

Relazione finale del corso - modulo due

SALMO - Solar Azimuth and eLevation Motorized IOcator

Corso di Progettazione e Prototipazione
di Sistemi Elettronici



Relazione finale del corso - modulo due

SALMO - Solar Azimuth and eLlevation Motorized IOcator

by

SALMO SOCIETY

Student Name	Student Number
Lisa Santarossa	209386
Thomas Nonis	209445
Tommaso Canova	209270
Simone Tollardo	209002
Gabriele Berretta	209466

Docente: M. Corrà

Dipartimento: Dipartimento di Ingegneria e Scienza dell'Informazione

Contents

1 Abstract	1
2 Analisi dello schematico	2
2.1 Control	3
2.2 Power	10
2.2.1 Input 5V	11
2.2.2 Input 12V	11
2.2.3 Regolatore di tensione	12
2.3 Peripherals	14
2.4 Panel sensing	18
2.5 Actuation	21
2.6 Finalizzazione schematico	23
3 Analisi del Layout PCB	24
3.1 Layout	24
3.2 Routing	26
3.3 Finitura	28
4 Firmware	30
4.1 Ambiente di sviluppo	30
4.2 Compilazione	30
4.3 Algoritmo per ottenere la posizione del sole	32
4.4 Implementazione software delle periferiche	34
4.5 Implementazione driver motori	36
4.6 Funzionamento generale del codice	38
5 Assemblaggio della scheda	40
6 Testing della scheda	46
7 Implementazioni future	48
8 Conclusioni	51
8.1 Resoconto esperienza	51
8.2 Lezioni imparate	51
9 Bibliografia	52
10 Allegati	53
10.1 Gerber	53

Abstract

La seconda parte del corso di Progettazione e Prototipazione di Sistemi Elettronici prevede la completa progettazione di una scheda elettronica che, nel nostro caso, ha come scopo quello di orientare in maniera ottimale un pannello fotovoltaico verso la posizione del sole nel cielo.

Per rendere più funzionale possibile "SALMO" si è deciso di rendere il tutto indipendente dalla posizione geografica, dalla collocazione spaziale del pannello fotovoltaico ed ovviamente dall'orario e dal giorno dell'anno. Il movimento del pannello è stato progettato in modo che la sua posizione sia sempre il più possibile perpendicolare ai raggi solari affinché la potenza generata dal pannello possa essere sempre massima.

Per raggiungere l'obiettivo sono state stilate le seguenti specifiche: due motori passo-passo unipolari per il movimento sui due assi (Azimuth e Elevation), GPS per rilevare la posizione geografica, magnetometro ed accelerometro per il feedback della posizione del pannello, circuiti di misura per tensione e corrente del pannello, display oled per la visualizzazione dei dati mediante interfaccia utente ed ovviamente un pannello fotovoltaico.

La progettazione della scheda è stata svolta in maniera precisa, rigorosa ed organizzata, seguendo quindi dei passi ben definiti in modo che ogni membro del team potesse lavorare parallelamente agli altri ed allo stesso tempo contribuire al lavoro di tutti senza perdere step fondamentali.

Inizialmente, facendo uso dei datasheet, abbiamo studiato approfonditamente le caratteristiche del microcontrollore *RP2040* e dei componenti principali della scheda, al fine di poter progettare la *PCB* in maniera più consapevole. Ogni studente del corso ha poi esposto alcune caratteristiche del microcontrollore, a partire dalla memoria fino alla struttura fisica passando per le varie interfacce di comunicazione, l'*ADC*, i *GPI/O* e l'alimentazione.

Ogni gruppo ha poi concordato con il professore il progetto da realizzare, fissando delle scadenze per la consegna dei vari schematici ed infine per la *BOM*.

Sono state decise inoltre delle specifiche di progetto uguali per ogni gruppo (come ad esempio le dimensioni della scheda (10x6 cm), il regolatore di tensione ecc..) in modo da avere delle *BOM* più uniformi possibili, senza decine di componenti alternativi non necessari. Dopo l'avvio vero e proprio del progetto in autonomia, le lezioni svolte in classe o in laboratorio erano principalmente volte alla correzione, alla revisione ed al debugging, facendo sì che ogni dubbio potesse essere chiarito immediatamente.

Finito lo sbroglio circuitale siamo passati alla generazione e all'invio dei file gerber al Professore, che ha successivamente provveduto a spedire questi ultimi al produttore. Analogamente, dopo aver completato la *BOM*, questa è stata inviata al Professore per il conseguente acquisto del materiale necessario. In seguito all'arrivo delle *PCB* e dei componenti essenziali, abbiamo iniziato l'assemblaggio sfruttando la strumentazione offerta dal laboratorio di elettronica presente all'interno del FabLab. Dopo aver completato l'assemblaggio, realizzato gran parte del firmware, svolto test a freddo prima (senza alimentazione) ed a caldo successivamente (con +5V USB e +12V), possiamo affermare che, benché ci siano ancora dei componenti mancanti, il prototipo è completamente funzionante (a meno di un piccolo errore, vedi cap. LED RGB).

Tutto il progetto della *PCB* è stato svolto utilizzando il software open-source KiCad 6.0, mentre per la progettazione del firmware abbiamo utilizzato l'*IDE VS Code* e la compilazione manuale via *CMake*. Per la gestione dell'intero progetto abbiamo deciso di avvalerci di un *VCS* (*Version Control System*): git.

La repository è hostata su GitHub e può essere visionata cliccando sul seguente link:
<https://github.com/thomasnonis/ppse-2021>.

2

Analisi dello schematico

I principali componenti utilizzati per la realizzazione del progetto sono:

- microcontrollore *RP2040*;
- memoria flash *QSPI W25Q16JV*;
- regolatore di tensione *TPS563201*;
- *GPS PAM7Q*;
- magnetometro *HMC5883L*;
- accelerometro e giroscopio *MPU6050*;
- 2 motori stepper unipolari con riduttore *28BYJ (ADAFRUIT 918)*;
- circuiti di misura tensione e corrente;
- display *OLED 128x64*;
- USB type B-mini;
- buzzer;
- led *RGB 1210*.

Per una maggior chiarezza abbiamo deciso di dividere lo schematico in più fogli, ognuno con una funzione ben precisa:

- **Control:** *MCU, Flash, Oscillatore al quarzo, pulsanti ecc..*
- **Power:** Regolatore di tensione, *USB*
- **Peripherals:** GPS, Magnetometro, Accelerometro, *OLED, Buzzer, Led RGB*
- **Panel sensing:** Current sensing, Voltage Sensing
- **Actuation:** Stepper Driver

