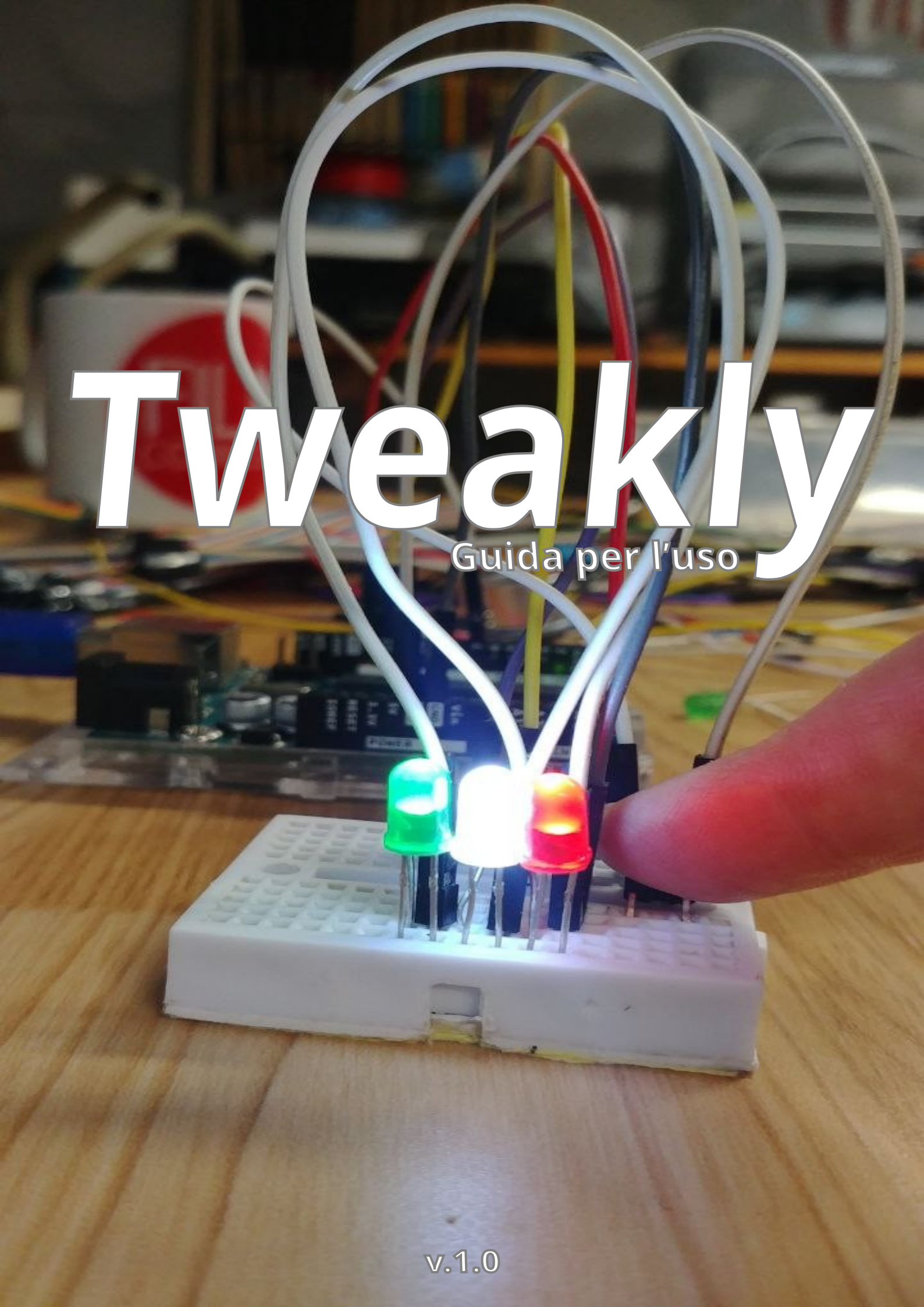


Tweakly

Guida per l'uso

v.1.0

Intro

Scrivere il primo programma con Tweakly

Gestione dei Pad

TickTimer

doList

inputHunter

Encoder rotativi

Scenari di utilizzo

Introduzione a Tweakly

Tweakly è una libreria per Arduino realizzata specificamente per semplificare la stesura di un codice e utilizzare meccanismi di programmazione avanzati su schede che non prevedono la possibilità di usare un RTOS.

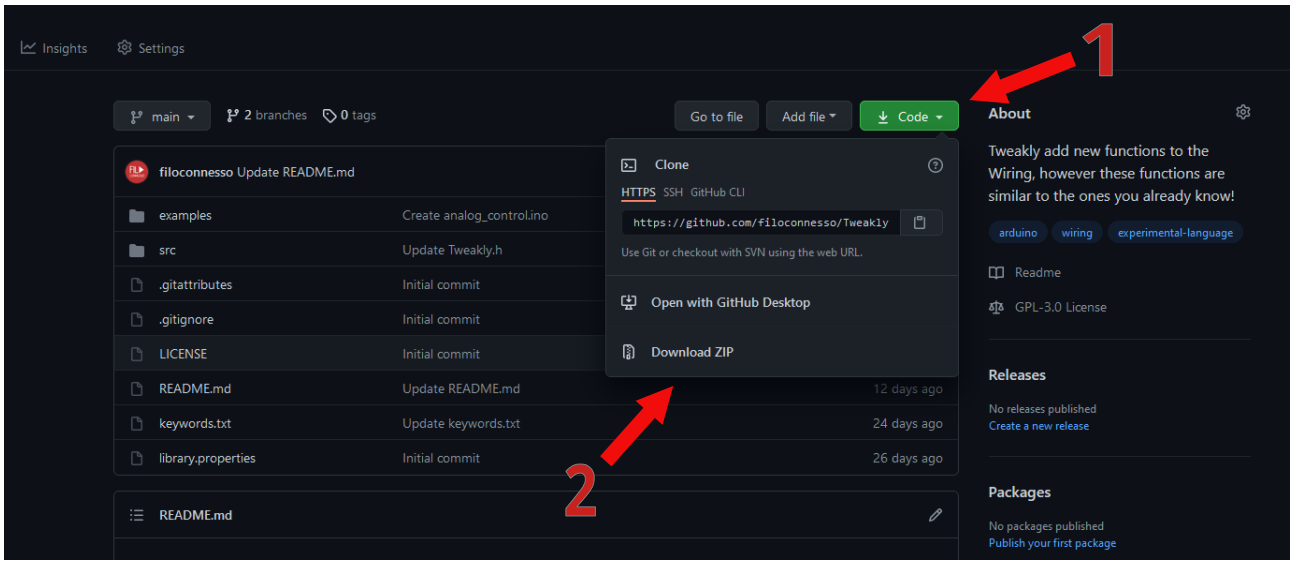
La libreria Tweakly è realizzata per essere il più possibile “Memory Friendly”, tanto che permette di essere eseguita, riducendo l’uso della memoria il più possibile, su micro-controllori con caratteristiche tecniche medio-basse, funzionando anche su Attiny85.

Il codice che fa funzionare Tweakly è scritto in C++ ed è eseguibile su hardware con Core software basato su Arduino.h. Tutte le funzionalità incluse in Tweakly non richiedono librerie secondarie e installazione di software aggiuntivo, ti basterà l’IDE di sviluppo di Arduino e sei pronto per iniziare!

Installazione di Tweakly

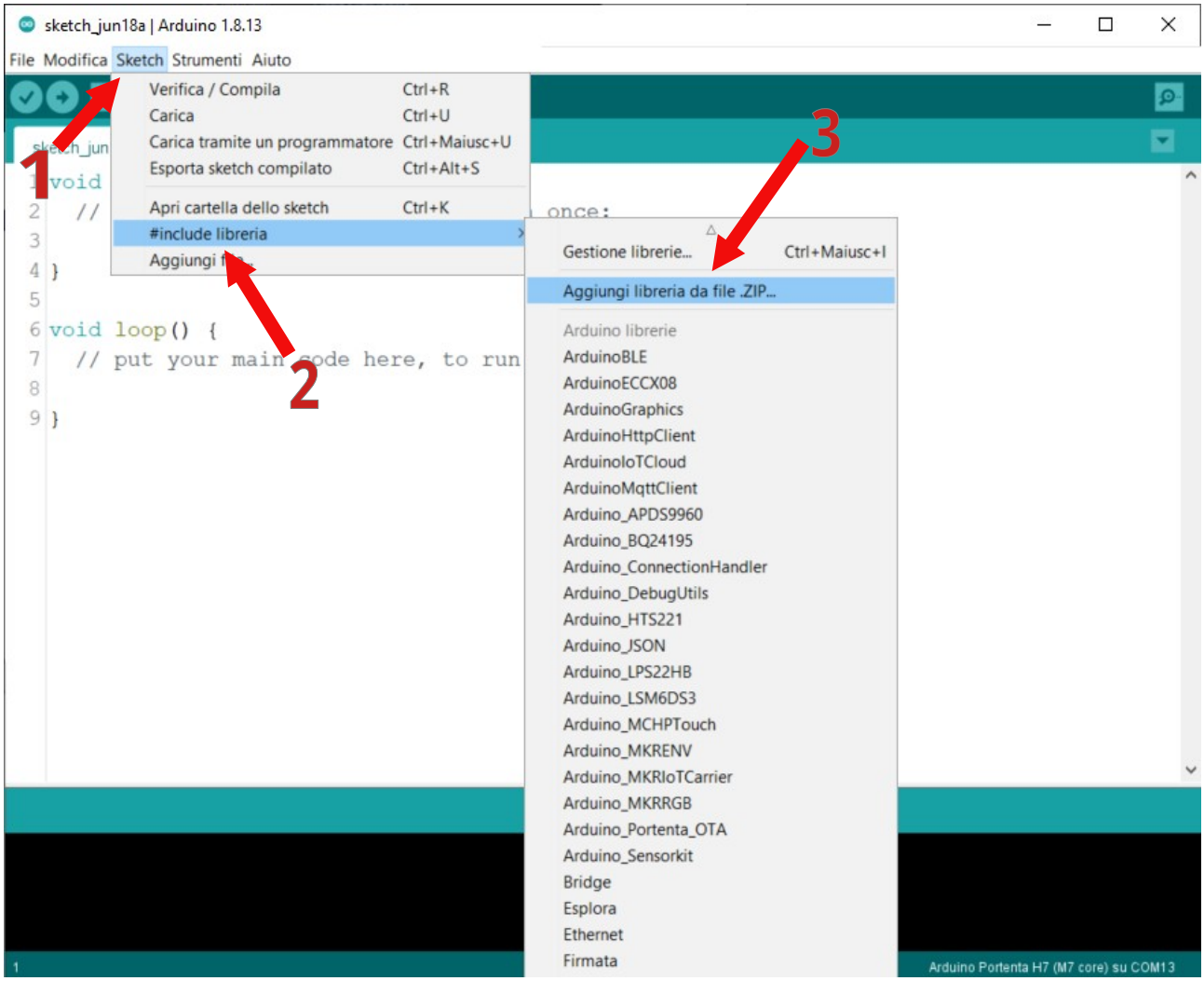
Puoi installare Tweakly nel tuo IDE di sviluppo scaricando il pacchetto zip da GitHub :

https://github.com/filoconnesso/Tweakly



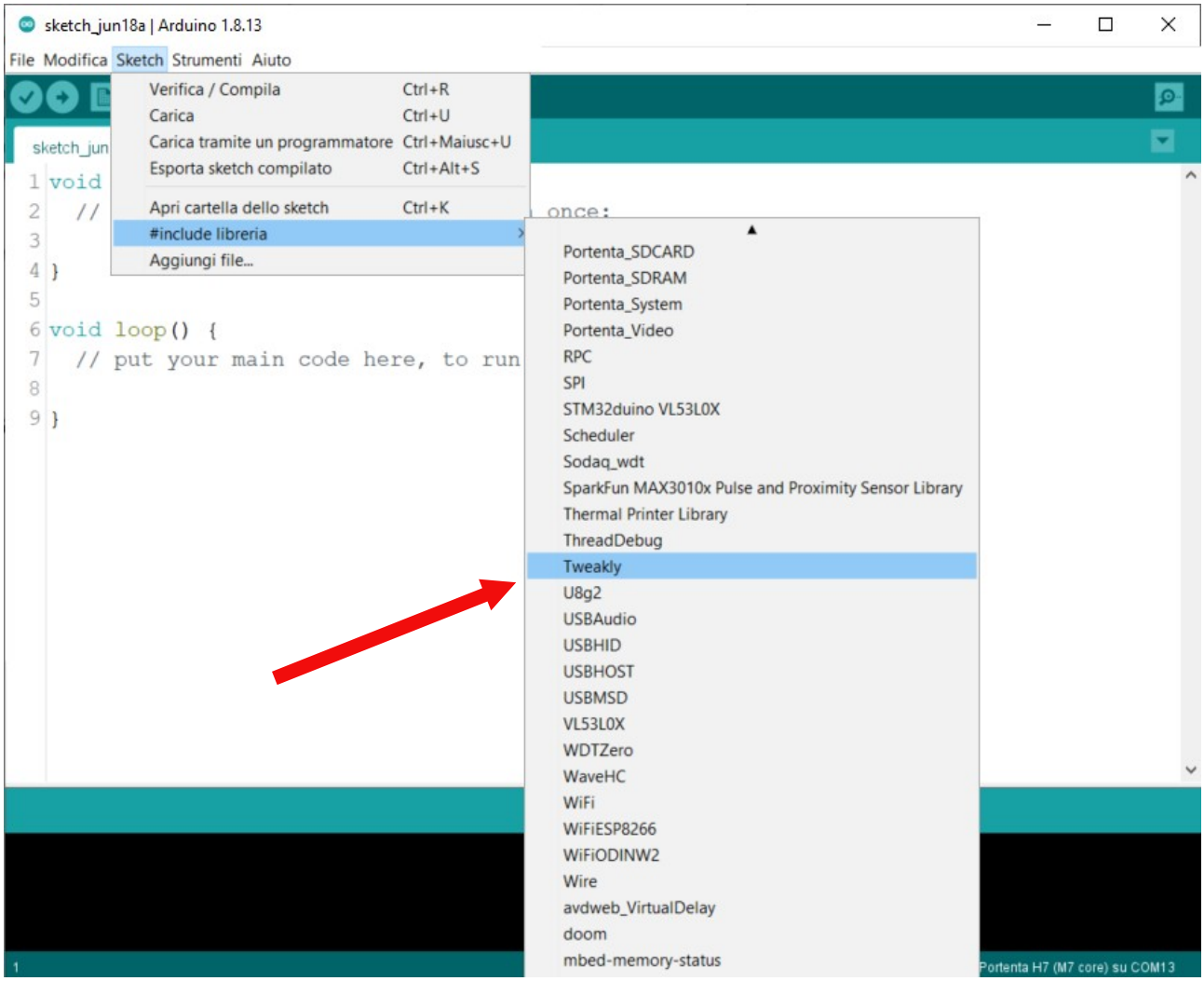
1. Fare click sul pulsante “Code”.
2. Scaricare il file .ZIP facendo click su “Download ZIP”.

Apri l’IDE di sviluppo di Arduino.



1. Aprire il menu “Sketch”.
2. Fare click su “#include libreria”.
3. Aggiungere il file .ZIP appena scaricato usando la funzione “Aggiungi libreria da file .ZIP”.

