development IDE.

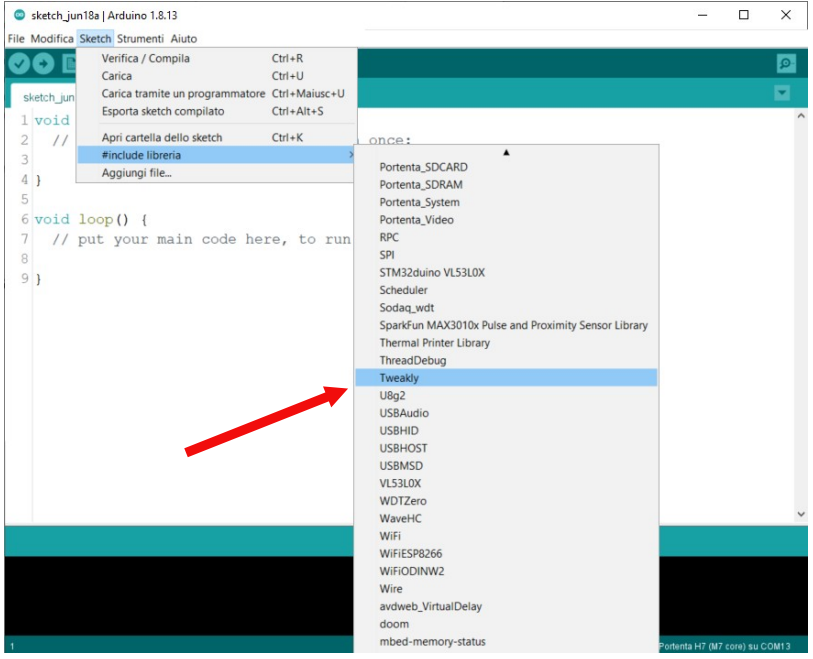


1. Open the "Sketch" menu.

2. Click on "#include Library".

3. Add the .ZIP file you just downloaded using the "Add Library from .ZIP File" feature.

If the library has been added correctly, you will see the "Tweakly" library in the list of libraries:



## Writing your first program with Tweakly | Blink – The Basics

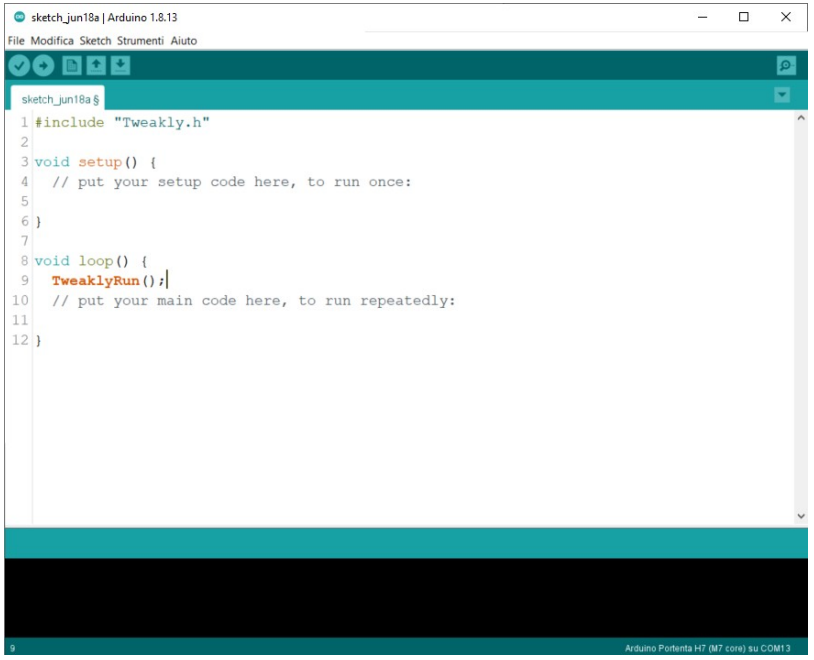
To test the correct functioning of the library we can write the classic Blink to make flash an LED, but using all of Tweakly's methods.

Before we begin, however, we need to understand two fundamental things:

1. In each of your codes it must include Tweakly, so you will need to write at the beginning of the code the instruction **#include "Tweakly.h"**.

2. To make Tweakly run correctly in your program's **loop()** function there must always be **TweaklyRun()**;.

Consequently, the basic structure of your program will be as follows:

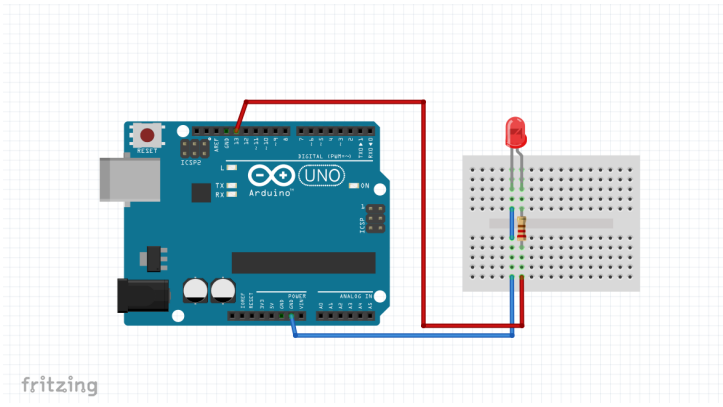


Next to write our Blink we need to initialize a pin like OUTPUT and to turn on and off with a timer, which has an activation period that we can decide according to our needs.

## 1.1 Writing your first program with Tweakly | Blink – The PAD

To control our LED we need a resistor, at least 220 Ohms to avoid that with prolonged use of the LED it will burn out.

Connect everything in the following way, using pin 13 of the Arduino (of course the choice of pin is free) :

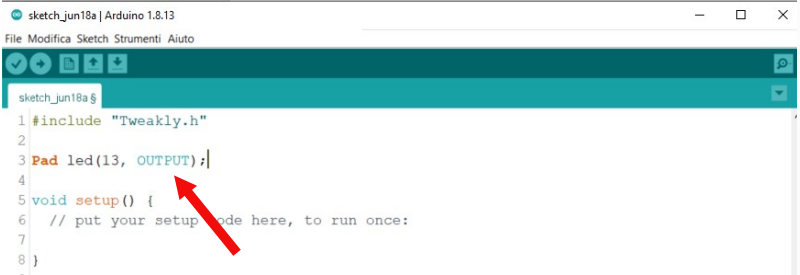


In our program we must indicate that the LED is connected to pin 13 and that this is an output, then set it as **OUTPUT**.

