

Project Report

TACHIMETRO PER BICICLETTA

Course Name: Architettura Degli Elaboratori



Submitted By: FRANCESCO CALICCHIO

Badge no: 122874

Dipartimento di Scienze e Tecnologie

July 15, 2023

Contents

1	Obiettivo del progetto	3
2	Calcolo dei dati	3
2.1	Velocità istantanea	3
2.2	Distanza totale percorsa	3
3	Hardware	4
3.1	Sensore LDR	4
3.1.1	Schema Logico	5
3.1.2	AO e DO	5
3.1.3	Analog Output	6
3.1.4	Digital Output	6
3.2	Display LCD 16x2	6
3.3	Push Button	7
3.3.1	Bouncing	8
3.3.2	Input Pullup	8
4	Funzionamento del sistema	8
4.1	Rilevamento di movimento	8
4.1.1	Interrupt Service Rountine (INT1)	9
4.2	Avvio ed Arresto del sistema	10
4.2.1	Interrupt Service Rountine (INT0)	11
4.3	Gestione del contrasto	12
4.3.1	Sostituzione analogRead()	13
4.3.2	Generazione segnale PWM	14
4.4	Gestione del display LCD	17
4.4.1	sendData()	17
4.4.2	sendCommand()	18
4.4.3	sendText()	18
4.4.4	sendNum()	19
4.5	Gestione del tempo	19
4.5.1	TIMER/COUNTER 1	20
4.5.2	ISR(TIMER1_COMPA_vect)	21
4.5.3	TIMER/COUNTER 2	22

4.5.4	ISR(TIMER2_COMPA_vect)	23
4.6	Aggiornrnamento del display	24
4.7	Funzionamento generale del sistema	25
5	Stati dell'automa	26
6	Funzioni Logiche	27
6.1	Automa ancora in fase di setup	27
6.2	Automa in fase di movimento	27
7	Variabili e Costanti	28
7.1	Variabili	28
7.2	Costanti	29
8	Potenziali miglioramenti hardware del sistema	31

1 Obiettivo del progetto

Il presente report descrive il processo di progettazione e sviluppo di un tachimetro per bicicletta con l'obiettivo di realizzare un dispositivo efficiente e funzionale. Il progetto si concentra sulla creazione di un sistema minimalista che consenta all'utente di utilizzare il tachimetro senza distrazioni durante l'utilizzo della bicicletta. A tal fine, è stato selezionato Arduino Uno come piattaforma principale, sfruttando il microcontrollore ATmega328P, per ospitare tutti i sensori necessari.

2 Calcolo dei dati

Nella presente sezione, verranno fornite le informazioni sul calcolo dei dati necessari per un potenziale utilizzatore del dispositivo. I parametri richiesti sono i seguenti:

2.1 Velocità istantanea

Per determinare la velocità istantanea di una bicicletta, sono necessari i seguenti dati:

- Circonferenza della ruota [m]
- Tempo necessario per completare una singola rivoluzione [ms]

$$\frac{\text{CirconferenzaRuota}}{T} * 36000 \quad [\text{km}/\text{h}] \quad (1)$$

Nella formula (1), il fattore di conversione 36000 viene utilizzato per convertire la velocità da mt/ms a km/h . Se il tempo (T) fosse espresso in secondi, sarebbe necessario moltiplicare per un fattore di conversione di 3.6 per ottenere la velocità in chilometri all'ora.

2.2 Distanza totale percorsa

Per calcolare la distanza totale percorsa, sono richiesti i seguenti dati:

- Numero totale di rivoluzioni compiute dalla ruota
- Circonferenza della ruota (espressa in metri)

L'equazione utilizzata per calcolare la distanza totale percorsa è la seguente:

$$\frac{nRivoluzioni * \text{CirconferenzaRuota}}{1000} \quad [\text{km}] \quad (2)$$

Nella formula (2), la divisione per 1000 viene eseguita per la conversione da m a km .

3 Hardware

La sezione seguente illustra l'hardware necessario per il progetto, considerando la presenza della scheda Arduino Uno.

3.1 Sensore LDR

Il sensore LDR (Light Dependent Resistor), noto anche come fotoresistore, è un particolare tipo di resistore in grado di variare il proprio valore in base all'intensità luminosa a cui è esposto.

Ogni sensore LDR ha un parametro di illuminazione in lux che corrisponde a un determinato valore di resistenza. Solitamente, viene scelto un valore standard di 10 lux che corrisponde a una resistenza di $50\text{ k}\Omega$ (il sensore LDR di Wokwi utilizza appunto questo valore standard). Il valore di 10 lux corrisponde, in pratica, all'intensità luminosa che si può avere al crepuscolo.

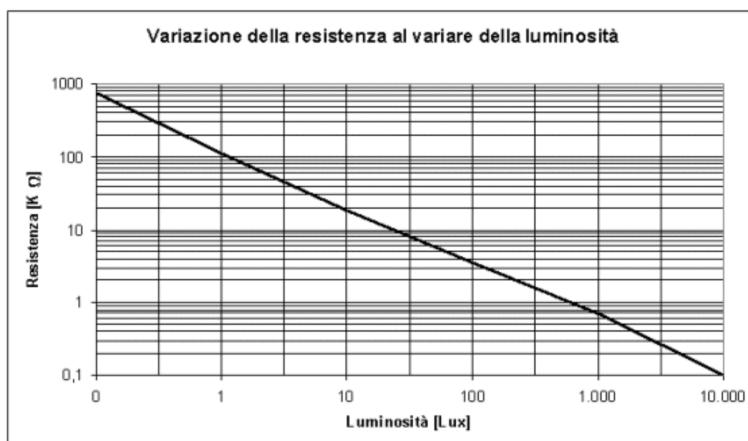


Figure 1: Variazione della resistenza al variare della luminosità

Come evidenziato dalla figura, il rapporto tra $k\Omega$ e lux è inversamente proporzionale: all'aumentare della resistenza, diminuisce l'intensità luminosa, mentre all'aumentare dell'intensità luminosa, diminuisce la resistenza.

Di conseguenza:

- In presenza di una **forte illuminazione**, si avrà il massimo voltaggio possibile e la resistenza più bassa.
- In condizioni di **completa oscurità**, si avrà il minimo voltaggio possibile e la resistenza più alta.

3.1.1 Schema Logico

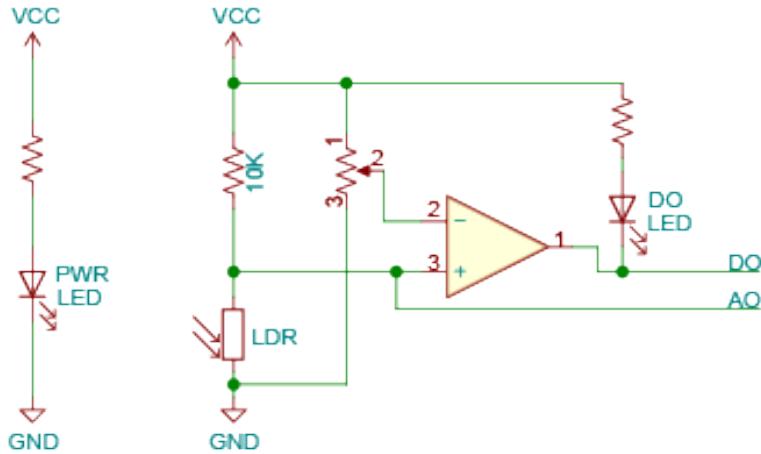


Figure 2: Schema Logico Sensore LDR

- **PWR LED:** Il LED di stato, indicante il passaggio di corrente all'interno del sensore e quindi confermando il corretto collegamento, è collegato in serie a una resistenza tra la tensione di alimentazione (VCC) e il terminale di terra (GND).
- Il fotoresistore (**LDR**) è collegato in serie con una resistenza di solito di $10\text{ k}\Omega$, tra la tensione di alimentazione (VCC) e il terminale di terra (GND).
- **Potenziometro:** Utilizzato per ottenere una tensione variabile da 0V a un massimo di VCC (nel nostro caso 5V).
- **DO LED:** Collegato in serie con una resistenza, il LED indica quando si supera o si scende al di sotto di una soglia predefinita di 100 lux. Questa informazione è rilevante se si considera l'output digitale (D0), che verrà approfondito successivamente.
- **Partitore di Tensione:** Utilizzato per suddividere la tensione tra due resistenze collegate in serie.