Se la libreria è stata aggiunta correttamente vedrete nella lista delle librerie la libreria “Tweakly” :

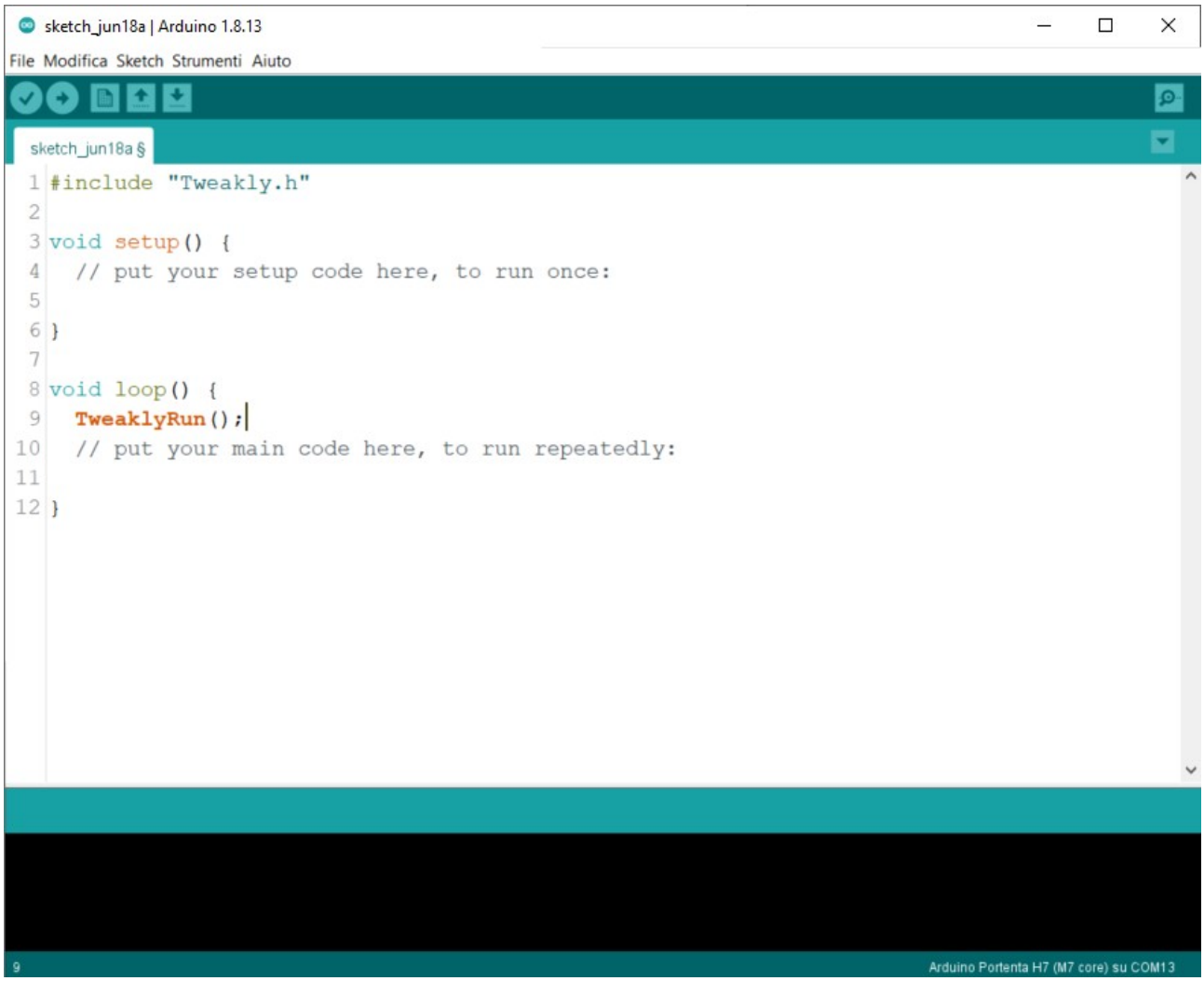


1.0 Scrivere il primo programma con Tweakly | Blink – Le basi

Per testare il corretto funzionamento della libreria possiamo scrivere il classico Blink per far lampeggiare un led, usando però tutti i metodi di Tweakly.
Prima di iniziare però abbiamo bisogno di comprendere due cose fondamentali :

1. In ogni tuo codice deve includere Tweakly, quindi sarà necessario scrivere a inizio codice l'istruzione **#include "Tweakly.h"**.
2. Per far si che Tweakly venga eseguita correttamente nella funzione **loop()** del tuo programma deve esserci sempre **TweaklyRun();**.

Di conseguenza la struttura base del tuo programma sarà la seguente :

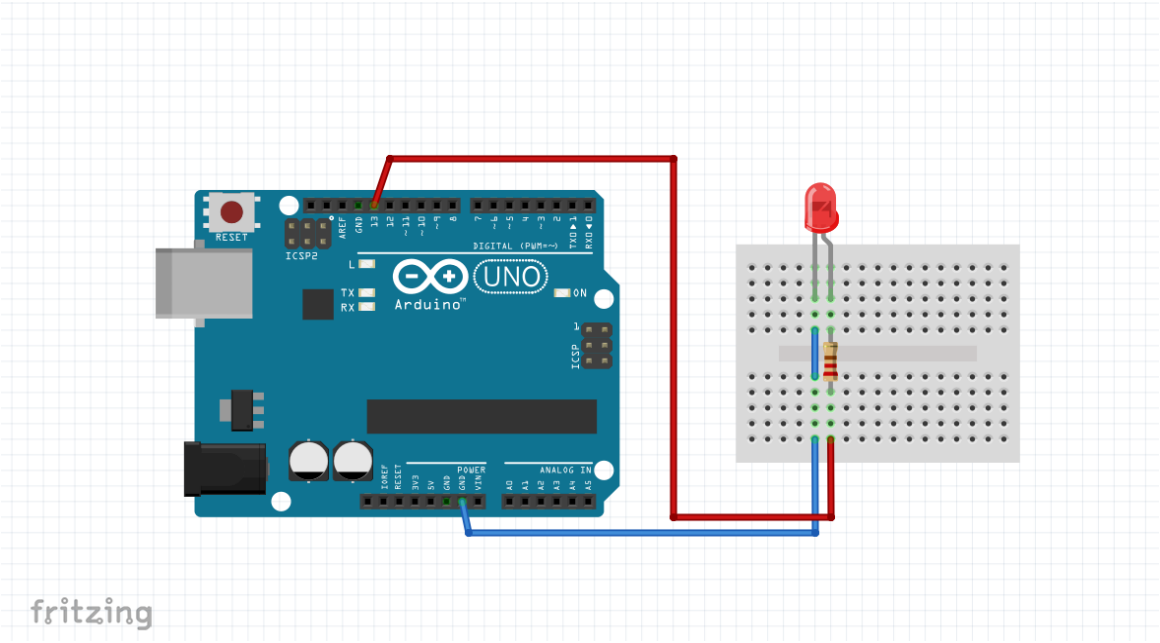


Successivamente per scrivere il nostro Blink abbiamo bisogno di inizializzare un pin come OUTPUT e di accendersi e spegnersi con un timer, il quale ha un periodo di attivazione che possiamo decidere in base alle nostre esigenze.

1.1 Scrivere il primo programma con Tweakly | Blink – Il PAD

Per controllare il nostro LED abbiamo bisogno di una resistenza, almeno di 220 Ohm per evitare che con un uso prolungato del LED questo si bruci.

Collega il tutto nel modo seguente, utilizzando il pin 13 di Arduino (ovviamente la scelta del pin è libera) :



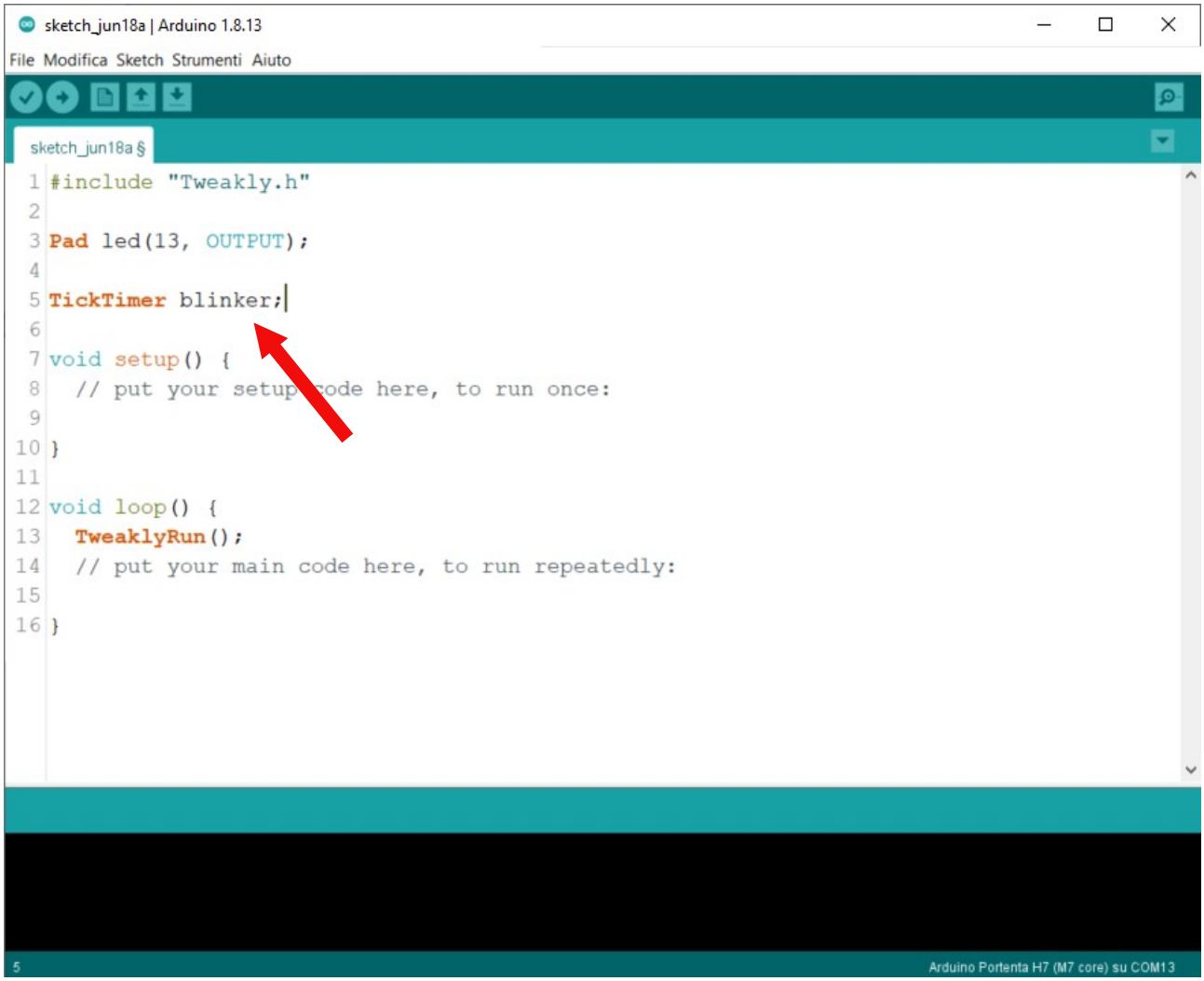
Nel nostro programma dobbiamo indicare che il led è collegato al pin 13 e che questo sia un'uscita, quindi settarlo come **OUTPUT**.

Per inizializzare un pin utilizziamo la classe **Pad** e creiamo un oggetto led, anche in questo caso la scelta del nome del pin è libera.

Il settaggio di un pin va eseguito prima del **setup()** :

1.2 Scrivere il primo programma con Tweakly | Blink – Il TickTimer

Ora dobbiamo usare la classe TickTimer per creare un timer, questo permetterà al nostro led di lampeggiare ogni qualvolta vogliamo. Per prima cosa creiamo un TickTimer in questo modo :

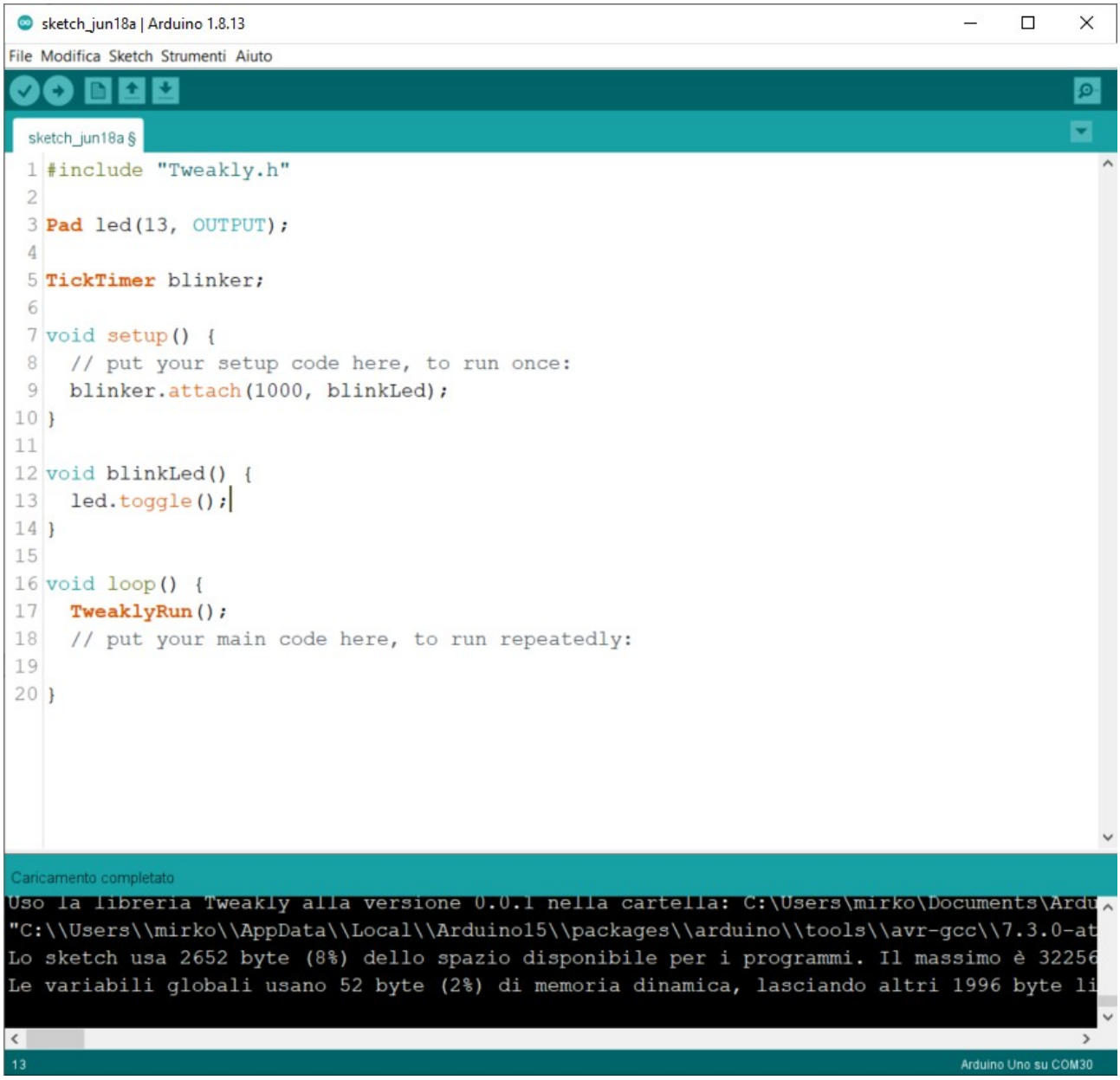


Nel **setup()** indichiamo al timer l'intervallo di tempo che deve passare da una chiamata e l'altra e cosa deve fare ogni volta che entra in azione, per fare tutto ciò utilizziamo il metodo **attach()** del **TickTimer** :

Per completare il nostro programma dobbiamo creare un'istruzione void, che chiameremo **blinkLed()**, e al suo interno daremo il comando di spegnimento e accensione del led utilizzando il metodo **toggle()** incluso nella classe **Pad**.