To initialize a pin we use the **Pad** class and create a led object, also in this case the

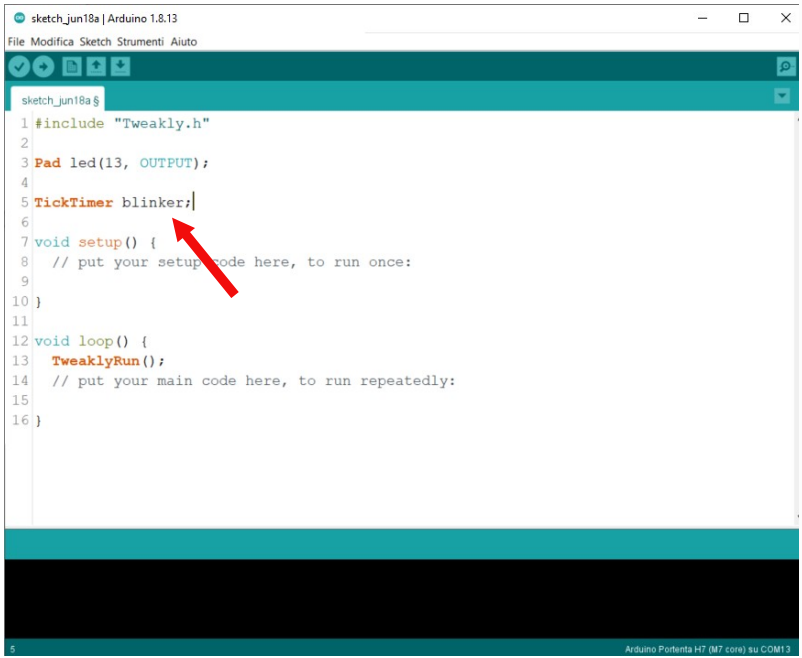
The choice of the pin name is free.

The setting of a pin must be done before **setup()** :

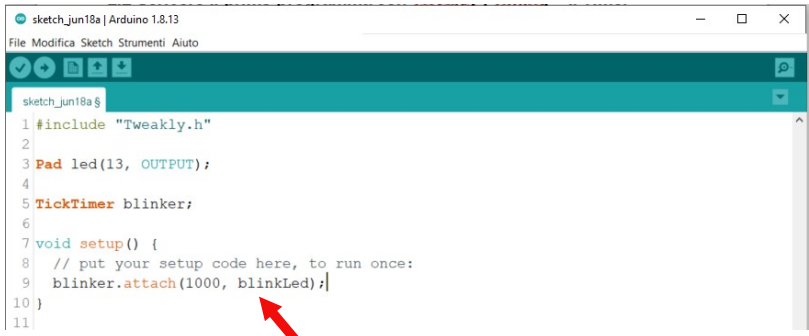


## 1.2 Writing your first program with Tweakly | Blink – The TickTimer

Now we need to use the TickTimer class to create a timer, this will allow our led to flashing whenever we want. First let's create a TickTimer like this:



In the **setup()** we tell the timer the time interval that should elapse between one call and the next and what it needs to do every time it goes into action, to do all this we use the **attach()** method of the **TickTimer** :

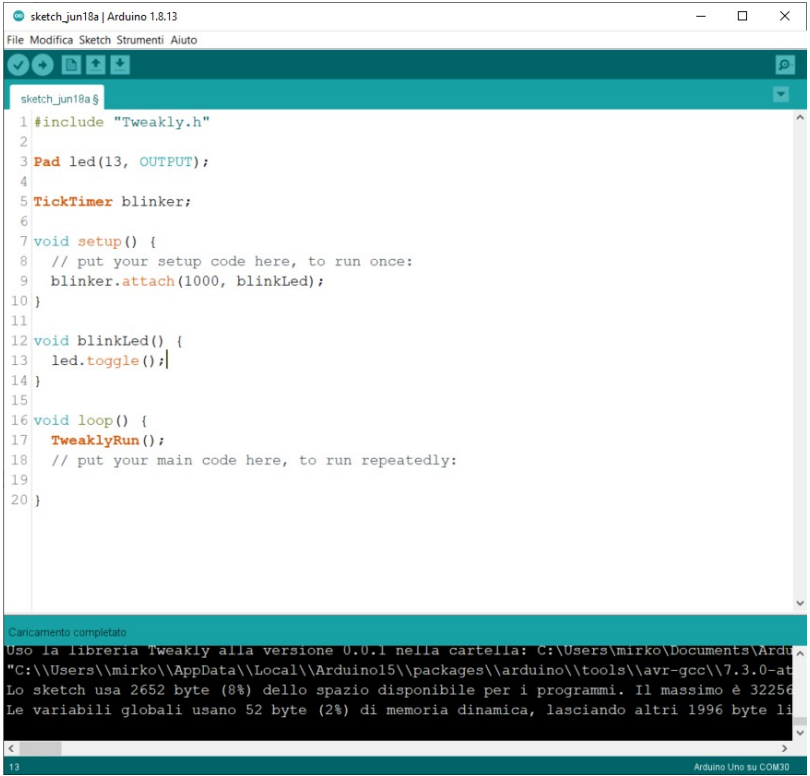


To complete our program we need to create a void statement, which we'll call **blinkLed(),** and inside it we will give the command to turn off and on the led using the **toggle()** method included in the **Pad** class.

The **blinkLed** instruction was assigned to the timer and then, according to the code we wrote, Every **1000 milliseconds** this is recalled and having **led.toggle()** inside, the connected LED at **pin 13** it will start to turn off and on **every second** :



You can see that in this first program we didn't use the **loop()** at all, of course however, all the operation runs behind **TweaklyRun();** which at this very moment is checking the status of pin 13 and the timing on your board. You can create multiple timers in your code, and control several LEDs together, they will flash differently without one affecting the other. On the other hand, this method is almost non-blocking and allows the **loop();** to turn regardless of the timer set.



**PLEASE NOTE** : The processes managed by Tweakly are not parallel, they are based on time control using **millis()**. You need to understand that in order to create code that is optimized using Tweakly can be useful to take advantage of all the methods offered by the library, and it is recommended to avoid using **delay()** within the program.

## 2.0 Pad Management | Initialization Modes

To manage Pads on Tweakly you can use the Pad class which allows you to create objects that they inherit functions to manage your board's pins.

The Pads can be initialized at the beginning of the code and depending on the type of pin they can be set in the following ways:

• OUTPUT

• INPUT

• INPUT\_PULLUP

• INPUT\_PULLDOWN

• PWM\_OUTPUT

• ANALOG\_INPUT

## OUTPUT

The pin is initialized as an output: useful for connecting LEDs, relays, motor drivers, TRIGGER pins. An ultrasonic sensor is anything that involves driving an external component.