3.1.2 AO e DO

Il sensore LDR di Wowki come si può vedere dalla seguente illustrazione:

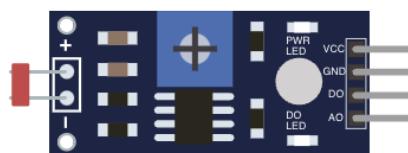


Figure 3: Sensore LDR

Presenta quattro pin:

- **VCC** per l'alimentazione a 5V
- **GND** per il collegamento a terra
- **AO** per l'uscita analogica
- **DO** per l'uscita digitale

3.1.3 Analog Output

L'uscita analogica del sensore consente di misurare il voltaggio utilizzando la funzione `analogRead()`. Come precedentemente menzionato, il voltaggio è direttamente proporzionale alla quantità di luce che colpisce il sensore: maggiore è l'intensità luminosa, minore sarà il voltaggio rilevato; viceversa, con una diminuzione dell'intensità luminosa, si avrà una il massimo voltaggio rilevato.

La funzione `analogRead()` restituisce un valore di 10 bit, che corrisponde a una gamma di 1024 valori possibili. Questo significa che il valore restituito dalla funzione `analogRead()` varierà da 0 a 1023 ($2^n - 1$ numeri in base 10 rappresentabili con 10bit), consentendo di rappresentare la tensione misurata in una scala numerica discretizzata.

3.1.4 Digital Output

L'uscita digitale, a differenza dell'uscita analogica, non restituisce un valore numerico, ma indica lo stato del sensore. Lo stato sarà **HIGH** (alto) se il valore in lux supera la soglia di 100 (circa 2.5V), mentre sarà **LOW** (basso) se il valore non la supera.

Lo stato dell'uscita digitale è visibile attraverso il DO LED, come già specificato nello schema logico. Quando lo stato è **HIGH**, il DO LED sarà acceso, mentre quando lo stato è **LOW**, il DO LED sarà spento.

3.2 Display LCD 16x2

Nel progetto è stato utilizzato un display LCD standard (non I2C) per visualizzare tutte le informazioni menzionate in precedenza e fornire un feedback visivo all'utente finale. Idealmente, il display dovrebbe essere posizionato sul manubrio della bicicletta per una comoda visualizzazione.

Il display LCD 16x02 è composto da 16 colonne e due righe, consentendo di visualizzare un output sullo schermo.

Esso è composto da una moltitudine di pin:

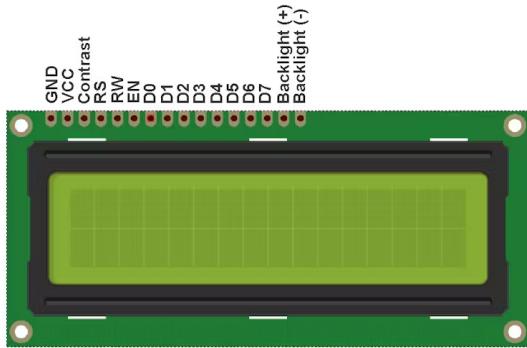


Figure 4: Display LCD 16x2

- **Pin Dati:** sono 8 vanno dal pin D0 al pin D7 e sono i pin adoperati per mandare i dati dall'arduino al display. Importante denotare che solo 4 di questi pin (da D4 a D7) sono stati utilizzati all'interno del progetto.
- **VSS e VDD:** il primo va collegato al GND e il secondo è per l'alimentazione a 5V,
- **V_O:** per il contrasto del display,
- **RS:** per mandare comandi al display,
- **E:** per abilitare il display,
- **RW:** read and write che va collegato a massa,
- **A e K:** sono rispettivamente anodo e catodo per la retroilluminazione, il primo va collegato ad un modulo I/O se si vuole controllare la retroilluminazione o in alternativa va collegato tramite una resistenza ai 5V ed il secondo va collegato a massa.

3.3 Push Button

Il Push Button utilizzato è un classico bottone a pressione come nella foto che segue:

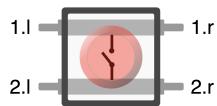


Figure 5: Schema Push Button

Il push button presenta due contatti, che corrispondono alle due "righe" di pin, quella superiore ed inferiore e vengono utilizzati per la connessione con Arduino.

3.3.1 Bouncing

Di norma, i pulsanti fisici meccanici presentano una caratteristica nota come "bouncing", che causa l'apertura e la chiusura ripetuta del circuito centinaia di volte quando viene premuto il pulsante. Tuttavia, nel contesto del progetto su Wokwi, questa caratteristica verrà disattivata per il pulsante.

Nel mondo reale, è possibile gestire il fenomeno del bouncing sia attraverso specifico codice di debouncing che mediante implementazioni hardware. Queste soluzioni consentono di stabilizzare il segnale del pulsante, eliminando i falsi contatti causati dal bouncing.

3.3.2 Input Pullup

Impostando i pin a cui sono collegati i push button come INPUT_PULLUP è possibile far sì che il push button in questione sia sempre su uno stato di HIGH e solo quando venga premuto, a seconda se è "buonced" o "de-bounced", passi momentaneamente a LOW.

4 Funzionamento del sistema

I moduli, appena descritti, di cui è composto il sistema, assolvono tutti a funzioni specifiche.

La disposizione degli ste:

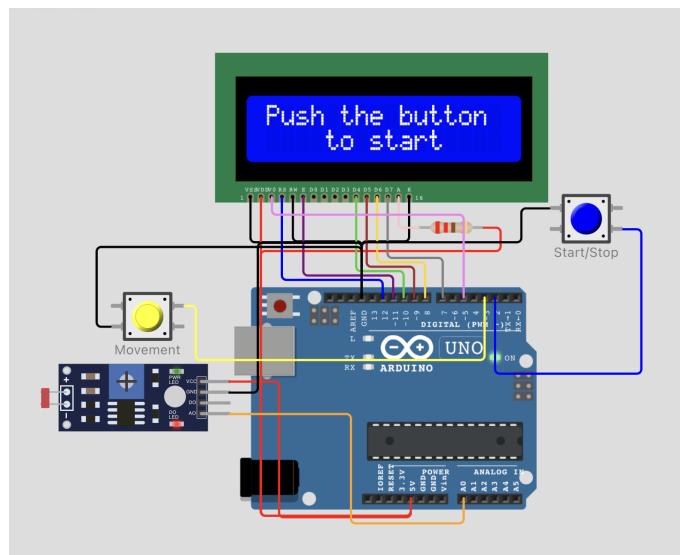


Figure 6: Schema di funzionamento generale

4.1 Rilevamento di movimento

Per problemi progettuali, che affronteremo poi, è stato deciso di usare un Push Button come descritto nella sezione 3.3.

Nella figura appena mostrata si fa riferimento al bottone giallo. Tale bottone è collegato al GND tramite il cavo nero, e tramite il cavo giallo è collegato al pin 3 della porta D.

La scelta del pin non è casuale, bensì necessaria per la maggiore ottimizzazione della macchina possibile.

Leggendo il data sheet di Atmega328p, infatti possiamo leggere che i pin disponibili per generare un'interruzione esterna si trovano sulla porta D, ai pin 2 e 3, come segue in figura:

Bit	7	6	5	4	3	2	1	0
Access						INT1	INT0	
Reset						R/W	R/W	

Figure 7: Pin per interrupt esterni disponibili sulla porta D

Impostando il pin 3 come `INPUT_PULLUP`, come descritto nella sezione 3.3.2, è possibile sfruttare la natura dei pulsanti per rilevare la transizione tra gli stati di `HIGH` e `LOW`. Nel metodo `setup()`, è possibile impostare un interrupt sul pin specificato per rilevare il cambio di stato utilizzando il parametro `CHANGE`. Di seguito è riportato il codice per il setup:

```
1 // Configurazione interrupt esterno su INT1 (pin 3)
2 EICRA |= (1 << ISC10); // Interruzione esterna su cambio di stato
3 EIMSK |= (1 << INT1); // Abilita interrupt esterno su INT1 (pin 3)
```

Nel dettaglio:

- **EICRA** è un registro che controlla la configurazione delle interruzioni esterne del microcontrollore. In questo caso, l'operatore `|=` viene utilizzato per impostare un bit specifico all'interno del registro. L'espressione `(1 << ISC10)` crea un valore in cui il bit `ISC10` (bit 1 del registro `EICRA`) è impostato a 1 e tutti gli altri bit sono a 0, consentendo l'attivazione dell'interrupt esterno su un cambiamento di stato.
- **EIMSK** è un altro registro che abilita gli interrupt esterni specifici del microcontrollore. Similmente a prima, l'operatore `|=` viene utilizzato per impostare un bit specifico all'interno del registro. L'espressione `(1 << INT1)` crea un valore in cui il bit `INT1` (bit 1 del registro `EIMSK`) è impostato a 1 e tutti gli altri bit sono a 0, abilitando l'interrupt esterno sul pin 3.