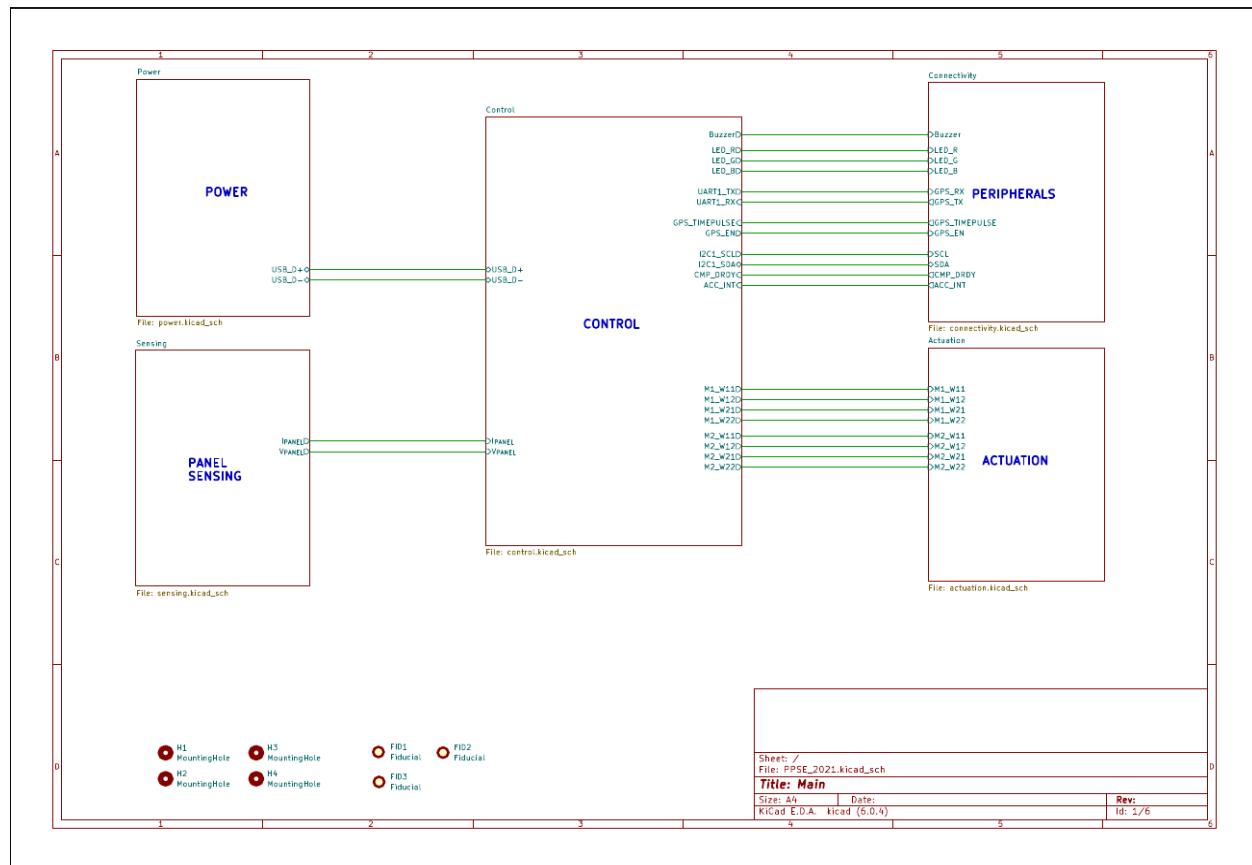


Figure 2.1: Foglio "Main" SALMO

2.1. Control

Il foglio “Control” include i componenti adibiti al controllo della scheda, quali:

- microcontrollore *RP2040*;
- memoria flash SPI *W25Q16JV*;
- cristallo al quarzo a 12 MHz;
- pulsanti di reset, boot, home e tracking enable;
- connettori di debug ed espansione.

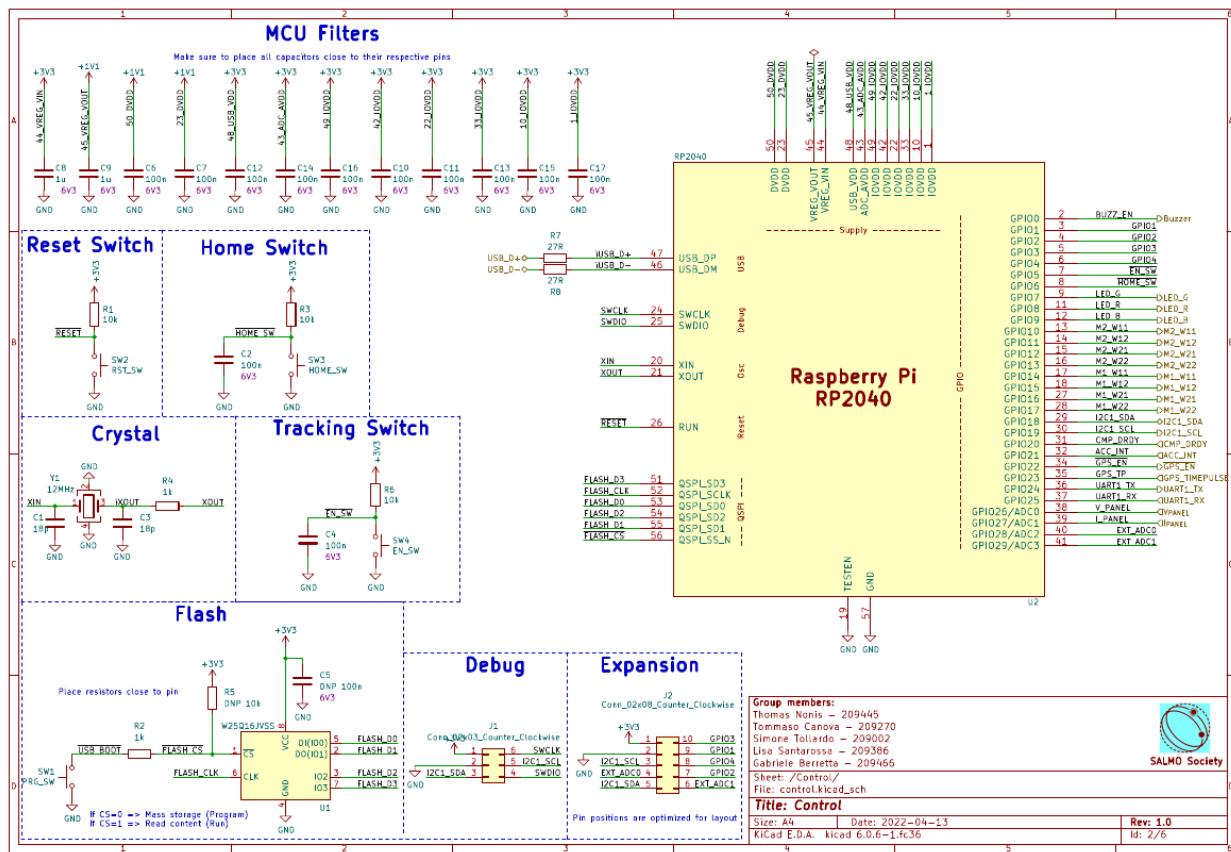


Figure 2.2: Foglio "Control" SALMO

Microcontrollore: RP 2040La scheda si avvale dell'ormai celeberrimo *MCU* di Raspberry Pi Ltd: *RP2040*.

Caratteristiche principali:

- 133 MHz dual core 32 bit ARM Cortex-M0+;
- 264 KB SRAM;
- Nessuna memoria flash interna;
- Controller bus *QSPI*, che supporta fino a 16 MB di memoria flash esterna;
- Controllore *DMA*;
- Crossbar *AHB* (Advanced High-performance Bus);
- Regolatore *LDO* programmabile per generare la tensione del core integrato;
- 2 *PLL* su chip per generare clock USB e core;
- 30 pin *GPIO*, di cui 4 utilizzabili optionalmente come ingressi analogici.

Periferiche:

- 2 *UART*;
- 2 controller *SPI*;
- 2 controller *I²C*;
- 16 canali *PWM*;
- Controller USB 1.1 e *PHY*, con supporto per host e dispositivo;
- 8 macchine a stati di input-output (*PIO*) programmate.

Il microcontrollore ha il compito di gestire le periferiche. In particolare:

- genera il segnale di rotazione dei motori quando previsto dall'algoritmo implementato (*GPIO* da 10 a 17);
- comunica con il GPS tramite interfaccia UART (*GPIO* 24 e 25);
- comunica con il magnetometro, con l'accelerometro e con il display OLED mediante interfaccia I₂C (*GPIO* 18 *SDA*, *GPIO* 19 *SCL*);
- effettua le misure di corrente e di tensione del pannello (*GPIO* 26 e 27);
- gestisce il buzzer (*GPIO* 0) ed un led *RGB* (*GPIO* 7, 8, 9);
- comunica con la memoria *flash* (pin da 51 a 56);
- gestisce l'interfaccia *USB* (pin 46 e 47).