L'istruzione **blinkLed** è stata assegnata al timer e quindi, secondo il codice che abbiamo scritto, ogni **1000 millisecondi** questa viene richiamata e avendo al suo interno **led.toggle(),** il led collegato al **pin 13** inizierà a spegnersi e accendersi **ogni secondo** :

Puoi notare che in questo primo programma non abbiamo utilizzato per nulla il **loop(),** ovviamente tutto il funzionamento comunque gira dietro a **TweaklyRun();** che in questo preciso momento sta controllando lo stato del pin 13 e il timing sulla tua board. Puoi creare più timer nel tuo codice e controllare più led insieme, quest'ultimi lampeggeranno in modo diverso senza che l'uno intacchi il timer dell'altro, questo metodo è pressoché non bloccante e permette al **loop();** di girare indipendentemente dal timer settato.



NOTA BENE : I processi gestiti da Tweakly non sono paralleli, si basano sul controllo del tempo mediante **millis().** E' necessario comprendere che per creare un codice ottimizzato mediante Tweakly può essere utile sfruttare tutti i metodi offerti dalla libreria ed è consigliato evitare l'uso dei **delay()** all'interno del programma.

2.0 Gestione dei Pad | Modi di inizializzazione

Per gestire i Pad su Tweakly puoi utilizzare la classe Pad che permette di creare oggetti che ereditano funzioni per gestire i pin della tua board.

I Pad possono essere inizializzati a inizio codice e in base al tipo di pin possono essere impostati con le seguenti modalità :

- OUTPUT
- INPUT
- INPUT_PULLUP
- INPUT_PULLDOWN
- PWM_OUTPUT
- ANALOG_INPUT

OUTPUT

Il pin viene inizializzato come uscita : utile per collegare led, relay, driver motori, pin TRIGGER di un sensore ultrasuoni e tutto quello che prevede il pilotaggio di un componente esterno.

INPUT

Il pin viene inizializzato come ingresso : utile per collegare pulsanti con un resistore esterno, sensori pir e tutti i componenti che possono inviare alla tua board un segnale digitale tramite un pin.

INPUT_PULLUP

Il pin viene inizializzato come ingresso utilizzando una resistenza interna collegata verso 5V o 3.3v (per le board a logica a 3.3v).

INPUT_PULLDOWN

Il pin viene inizializzato come ingresso utilizzando una resistenza interna collegata verso GND (alcune board lo supportano).

PWM_OUTPUT

Modo di attivazione applicabile a tutti i pin PWM, questo permetterà di sfruttare alcune funzionalità aggiuntive rispetto ai pin digitali.

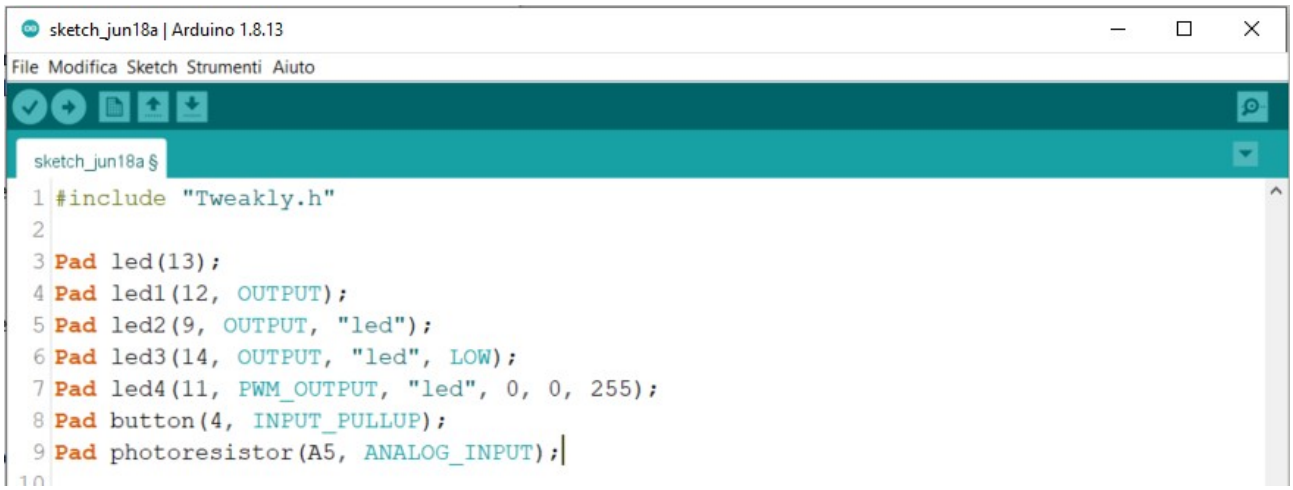
ANALOG_INPUT

Modo di attivazione dedicato ai pin analogici della tua board.

2.1 Gestione dei Pad | Inizializzazione

Per inizializzare un pad è possibile richiamare la classe Pad e dare un nome all'oggetto, è necessario comprendere che ogni oggetto Pad deve avere un nome diverso da tutti gli altri oggetti inizializzati nel tuo codice.

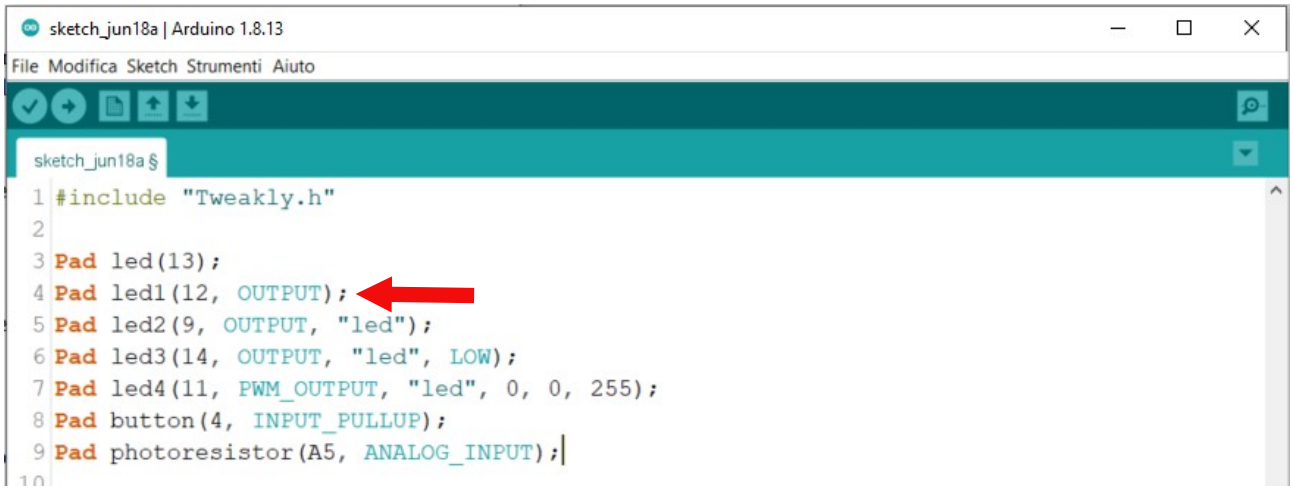
Un esempio di inizializzazione di uno o più Pad può essere il seguente :



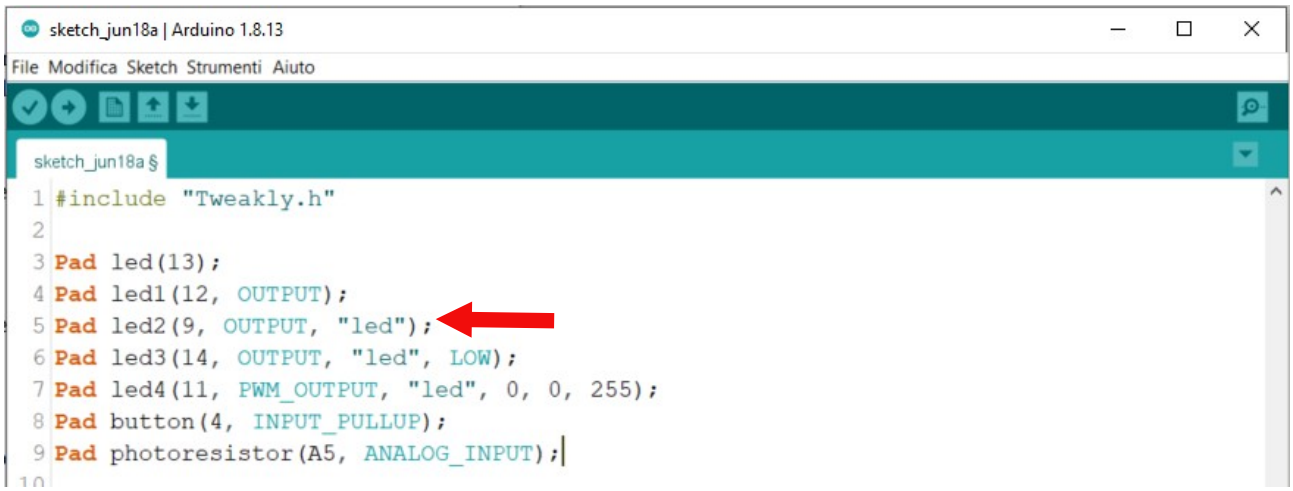
In base alle nostre esigenze possiamo inizializzare N oggetti Pad per avere accesso in modo diverso a ciascun pin della nostra board.

Se indicassimo solo il numero del pin, questo viene inizializzato in automatico come OUTPUT e verrà spento a inizio programma.

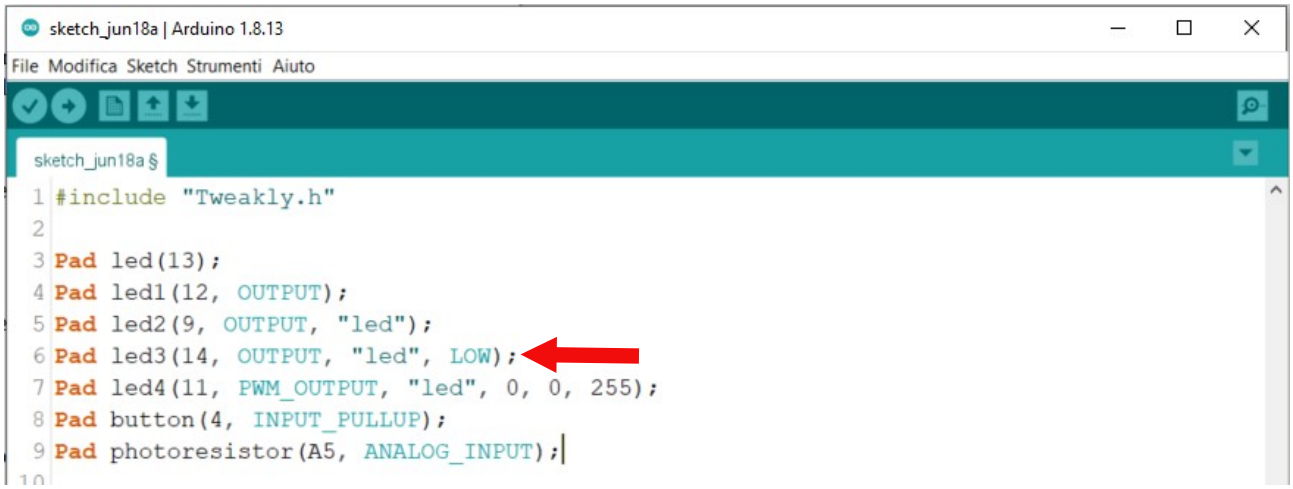
In caso volessimo avere una modalità diversa dovremmo indicare, oltre al numero del pin, anche il suo modo d'uso :



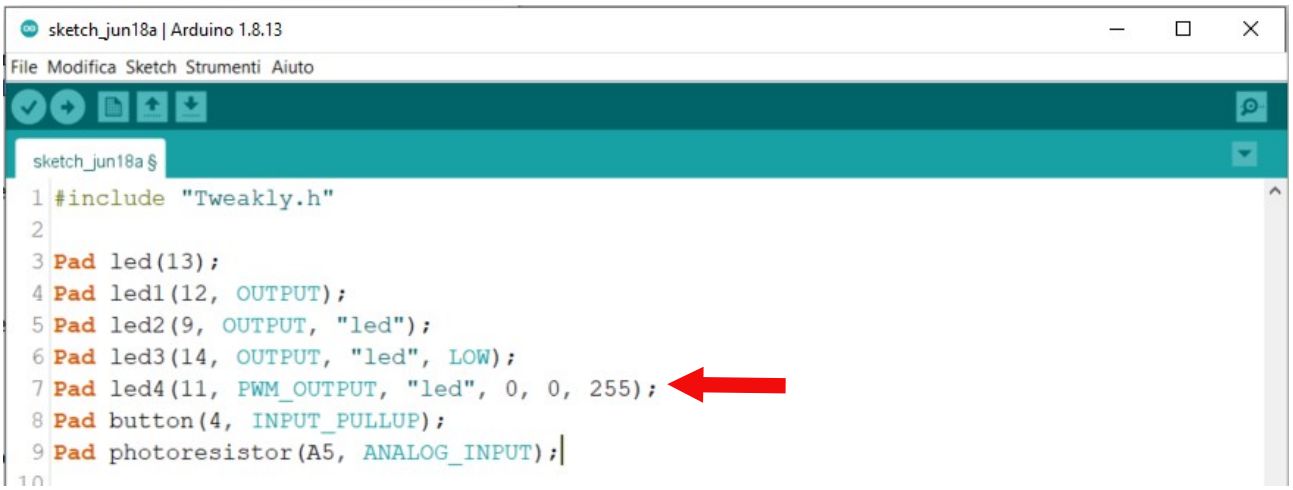
Il terzo paramentro che può essere utile per organizzare i pin è quello della classe. Una classe di pin può essere utile per permetterci di impartire azioni a un gruppo specifico di pin, magari accenderli o spegnerli tutti in un colpo con le funzioni di azione multipla che vedremo più avanti :



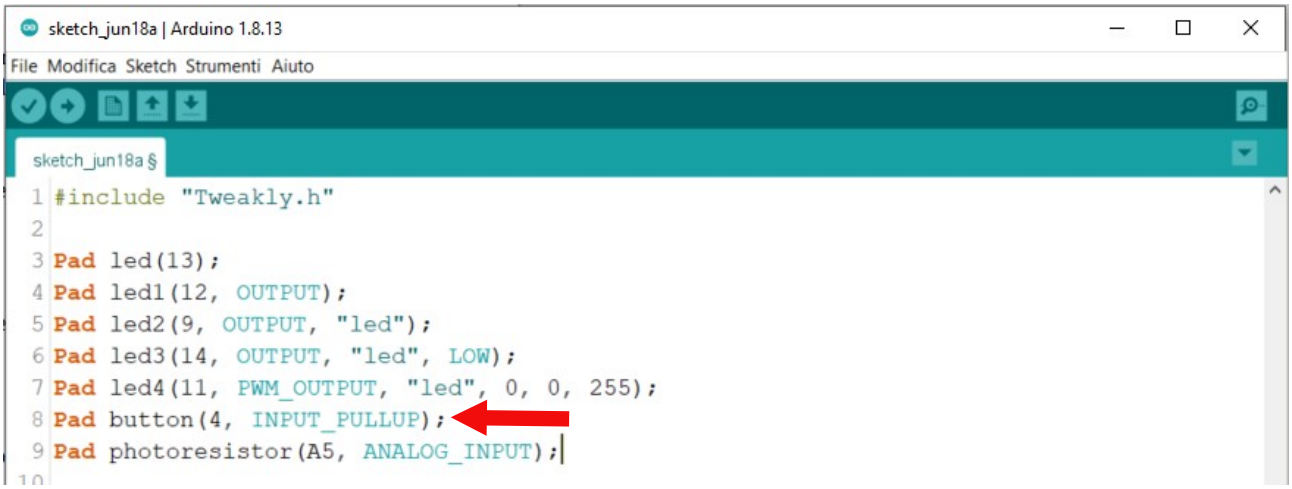
Se ci fosse la necessità, potremmo decidere di accendere o spegnere il pin (settando il quarto parametro **LOW** o **HIGH**) all'inizializzazione del pin stesso :