In termini semplici quello che succede è che quando viene rilevato un cambio di stato sul PD3 il microcontrollore gestisce l'evento attivando una routine di interrupt specifica (`ISR`).

4.1.1 Interrupt Service Rountine (INT1)

L'`ISR` avviata quando si verifica il cambio di stato sul PD3, recupera tutti i dati necessari per il calcolo della velocità istantanea (1) e distanza totale percorsa (2).

Vengono sfruttate le particolarità fisiche del push button abilitato come `INPUT_PULLUP` descritte nella sezione 3.3.2 per andare a simulare quella che è una rivoluzione della ruota della bicicletta.

Nei tachimetri più tradizionali, come verrà affrontato più avanti, è presente solitamente un sensore ad effetto Hall che grazie alle sue caratteristiche fisiche è capace di rilevare il cambio di stato (di conseguenza quando avviene la rivoluzione e quando termina) e quindi calcolare i dati necessari.

Quello che avviene nella routine di interrupt quindi è che al rilevamento della rivoluzione (il PD3 a cui è collegato il push button diventa LOW) viene:

- assegnato il valore `true` ad una variabile, ciò servirà per poi calcolare i dati all'interno del loop
- viene incrementato il counter delle rivoluzioni, ciò è necessario per calcolare la distanza totale percorsa, come nella formula (2).

Quando il PD3 torna HIGH, il suo stato "normale" il sistema salva l'istante (in ms) in cui il pin ritorna HIGH, ciò è necessario per calcolare T nel loop, ovvero la differenza di tempo che intercorre tra una rivoluzione ad un'altra (il tempo per fare una rivoluzione).

Di seguito è rappresentato il diagramma di flusso per la routine:

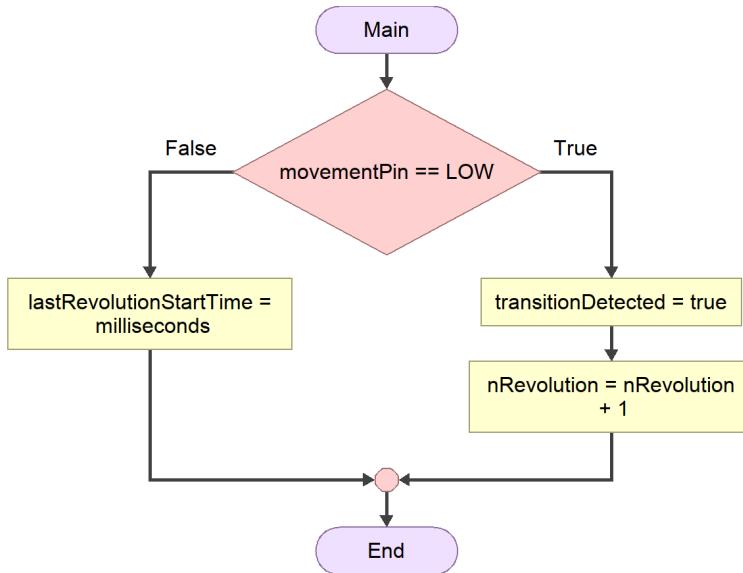


Figure 8: Diagramma di flusso per ISR(INT0_vect)

4.2 Avvio ed Arresto del sistema

Il sistema prevede un secondo push button (quello di colore blu nell'immagine 6). Tale pulsante è responsabile dell'avviamento e dell'arresto del sistema, il pulsante è collegato (oltre che al GND) al PD2, su questo pin come esplicitato nell'immagine 7 è possibile quindi generare una routine di interrupt, in questo caso quando avviene un cambio di stato sul suddetto.

Per abilitare l'interrupt su quel pin al cambio di stato si procede in modo quasi analogo al paragrafo 4.1:

```

1 // Configurazione interrupt esterno su INT0 (pin 2)
2 EICRA |= (1 << ISCO0); // Interruzione esterna su cambio di stato
3 EIMSK |= (1 << INT0); // Abilita interrupt esterno su INT0 (pin 2)

```

I registri utilizzati sono gli stessi per entrambi i pin, vengono semplicemente "shiftati" bit differenti (essendo pin differenti).

Il pulsante come accennato ha quindi due funzioni:

- **Avvio:** quando viene rilevato un "single click" del pulsante la macchina passa da uno stato di attesa di interazione da parte dell'utilizzatore, in cui si trova ancora nel setup, e in cui manda a display la scritta "Push the button to start", allo stato di effettivo avviamento, in cui si si "entra" nel loop.
- **Arresto:** quando il pulsante viene premuto per più di 1 secondo, l'automa torna al suo stato iniziale (setup), viene riportato il Program Counter (PC) a 0.

4.2.1 Interrupt Service Rountine (INT0)

L'ISR dell'INT0, ovvero la routine viene avviata quando si genera un cambio di stato, avrà il compito di capire quando il pulsante è stato premuto e per quanto tempo e in base a questo assegnare alle giuste variabili il valore logico **true**;

Di seguito è riportato il diagramma di flusso per l'ISR(INT0_vect):

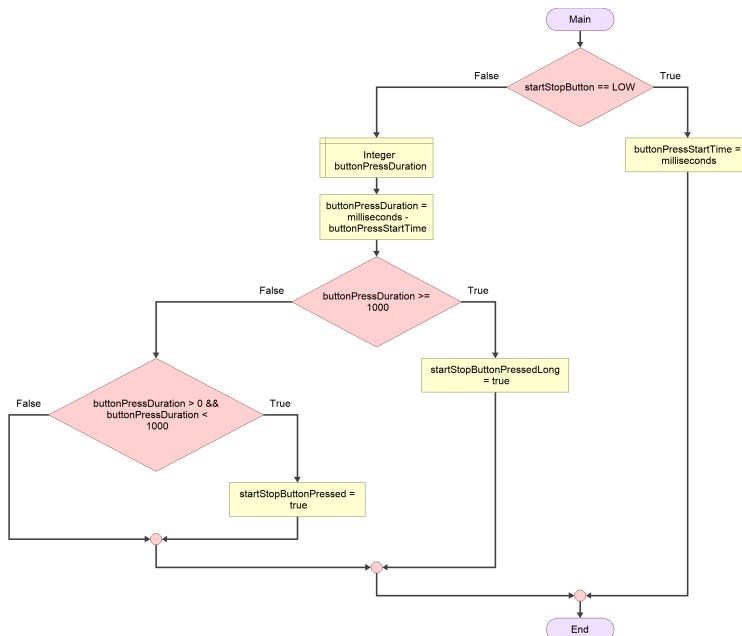


Figure 9: Diagramma di flusso per ISR(INT1_vect)

4.3 Gestione del contrasto

Uno dei moduli fondamentali per il sistema è quello della gestione del contrasto del display.

Come visto al paragrafo 3.2 è il pin V0 del display il responsabile della regolazione del contrasto, anche se è bene sottolineare come il contrasto del display nell'ambiente di simulazione Wokwi non si difatti simulabile.

Il contrasto dovrà dipendere dalla luminosità rilevata dal sensore LDR che si vede nell'immagine 3, maggiore sarà luminosità maggiore sarà il contrasto del display per garantire la massima visibilità, viceversa minore luminosità minore contrasto.

Il sensore è quindi collegato, oltre che ai 5V e al GND, tramite il pin A0 al pin A0 della porta C di Arduino. Il funzionamento dell'uscita analogica è stato approfondito al paragrafo 3.1.3.

Seguendo la tabella di valori del sensore:

Condition	Illumination (lux)	LDR Resistance	Voltage*	analogRead() value
Full moon	0.1	1.25MΩ	4.96	1016
Deep twilight	1	250kΩ	4.81	985
Twilight	10	50kΩ	4.17	853
Computer monitor**	50	16.2kΩ	3.09	633
Stairway lighting	100	9.98kΩ	2.50	511
Office lighting	400	3.78kΩ	1.37	281
Overcast day	1,000	1.99kΩ	0.83	170
Full daylight	10,000	397Ω	0.19	39
Direct sunlight	100,000	79Ω	0.04	8

Figure 10: Tabella di valori del sensore LCR

Si può notare come illuminazione (luce che "entra" nel sensore) e voltaggio siano indirettamente proporzionali.

E' importante ricordare che la funzione `analogRead()`, come già spiegato al paragrafo 3.1.3, restituisce un range di valori discretizzati che va da 0 a 1023 e che rappresentano il voltaggio "letto" su un determinato pin analogico.

Il sistema deve prevedere che venga generato un segnale PWM (Pulse-Width Modulation) il cui duty cycle dipenda dalla luminosità rilevata.

4.3.1 Sostituzione analogRead()

La funzione `analogRead()` è una funzione della libreria standard di Arduino, per quanto comoda dato che in automatico effettua una conversione ADC (Analog to Digital Converter) presenta delle notevoli inefficienze nell'economia generale del sistema.