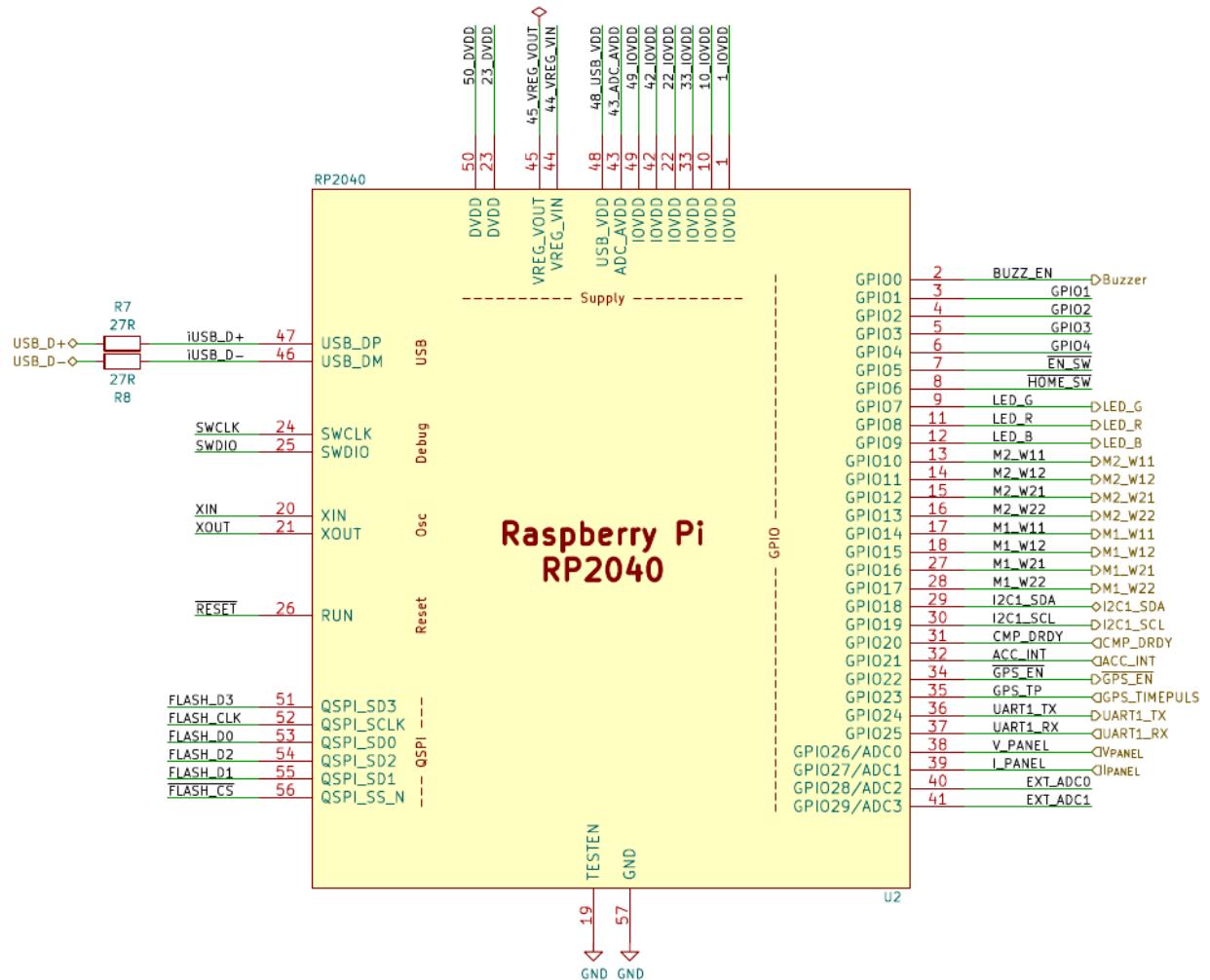


Figure 2.3: Schema pin microcontrollore RP2040

Per ridurre il rumore ad alta frequenza presente sulla linea di alimentazione abbiamo previsto dei condensatori di filtro per cortocircuitare a massa la componente alternata (andranno poi posizionati il più vicino possibile ai pin del MCU).

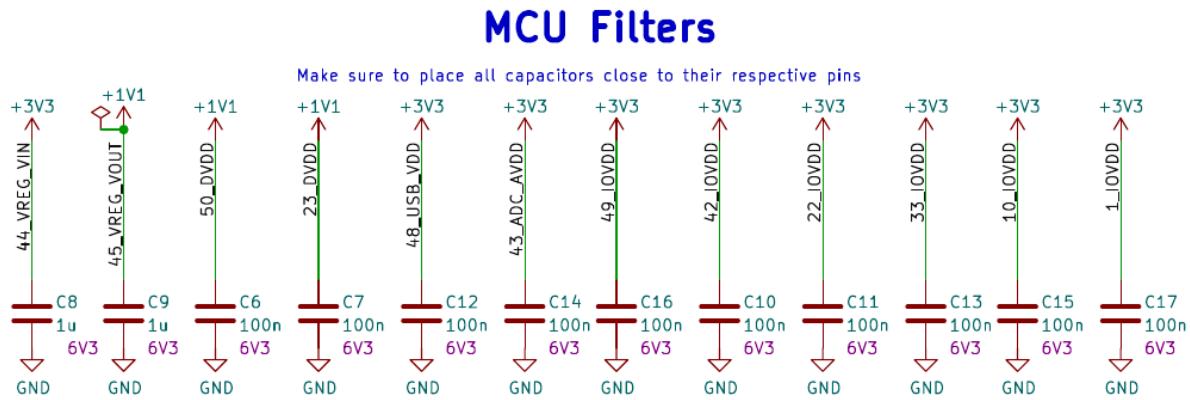


Figure 2.4: Condensatori di bypass MCU

Pulsanti

Per il progetto *SALMO* abbiamo previsto quattro pulsanti tattili:

- Un pulsante di reset (*SW2*) per poter resettare il microcontrollore;
- Un pulsante per portare il microcontrollore in modalità *bootloader* (*SW1*), in modo tale da poter ``flashare'' il programma via USB.
- Un pulsante (*SW3*), denominato "Home", che permette di portare il sistema di puntaamento del pannello solare ad una posizione predefinita (0° Nord, 45° Elevazione);
- Un pulsante (*SW4*), denominato ``Tracking Enable'', che permette di far partire il sistema di puntaamento del pannello;

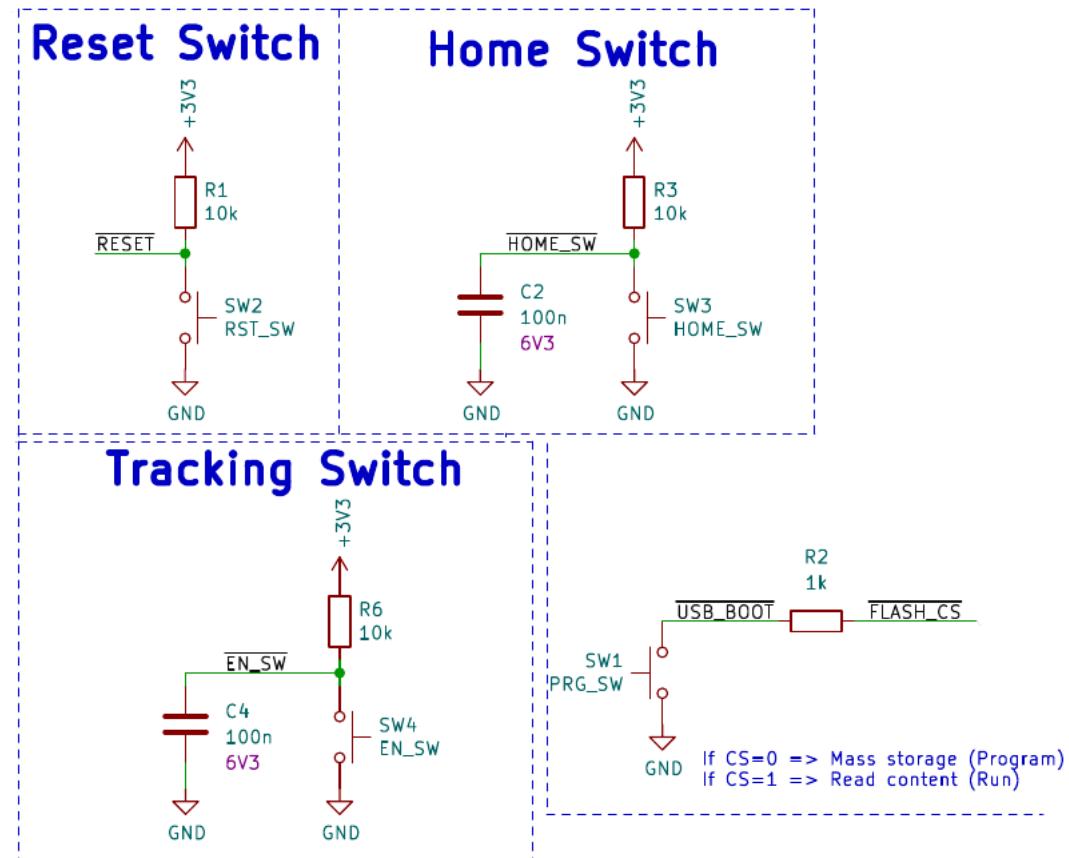


Figure 2.5: Pulsanti SALMO

Per ogni pulsante abbiamo previsto un resistore di pull-up da $10k\Omega$, che nel caso del pulsante di programmazione è contrassegnato come *DNP* (Do Not Place), poiché la memoria fornisce un *pull-up* interno che lo rende non necessario.