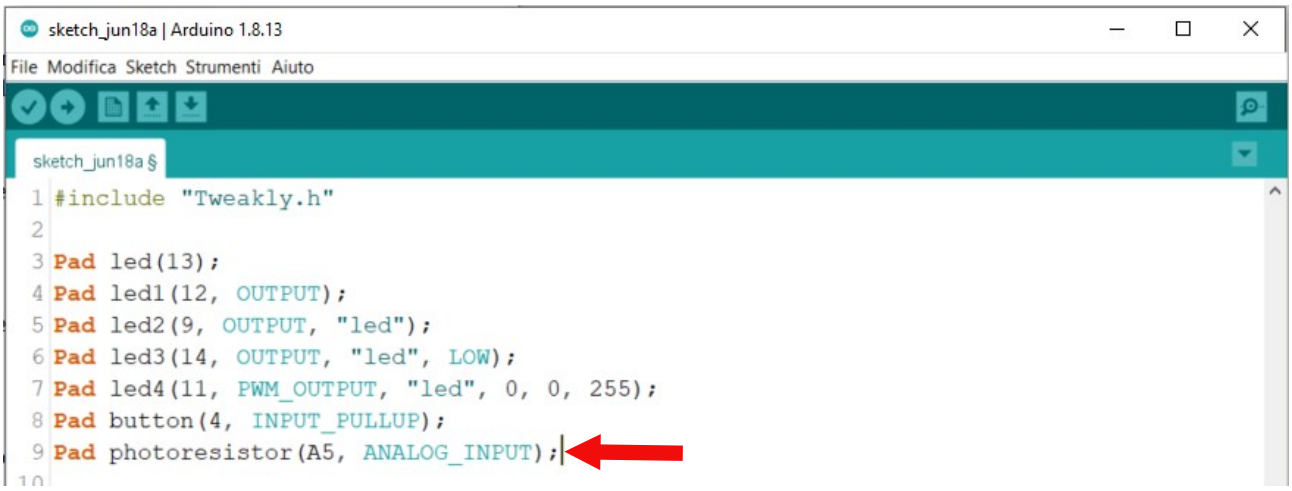
Se volessimo utilizzare a pieno un pin **PWM,** è necessario oltre che settare il modo d'uso in **PWM_OUTPUT,** andrà indicato il valore iniziale del pin in base al suo futuro utilizzo e il valore minimo e massimo che vogliamo che il nostro pin assuma , di default da 0 a 255 :



Se sul pin fosse collegato un pulsante senza resistenza dobbiamo utilizzare il modo d'uso **INPUT_PULLUP** :



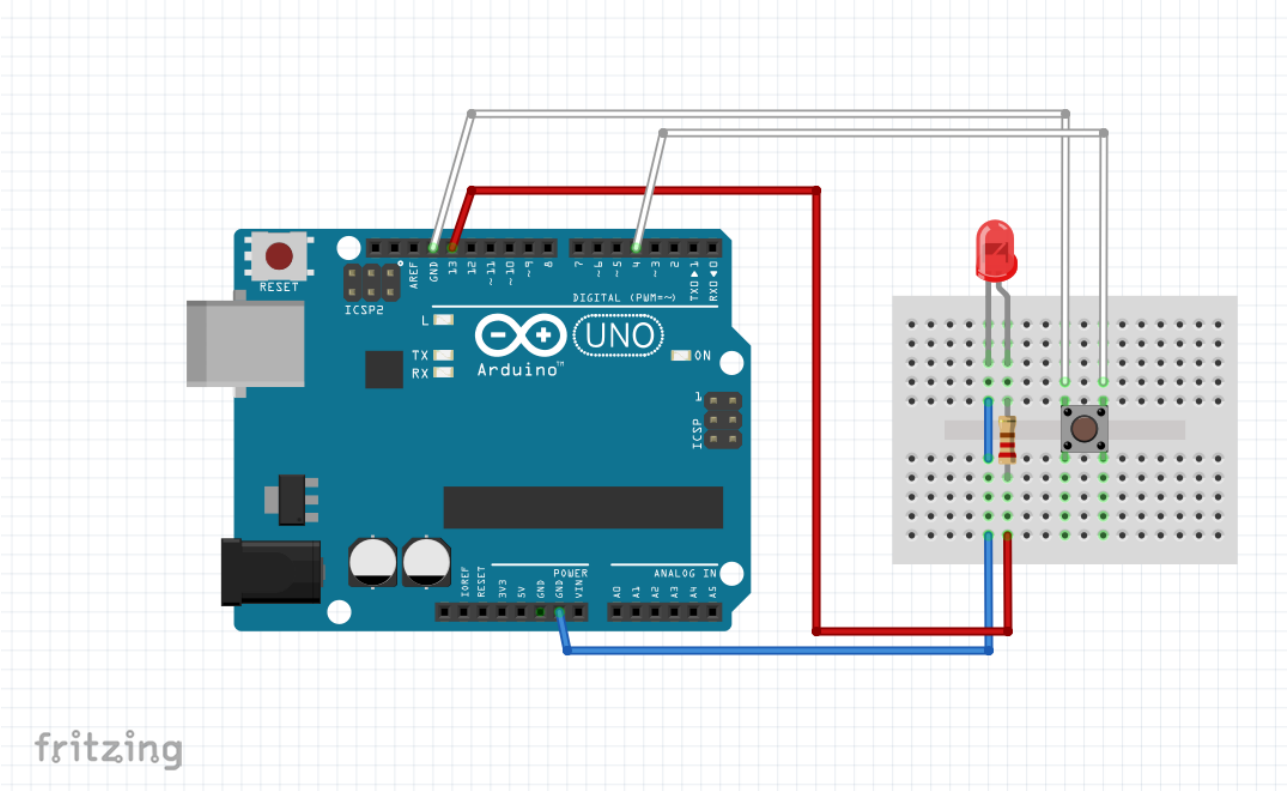
Per avere pieno accesso ai pin analogici con le funzionalità avanzate di Tweakly, infine, c'è il modo d'uso **ANALOG_INPUT** :



In definitiva possiamo notare che il modo di dichiarare e inizializzare i pin è il medesimo su tutte le modalità disponibili, questo permette di utilizzare su tutti i pin la stessa logica di controllo e di conseguenza avere sempre il medesimo metodo di gestione di ogni singolo pin.

2.2 Gestione dei Pad | read e write (metodi espliciti)

Possiamo leggere e scrivere un valore su un pin tramite i metodi **read()** e **write(),** questi metodi fanno parte della classe Pad e sono definiti metodi espliciti, possono essere utilizzati come l’esempio qui sotto :



Nel codice precedente il led sul pin 13 si accenderà ogni volta che il pulsante sul pin 4 verrà premuto. Utilizzando **INPUT_PULLUP** il valore che abbiamo sul pin è sempre 1 e va a 0 solamente se il pulsante viene premuto e di conseguenza il pin 4 viene collegato a GND, per questo motivo è stato usato il valore negato della variabile btn all'interno del metodo write con **!btn**.

2.3 Gestione dei Pad | on e off (metodi espliciti)

Se gestisci dei pin digitali in alternativa al **write()** puoi usare altri due metodi espliciti e cioè **on()** e **off().** Questi due metodi possono essere usati per esempio come mostrato qui sotto :

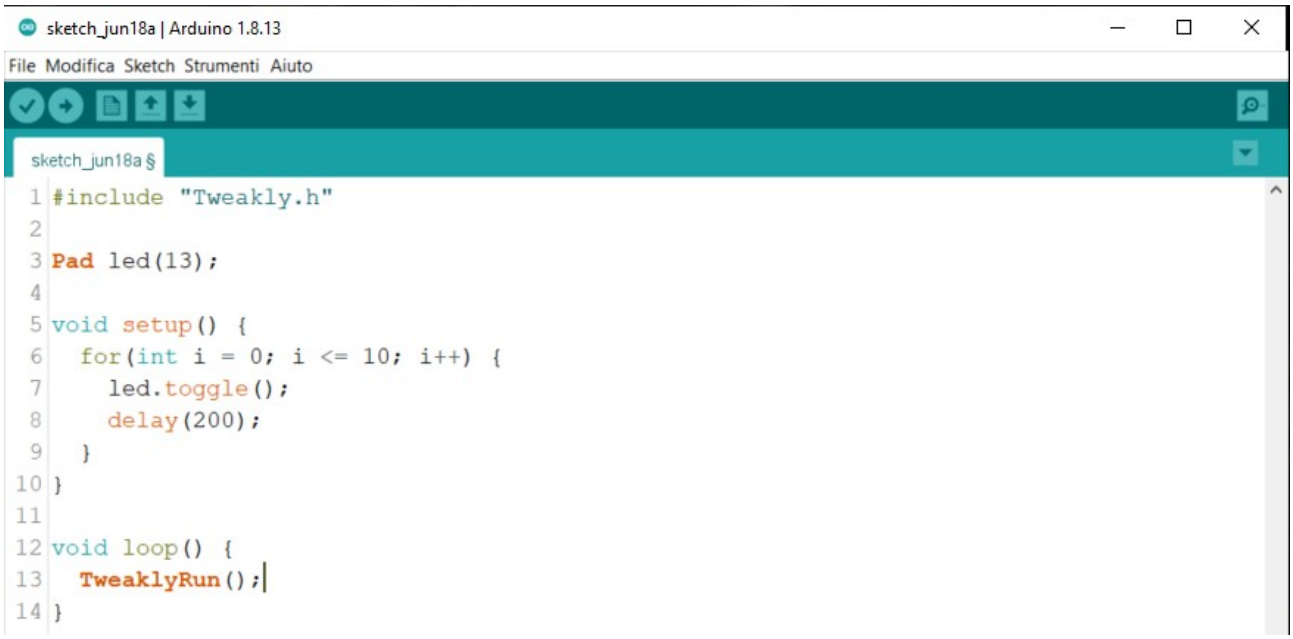
2.4 Gestione dei Pad | write (metodo implicito)

Puoi assegnare un valore a un pin in modo implicito utilizzando l'operatore "=". Di seguito c'è un esempio per capire l'utilizzo del metodo :

Questo metodo è valido anche per i pin PWM, sarà possibile quindi assegnare valori interi al nostro Pad con l'operatore "=".

2.5 Gestione dei Pad | toggle

Il metodo **toggle()** permette di commutare in modo del tutto automatico lo stato di un pin digitale che da 0 passa a 1 e viceversa. Il metodo può essere utilizzato per realizzare un semplice blink o applicazioni dove è necessaria la commutazione dello stato di un pin :



Nell’esempio qui sopra il led lampeggia 10 volte all’avvio della board e successivamente entra nel loop.

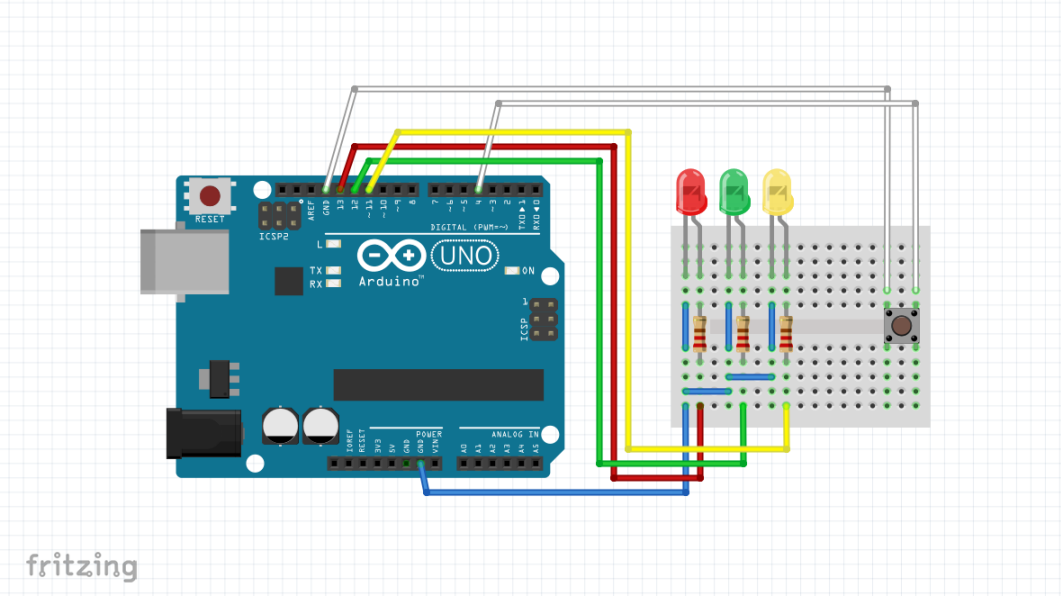
2.6 Gestione dei Pad | pinNumber

E’ possibile restituire il numero del pin impostato nell’inizializzazione con **pinNumber().** Questo metodo permette di gestire il pin con funzioni secondarie dove è necessario indicare il numero del pin :

2.7 Gestione dei Pad | lock e unlock

Tweakly permette di utilizzare dei metodi di controllo su gruppi di pin, questi metodi potrebbero sovrascrivere le funzioni di scrittura dei valori sui pin durante l'esecuzione del codice, producendo comportamenti indesiderati del tuo programma. I metodi **lock()** e **unlock()** bloccano l'azione delle seguenti funzioni **digitalWriteAll, digitalWriteClass, digitalToggleAll,analogWriteAll, analogWriteClass e analogWriteProgressive.**

Un possibile utilizzo di questa funzione potrebbe essere quella di bloccare l'esecuzione delle azioni su un pin gestito da un pulsante :



Per realizzare questo programma ti basta scrivere il codice seguente :

Mentre i due led collegati sui pin 12 e 11 continuano a lampeggiare grazie al **TickTimer** settato a 1 secondo di intervallo, il led rosso collegato sul pin 13 è gestibile solamente dal pulsante.

sketch_jun18a | Arduino 1.8.13

File Modifica Sketch Strumenti Aiuto

sketch_jun18a §

```
#include "Tweakly.h"

Pad red(13);
Pad green(12);
Pad yellow(11);
Pad button(4, INPUT_PULLUP);

TickTimer blinker;

void setup() {
  blinker.attach(1000, []{ digitalToggleAll(); });
  red.lock();
}

void loop() {
  TweaklyRun();
  bool btn = button.read();
  red.write(!btn);
}
```

È possibile sbloccare il pin utilizzando il metodo **unlock(),** il pin sarà sempre gestibile dal pulsante ma in questo caso **red.write(!btn)** e il **digitalToggleAll()** entreranno in conflitto, creando un effetto di **flicker** sul led del pin 13 poiché entrambi le funzioni accendendo e spegnendo il led.