Questa funzione è difatti in grado di discretizzare il voltaggio su un dato pin analogico (PORTC) impostando automaticamente i parametri di tensione e frequenza di campionamento predefiniti.

Per una maggiore efficienza e per avere più flessibilità nell'impostare i vari parametri è necessario ricorrere a notazioni diverse che prevedono registrazioni dirette per configurare l'ADC.

Il sistema dovrà quindi prevedere nel setup la seguente scrittura:

```
1 // Abilita la tensione di riferimento AVCC (5V)
2 ADMUX |= (1 << REFS0);
3 // Imposta la prescaler su 128 per ottenere una frequenza di campionamento di 125 kHz
4 ADCSRA |= (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);
5 // Abilita il convertitore ADC
6 ADCSRA |= (1 << ADEN);
```

Quello che accade nel seguente codice è:

- Viene abilitata la tensione di riferimento AVCC (5V), dato che il senore LDR ha una tensione variabile in un intervallo [0 – 5V], impostando il bit REFS0 del registro ADMUX.
- Si accede al registro ADCSRA per impostare un prescaler su un valore di 128, ciò è necessario perchè seguendo la formula:

$$\frac{CPU\ frequency}{prescaler} = \frac{16000000}{128} = 125000Hz = 125kHz \quad (3)$$

Otterremo come visibile una frequenza di campionamento di 125kHz, questa frequenza è responsabile della "risoluzione" con cui verranno "letti" i valori di voltaggio.

- Infine viene abilitato il convertitore ADC controllando il bit ADEN sempre del registro ADCSRA.

Nel loop invece si procede con:

```
1 // Avvia la conversione ADC per il sensore LDR
2 ADCSRA |= (1 << ADSC);
3 // Attendi la fine della conversione ADC
4 while (ADCSRA & (1 << ADSC));
5 // Leggi il valore convertito dal sensore LDR
6 lcdValue = ADC;
```

- Alla prima riga di codice viene avviata la conversione ADC per il sensore LDR impostando il bit ADSC del resistro ADCSRA.

- La linea di codice "while (ADCSRA & (1 << ADSC));" è un ciclo while che controlla lo stato del bit ADSC nel registro ADCSRA. Questa riga di codice viene utilizzata per attendere la fine della conversione ADC prima di procedere con la lettura del valore convertito. Il bit ADSC nel registro ADCSRA (ADC Start Conversion) viene impostato a 1 quando viene avviata una conversione ADC. Durante la conversione, il microcontrollore esegue continuamente la conversione finché il bit ADSC rimane alto. Quando la conversione è completata, il bit ADSC viene automaticamente azzerato dal microcontrollore.
- In conclusione viene letto il valore digitalizzato sul registro ADC e viene salvato sulla variabile globale `ldrValue`.

4.3.2 Generazione segnale PWM

Come già accennato al paragrafo 4.3, il sistema deve essere in grado di generare un segnale PWM il cui duty cycle dipenda dalla luminosità rilevata.

E' bene sottolineare che non tutti i pin digitali (sia PORTB che PORTD) sono in grado di generare un segnale PWM.

Timer	Frequenza Base	Output Compare		Pin Arduino	Frequenza Default
TCCRO	62500Hz	OC0	OC0A	6	976Hz
	62500Hz		OC0B	5	976Hz
TCCR1	31250Hz	OC1	OC1A	9	488Hz
	31250Hz		OC0B	10	488Hz
TCCR2	31250Hz	OC2	OC1A	11	488Hz
	31250Hz		OC2B	3	488Hz

Figure 11: Frequenze segnale PWM su Arduino Uno

Come si può vedere solo i pin 3,5,6,9,10,11, sono predisposti per generare un segnale PWM. Sia perchè gli altri pin della scheda arduino sono impegnati ad ospitare altri moduli e sia perchè, come si vedrà più avanti nell'analisi, il TIMER1 e TIMER2 sono impegnati ad assolvere altre funzioni, è stato scelto di procedere con l'utilizzo del PD5 che fa riferimento al TCCRO.

Una volta individuato il timer interno da utilizzare si può procedere con la stesura del codice:

- Nel `setup()`:

```

1 TCCROA = _BV(COM0B1) | _BV(WGM01) | _BV(WGM00); //impostazione del PWM
2 TCCROB = _BV(CS01); //impostazione della frequenza di clock a 64
3

```

- `_BV(COM0B1)` abilita il bit COM0B1 nel registro TCCROA. Questo bit specifica la modalità di uscita del segnale PWM sul pin di output specifico. La configurazione specifica COM0B1 indica che il segnale

PWM verrà fornito al pin `OCR0B`. `_BV(WGM01)` e `_BV(WGM00)` abilitano rispettivamente i bit `WGM01` e `WGM00` nel registro `TCCR0A`. Questi bit determinano la modalità di funzionamento del timer hardware utilizzato per generare il segnale PWM. La configurazione specifica `WGM01` e `WGM00` corrisponde alla modalità di generazione di PWM a 8 bit. Nella tabella che segue si possono vedere tutte le possibili configurazioni:

<code>WGM00</code>	<code>WGM01</code>	<code>Timer0 mode selection bit</code>
0	0	Normal
0	1	CTC (Clear timer on Compare Match)
1	0	PWM, Phase correct
1	1	Fast PWM

Figure 12: TIMER0 mode selection bit

- `_BV(CS01)` abilita il bit `CS01` nel registro `TCCR0B`. Questo bit rappresenta il prescaler del timer hardware utilizzato per controllare la frequenza di clock. Quindi, questa riga di codice imposta il registro `TCCR0B` in modo da impostare il prescaler a 64. Questo determina la frequenza di clock del timer a una frequenza predefinita nel microcontrollore, che influenzera la frequenza del segnale PWM generato.

Tale frequenza è calcolabile con la formula (3) come segue:

$$\frac{CPU\ frequency}{prescaler} = \frac{16000000}{64} = 250000Hz = 250kHz \quad (4)$$

- Nel `loop()`:

```
1 OCR0B = map(ldrValue, 0, 1023, 255, 0); //conversione del valore letto in un valore PWM
2
```

- Come dichiarato prima, il valore PWM dovrà essere fornito al pin `OCR0B`. Questo pin richiede un valore con una profondità di 8 bit per generare il segnale, ovvero un intervallo di valori compresi tra $[0, 255]$ dato che $2^8 - 1 = 255$. Il valore del sensore LDR come visto al paragrafo 4.3.1 è stato discretizzato con una profondità di 10 bit, ovvero con 1023 valori possibili ($2^{10} - 1 = 1023$). Per questa motivazione il valore ottenuto dalla conversione (`ldrValue`) dovrà necessariamente essere mappato, per fare ciò viene utilizzata la funzione `map()` della libreria base di Arduino, come nel codice appena mostrato.
- Quello che succede all'interno del sistema è quindi che il valore mappato andrà a regolare, tramite il registro `OCR0B`, il duty cycle del segnale PWM, vale a dire: l'ampiezza del segnale analogico che si desidera rappresentare. Il duty cycle in un segnale PWM rappresenta quindi la percentuale di

tempo in cui il segnale è alto rispetto al periodo totale e viene utilizzato per controllare l'ampiezza o l'intensità di un segnale analogico, può variare da 0% (segnalet sempre basso) al 100% (segnalet sempre alto). Ad esempio, se il duty cycle è del 50%, significa che il segnale è alto per la metà del periodo e basso per l'altra metà.

Il tutto è facilmente comprensibile tramite la seguente immagine:

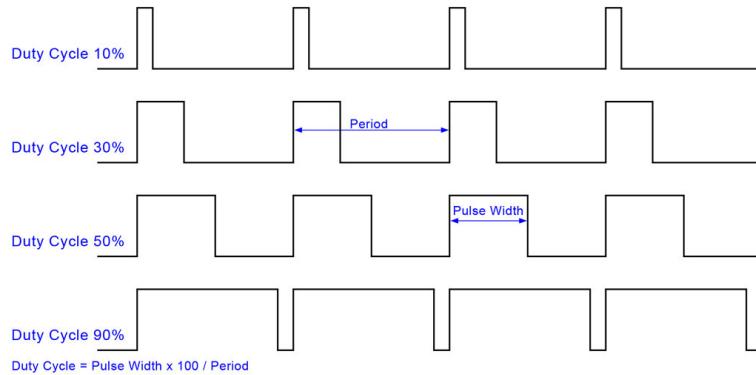


Figure 13: Duty-Cycle di un segnale PWM

- Si può notare che il valore è stato però mappato "al contrario", cioè perchè come visionabile nella tabella 10, al crescere della luminosità il valore letto sarà molto piccolo e di conseguenza al diminuire della luminosità il valore letto sarà molto alto.