## INPUT

The pin is initialized as an input: useful for connecting buttons with an external resistor, sensors PIR and all components that can send a digital signal to your board via a PIN.

## INPUT\_PULLUP

The pin is initialized as an input using an internal resistor connected to 5V or 3.3v (for 3.3V logic boards)

## INPUT\_PULLDOWN

The pin is initialized as an input using an internal resistor connected to GND (some boards support it).

## PWM\_OUTPUT

Activation mode applicable to all PWM pins, this will allow you to take advantage of some features additional to digital pins.

## ANALOG\_INPUT

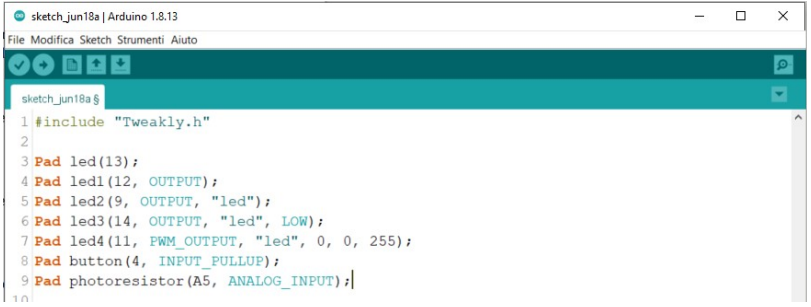
Activation mode dedicated to the analog pins of your board.

## 2.1 Pad Management | Initialization

To initialize a pad you can invoke the Pad class and give the object a name.

Understand that each Pad object must have a different name from all other initialized objects in your code.

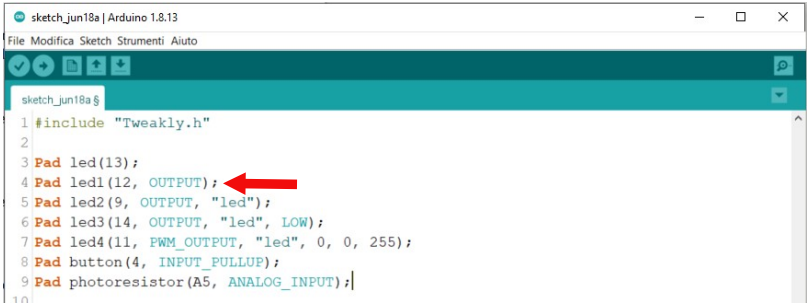
An example of initialization of one or more Pads can be the following:



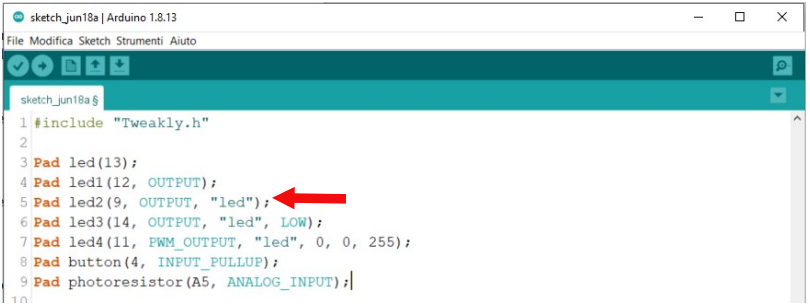
According to our needs, we can initialize N Pad objects to have access in a different way to each pin on our board.

If we only indicate the pin number, it is automatically initialized as OUTPUT and will be turned off at the beginning of the program.

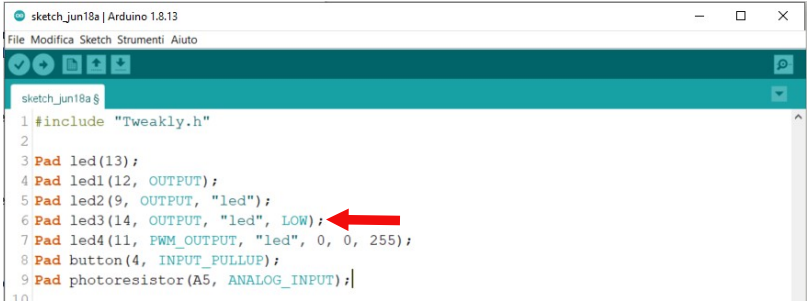
If we want to have a different modality, we would have to indicate, in addition to the pin number, also the How to use it :



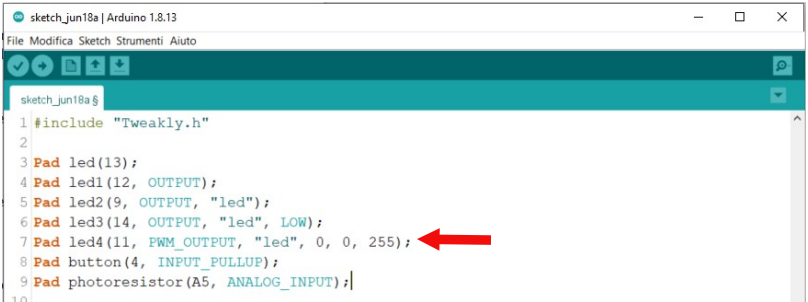
The third parameter that can be useful for organizing the pins is that of the class. A class of pins It can be useful to allow us to give actions to a specific group of pins, perhaps turn them on or turn them all off in one fell swoop with the multi-action functions that we will see later :



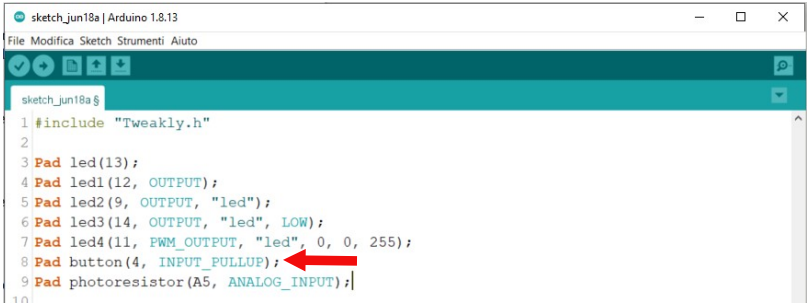
If the need arises, we could decide to turn on or off the pin (by setting the fourth parameter **LOW** or **HIGH**) when initializing the pin itself :