Puoi eseguire un lock e un unlock dei pin in modo veloce su tutti i Pad inizializzati come **OUTPUT** e **PWM_OUTPUT** con le seguenti funzioni :

- digitalLockAll();
- digitalUnlockAll();
- analogLockAll();
- analogUnlockAll();

2.8 Gestione dei Pad | Funzioni di azione multipla “Wiring Style”

E’ possibile eseguire azioni multiple sui pin impostati come **OUTPUT** o **PWM_OUTPUT** con le funzioni di azione multipla “Wiring Style”. Le funzioni di questo pacchetto di features di Tweakly sono molto simili alle funzionalità base del Wiring ma eseguono azioni su tutti i pin o su una classe di pin :

digitalLockAll();

Protegge tutti i pin OUTPUT da alcune funzioni di azione multipla.

digitalUnlockAll();

Elimina la protezione da tutti i pin OUTPUT per utilizzarli con alcune funzioni di azione multipla.

analogLockAll();

Protegge tutti i pin PWM_OUTPUT da alcune funzioni di azione multipla.

analogUnlockAll();

Elimina la protezione da tutti i pin PWM_OUTPUT per utilizzarli con alcune funzioni di azione multipla.

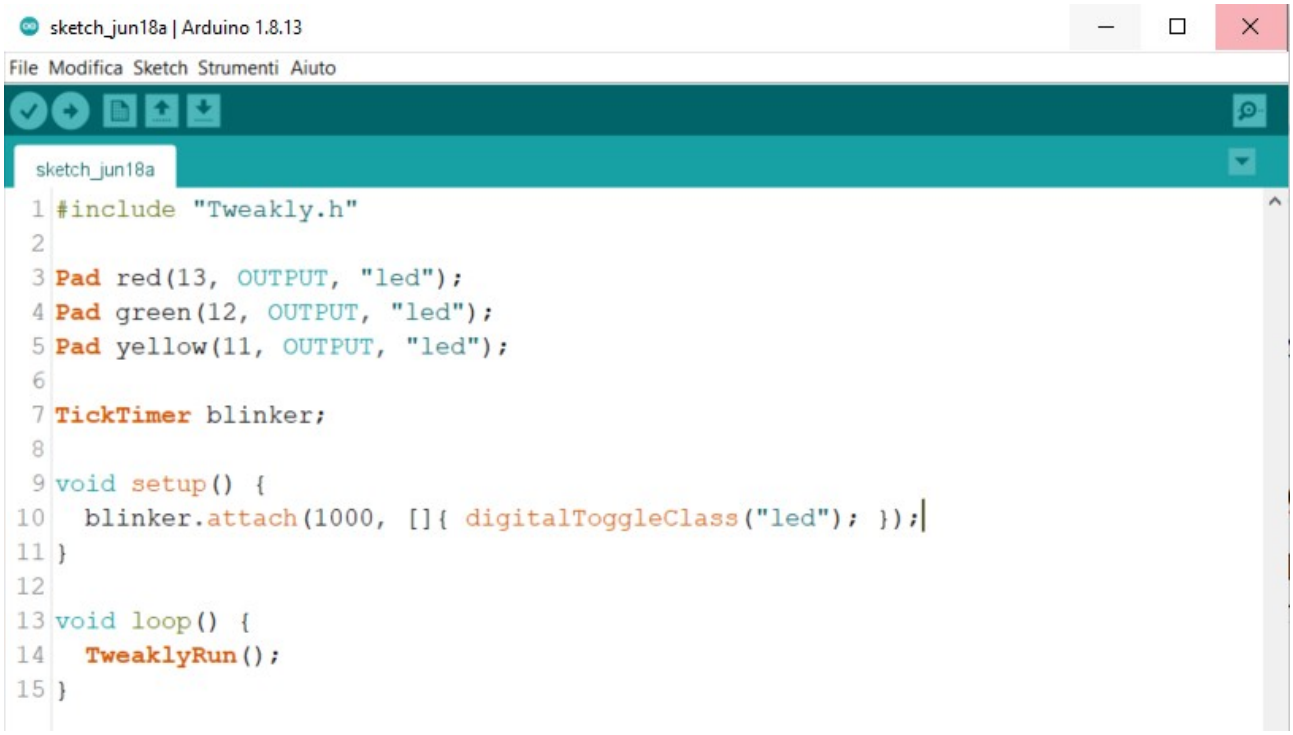
digitalToggleAll();

Commuta tutti gli stati dei pin inizializzati come OUTPUT, i pin che sono settati a 0 commutano a 1 e viceversa.

digitalToggleClass(const char* class);

Indicando la classe dei pin all’interno delle parentesi, commuta lo stato di tutti i pin contenuti in quella classe, tutti i pin settati a 0 commutano a 1 e viceversa.

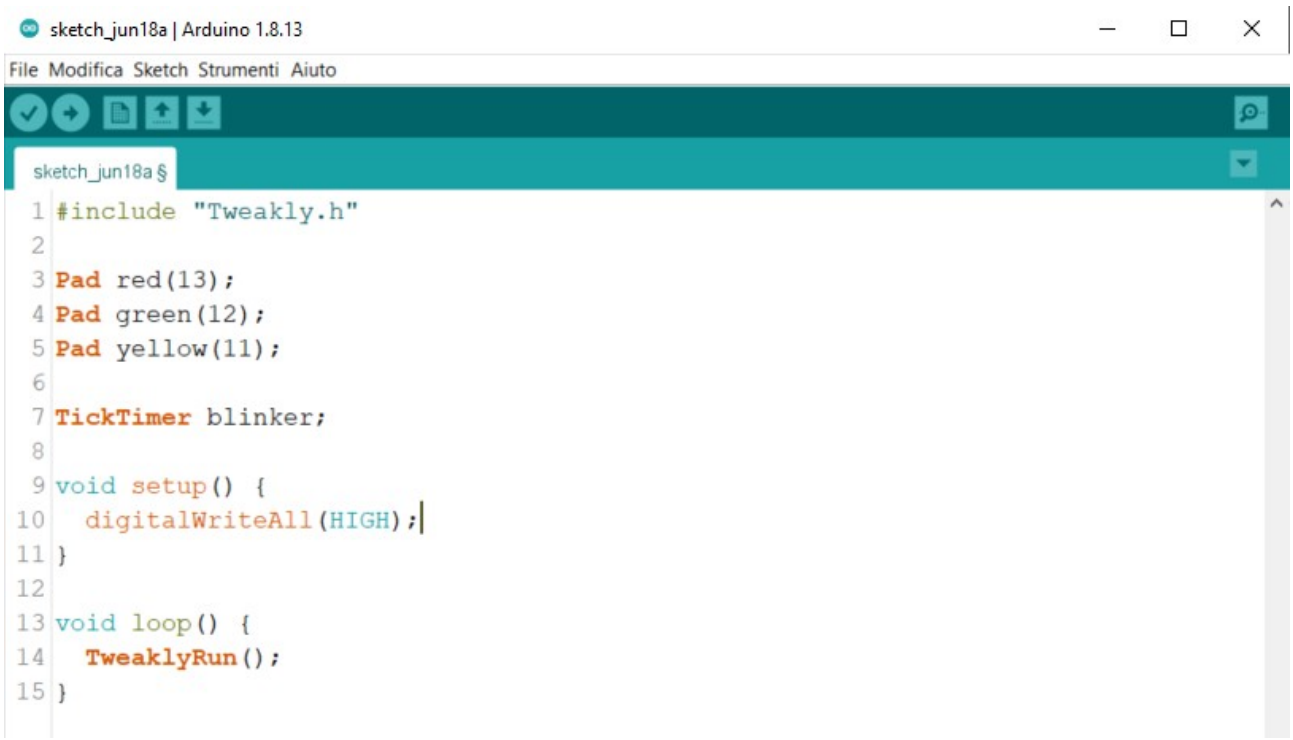
Un esempio di **digitalToggleClass()** è il seguente :



Tutti i led presenti nella classe “led” verranno spenti e accesi con un intervallo di tempo di 1 secondo.

digitalWriteAll(uint8_t new_value);

digitalWriteAll() permette di settare un valore (0 o 1) su tutti i pin inizializzati come OUTPUT :



digitalWriteClass(const char* class, uint8_t new_value);

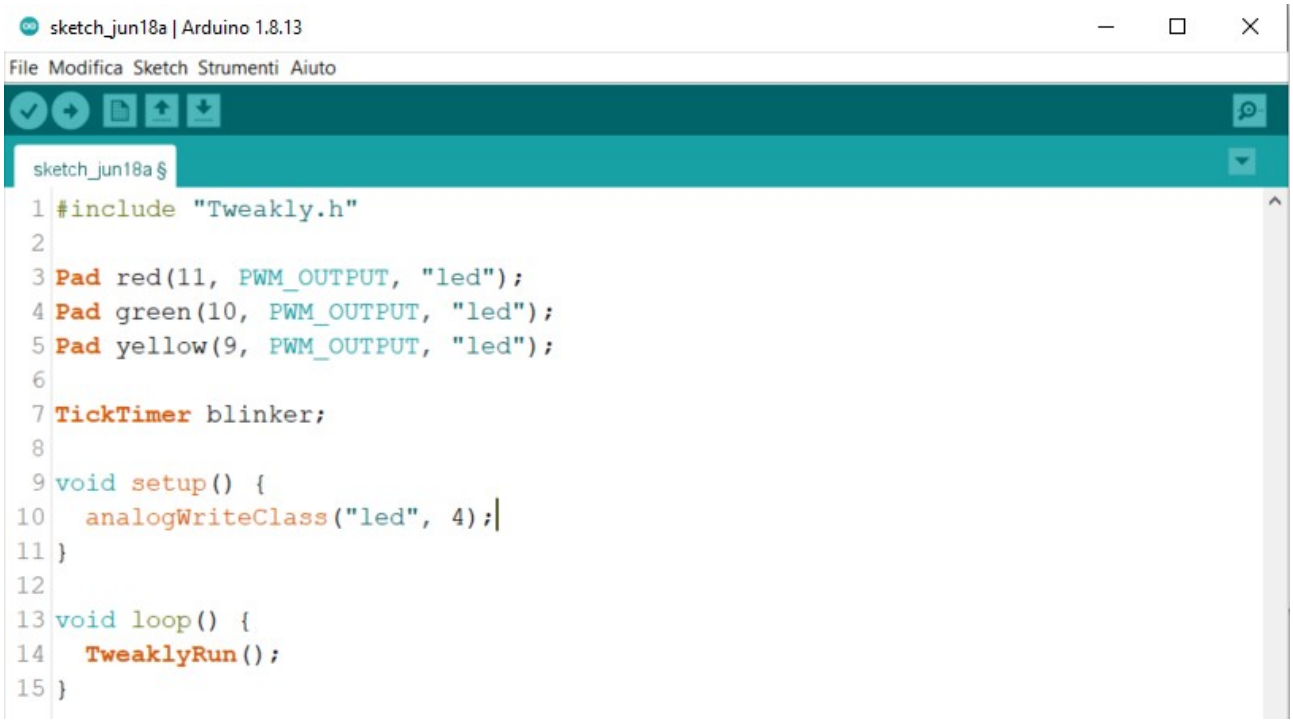
DigitalWriteClass() permette di settare un valore (0 o 1) su tutti i pin appartenenti a una specifica classe e inizializzati come OUTPUT :

analogWriteAll(unsigned int new_value);

analogWriteAll() permette di settare un valore intero su tutti i pin inizializzati come PWM_OUTPUT :

analogWriteClass(const char* class, unsigned int new_value);

AnalogWriteClass() permette di settare un valore intero su tutti i pin appartenenti a una classe specifica di pin inizializzati come PWM_OUTPUT :



analogWriteProgressive(uint8_t pin_number, unsigned long delay, uint8_t mode);

analogWriteProgressive() è una funzionalità molto utile per settare in modo progressivo un valore min a un valore max, indicando un delay non bloccante. Questa funzione permette di creare effetti FADE con i led o diminuire e aumentare la velocità di un motore in modo graduale.

In base al metodo di commutazione del valore il tempo per passare dal valore min al valore max varia. I metodi di commutazione sono i seguenti :

TO_HIGH : Da un valore minimo a un valore massimo con un intervallo di tempo tra un cambio di valore e l'altro. Un delay settato a 1000 millisecondi permette un passaggio da un valore 0 a un valore 255 in 255 secondi.

TO_LOW : Da un valore massimo a un valore minimo con un intervallo di tempo tra un cambio di valore e l'altro. Un delay settato a 1000 millisecondi permette un passaggio da un valore 255 a un valore 0 in 255 secondi.

TO_PULSE : Da un valore massimo a un valore minimo e viceversa con un intervallo di tempo tra un cambio di valore e l'altro, per sempre. Un delay settato a 1000 millisecondi permette un pasaggio da un valore minimo a un valore massimo e viceversa in 255 secondi, per sempre.

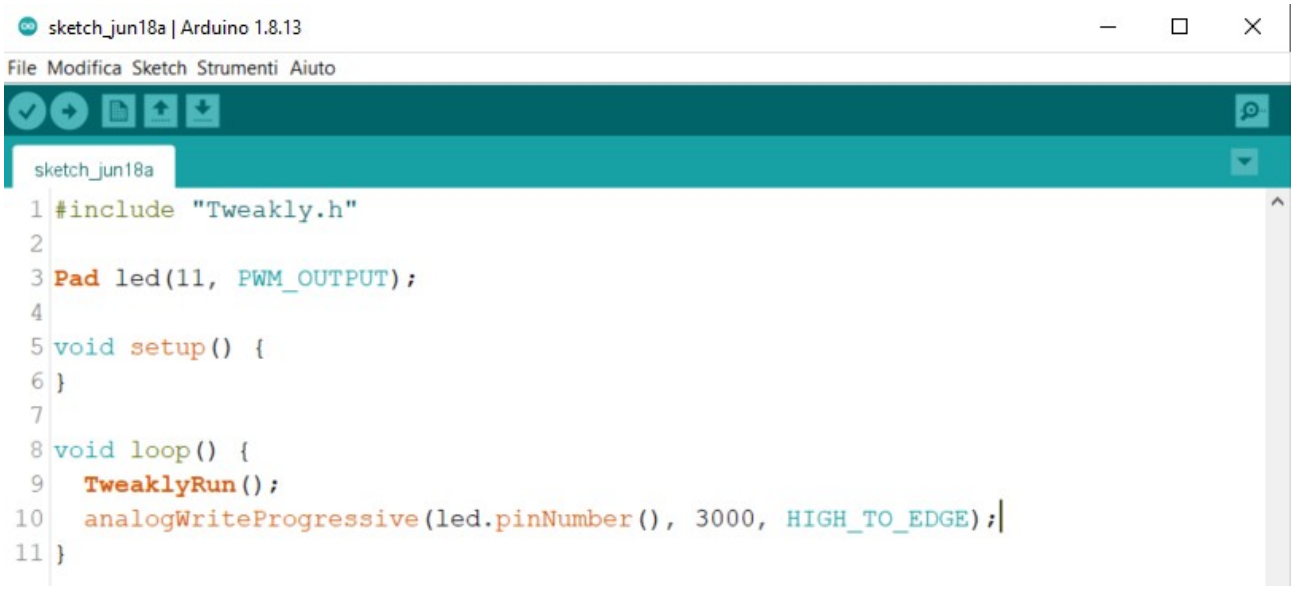
HIGH_TO_EDGE : Da un valore massimo e un valore minimo in un tempo di delay indicato. Un delay settato a 1000 millisecondi permette un passaggio da un valore 255 a un valore 0 in 1 secondo.

LOW_TO_EDGE : Da un valore minimo a un valore massimo in un tempo di delay indicato. Un delay settato a 1000 millisecondi permette un passaggio da un valore 0 a un valore 255 in 1 secondo.