Ai fini del progetto però dovrà essere esattamente l'opposto, quando sarà rilevata poca luce dovrà essere generato un voltaggio molto piccolo, dato che nella pratica non ci sarà bisogno di troppo contrasto per leggere il display. E al contrario quando ci sarà molta luce, sarà difficoltoso leggere il display e quindi andrà aumentato il contrasto (e quindi il voltaggio).

A questo punto il sengale PWM, in dipendenza dal duty cycle è stato correttamente generato, come detto prima però non si può avere un effettivo riscontro dato che il display LCD, nell'ambiente Wokwi non permette di simulare il contrasto.

Per assolvere al problema è stato utilizzato un LED collegato sempre al PD5.

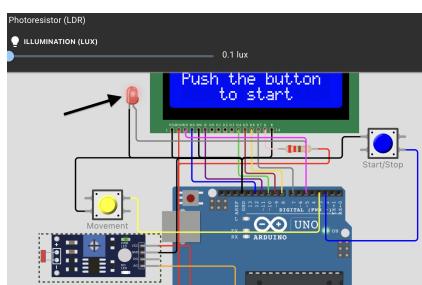


Figure 14: Poca luce, poco contrasto

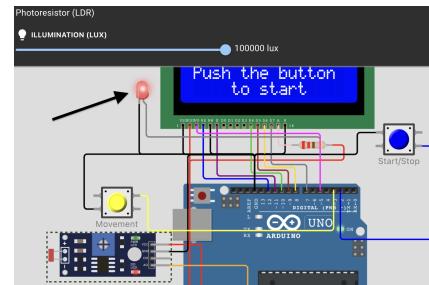


Figure 15: Tanta luce, tanto contrasto

4.4 Gestione del display LCD

Il display LCD 16x2 visto al paragrafo 3.2 è solitamente gestito tramite la libreria "Liquid Crystal", da importare nel proprio editor.

In questo progetto non è stata utilizzata alcuna libreria per la gestione del display ma bensì sono state create delle funzioni ad hoc per la scrittura dei dati sul display e per l'invio dei comandi al display.

4.4.1 sendData()

E' la funzione principale su cui si basano tutte le altre funzioni secondarie del display.

Questa funzione invia un singolo byte di dati al display LCD. Prima di inviare i dati, accende il display impostando il pin EN (Enable) a basso. Successivamente, i bit più significativi (i bit 4,5,6,7) del byte data vengono inviati ai pin D4, D5, D6 e D7 del display, che sono collegati ai pin del microcontrollore Arduino. I bit vengono inviati in sequenza utilizzando operazioni di shift e maschere bitwise. Dopo aver inviato i 4 bit più significativi, il pin EN viene impostato ad alto per indicare la fine dell'invio dei dati. Successivamente, il pin EN viene riportato a basso per accendere il display nuovamente. Infine, vengono inviati i 4 bit meno significativi (i bit 0,1,2,3) dello stesso byte data utilizzando la stessa procedura.

In sintesi, la funzione sendData si occupa di impostare i pin di dati del display LCD (D4-D7) con i bit appropriati del byte data, consentendo al display di ricevere e visualizzare il dato corrispondente. Inoltre, la funzione controlla il pin di controllo EN (Enable) per accendere e spegnere il display durante la trasmissione dei dati.

Il codice di questa funzione è il seguente:

```
1 void sendData(byte data) {
2     PORTB &= ~BV(3); //EN pin su LOW -> Accensione del display
3
4 // Invio dei 4 bit piu significativi, primo nibble
5 // Imposta il pin D4 con il bit meno significativo di 'data'
6     PORTB = (PORTB & ~BV(pinD4)) | (((data >> 4) & 1) << 2);
7 // Imposta il pin D5 con il secondo bit meno significativo di 'data'
8     PORTB = (PORTB & ~BV(pinD5)) | (((data >> 5) & 1) << 1);
9 // Imposta il pin D6 con il terzo bit meno significativo di 'data'
10    PORTB = (PORTB & ~BV(pinD6)) | (((data >> 6) & 1) << 0);
11 // Imposta il pin D7 con il quarto bit meno significativo di 'data'
12    PORTD = (PORTD & ~BV(pinD7)) | (((data >> 7) & 1) << 7);
13
14    PORTB |= BV(pinEN); //EN pin su HIGH -> Spegnimento del display
15    PORTB &= ~BV(pinEN); //EN pin su LOW -> Accensione del display
16
17 // Invio dei 4 bit meno significativi, secondo nibble
```

```

18 // Imposta il pin D4 con il bit meno significativo di 'data'
19 PORTB = (PORTB & ~_BV(pinD4)) | (((data >> 0) & 1) << 2);
20 // Imposta il pin D5 con il secondo bit meno significativo di 'data'
21 PORTB = (PORTB & ~_BV(pinD5)) | (((data >> 1) & 1) << 1);
22 // Imposta il pin D6 con il terzo bit meno significativo di 'data'
23 PORTB = (PORTB & ~_BV(pinD6)) | (((data >> 2) & 1) << 0);
24 // Imposta il pin D7 con il quarto bit meno significativo di 'data'
25 PORTD = (PORTD & ~_BV(pinD7)) | (((data >> 3) & 1) << 7);
26
27 PORTB |= _BV(3); //EN pin su HIGH -> Spegnimento del display
28 }

```

4.4.2 sendCommand()

Questa funzione invia un comando al display LCD. In particolare, imposta il pin RS (Register Select) a basso per indicare che si sta inviando un comando (non dei dati). Il parametro command rappresenta il comando da inviare.

Il codice è il seguente:

```

1 void sendCommand(byte command) {
2     PORTB &= ~_BV(pinRS); // Impostazione del pin RS a basso per i comandi
3     sendData(command);
4 }

```

Nel concreto quello che fa la funzione è impostare tutti i parametri necessari al corretto funzionamento del display (posizionamento cursore, pulizia display, accensione display, ecc...)

4.4.3 sendText()

La funzione `sendText(const char* text)`, dopo aver impostato il pin RS ad alto per indicare al display che sono in arrivo dei caratteri e non dei comandi, utilizza un puntatore text per accedere ai caratteri all'interno della stringa. Ad ogni iterazione del while, il carattere corrente puntato da text viene inviato al display LCD utilizzando la funzione `sendData()`. Quindi, il puntatore text viene incrementato per puntare al carattere successivo nella stringa. Questo processo continua fino a quando viene raggiunto il carattere nullo ('\0'), che indica la fine della stringa.

Ciò è possibile perché i tipi `char` nel linguaggio C sono composti da 8 bit, quindi "traducibili" dalla funzione `sendData(byte data)` che ha punto come parametro un byte (8 bit).

Il codice della funzione è il seguente:

```

1 void sendText(const char* text) {
2     PORTB |= _BV(pinRS); // Impostazione del pin RS ad alto per i dati

```

```

3   while (*text) {
4     sendData(*text);
5     text++;
6   }
7 }
```

4.4.4 sendNum()

La funzione `sendNum(volatile float num)` converte un numero in formato floating-point (float) in una stringa e lo invia al display LCD. Utilizza la funzione `dtostrf` per effettuare la conversione del numero in una stringa di caratteri. Il numero viene formattato con una cifra totale e nessuna cifra decimale. La stringa risultante viene quindi inviata al display utilizzando la funzione `sendText`.

D'altra parte la funzione `sendNumFloat(volatile float num)` è molto simile alla funzione `sendNum()` ma formatta il numero con cinque cifre totali e due cifre decimali. La stringa risultante viene sempre inviata al display utilizzando la funzione `sendText`.

Il codice di entrambe le funzioni è il seguente:

```

1 void sendNum(volatile float num) {
2   char buffer[10];
3   dtostrf(num, 1, 0, buffer); // Converte il numero in formato stringa con 5 cifre totali e 2
4   decimali
5   sendText(buffer);
6 }
7 void sendNumFloat(volatile float num) {
8   char buffer[10];
9   dtostrf(num, 5, 2, buffer); // Converte il numero in formato stringa con 1 cifra totale
10  sendText(buffer);
11 }
```

4.5 Gestione del tempo

Come evidenziato nella formula (1) e ai paragrafi 4.2.1 e 4.1.1 e come si evidenzierà poi, è fondamentale conoscere il tempo in millisecondi in cui si trova la macchina in un dato istante.

Per evitare l'utilizzo della funzione `millis()` o `micros()` sono stati utilizzati i timer interrupt interni ad Arduino per una maggiore efficienza del sistema.

Come già visto il `TIMER0` è già stato impiegato per generare il segnale PWM al paragrafo 4.3.2.

Restano quindi `TIMER1` e `TIMER2`.