Per i pulsanti di *reset* e programmazione non abbiamo previsto alcun filtro di debouncing come consigliato dai relativi datasheet, mentre per i restanti abbiamo aggiunto un condensatore da $100nF$ per introdurre una costante di tempo di $1ms$.

La resistenza R2 in serie al pulsante di programmazione è stata inserita seguendo i consigli del manuale di sviluppo hardware del RP2040 (probabilmente il pin \overline{CS} della flash non ha una resistenza di limitazione interna).

I pulsanti scelti sono dei generici *tactile switch* da 6mm a tecnologia TH .

Quarzo

Per generare il segnale di clock del microcontrollore abbiamo selezionato un quarzo da 12 MHz . In particolare abbiamo utilizzato un quarzo a 4 pin con package 3225, essendo questo uno dei constraint imposti dal professore ad inizio progetto. Per il dimensionamento del resistore in serie e dei condensatori abbiamo seguito le linee guida del manuale di sviluppo hardware del RP2040.

$$C_1 = C_2 = 2 \cdot (C_L - C_{\text{Parassita}}) , \text{ dove } C_L \text{ è la capacità di load ottimale del quarzo.}$$

Ipotizzando quindi una capacità parassita pari a $5pF$ ed una capacità di load di $14pF$ otteniamo $C_1 = C_2 = 18pF$.

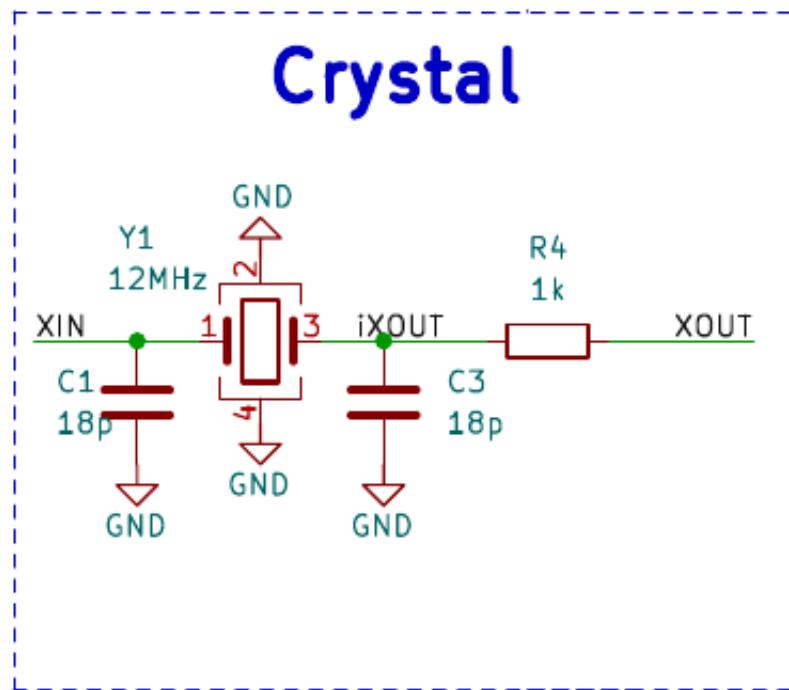


Figure 2.6: Circuito del cristallo di quarzo di SALMO

La capacità di carico è particolarmente importante perché, essendo la capacità totale vista dai due pin del cristallo XIN e XOUT, può portare il cristallo ad oscillare in un punto specifico tra la sua frequenza minima e massima. Cambiando la capacità del carico si otterrà quindi una diversa frequenza di oscillazione. Ecco perché il produttore di cristalli fornisce la frequenza del cristallo a una capacità di carico specifica, che in questo caso è di $14pF$.

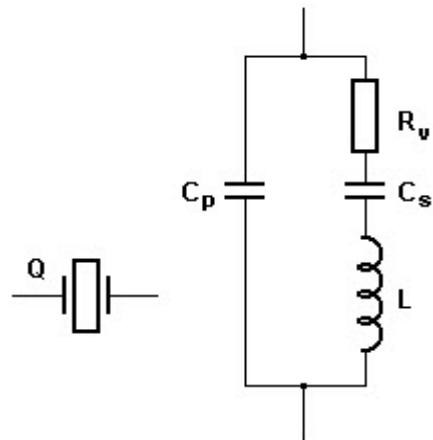


Figure 2.7: Modello RLC di un cristallo al quarzo in risonanza parallelo

Memoria flash

Essendo il microcontrollore sprovvisto di una memoria flash interna, è necessario installarne una esterna in cui poter salvare il firmware compilato in formato binario *.uf2*, il quale all'accensione verrà eseguito dal bootloader. La memoria flash utilizzata è la *W25Q16JV*, caratterizzata da una capacità di 16MB e dall'interfaccia di comunicazione Quad-SPI. Il protocollo QSPI è particolarmente diffuso nel ramo delle comunicazioni con memorie flash, come nel caso presente sulla scheda. Rispetto allo standard SPI, sono presenti 4 linee di dato bidirezionali, rispettivamente *I0*, *I1*, *I2*, *I3* (che vanno dunque a sostituire *MISO* e *MOSI*), che permettono un trasferimento dati dalle 4 alle 6 volte più veloce rispetto ad una normale interfaccia SPI.