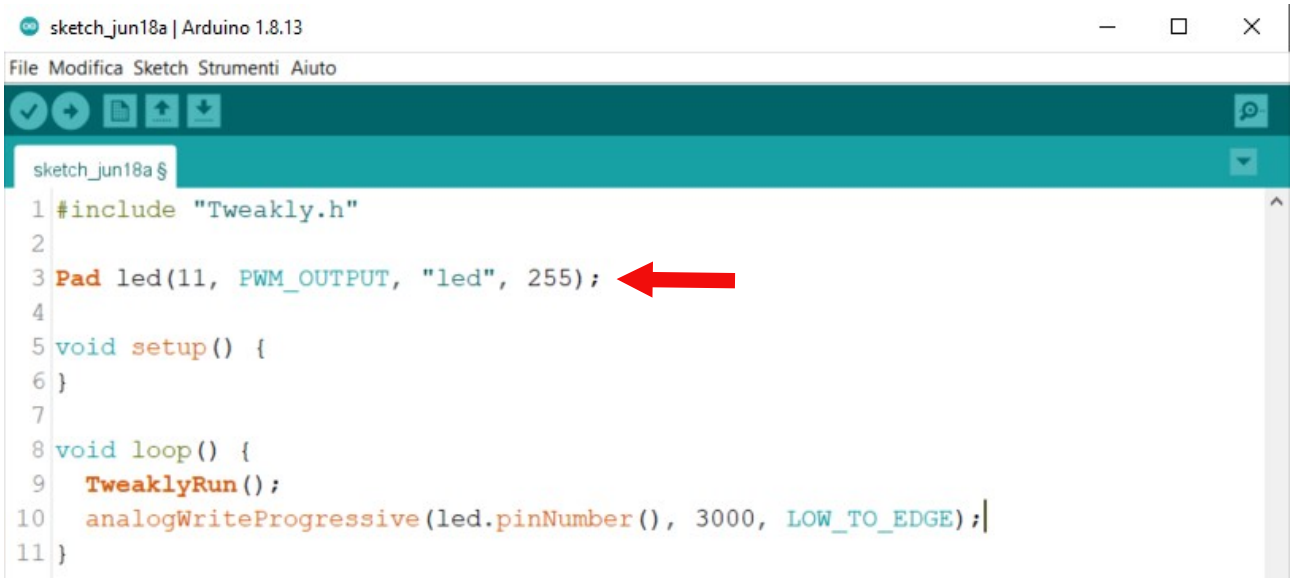
PULSE_TO_EDGE : Da un valore minimo a un valore massimo e viceversa per sempre. Un delay settato a 1000 millisecondi permette un passaggio da un valore minimo a un valore massimo e viceversa in 1 secondo.

Un esempio per la realizzazione dell'effetto FADE applicato a un led può essere eseguito con il codice seguente :

Se volessimo accendere pienamente il led in 3 secondi possiamo usare il metodo **HIGH_TO_EDGE** che definisce il fronte di salita del valore PWM :

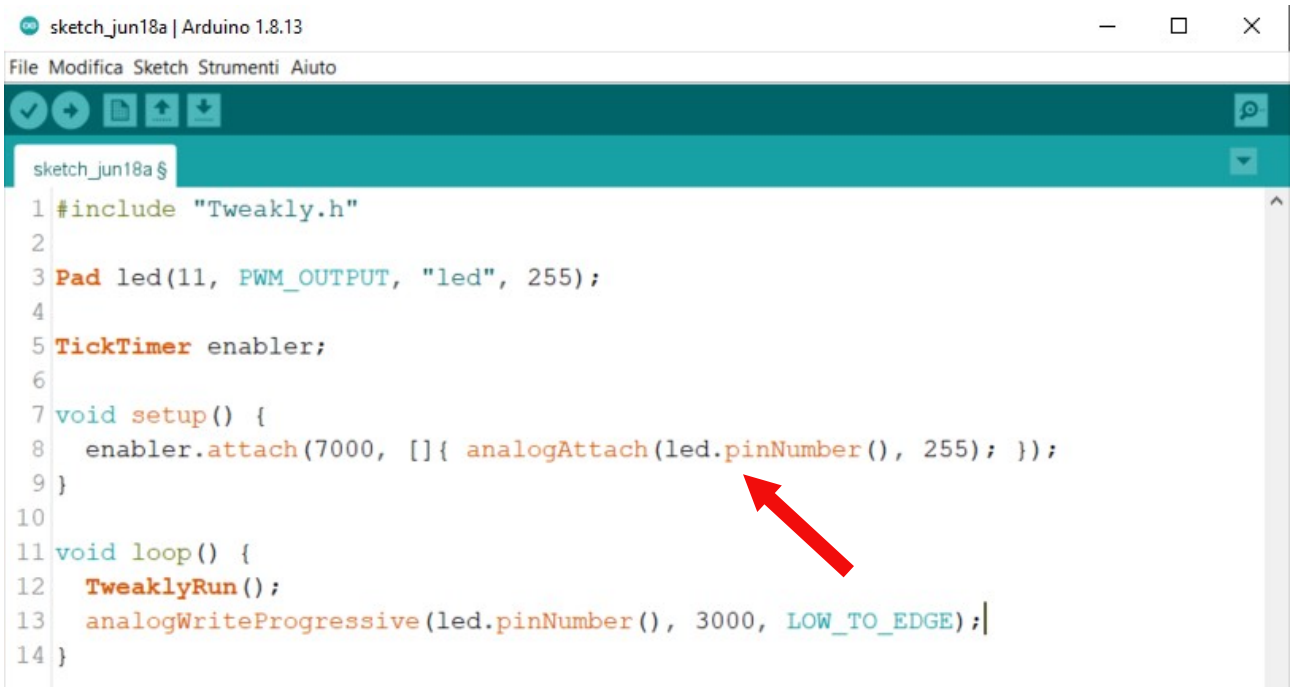


In caso fosse necessario partire da un valore alto e scendere a un valore basso utilizzando i metodi **TO_LOW** e **LOW_TO_EDGE** è necessario inizializzare il pin con un valore predefinito pari al valore massimo desiderato :



Utilizzando i modi **TO_HIGH, TO_LOW, HIGH_TO_EDGE e LOW_TO_EDGE** il pin utilizzato nel settaggio progressivo viene disabilitato una volta che il valore finale prestabilito viene raggiunto, per riabilitare il pin sarà necessario utilizzare un richiamo alla funzione **analogAttach()**.

Per riabilitare il pin possiamo usare un TickTimer come nell'esempio qui sotto o qualsiasi altra funzionalità che ci permetta di riaccedere al settaggio del pin pwm desiderato :



Ogni 7 secondi il pin tornerà al valore 255 e di conseguenza **analogWriteProgressive()** spegnerà di nuovo in modo graduale il led.

analogAttach(uint8_t pin, unsinged int new_value);

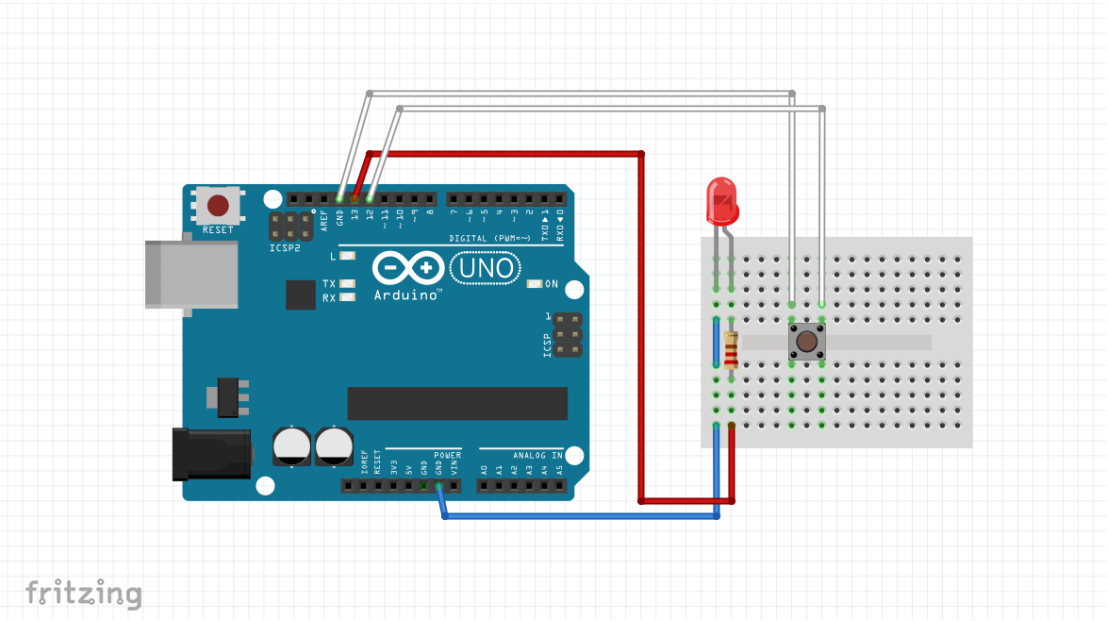
analogAttach() permette di riabilitare un pin PWM quando questo viene disabilitato con un metodo progressivo della funzione **analogWriteProgressive()** o dopo aver chiamato la funzione **analogDetach()**.

analogDetach(uint8_t pin);

Permette di disabilitare un pin PWM indicando il pin da disabilitare.

2.9 Gestione dei Pad | Pulsanti e metodi di debounce

Su Tweakly hai la possibilità di gestire i pulsanti in 3 metodi diversi, il primo metodo è senza debouncing e il valore che viene letto dal pin può essere letto con il metodo **read() :**



digitalPushButton(uint8_t pin);

digitalPushButton() permette di leggere lo stato privo di bouncing di un pulsante collegato a un pin di input :

digitalSwitchButton(uint8_t pin);

digitalSwitchButton() permette di leggere lo stato privo di bouncing di un pulsante collegato a un pin di input con la modalità interruttore. Questa particolare funzione mantiene lo stato dell'ultima pressione e commuta ogni volta che viene premuto il pulsante assegnato al pin :

Se premiamo il pulsante il led si accende e resta acceso fino alla prossima pressione del pulsante per poi spegnersi.

3.0 Timing | TickTimer

Nei capitoli precedenti di questa guida abbiamo utilizzato varie volte i **TickTimer,** questa classe permette di creare dei timer non bloccanti che eseguono le operazioni con un intervallo di tempo specificato in millisecondi.

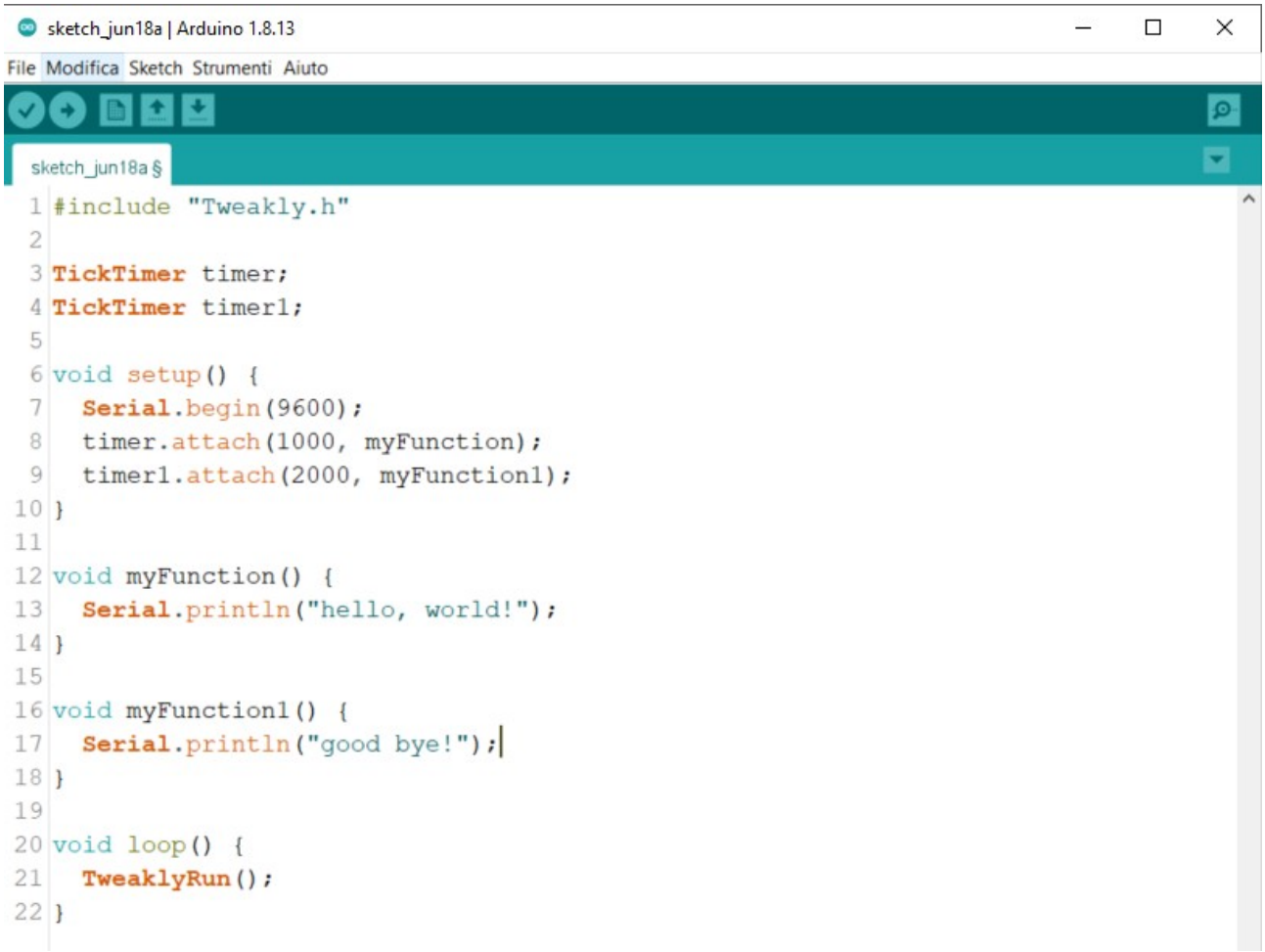
Il TickTimer ha diversi metodi che possono essere utilizzati per gestire un timer specifico.

attach(unsigned long delay, callback);