4.5.1 TIMER/COUNTER 1

Lo scopo del Timer/Counter 1 sarà quello di incrementare una variabile `seconds`, appunto ogni secondo, per ottenere un valore di tempo quanto più preciso possibile nell'ordine dei secondi. Ciò sarà necessario in un secondo momento, quando vorrà poi essere mandato a schermo un "orologio" che mostri il tempo in HH:MM:SS.

Come si può leggere dal datasheet di ATM328p per il setup del Timer/Counter 1 è necessario andare a controllare i seguenti registri:

TCCR2A – Timer/Counter Control Register A

Bit	7	6	5	4	3	2	1	0	
(0xB0)	COM2A1	COM2A0	COM2B1	COM2B0	–	–	WGM21	WGM20	TCCR2A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

TCCR2B – Timer/Counter Control Register B

Bit	7	6	5	4	3	2	1	0	
(0xB1)	FOC2A	FOC2B	–	–	WGM22	CS22	CS21	CS20	TCCR2B
Read/Write	W	W	R	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

TIMSK2 – Timer/Counter2 Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0	
(0x70)	–	–	–	–	–	OCIE2B	OCIE2A	TOIE2	TIMSK2
Read/Write	R	R	R	R	R	R/W	R/W	R/W	

TIFR2 – Timer/Counter2 Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0	
0x17 (0x37)	–	–	–	–	–	OCF2B	OCF2A	TOV2	TIFR2
Read/Write	R	R	R	R	R	R/W	R/W	R/W	

Figure 16: TIMER/COUNTER 2 Register

(Nell'immagine sono specificati i registri e i bit per il timer/counter 2, nel timer1 la denominazione dei registri rimane la stessa, cambia solo il "numero" all'interno del bit/registro, es: TCCR1A, TCCR1B, ecc...)

Per inizializzare ed impostare il timer/counter è necessario quindi andare a controllare i vari bit dei registri in figura 16

Nel dettaglio:

```

1 TCCR1A = 0;
2 TCCR1B = 0;
3 TCNT1 = 0;
```

Le prime due righe di codice ci permettono di andare a ad azzerare i due registri di controllo. Accedendo al registro TCNT1 invece, si va ad azzerare il conteggio del timer.

```

1 OCR1A = 0x3D08;
2 TCCR1B |= (1 << WGM12);
3 TCCR1B |= (1 << CS12) | (1 << CS10);
```

```
4 TIMSK1 |= (1 << OCIE1A);
```

- assegnando al registro **OCR1A** (Output Compare Register 1A) il valore di confronto 0x3D08 (che corrisponde al valore 15624_{10}), ciò indica il valore a cui il timer verrà confrontato.

Il valore non è casuale ma frutto di un'equazione:

$$OCR1A = \frac{CPU\ frequency}{prescaler * InterruptFrequency} - 1 = \frac{16000000Hz}{1024 * 1Hz} - 1 = 15624 \quad (5)$$

$1Hz = 1s$

- Controllare il bit **WGM12** del registro **TCCR1B** è necessario nel momento in cui si ha bisogno di confrontare il valore del timer con il valore del registro **OCR1A**.
- Come visto nell'equazione (5), il prescaler per ottenere il risultato desiderato deve essere impostato a 1024 è bene leggere la documentazione e capire come proseguire:

Table 14-9. Clock Select Bit Description

CS02	CS01	CS00	Description
0	0	0	No clock source (Timer/Counter stopped)
0	0	1	$clk_{I/O}/(no\ prescaling)$
0	1	0	$clk_{I/O}/8$ (from prescaler)
0	1	1	$clk_{I/O}/64$ (from prescaler)
1	0	0	$clk_{I/O}/256$ (from prescaler)
1	0	1	$clk_{I/O}/1024$ (from prescaler)
1	1	0	External clock source on T0 pin. Clock on falling edge.
1	1	1	External clock source on T0 pin. Clock on rising edge.

Figure 17: Prescaler

E' quindi necessario andare a modificare i bit **CS12** e **CS10** del registro **TCCR1B**, affinchè si ottenga la stringa di bit 101.

- Infine modificare il bit **OCIE1A** del registro **TIMSK1** abilita l'interrupt per il Timer/Counter 1 ogni volta che il valore del registro **TCNT1** (conteggio del timer) corrisponde al valore di **OCR1A**.

4.5.2 ISR(TIMER1_COMPA_vect)

Nell' Interrupt Service Routine riferente a questo Timer/Counter c'è solo bisogno di incrementare di 1 la variabile **seconds** come già detto in precedenza, solo se la condizione **startStopButtonPressed** è verificata, vale a dire: se il pulsante di avvio visto nel paragrafo 4.2 è stato premuto, ovvero se il programma è avviato, in caso contrario infatti, non c'è alcun bisogno di far partire l'orologio, dato che l'utente ancora non ha interagito e quindi non è pronto alla marcia.

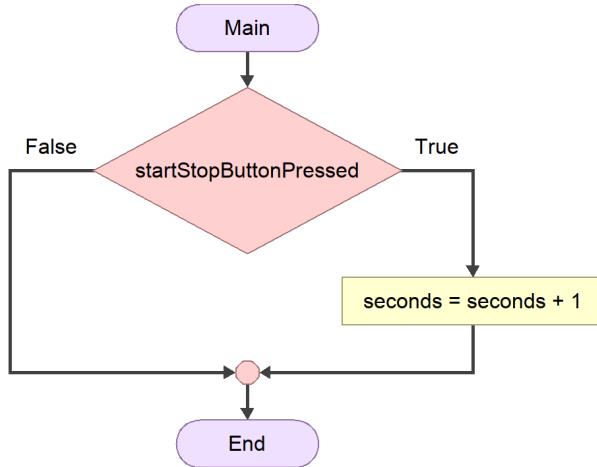


Figure 18: Diagramma di flusso ISR(TIMER1_COMPA_vect)

Il funzionamento di questa ISR segue il diagramma di flusso:

Il codice di riferimento è il seguente:

```

1 ISR(TIMER1_COMPA_vect) {
2     if(startStopButtonPressed){seconds++;}
3 }
```

4.5.3 TIMER/COUNTER 2

Per il Timer/Counter 2, si procedere in modo analogo al Timer/Counter 1.

E' bene sottolineare che la differenza sostanziale tra il Timer/Counter 1 e 2, è che il primo è un timer a 16 bit, ciò significa che può memorizzare un valore massimo di $2^{16} = 65535$, mentre il secondo arriva a 8 bit, quindi $2^8 = 256$.

Il timer/counter 1 è stato usato per il calcolo dei secondi dato che $2^8 < 15624 < 2^{10}$, esso non poteva essere utilizzato per il timer/counter 2 dato che il valore di confronto era più grande rispetto al valore massimo memorizzabile da quel timer/counter.

Lo scopo del Timer/Counter 2 è quello di incrementare la variabile `milliseconds` ogni, appunto, millisecondo.

Per fare ciò si procede in modo simile al Timer/Counter 1 ma con prescaler e valore di confronto diversi, ed ovviamente anche nome di bit e registri.

$$OCR2A = \frac{CPU\ frequency}{prescaler * InterruptFrequency} - 1 = \frac{16000000\ Hz}{64 * 1000\ Hz} - 1 = 249 \quad (6)$$

Il codice per il setup del Timer/Counter 2 è intuitivamente il seguente:

```
1 TCCR2A = 0;
2 TCCR2B = 0;
3 TCNT2 = 0;
4 OCR2A = 249;
5 TCCR2A |= (1 << WGM21);
6 TCCR2B |= (1 << CS22);
7 TIMSK2 |= (1 << OCIE2A);
```

4.5.4 ISR(TIMER2_COMPA_vect)

L'ISR in questo caso prevede solo l'incremento della variabile `milliseconds` senza verificarsi di alcuna condizione, dato che non necessario.