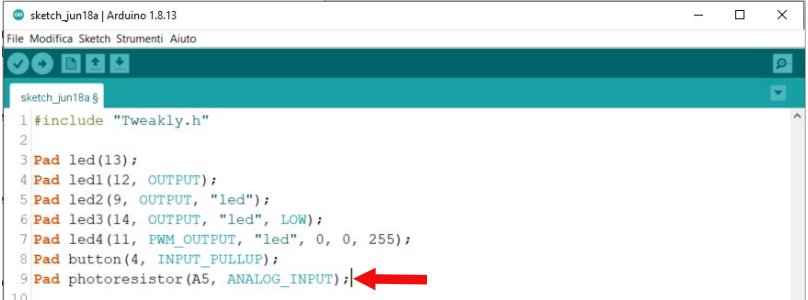
If we want to fully use a **PWM** pin, it is necessary not only to set the mode of use in **PWM\_OUTPUT**, the initial value of the pin should be indicated based on its future use and the value minimum and maximum that we want our pin to assume, by default from 0 to 255 :



If a button without resistance is connected to the pin, we must use the mode of use **INPUT\_PULLUP** :



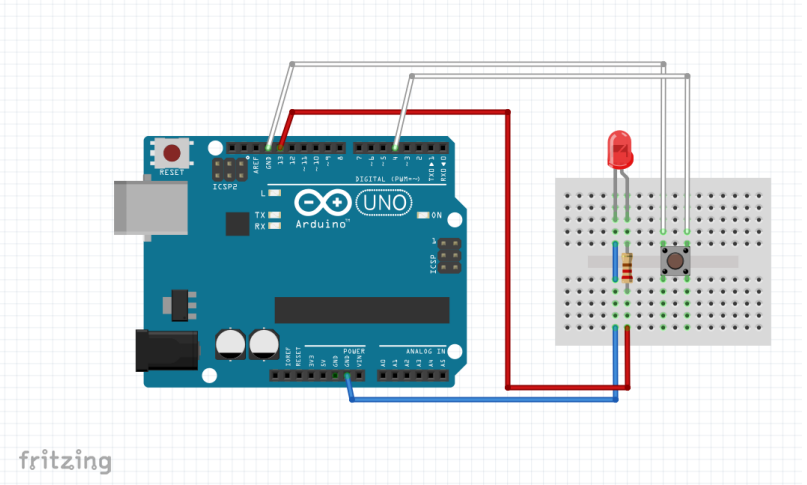
To have full access to analog pins with Tweakly's advanced features, finally, there is the of use **ANALOG\_INPUT** :



Ultimately, we can see that the way of declaring and initializing pins is the same on all available modes, this allows the same control logic to be used on all pins and As a result, always have the same method of managing each individual pin.

## 2.2 Pad Management | read and write (explicit methods)

We can read and write a value to a pin using the **read()** and **write()** methods, these methods are part of the Pad class and are defined as explicit methods, they can be used as The example below:





In the code above, the LED on pin 13 will light up whenever the button on pin 4 is Pressed. Using **INPUT\_PULLUP** the value we have on the pin is always 1 and goes to 0 only if the button is pressed and pin 4 is connected to GND, for this Reason: The negated value of the btn variable was used within the write method with **!btn**.

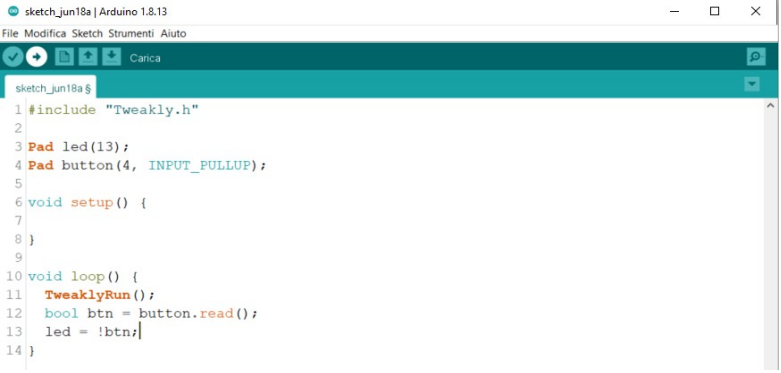
## 2.3 Pad Management | on and off (explicit methods)

If you manage digital pins, as an alternative to **write()** you can use two other explicit methods, namely **on()** and **off().** These two methods can be used for example as shown below:



## 2.4 Pad Management | write (implicit method)

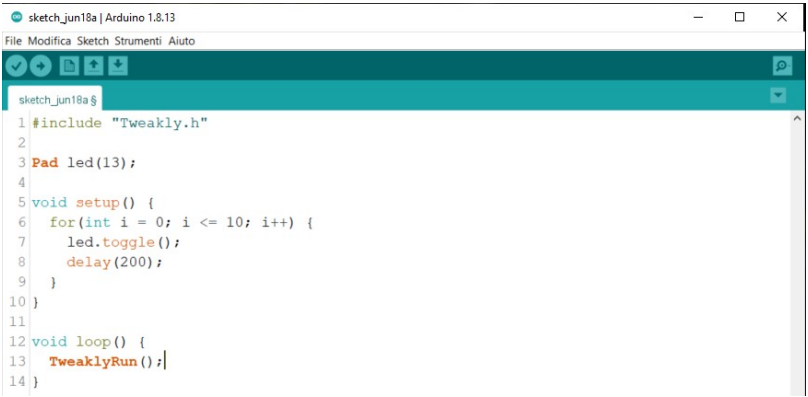
You can assign a value to a pin implicitly using the "=" operator. Below is a Example to understand the use of the method:



This method is also valid for PWM pins, so it will be possible to assign integer values to our Pad with the "=" operator.

## 2.5 Pad Management | Toggle

The **toggle()** method allows you to switch the status of a digital pin fully automatically that goes from 0 to 1 and vice versa. The method can be used to make a simple blink or Applications where switching the state of a pin is required:



In the example above, the LED flashes 10 times when the board is started and then enters the Loop.

## 2.6 Pad Management | pinNumber

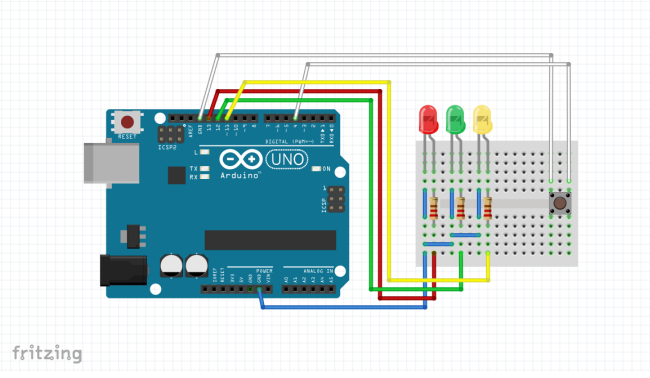
You can return the pin number set in the **pinNumber()** initialization. This method allows you to manage the PIN with secondary functions where it is necessary to indicate the PIN number. pin:



## 2.7 Pad Management | Lock and Unlock

Tweakly allows you to use control methods on groups of pins, these methods may be override the functions that write values to pins during code execution, producing undesirable behaviors of your program. The **lock()** and **unlock()** methods block the action of **digitalWriteAll**, **digitalWriteClass**, **digitalToggleAll**,**analogWriteAll**, **analogWriteClass** and **analogWriteProgressive**.