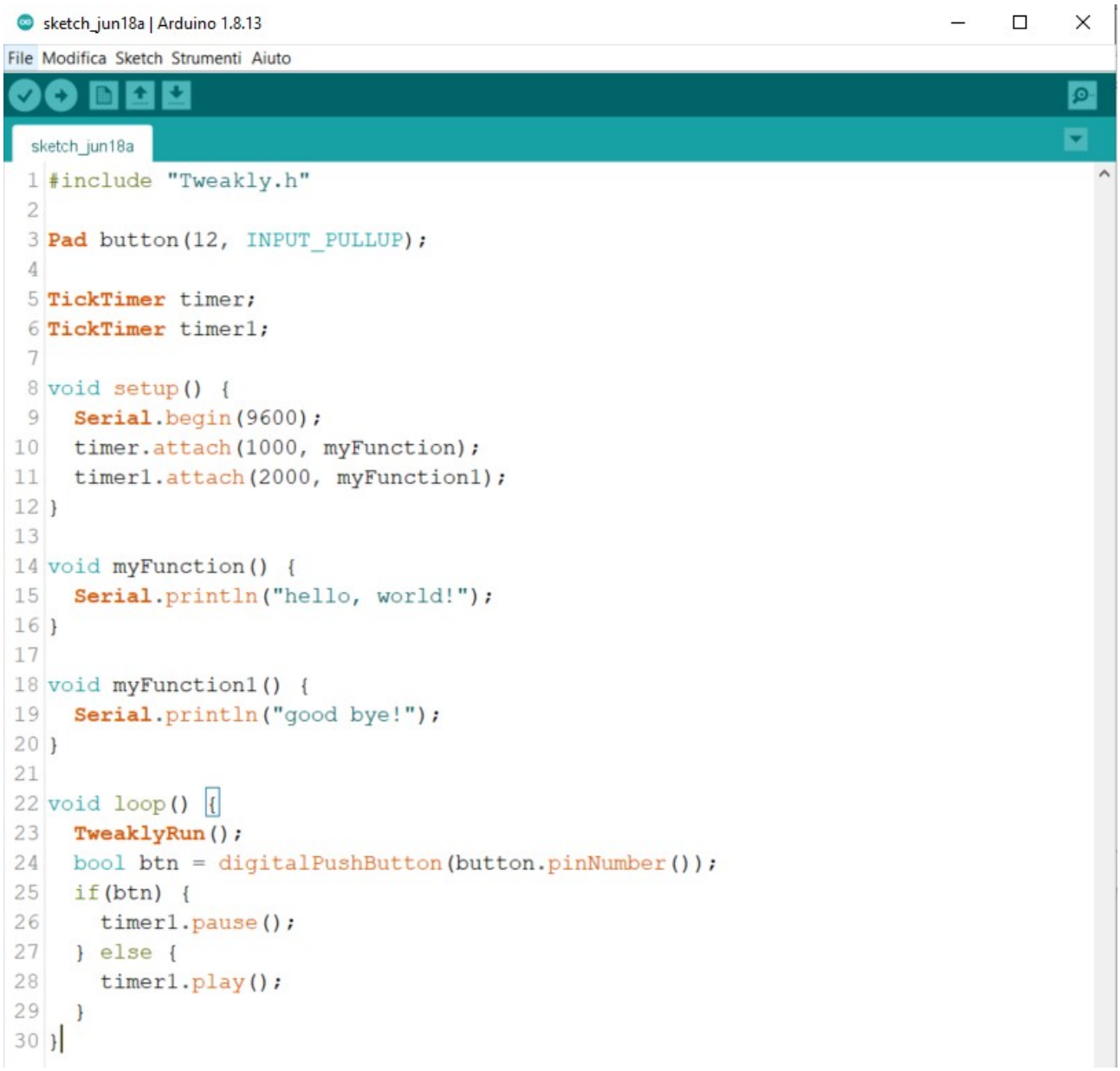
attach() permette di inizializzare un timer, può essere richiamato in qualsiasi parte del tuo codice, anche se è sempre preferibile richiamare la configurazione di un nuovo timer nel **setup() :**

Nel codice dell’esempio indicato qui sopra ogni secondo viene stampata sul monito seriale la scritta “hello, world!”.

E’ possibile creare N timer nel tuo codice e ogni timer può avere un tempo di azione diverso :



Se avessimo la necessità di interrompere un TickTimer possiamo usare la funzione **pause()** e usare la funzione **play()** per ripristinare la sua esecuzione :



In caso di pressione del pulsante il **timer1** viene messo in pausa e non esegue le istruzioni che gli abbiamo assegnato.

3.1 Timing | Assegnare istruzioni tramite lambda a TickTimer

E’ possibile assegnare istruzioni al timer utilizzando lambda e risparmiare righe di codice, alcuni esempi qui di seguito :

oppure :

4.0 doList | Metodo di scorrimento esplicito

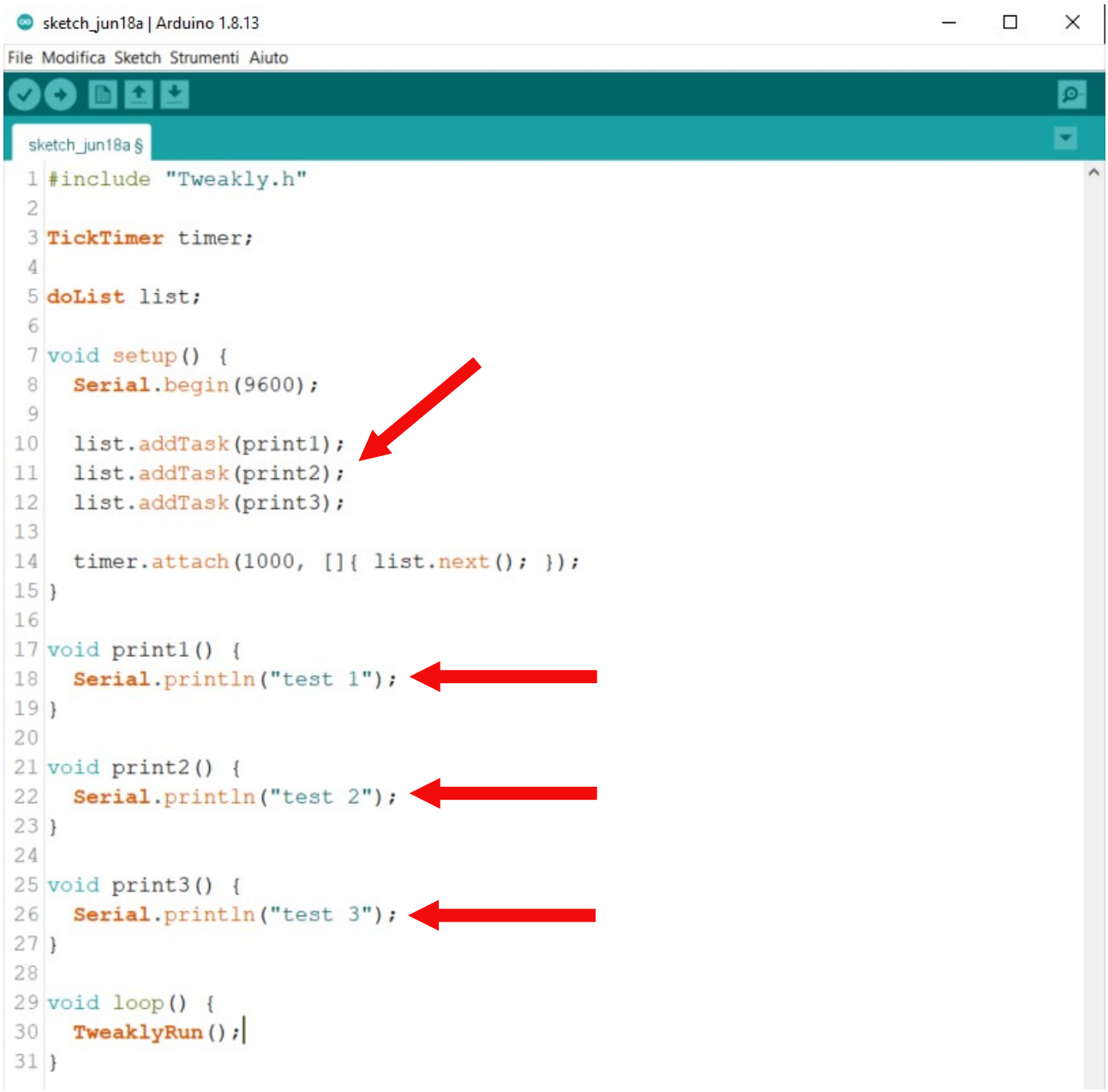
Tweakly include una classe chiamata **doList** che permette di creare “liste della spesa” in grado di contenere delle istruzioni e che possono essere eseguite a ogni richiamo della lista mediante il metodo **next()**. Le istruzioni all’interno della lista possono essere eseguite una alla volta, ad ogni richiamo del metodo **next()** viene eseguita la funzione successiva.

Per sperimentare con questo metodo possiamo scrivere un codice di questo tipo :

Per assegnare un’istruzione alla lista utilizziamo il metodo **addTask** e all’interno delle parentesi tonde di quest’ultimo indichiamo una o più istruzioni da eseguire al richiamo della task o in alternativa è possibile richiamare una funzione presente nel nostro programma.

Nel caso dell’esempio, ogni secondo verranno scritti sulla seriale i testi “test 1”, “test 2” e “test 3” in modo cliclico e uno alla volta, ripartendo da capo ogni volta che le istruzioni della lista sono finite. Questa funzionalità permette per esempio di cambiare scena su un display lcd e mostrare ad ogni intervallo di tempo una schermata differente da quella precedente.

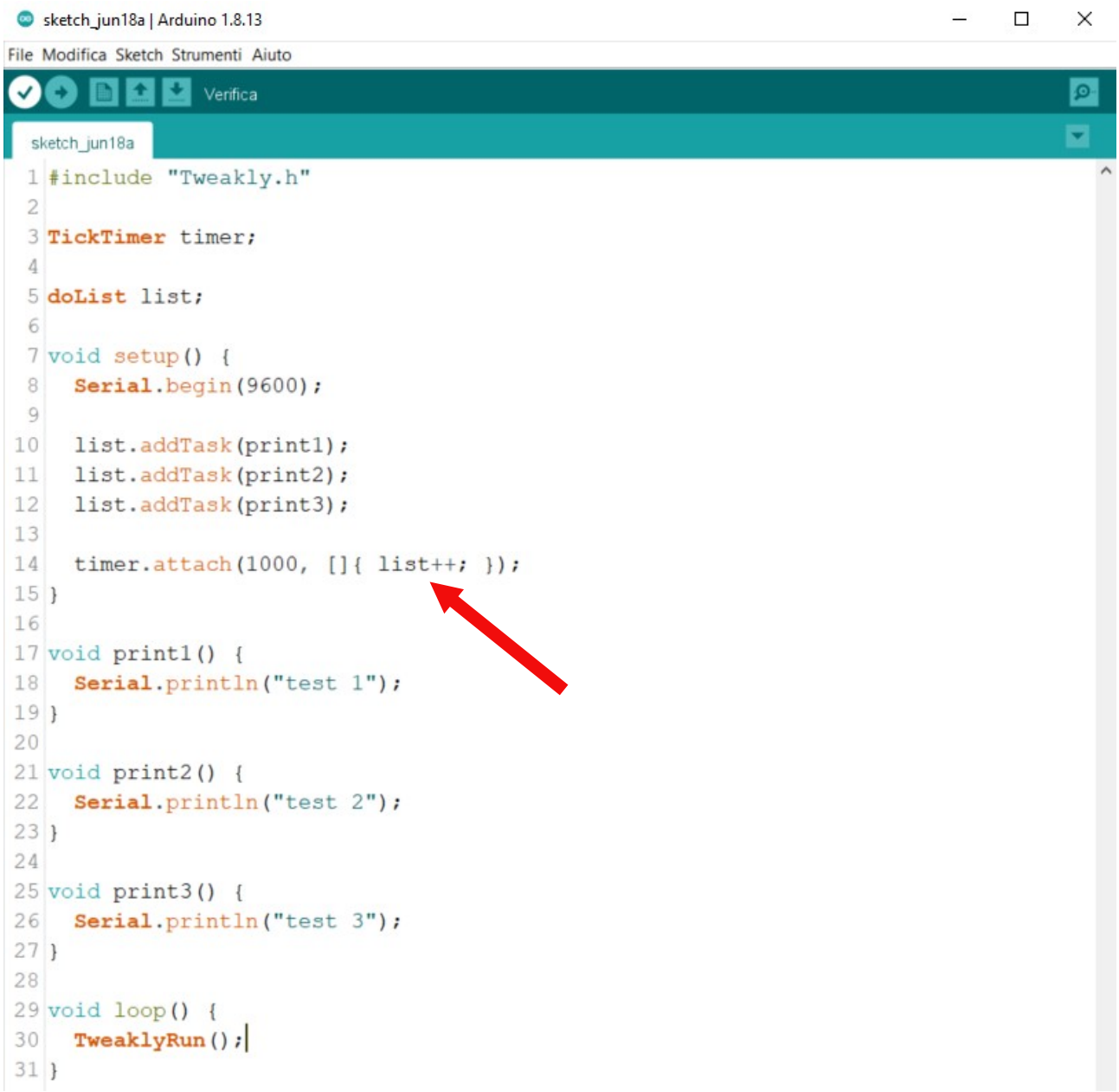
Nell’esempio qui sopra è stata utilizzata lambda per assegnare un’azione alla task della lista, tuttavia è possibile riprodurre lo stesso programma in questo modo :



Ovviamente è possibile creare N liste nel codice, gestendo numerosi cambi di istruzione all’interno del programma.

4.1 doList | Metodo di scorrimento implicito

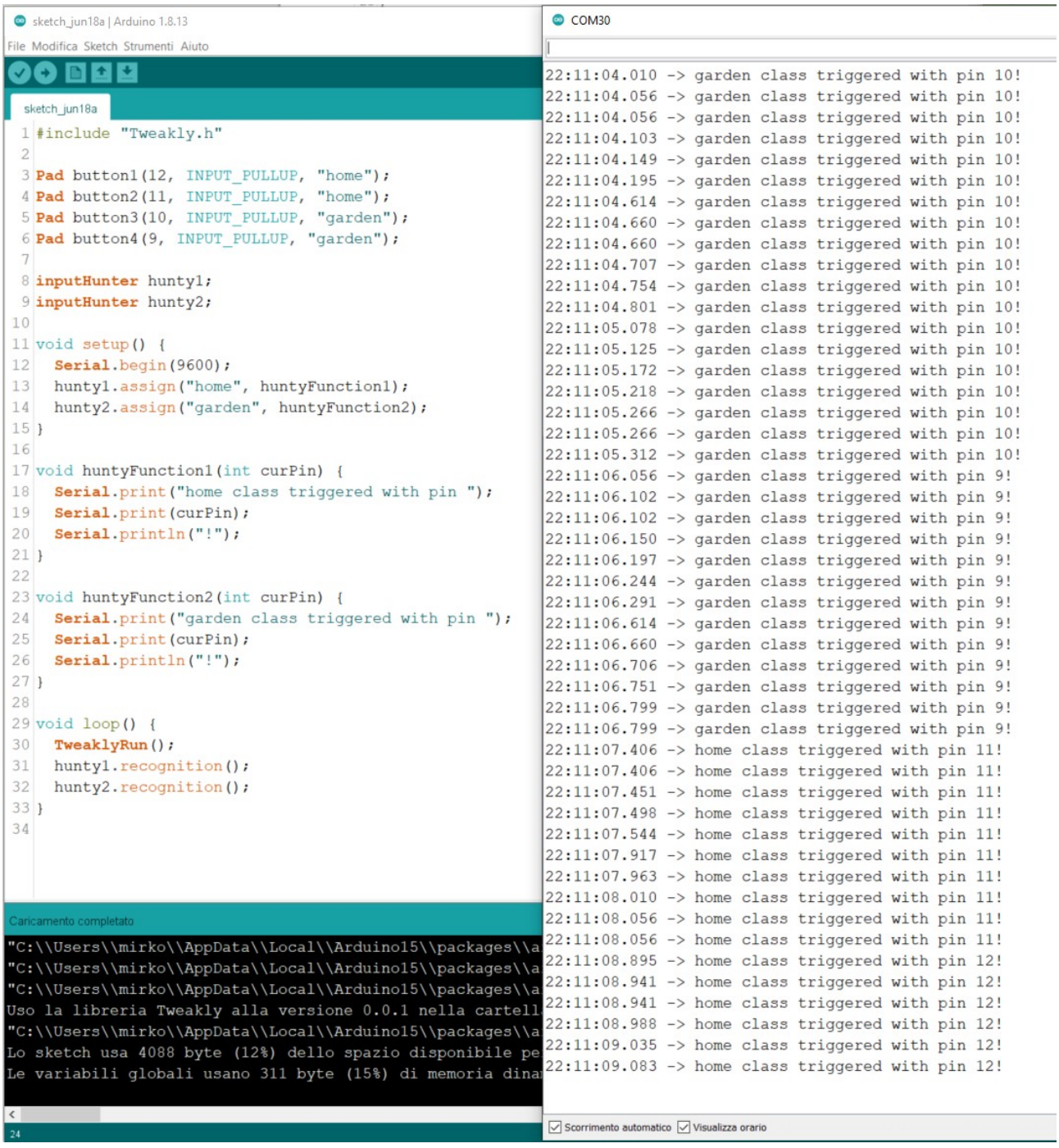
Possiamo scorrere una lista in modo implicito utilizzando l'operatore "++" in alternativa al metodo **next() :**