Segue quindi un Flowchart di questo tipo:

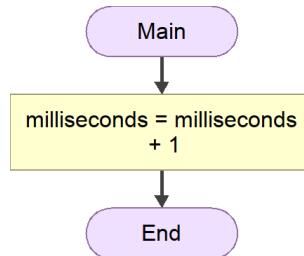
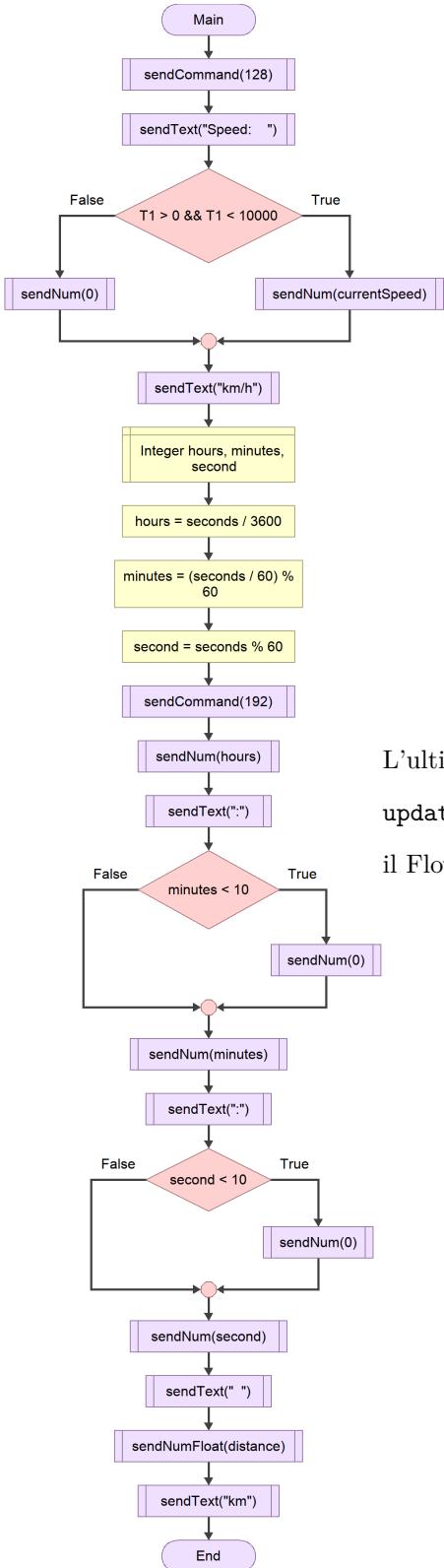


Figure 19: Diagramma di flusso ISR(TIMER2_COMPA_vect)

Ed il codice del tipo:

```
1 ISR(TIMER2_COMPA_vect) {
2     milliseconds++;
3 }
```

4.6 Aggiornamento del display



L'ultima funzione per raccogliere tutti i dati raccolti e mostrarli a schermo è updateLCD(), non c'è necessità di spendere troppe parole a riguardo, dato che il Flowchart che segue è già di per sé esplicativo:

4.7 Funzionamento generale del sistema

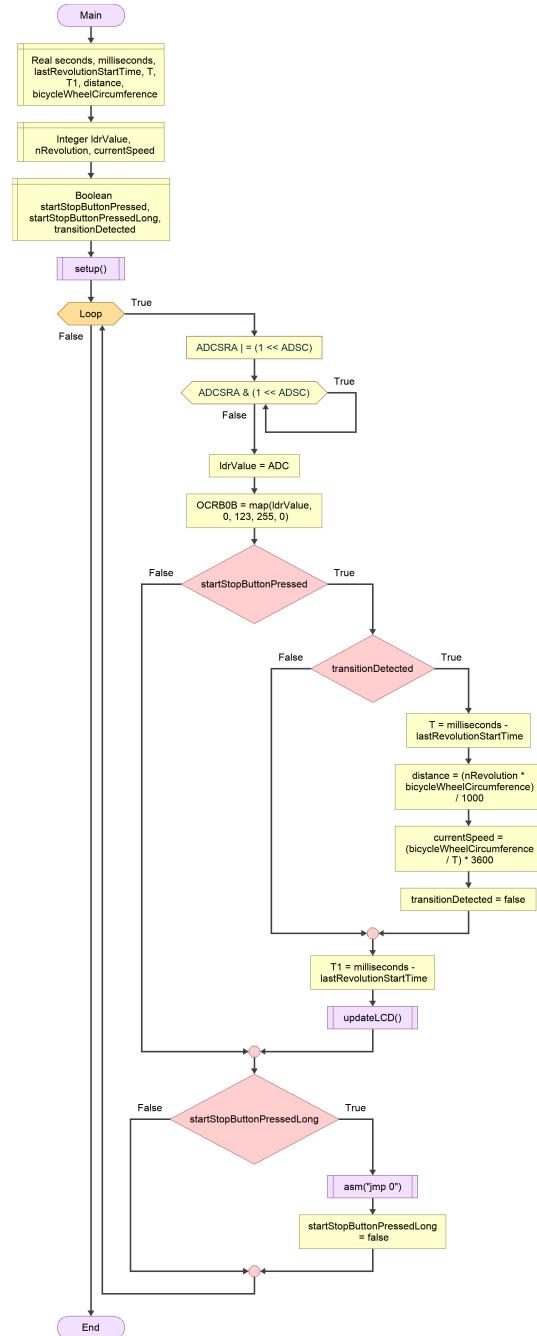


Figure 20: Diagramma di flusso generale del funzionamento del sistema

Per il codice del progetto completo e per vedere il progetto funzionante il link è il seguente:

<https://wokwi.com/projects/370181612744975361>

5 Stati dell'automa

Lo statechart della macchina può essere disamminato in più parti come si vede nella seguente immagine:

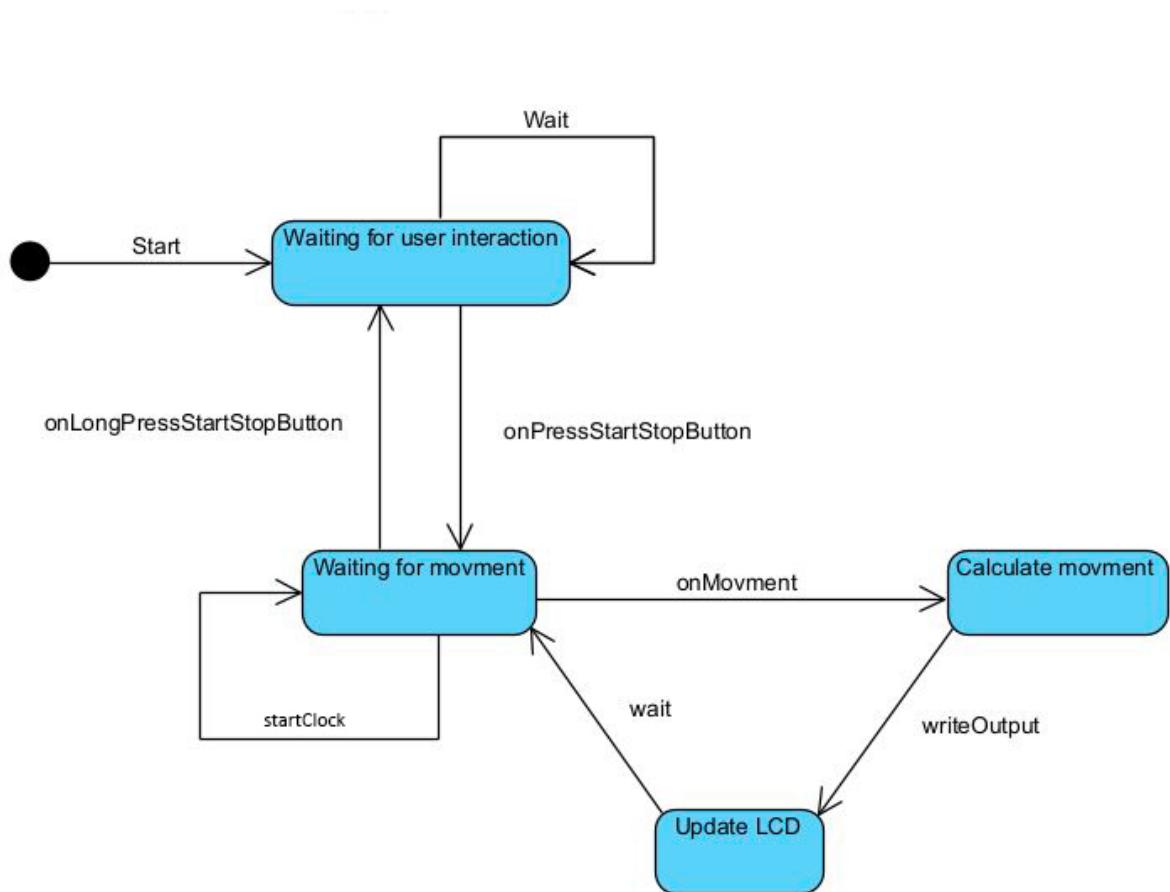


Figure 21: Statechart dell'atuoma

- All'**avvio** della macchina essa è inattiva e si trova in uno stato di attesa.
- se c'è **interazione** da parte dell'utente, ovvero se viene premuto il pulsante di start allora si entra in un altro stato di attesa, in cui l'utente si ha avviato l'automa ma esso aspetta ancora che esso incominci la marcia, nel frattempo il tempo inizia a scorrere, ma gli altri dati non si aggiornano ancora.
- se viene rilavato **movimento** allora si procede con il calcolo del movimento
- i dati raccolti vengono **mostrati** a schermo.
- si **attende** nuovamente l'interazione dell'utente o la marcia.
- se avviene una **lunga pressione** l'automa ritorna allo stato di attesa precedente (setup()).