One possible use of this feature could be to block the execution of actions On a button-operated pin:



To create this program, you just need to write the following code:

While the two LEDs connected on pins 12 and 11 continue to flash thanks to the **TickTimer** set to 1 Second of interval, the red LED connected on pin 13 can only be managed by the button.



You can unlock the pin using the **unlock()** method, the pin will always be manageable by the button but in this case **red.write(!btn)** and the **digitalToggleAll()** will conflict, creating an effect of **flicker** on the pin 13 LED as both functions by turning the LED on and off.

You can quickly lock and unlock pins on all pads initialized as **OUTPUT**and **PWM\_OUTPUT** with the following functions:

• digitalLockAll();

• digitalUnlockAll();

• analogLockAll();

• analogUnlockAll();

## 2.8 Pad Management | "Wiring Style" multi-action functions

It is possible to perform multiple actions on pins set as **OUTPUT** or **PWM\_OUTPUT** with the "Wiring Style" multi-action functions. The functions of this Tweakly feature pack they are very similar to the basic functionality of Wiring but perform actions on all pins or on a class of pins :

## digitalLockAll();

Protects all OUTPUT pins from some multi-action functions.

## digitalUnlockAll();

Unprotect all OUTPUT pins for use with some multi-action functions.

## analogLockAll();

Protects all pins PWM\_OUTPUT from some multi-action features.

## analogUnlockAll();

Unprotect all pins PWM\_OUTPUT to use them with some action features multiple.

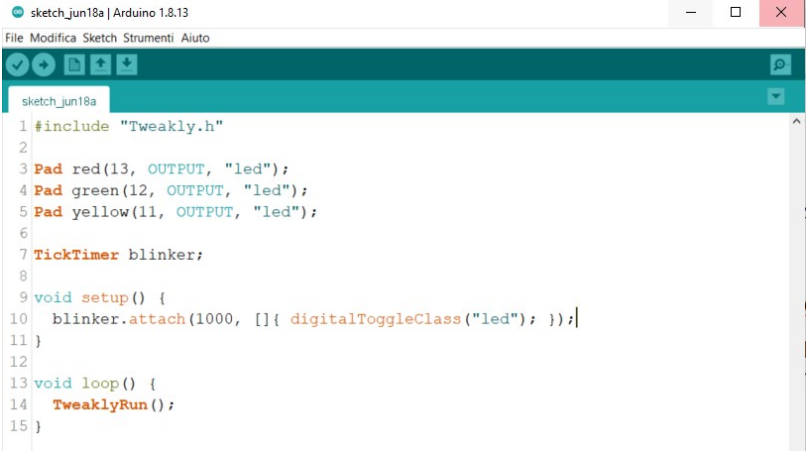
## digitalToggleAll();

Switches all initialized pin states as OUTPUT, pins that are set to 0 switch to 1and vice versa.

## digitalToggleClass(const char\* class);

By indicating the pin class inside the parentheses, it toggles the state of all pins contained in that class, all pins set to 0 switch to 1 and vice versa.

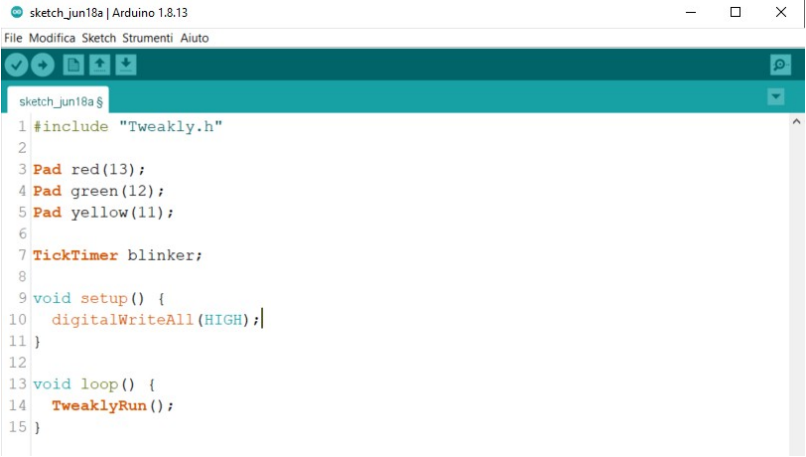
An example of **digitalToggleClass()** is as follows:



All LEDs in the "led" class will be turned off and on with a time interval of 1 second.

## digitalWriteAll(uint8\_t new\_value);

**digitalWriteAll()** allows you to set a value (0 or 1) on all initialized pins as OUTPUT:



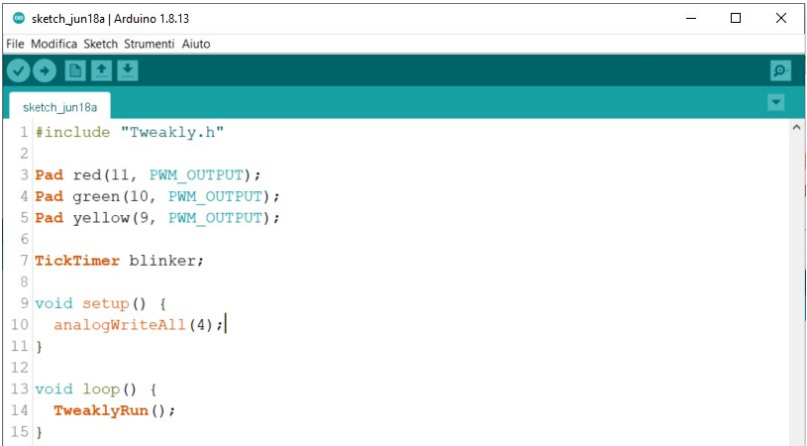
## digitalWriteClass(const char\* class, uint8\_t new\_value);

**DigitalWriteClass()** allows you to set a value (0 or 1) on all pins belonging to a specific class and initialized as OUTPUT :



## analogWriteAll(unsigned int new\_value);

**analogWriteAll()** allows you to set an integer value on all initialized pins as PWM\_OUTPUT :



## analogWriteClass(const char\* class, unsigned int new\_value);

**AnalogWriteClass()** allows you to set an integer value on all pins belonging to a class Specifying pins initialized as PWM\_OUTPUT :



## analogWriteProgressive(uint8\_t pin\_number, unsigned long delay, uint8\_t mode);

**analogWriteProgressive()** is a very useful feature for setting a value progressively min to a max value, indicating a non-blocking delay. This feature allows you to create effects FADE with LEDs or decrease and increase the speed of a motor gradually.