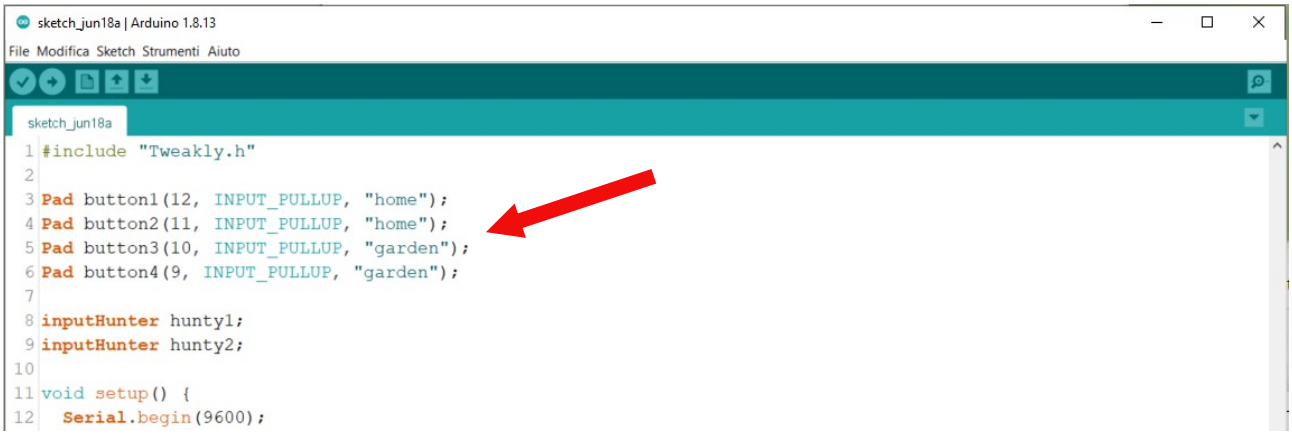
Il metodo di scorrimento implicito della classe doList velocizza la scrittura del codice e riduce le digitazioni, tuttavia i due metodi di scorrimento mostrati precedentemente sono identici e non presentano differenze di funzionamento.

5.0 inputHunter

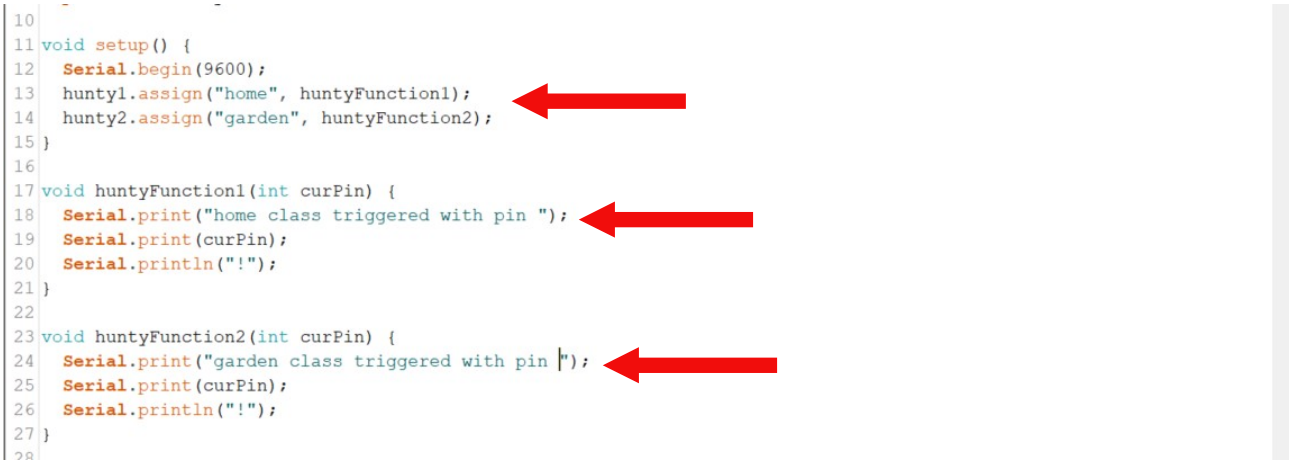
Tra le features più interessanti di Tweakly ci sono gli **inputHunter**, questa funzione permette di creare dei controlli che girano su una classe di pin e intercettare quali sono i pin premuti. Un utilizzo classico di questa funzionalità è la possibilità di leggere quanto uno stato di un gruppo di sensori all'interno di una stanza cambia, permettendo quindi di creare applicazioni dedicate a impianti di allarme e simili.



Una volta creato l'oggetto **inputHunter** e avergli dato un nome, dobbiamo essere sicuri di aver settato una classe ai nostri pin.

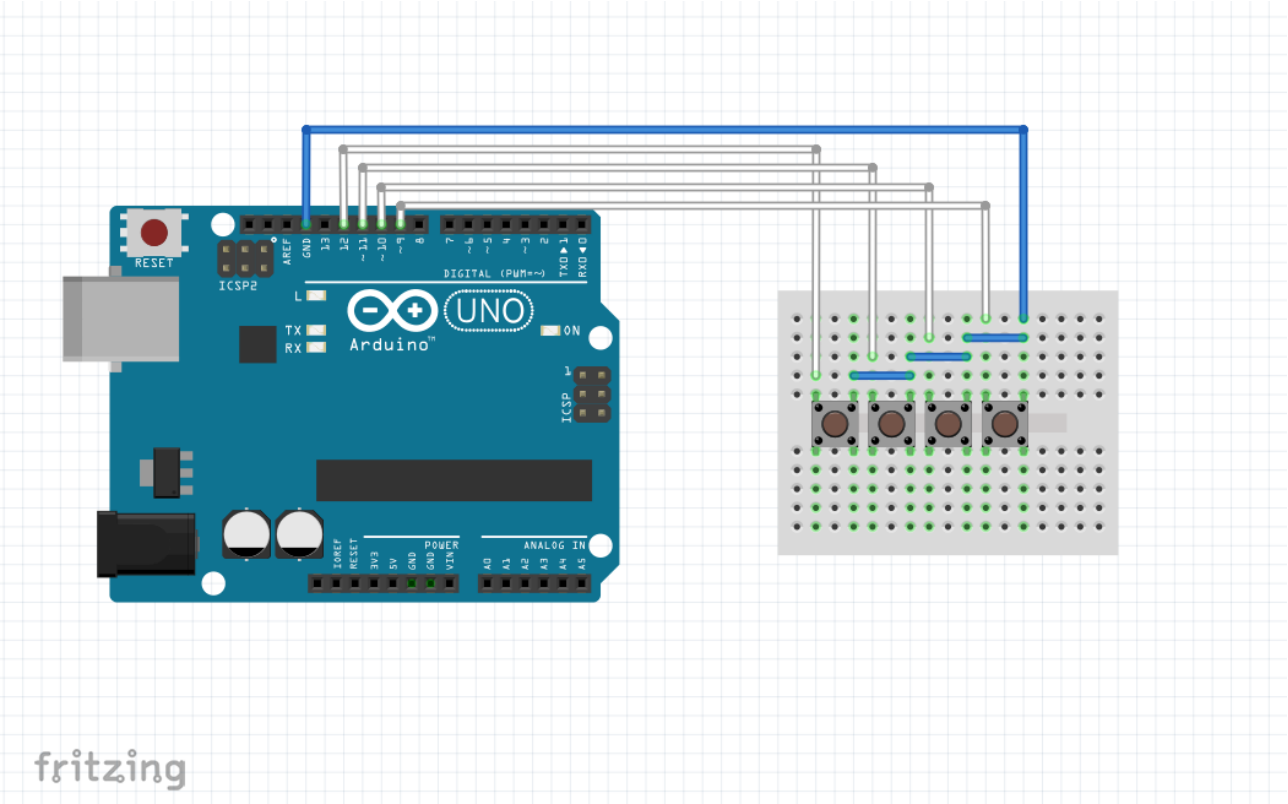


In questo esempio abbiamo dato a due pulsanti la classe “home” e ad altri due pulsanti la classe “garden”. Successivamente, nel setup() assegniamo ai nostri due inputHunter le due classi dei pin e settiamo una funzione di callback che verrà richiamata ogni volta che premiamo un pulsante di quella determinata classe :



Come si può ben notare le funzioni **huntyFunction1** e **huntyFunction2** assegnate rispettivamente ai cacciatori di input **hunty1** e **hunty2** hanno un argomento di tipo int chiamato **curPin**, quest’ultimo è un argomento obbligatorio che può essere rinominato come vogliamo e che conterrà il numero del pin che ha commutato il suo valore.

E’ possibile controllare quale pin ha cambiato di stato e utilizzare un costrutto decisionale per eseguire delle istruzioni specifiche in base agli ingressi o in alternativa creare una funzione globale in base alla classe cambiata nel corso del programmata.



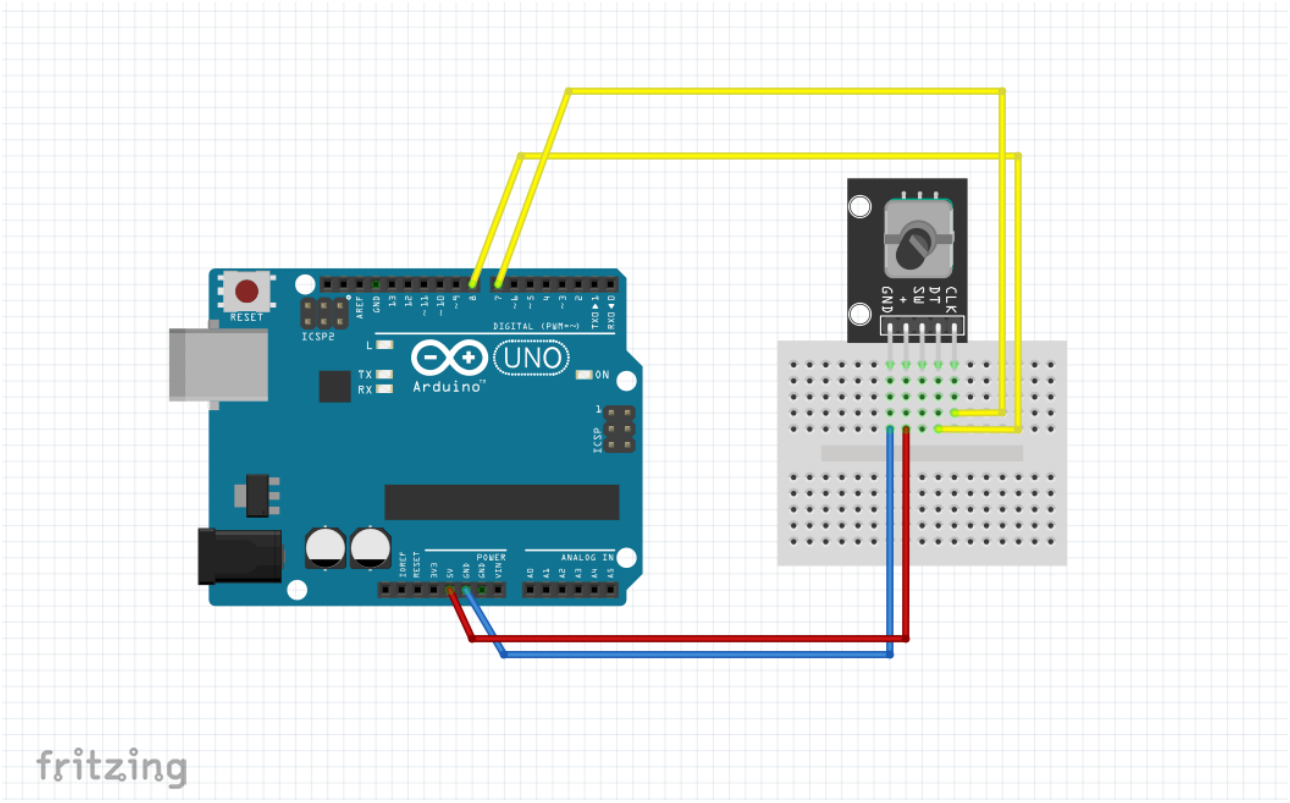
Avendo collegato i pulsanti come nello schema sopra indicato è possibile controllare con precisione quali sono i pulsanti premuti e in quale classe sono stati settati.

Per poter far funzionare gli **inputHunter** in modo corretto, all'interno del loop è necessario richiamare il metodo **recognition()** per far si che gli stai gli input settati precedentemente vengano tenuti sotto controllo.

```
29 void loop() {
30   TweaklyRun();
31   hunty1.recognition();
32   hunty2.recognition();
33 }
34
```

6.0 Gestione degl Encoder rotativi

Il supporto Nativo agli encoder integrato in Tweakly permette di gestire in modo molto preciso gli encoder rotativi e determinare il verso di rotazione di quest'ultimi.



sketch_jun18a | Arduino 1.8.13

File Modifica Sketch Strumenti Aiuto

sketch_jun18a

```
#include "Tweakly.h"

Pad dt(8, INPUT_PULLUP);
Pad clk(7, INPUT_PULLUP);

void setup() {
  Serial.begin(9600);
  encoderAttach(dt.pinNumber(), clk.pinNumber(), encoderChange);
}

void encoderChange(bool curDir) {
  if(curDir) {
    Serial.println("left");
  } else {
    Serial.println("right");
  }
}

void loop() {
  TweaklyRun();
}
```

Per prima cosa è necessario settare i pin dt e clk come **INPUT_PULLUP :**

successivamente usando la funzione encoderAttach() nel setup indichiamo il pin DT, il pin CLK e la funzione di callback che nell'esempio in questione abbiamo chiamato **encoderChange :**

```
void setup() {
  Serial.begin(9600);
  encoderAttach(dt.pinNumber(), clk.pinNumber(), encoderChange);
}

void encoderChange(bool curDir) {
  if(curDir) {
    Serial.println("left");
  } else {
    Serial.println("right");
  }
}
```

l'argomento **curDir** può essere rinominato in base alle nostre esigenze e può assumere un valore **true** e **false** in base alla rotazione dell'encoder.

Agendo sull'encoder rotativo vedremo nel monitor seriale il testo **right** o **left** in base alla direzione della rotazione.

7.0 Scenari di utilizzo

Tweakly è una libreria che non si pone nell'intendo di rimpiazzare librerie e metodi di programmazione ma al contrario punta a un formattazione del codice basato su callback, riducendo la digitazione di codice e offrendo un metodo di sviluppo moderno su board poco potenti e che non permettono l'esecuzione di un RTOS. Tuttavia, Tweakly non è un RTOS ma una sorta di Framework che potenzia l'uso del Wiring su tutte le board basate su core Arduino.

Tweakly è perfetta per lo sviluppo di :

- Applicazioni domotiche
- Controllo motori
- Realizzazione di tastierini
- Semplificazione di progetti scolastici
- Programmi che devono girare su applicazioni web affiancati a piattaforme per l'IoT come Blynk, Arduino IoT Cloud e simili, avendo metodi non bloccanti i client che dialogano con un server non verranno interrotti.
- Applicazioni di Interaction Design