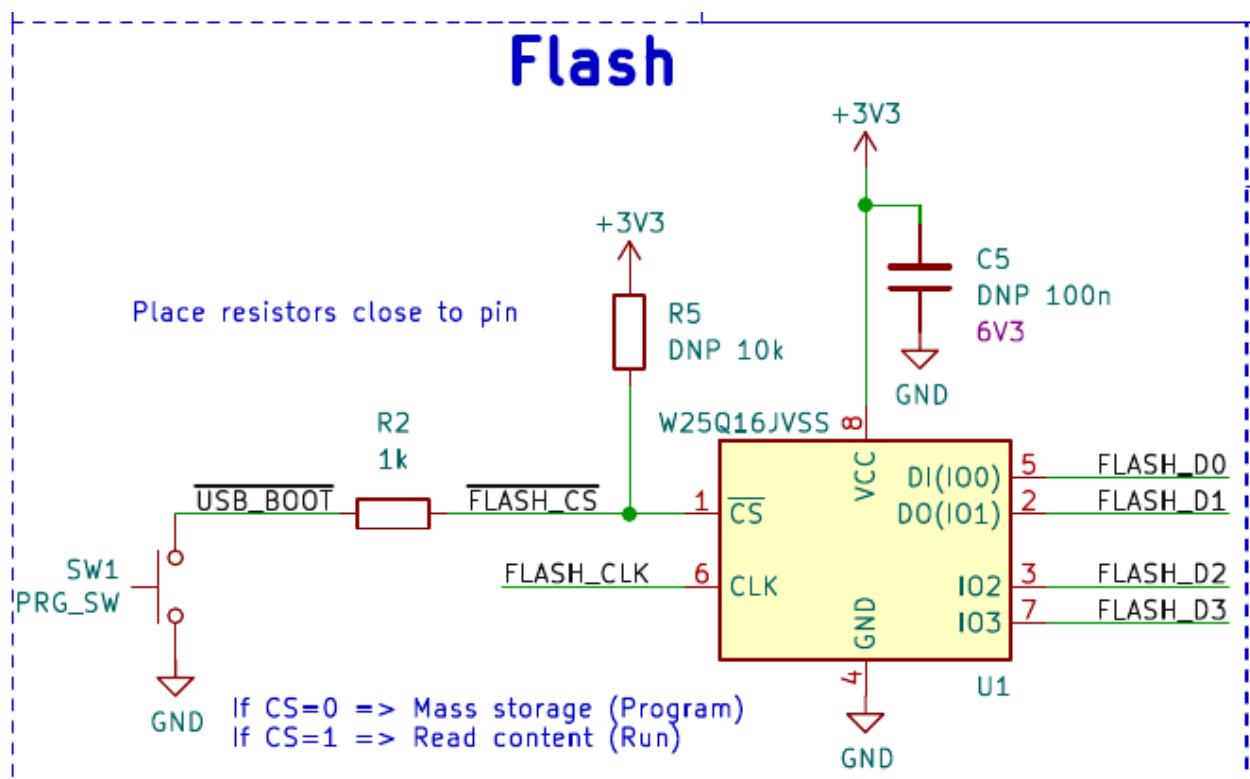


Figure 2.8: Circuito flash SALMO

Il pin \overline{CS} permette di abilitare o disabilitare la comunicazione QSPI, quando $\overline{CS}=1$ i pin di comunicazione rimangono ad alta impedenza, mentre quando $\overline{CS}=0$ è possibile interfacciarsi con la memoria per memorizzare il programma. Sul pin di alimentazione è presente un condensatore di decoupling da 100nF per filtrare rumore ad alta frequenza sulla linea di alimentazione.

Debug ed espansione

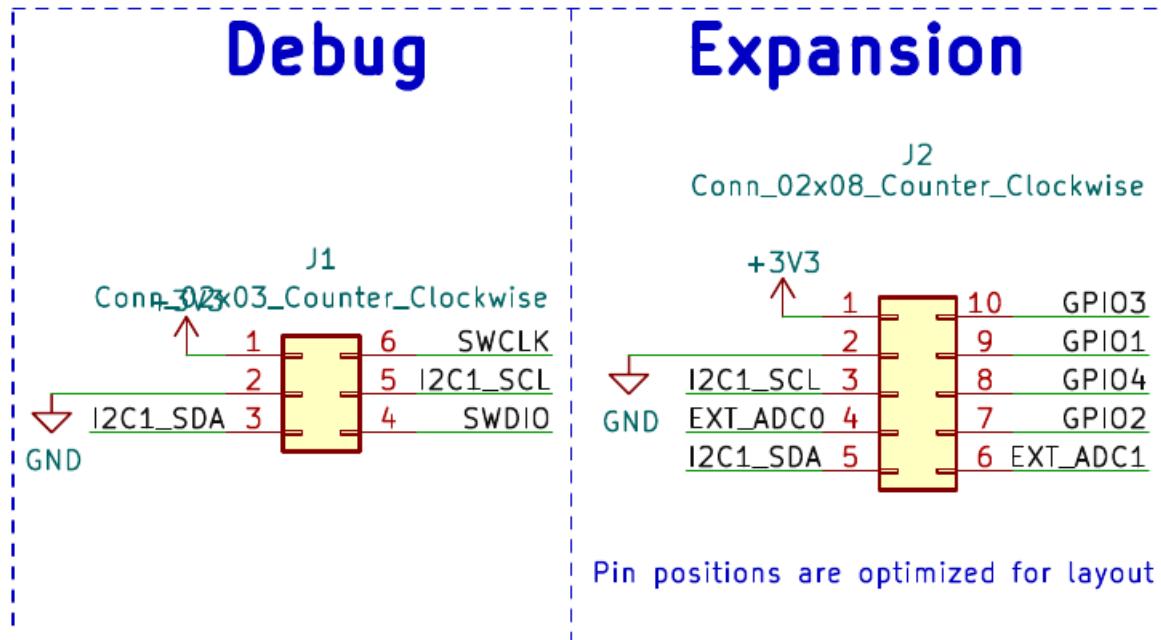


Figure 2.9: Connettori Debug ed Expansion SALMO

Per permettere al microcontrollore di comunicare con eventuali unità esterne, sulla scheda abbiamo inserito un connettore a 10 pin con le linee I^2C , i due canali non utilizzati del convertitore analogico digitale ed i rimanenti $GPIO$ non utilizzati.

Abbiamo previsto anche un connettore per il debug con interfaccia SWD (Serial Wire Debug), alimentazione per il microcontrollore e bus I^2C . Per il debug è necessario utilizzare un altro RP2040, potrebbe essere particolarmente conveniente utilizzare un Pi Pico come debugger, viste le due dimensioni. Basta collegare le linee di SWData e SWClock tra loro e la relativa alimentazione necessaria al debugger, dopodichè si può flashare quest'ultimo con *picoprobe.uf2* (programma situato nella cartella degli esempi offerti per RP2040, vedesi capitolo [Firmware](#)), per poi interfacciarsi via usb utilizzando GDB ed OpenOCD.

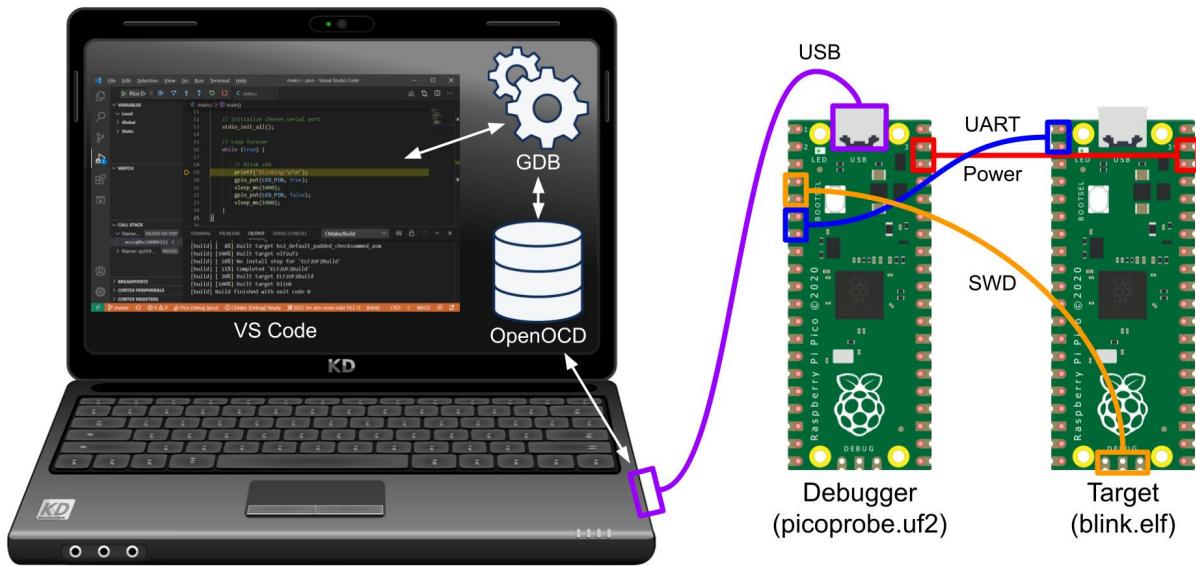


Figure 2.10: Immagine illustrativa del debug di un Pi Pico mediante un altro Pi Pico