Depending on the value switching method, the time to go from the min value to the max value varies. The switching methods are as follows:

**TO\_HIGH** : From a minimum value to a maximum value with a time interval between a change of value and the other. A delay set to 1000 milliseconds allows a change from a value of 0 to a value 255 in 255 seconds.

**TO\_LOW** : From a maximum value to a minimum value with a time interval between a change of value and the other. A delay set to 1000 milliseconds allows a change from a value of 255 to a value 0 in 255 seconds.

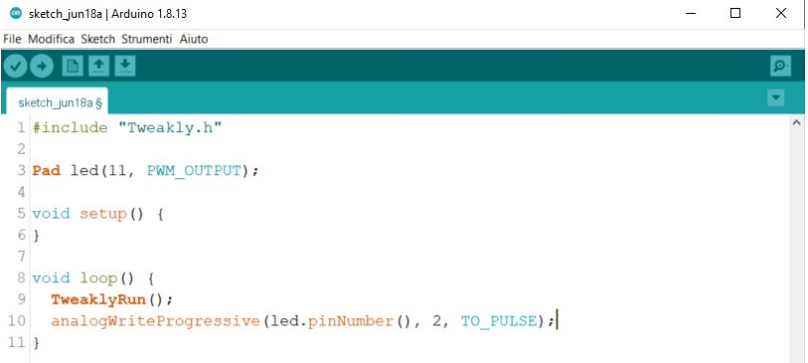
**TO\_PULSE** : From a maximum value to a minimum value and vice versa with a time interval between One change of value and the other, forever. A delay set to 1000 milliseconds allows a Go from a minimum value to a maximum value and vice versa in 255 seconds, forever.

**HIGH\_TO\_EDGE** : From a maximum value and a minimum value in a given delay time. A Delay set to 1000 milliseconds allows a change from a value of 255 to a value of 0 in 1 second.

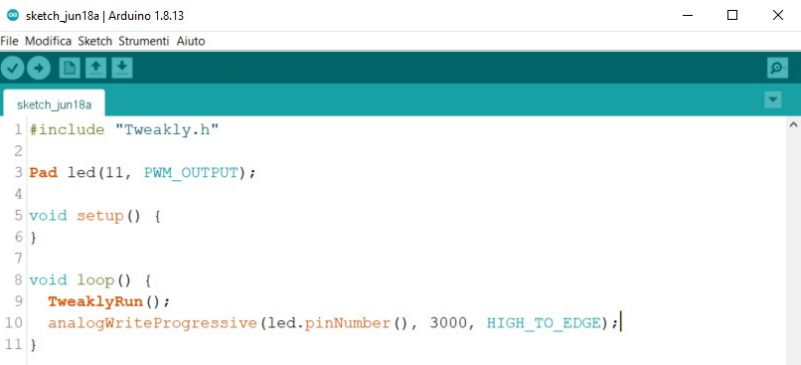
**LOW\_TO\_EDGE** : From a minimum value to a maximum value in a given delay time. A Delay set to 1000 milliseconds allows a change from a value of 0 to a value of 255 in 1 second.

**PULSE\_TO\_EDGE** : From a minimum value to a maximum value and vice versa forever. A delay set to 1000 milliseconds, it allows a transition from a minimum to a maximum value and vice versa in 1 second.

An example for the realization of the FADE effect applied to a LED can be performed with the the following code :



If we want to fully turn on the LED in 3 seconds we can use the **HIGH\_TO\_EDGE** that defines the rising edge of the PWM value:



In case you need to start from a high value and go down to a low value using the **TO\_LOW** and **LOW\_TO\_EDGE** you need to initialize the pin with a default value of Maximum desired value :



Using the TO**\_HIGH, TO\_LOW, HIGH\_TO\_EDGE,** and **LOW\_TO\_EDGE** modes the pin used in the progressive setting is disabled once the final preset value is reached, to re-enable the pin you will need to use an invocation to the **analogAttach()** function.

To re-enable the pin we can use a TickTimer as in the example below or any other functionality that allows us to re-access the desired PWM pin setting:



Every 7 seconds the pin will revert to 255 and as a result **analogWriteProgressive()** will turn off by gradually new LED.

## analogAttach(uint8\_t pin, unsinged int new\_value);

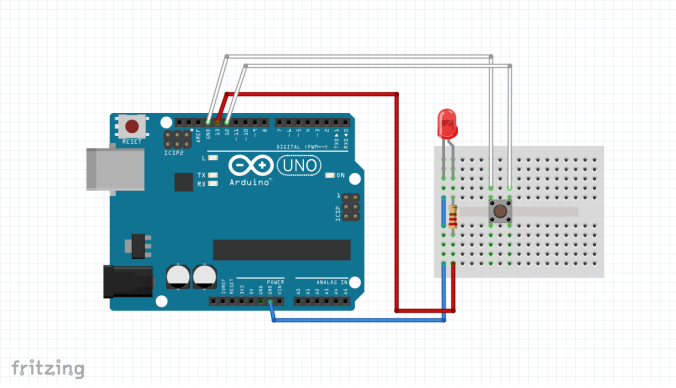
**analogAttach()** allows you to re-enable a PWM pin when it is disabled with a of the **analogWriteProgressive()** function, or after calling the **analogDetach()**.

## analogDetach(uint8\_t pin);

Allows you to disable a PWM pin by indicating the pin to be disabled.

## 2.9 Pad Management | Debounce Buttons and Methods

On Tweakly you have the option to manage the buttons in 3 different methods, the first method is without debouncing and the value that is read from the pin can be read with the **read()** method:



## digitalPushButton(uint8\_t pin);

**digitalPushButton()** allows you to read the bouncing-free state of a button connected to a Input pin :



## digitalSwitchButton(uint8\_t pin);

**digitalSwitchButton()** allows you to read the bouncing-free state of a button connected to a input pin with switch mode. This particular function maintains the state of the last Press and toggle each time the button assigned to the pin is pressed:



If we press the button, the LED lights up and stays on until the next time we press the button to then turn off.

## 3.0 Timing | TickTimer

In the previous chapters of this guide, we have used TickTimers several times, this class It allows you to create non-blocking timers that perform operations with a time interval specified in milliseconds.

The TickTimer has several methods that can be used to manage a specific timer.