6 Funzioni Logiche

Riguardando i diagrammi di flusso 20 e lo state chart 21 è possibile, facendo riferimento all'algebra booleana capire a da quali funzioni logiche il funzionamento del sistema dipende.

Si distinguono due possibili scenari:

6.1 Automa ancora in fase di setup

Come visto nell'analisi e nel diagramma degli stati, l'automa appena avviato si trova in uno stato iniziale di attesa di interazione.

PD2	PD3	SETUP
0	0	0
0	1	0
1	0	0
1	1	1

Table 1: Automa in fase di setup

Come già detto l'automa se il PD2 è HIGH e così anche il PD3, vale a dire che sia il push button di start/stop visto al paragrafo 4.2 che il pushbutton che simula il movimento 4.1 sono attualmente inattivi (non premuti), e quindi non c'è interazione e la macchina resta nel setup.

Questo stato corrisponde ad una funzione logica AND:

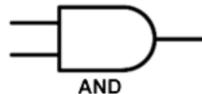


Figure 22: Funzione logica AND

6.2 Automa in fase di movimento

L'automa si trova in uno stato di movimento se esso è uscito dal setup e se c'è un effettivo rilevamento di movimento, ovvero se si è in marcia.

PD2	PD3	MOVEMENT
0	0	1
0	1	0
1	0	0
1	1	0

Table 2: Automa in fase di movimento

In questo caso quindi PD2 e PD3 saranno entrambi su **LOW**, vale a dire che il push button di start/stop è stato premuto e il push button di movimento anche.

Questo stato è quindi corrispondente alla funzione logica NOR

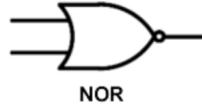


Figure 23: Funzione logica NOR

7 Variabili e Costanti

Per una più efficiente gestione della memoria si può procedere con la modifica di alcuni tipi di variabili e costanti all'interno del codice.

7.1 Variabili

Per quanto riguarda le variabili, quello che bisogna fare è andare a contestualizzare la variabile all'interno del codice e capire che valori può assumere quella variabile e di conseguenza assegnare un determinato tipo, si può passare da questo codice:

```

1 int ldrValue = 0;
2 volatile bool startStopButtonPressed = false;
3 volatile bool startStopButtonPressedLong = false;
4 volatile unsigned long buttonPressStartTime = 0;
5 volatile unsigned long transitionTime = 0;
6 volatile bool transitionDetected = false;
7 volatile int seconds = 0;
8 volatile unsigned long milliseconds = 0;
9 float lastRevolutionStartTime = 0;
10 volatile int nRevolution = 0;
11 volatile float distance = 0;
12 volatile unsigned int currentSpeed = 0;
13 volatile float T = 0;
14 volatile float T1 = 0;
```

a questo:

```

1 uint16_t ldrValue = 0;
2 volatile bool startStopButtonPressed = false;
3 volatile bool startStopButtonPressedLong = false;
4 volatile unsigned long buttonPressStartTime = 0;
5 volatile unsigned long transitionTime = 0;
```

```

6 volatile bool transitionDetected = false;
7 volatile long seconds = 0;
8 volatile long milliseconds = 0;
9 float lastRevolutionStartTime = 0;
10 volatile int nRevolution = 0;
11 volatile float distance = 0;
12 volatile uint8_t currentSpeed = 0;
13 volatile float T = 0;
14 volatile float T1 = 0;

```

Le modifiche apportate riguardano le variabili:

- **ldrValue**: si passa da `int` (*4byte = 32bit*) a `uint16_t`, questo è possibile perché `ldrValue` come visto al paragrafo 4.3 può assumere valori compresi tra 0 e 1023, e `uint16_t` arriva fino a 65535.
- **currentSpeed**: si passa in questo caso da `int` a `uint8_t`, 255 valori sono sufficienti per rappresentare la velocità massima, ovvero $255km/h$, irraggiungibile per una bicicletta.

Per quanto riguarda le altre variabili purtroppo si è costretti a rimanere con i tipi prima attribuiti, dato che ad esempio la variabile `nRevolution` non può essere rappresentata con `uint16_t` dato che ciò comporterebbe un valore massimo, come già detto, di 65535, non propriamente sufficiente per questo dato. Ciò corrisponde alla percorrenza massima di circa $140km$, dato che:

$$distanza = nRevolution * bicycleWheelCircumference = 65535 * 2,1206 \cong 140km \quad (7)$$

7.2 Costanti

Per quanto riguarda le costanti sono le seguenti:

```

1 const int PWM_PIN = 5;
2 const int pinRS = 4;
3 const int pinEN = 3;
4 const int pinD4 = 2;
5 const int pinD5 = 1;
6 const int pinD6 = 0;
7 const int pinD7 = 7;
8 const int startStopButton = 2;
9 const int movementPin = 3;
10 const float bicycleWheelCircumference = 2.1206;

```

è possibile migliorare in questo modo il codice:

```

1 const uint8_t PWM_PIN = 5;
2 const uint8_t pinRS = 4;

```

```

3 const uint8_t pinEN = 3;
4 const uint8_t pinD4 = 2;
5 const uint8_t pinD5 = 1;
6 const uint8_t pinD6 = 0;
7 const uint8_t pinD7 = 7;
8 const uint8_t startStopButton = 2;
9 const uint8_t movementPin = 3;
10 const float bicycleWheelCircumference = 2.1206;

```

Come già detto prima essendo valori compresi nel range [0-255] è possibile rappresentare tali valori tramite `uint8_t`.

Per una maggiore efficienza si potrebbe pensare anche di sostituire:

```
1 const float bicycleWheelCircumference = 2.1206;
```

con:

```
1 const uint16_t bicycleWheelCircumference_mm = 2121;
```

Ovvero esprimere il valore della circonferenza in mm per utilizzando il tipo `uint16_t`. Ovviamente a questo andrebbe aggiunta la modifica della formula (1) e (2).

Tali variabili giocano un ruolo fondamentale nella comprensione del codice quando si vanno ad impostare i vari pin nel `setup()`.

```

1 // Inizializzazione dei pin del display come output
2 DDRB |= (1 << pinRS);
3 DDRB |= (1 << pinEN);
4 DDRB |= (1 << pinD4);
5 DDRB |= (1 << pinD5);
6 DDRB |= (1 << pinD6);
7 DDRD |= (1 << pinD7);
8
9 DDRD |= (1 << PWM_PIN); //Abilita il PWM_PIN come output
10
11 // Abilita il pushButton Movement come input-pullup
12 DDRD &= ~(1 << movementPin);
13 // Imposta il pull-up interno
14 PORTD |= (1 << movementPin);
15
16 //Abilita i pin del pushButton Start come input-pullup
17 // Imposta il pin come input
18 DDRD &= ~(1 << startStopButton);
19 // Imposta il pull-up interno
20 PORTD |= (1 << startStopButton);

```

8 Potenziali miglioramenti hardware del sistema

E' bene sottolineare che è stato utilizzato un push button per simulare il movimento, per necessità di utilizzare un'interrupt generato da tale modulo, e soprattutto l'ambiente di simulazione Wowki non fornisce i sensori adatti per questo compito, difatti nella realtà dei fatti il meccanismo non potrebbe essere replicato.

Alcuni sensori decisamente più pertinenti (capaci comunque di essere controllati tramite routine di interrupt), sono i seguenti:

- **Sensore ad effetto Hall:** Questo sensore sfrutta un principio elettromagnetico noto come effetto Hall. I sensori a effetto Hall rilevano la presenza, l'intensità e la polarità di un campo magnetico ed emettono un segnale di uscita variabile in funzione di tali parametri.
Posizionando quindi tale sensore sulla forcella della bicicletta e posizionando un magnete (in modo tale da poter permettere la variazione di campo elettrico) sul raggio della ruota ad esempio, è facilmente possibile replicare, in modo decisamente più attendibile, il mio stesso esperimento
- **Sensore IR:** Con tale sensore, anche noto come sensore ad infrarossi, è possibile sfruttare il concetto della variazione di luce, tramite il posizionamento del nastro come nell'immagine che segue, in questo caso non sarà tanto la variazione della luce emessa dal materiale ad influire sulla variazione di stato del sensore ma bensì la temperatura emessa dal materiale.



Figure 24: Render della potenziale disposizione del nastro

N.B. Il sistema risulta comunque essere quasi interamente retro-compatibile con gli altri due sensori citati, se non per eventuali minime variazioni di codice e wiring.