2.2. Power

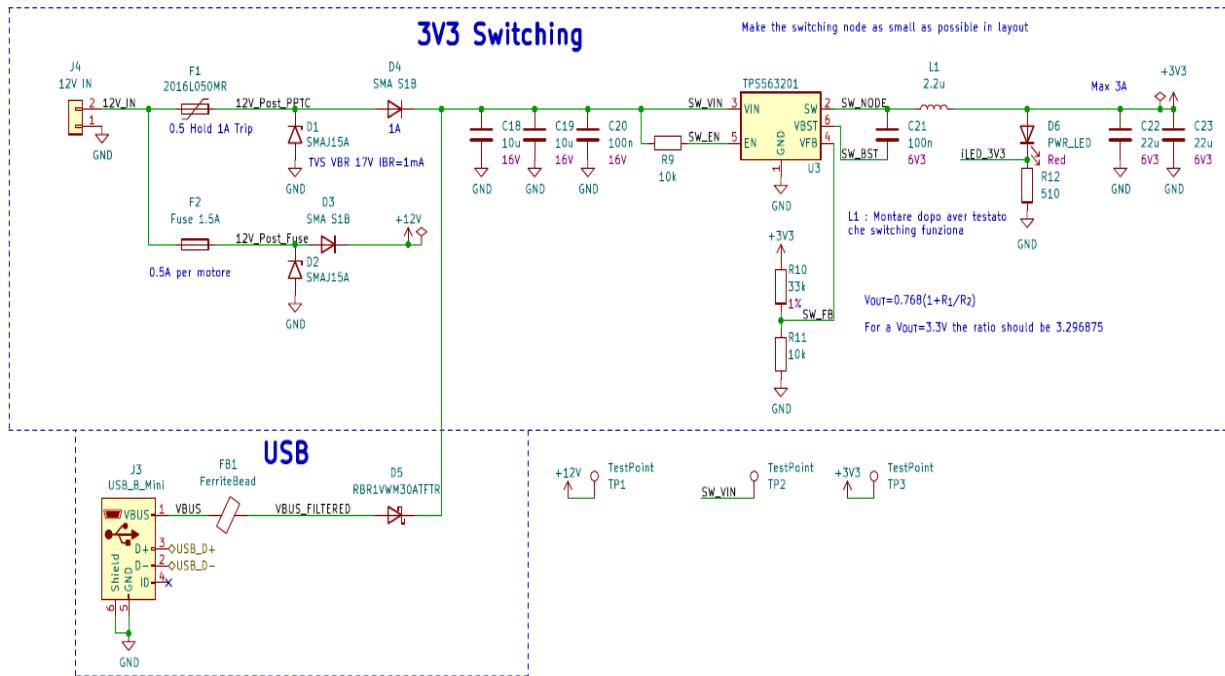


Figure 2.11: Foglio "Power" SALMO

Per il funzionamento della scheda sono necessari due livelli di tensione: 12V per il controllo dei motori e per il led RGB e 3.3V per tutto il resto. I 12V provengono direttamente da un alimentatore

esterno (collegato al connettore J4), mentre i 3.3V vengono convertiti da un convertitore switching di tipo buck (step-down). È inoltre prevista la possibilità di alimentare la logica di controllo tramite un connettore USB (J3). Suddividiamo quindi il circuito di alimentazione in tre macro-blocchi: input 5V, input 12V e regolatore di tensione.

2.2.1. Input 5V

Per permettere di alimentare la scheda anche senza la necessità di collegare un alimentatore da 12V, principalmente per il debug e lo sviluppo firmware, abbiamo previsto un connettore USB di tipo B-mini (il pin ID è stato lasciato flottante non dovendo utilizzare la funzione USB OTG, fissando il dispositivo come device).

Questo fornisce una tensione di 5V in ingresso al regolatore switching, che permette di generare i 3.3V necessari alla scheda. Per evitare che i 12V possano propagarsi sulla rail di alimentazione dell'USB abbiamo aggiunto in serie un diodo (D5) di tipo Schottky, così da avere perdite minori grazie alla caduta di tensione caratteristica di soli 0,33V. Abbiamo inoltre aggiunto una ferrite bead, caratterizzata da una resistenza di 120Ω a 100MHz, per ridurre i rumori di rumore ad alta frequenza.

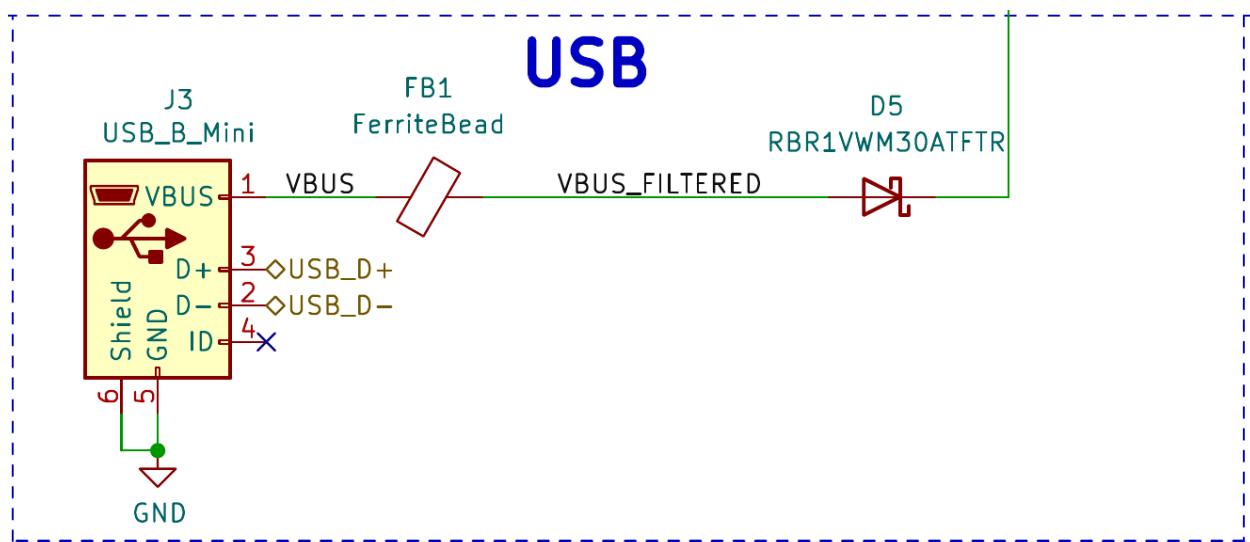


Figure 2.12: Alimentazione via USB

2.2.2. Input 12V

Per il regime di funzionamento normale l'alimentazione proviene dal connettore J4 ad una tensione nominale di 12V. Questo fornisce alimentazione sia ai motori e al led RGB, che al convertitore switching, tramite due rami separati. Per l'alimentazione "di potenza" abbiamo previsto un circuito di protezione composto da un fusibile per protezione da sovraccorrenti, un TVS per protezione da sovratensioni transitorie ed un diodo per bloccare correnti inverse. La corrente massima assorbita da ciascun motore è di circa 0.5A, perciò abbiamo scelto un fusibile da 1.5A (tenendo conto di qualche centinaio di mA per il led RGB e qualche altro per avere una minima tolleranza).

Il TVS invece è uno SMAJ15A, in grado di dissipare 400W per transitori di durata fino a 10us, mentre il diodo in serie è un SMA S1B (D4), un rettificatore generico.

Il ramo che riguarda l'alimentazione del convertitore switching è protetto in modo simile, con la sola differenza che la protezione dalle sovraccorrenti avviene tramite un fusibile resettabile, o PPTC (Polymeric Positive Temperature Coefficient).

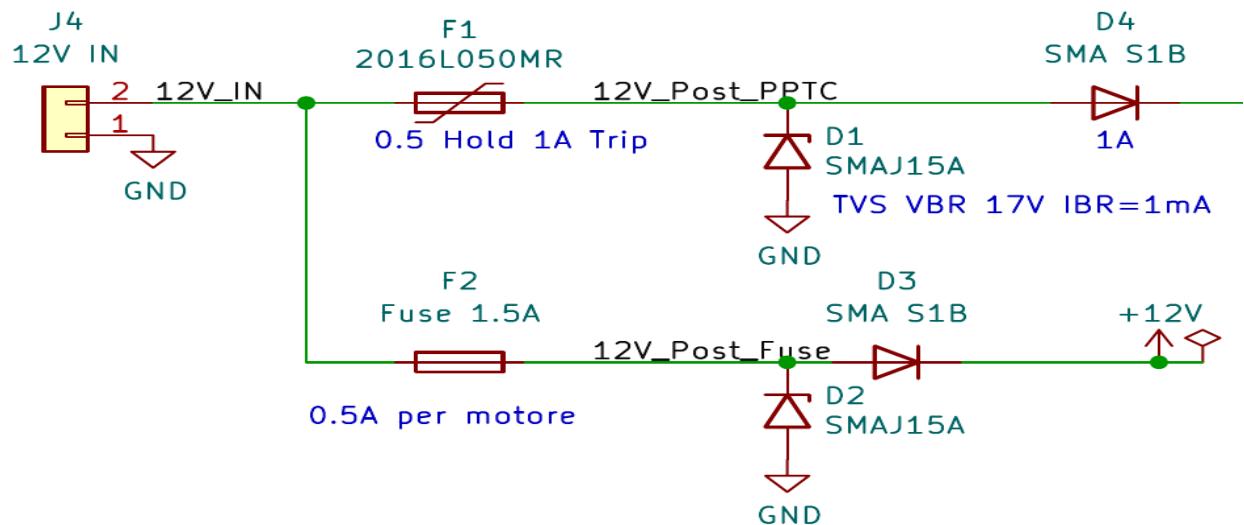


Figure 2.13: Rami 12V e regolatore di tensione

2.2.3. Regolatore di tensione

I diodi D4 e D5 svolgono la funzione equivalente di OR logico per garantire in ogni caso l'alimentazione al regolatore switching, un *TPS563201*, con tensione in ingresso minima di 4.5V e massima di 17V. Per avere in output una tensione di 3.3V, il pin di feedback VFB del regolatore deve essere connesso, tramite un partitore, alla linea di uscita.