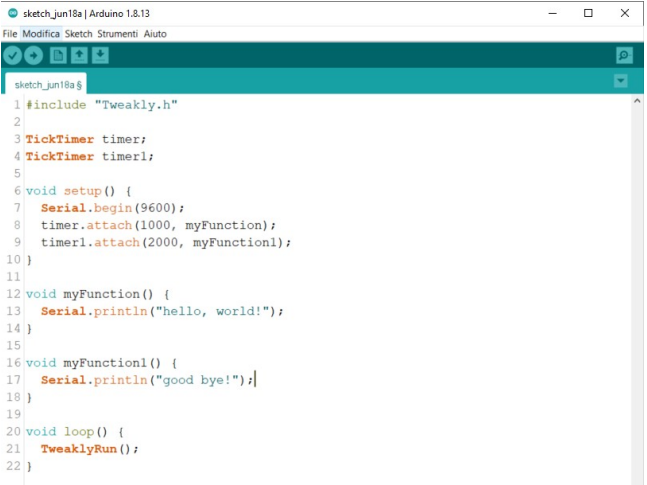
### attach(unsigned long delay, callback);

attach() allows you to initialize a timer, it can be invoked in any part of your code, although it is always preferable to call the configuration of a new timer in the setup() :

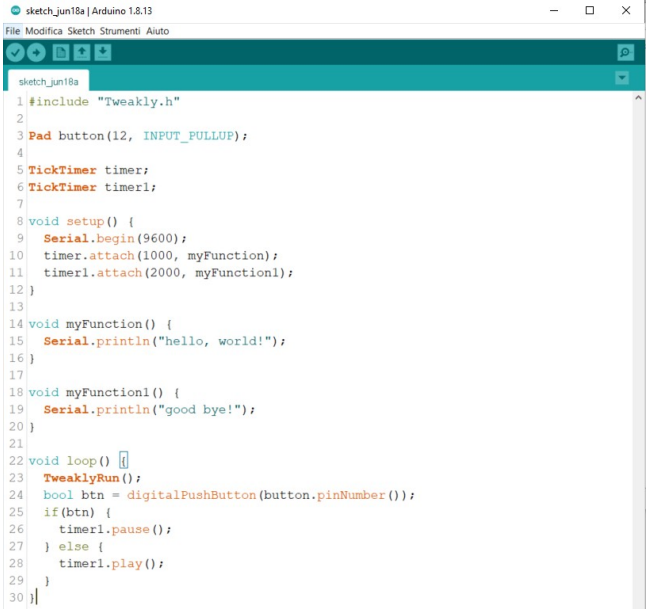


In the code in the example shown above, every second the text is printed on the serial warning "Hello, world!"

It is possible to create N timers in your code and each timer can have a different action time:



If we need to interrupt a TickTimer we can use the pause() function and use the play() function to restore its execution :



If the button is pressed, the timer1 is paused and does not execute the instructions that we assigned.

## 3.1 Timing | Assign statements via lambda to TickTimer

It is possible to assign statements to the timer using lambda and save lines of code, some

Examples below:



or:



## 4.0 doList | Explicit Scrolling Method

Tweakly includes a class called doList that allows you to create "shopping lists" that can contain instructions and which can be executed each time the list is called up using the next() method. The instructions within the list can be executed one at a time, at each time Invoking the next() method executes the next function.

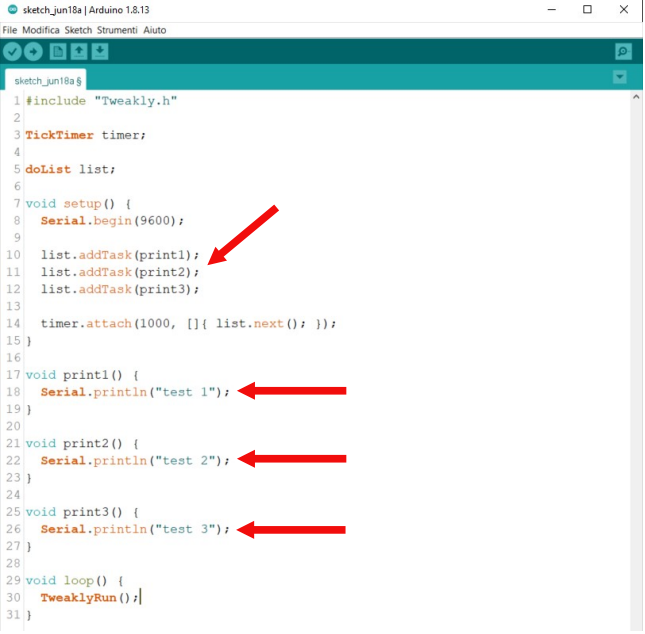
To experiment with this method we can write code like this:



To assign a statement to the list we use the addTask method and inside the parentheses round of the latter we indicate one or more instructions to be executed when the task is called up or in Alternatively, you can call up a function in our program.

In the case of the example, the texts "test 1", "test 2" and "test 3" will be written to the serial every second cyclically and one at a time, starting from the beginning each time the instructions in the list are Finished. This feature allows, for example, to change scenes on an LCD display and show Each time interval a different screen from the previous one.

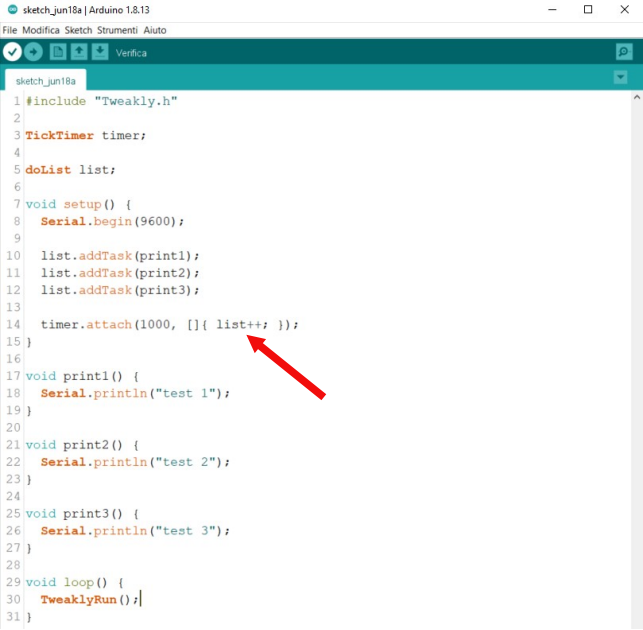
In the example above, lambda was used to assign an action to the list task. However, you can play the same program like this:



Of course, it is possible to create N lists in the code, handling numerous instruction changes within of the program.