Il rapporto di partizione è dettato dalla seguente formula:

$$V_{out} = 0.768 \left(1 + \frac{R_{10}}{R_{11}} \right)$$

per cui ne consegue che per avere $V_{OUT}=3.3V$, deve essere $\frac{R_{10}}{R_{11}} = 3.3$.

Al fine di segnalare la presenza dei 3.3V, abbiamo previsto un led rosso con resistenza in serie da 510Ω .

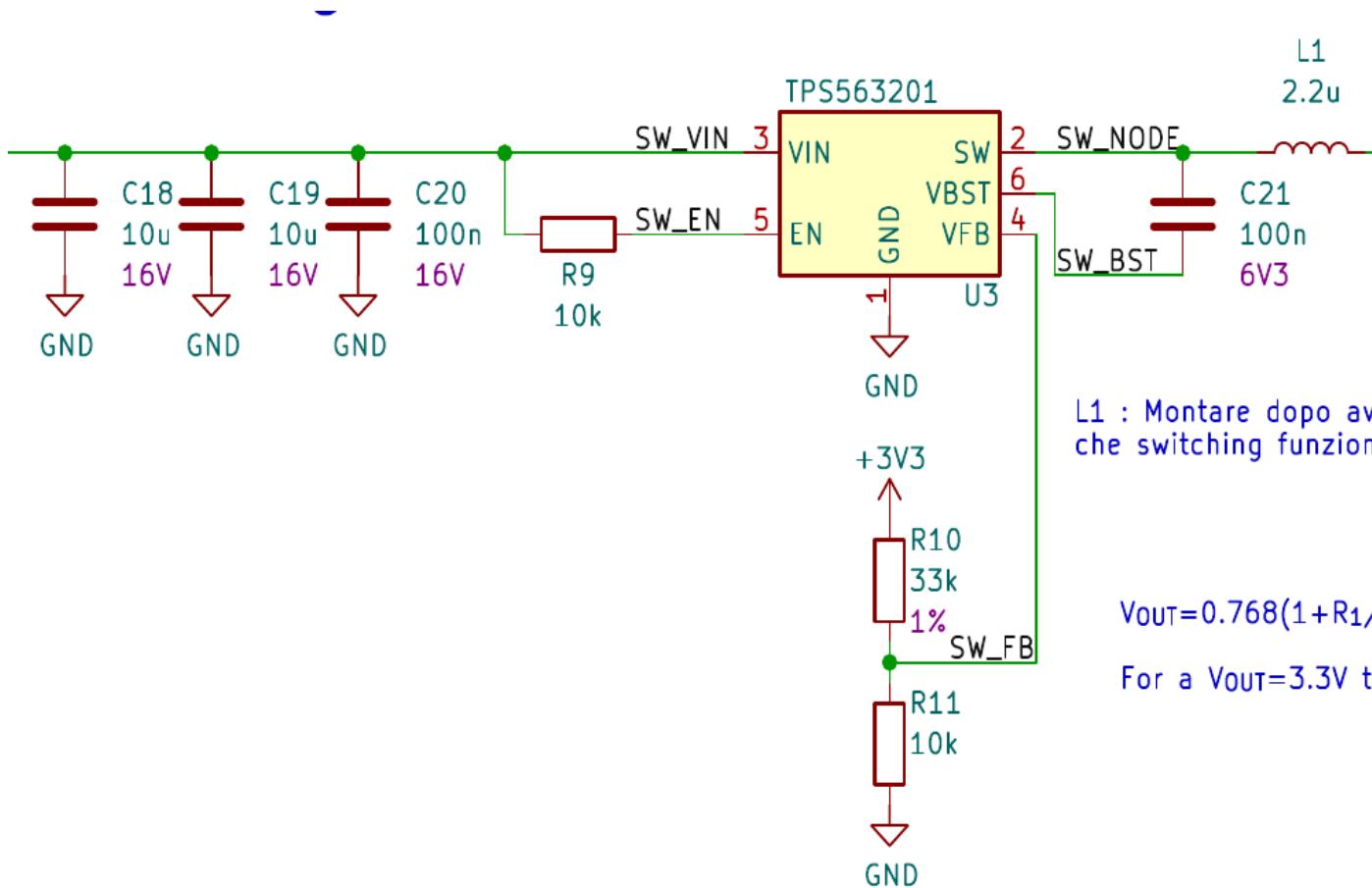


Figure 2.14: Circuito regolatore di tensione buck

Per il dimensionamento dei filtri di ingresso e uscita ci siamo affidati alle design guidelines del datasheet del dispositivo.

All'ingresso del regolatore di tensione abbiamo posto tre condensatori (due da 10u e uno da 100n) con lo scopo di ridurre il ripple in uscita dal regolatore switching. Si inseriscono 3 condensatori poiché, essendo componenti reali e presentando di conseguenza una resistenza ed una induttanza parassita, la loro combinazione darà luogo ad un condensatore con caratteristiche migliori rispetto ad un condensatore di capacità pari alla loro somma. Sul nodo di uscita abbiamo inserito in serie un'induttanza da 2.2uH ed in parallelo due condensatori da 22u.

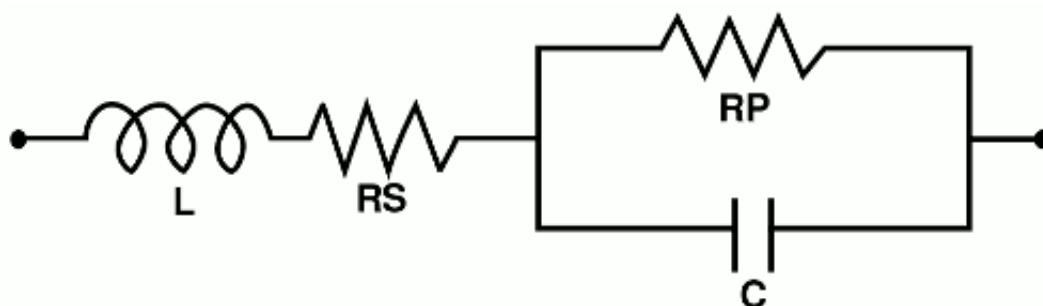


Figure 2.15: Modello RLC di un condensatore reale

Sempre seguendo le linee guida del costruttore, prima dell'induttore, quindi sul nodo di commutazione, abbiamo inserito un condensatore da 100n collegato al pin di ballast. Il regolatore prevede

inoltre un pin di abilitazione che abbiamo deciso di collegare permanentemente, tramite un resistore da 10k, alla linea di ingresso, cosicché la scheda venga alimentata non appena ci sia una tensione adeguata in ingresso.

Abbiamo infine previsto dei test point per il controllo delle varie tensioni sul pannello in fase di debug.