## 4.1 doList | Implicit Scrolling Method

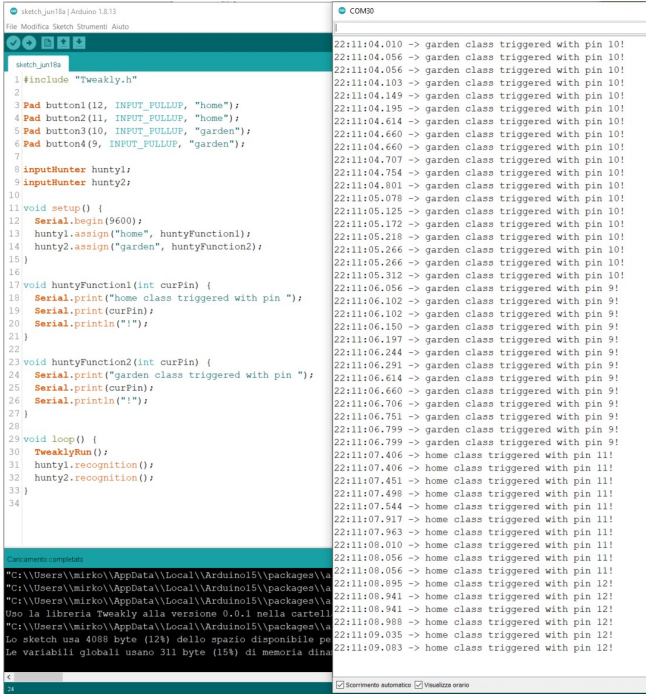
We can implicitly iterate through a list using the "++" operator as an alternative to the next() :



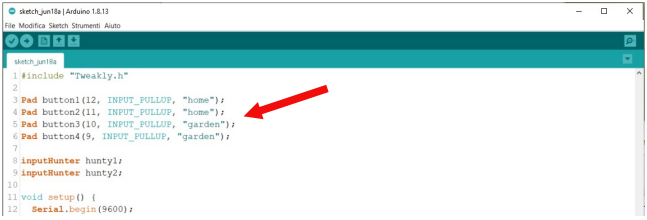
The implicit scrolling method of the doList class speeds up code writing and reduces keys, however, the two scrolling methods shown above are identical and do not have differences in operation.

## 5.0 inputHunter

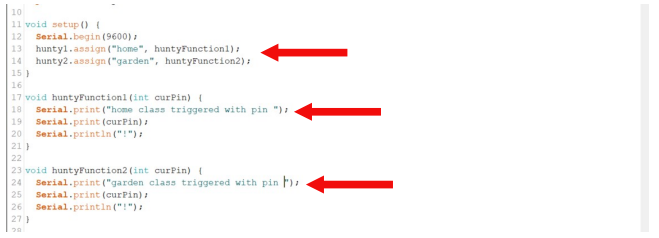
Among the most interesting features of Tweakly are the inputHunter, this function allows you to Create controls that run on a class of pins and intercept which pins are pressed. A The classic use of this feature is the ability to read as much as a state of a group of sensors within a room changes, thus allowing you to create applications dedicated to alarm systems and the like.



Once we've created the inputHunter object and given it a name, we need to make sure we have set a class to our pins.



In this example, we've given two buttons the "home" class and two other buttons the "home" class. "garden". Next, in setup() we assign to our two inputHunters the two pin classes and We set up a callback function that will be invoked every time we press a particular class :



As you can see, the huntyFunction1 and huntyFunction2 functions assigned respectively to input hunters hunty1 and hunty2 have an int argument called curPin, The latter is a mandatory topic that can be renamed as we wish and that will contain the number of the pin that switched its value.

You can check which pin has changed state and use a decision construct to Perform specific instructions based on the inputs or alternatively create a global function based on the class changed during the course of the programme.

In order for inputHunters to work properly, inside the loop you need to Invoke the recognition() method to make it feel the inputs you have previously set kept under control.



## 6.0 Rotary Encoder Management

The native encoder support built into Tweakly allows you to manage the encoders in a very precise way. rotary encoders and determine the direction of rotation of the encoders.



First you need to set the dt and clk pins as INPUT\_PULLUP :



then using the encoderAttach() function in the setup we indicate the DT pin, the CLK pin and the callback function that in this example we called encoderChange:



The curDir argument can be renamed according to our needs and can take on a value true and false based on the rotation of the encoder.

By acting on the rotary encoder we will see in the serial monitor the text right or left depending on the direction of rotation.

## 7.0 Usage Scenarios

Tweakly is a library that does not intend to replace libraries and methods of programming, but on the contrary, it aims at a formatting of the code based on callbacks, reducing typing code and offering a modern method of development on low-powered boards that do not allow the execution of an RTOS. However, Tweakly is not an RTOS but a kind of Framework that enhances the use of Wiring on all Arduino core-based boards.

Tweakly is perfect for the development of:

* Home automation applications
* Motor control
* Creation of keypads
* Simplification of school projects
* Programs that need to run on web applications alongside IoT platforms such as Blynk, Arduino IoT Cloud and the like, having methods that do not block clients that communicate with a server will not be interrupted.
* Interaction Design Applications

README.md

## Usage

Tweakly bases its operation on **millis()** and allows you to create non-blocking code for your board.

It is essential to know that to work correctly you need to use methods that do not block the execution of your code, the main libraries available for the sensors may have delays inside them, but after the tests done in the development phase we have could verify that most of the libraries are tweakly-compatible.

Tweakly takes pin status into account with a built-in pin manager, so use that to initialize your pins:

Pad led(13);

With Tweaky's pin manager you can read and write values ​​on all initialized pins and give them a name as you like, making your life a lot easier!

led.write(1);

led.read(); //1

If you want to create non-blocking code, rely on TickTimers.

TickTimer blinker;

Initialize it in **setup()** with:

blinker.attach(1000, [] {

led.toggle();

});

And your led will start blinking without blocking your code! Is not it fantastic ?

For more features, please see the [*Examples Folder*](https://github.com/filoconnesso/Tweakly/tree/main/examples)

## Tricks

Use a fast baud rate to not affect the performance of your code:

Serial.begin(115200);

Use stringAssembler to assemble strings and format a dataset together:

int sensor\_value;

char sensor\_name;

...

String message = stringAssembler("your sensor %s has read %d C° \n", sensor\_name, sensor\_value);

Use Echo Stream for print to file and Serial :

Echo printer;

Serial.begin(115200);

printer.attach(&Serial);

printer.print("Hello, Friend \n");

Tweakly's new "Clock" component was used for the clock, which allows you to create virtual clocks and manage them separately.

All the code is handled by various TickTimers that guarantee non-blocking execution of everything.

Code : <https://create.arduino.cc/editor/filoconnesso/0347b6f1-775b-4f70-bb65-e5fa197c5b39/preview>