Figure 2.16: Testpoint per la misura delle tensioni di alimentazione

2.3. Peripherals

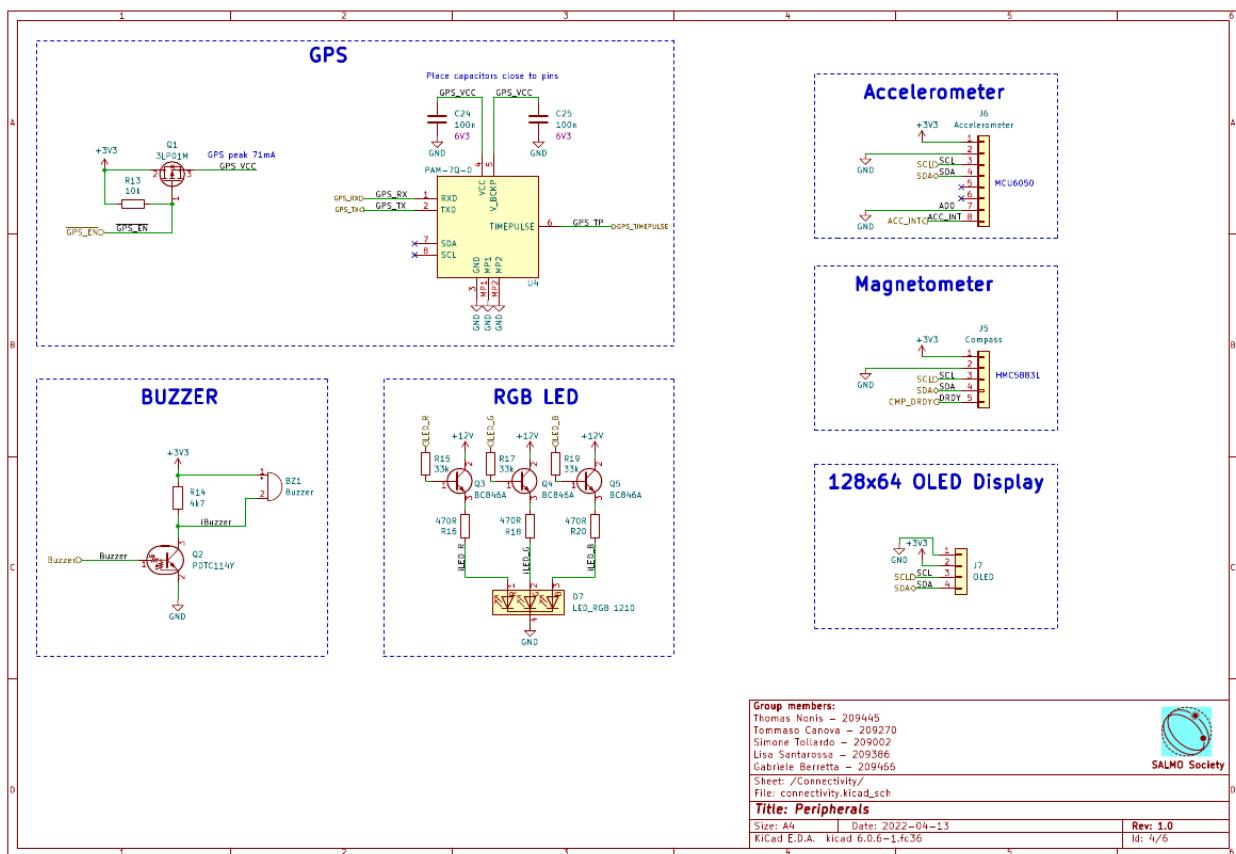


Figure 2.17: Foglio "Peripherals" SALMO

GPS

Il dispositivo GPS utilizzato è lo *U-BLOX PAM7Q*. Questo componente permette di interfacciarsi con i satelliti della rete GPS per ottenere le coordinate geografiche del dispositivo, rendendo possibile il calcolo della posizione del sole rispetto al pannello attraverso l'algoritmo di tracciamento, per poi stabilire il movimento dei motori.

Il dispositivo utilizza il protocollo standard *NMEA* (ma può essere configurato anche per utilizzare il protocollo proprietario *U-BLOX*) per comunicare attraverso la linea seriale.

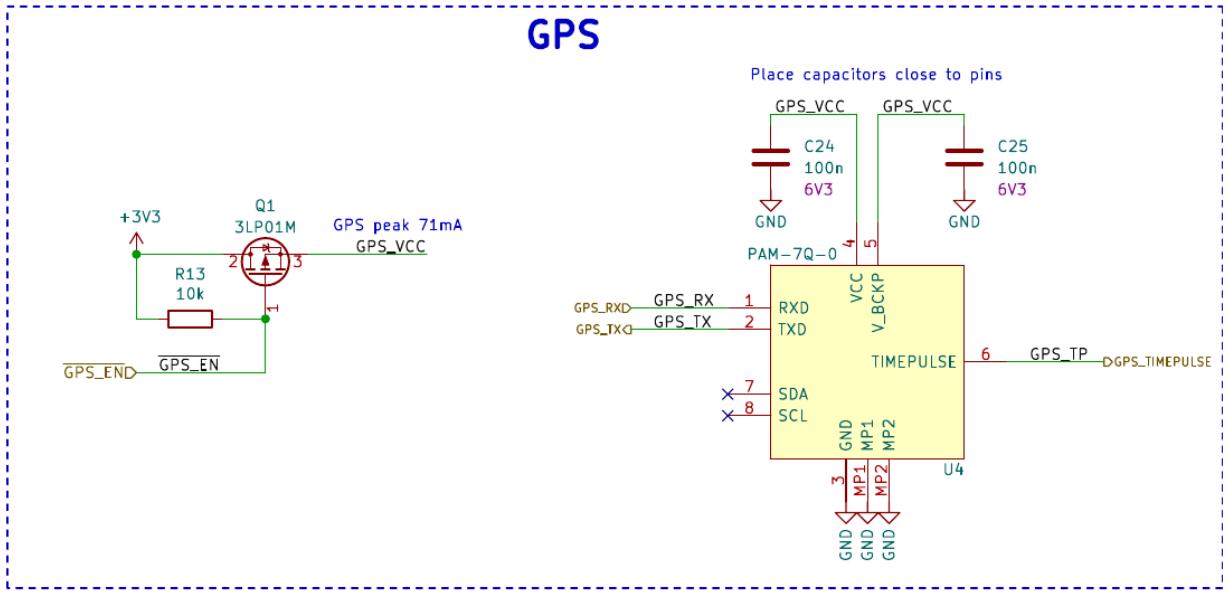


Figure 2.18: Circuito del GPS PAM7Q

Il dispositivo è alimentato a 3.3V tramite i pin 4 e 5, accompagnati dai soliti condensatori di decoupling da 100n.

Si noti che *V_BCKP* e *VCC* sono entrambe connesse alla stessa alimentazione, in quanto non è stata prevista una batteria tampone. Sono presenti nuovamente i condensatori da 100nF tra *GPS_VCC* e ground servono a bypassare il rumore ad alta frequenza verso massa.

L'alimentazione è interrotta da un transistor *PMOS* (*Q1*) e viene abilitata solo quando il transistor viene polarizzato correttamente, ovvero quando $\sim(\text{GPS_EN})$ si trova a livello logico basso. La resistenza da $10\text{k}\Omega$ tra gate e source del pmos serve a mantenere il transistor in cut off quando il pin $\sim(\text{GPS_EN})$ non è pilotato (floating).

I pin 1 e 2, cioè *RXD* e *TXD*, sono rispettivamente i pin di ricezione e trasmissione (dal punto di vista del GPS) per la comunicazione *UART*.

Il pin 6 è dedicato al time pulse, ovvero a generare un segnale elettrico a 1 *PPS* (pulse per second). Questa uscita può essere utilizzata, ad esempio, per riattivare il MCU da una modalità di sospensione una volta al secondo per comunicare con esso. Il *GPS PAM7Q* presenta anche un interfaccia seriale *I2C* ma, dato che per il progetto *SALMO* abbiamo scelto un interfacciamento via *UART*, i pin 7 e 8 (*SDA* e *SCL*) non sono stati connessi.

I mounting point *MP1* e *MP2* sono collegati a massa come indicato da datasheet.



Figure 2.19: GPS U-Blox PAM7Q

Magnetometro ed Accelerometro

Al fine di poter gestire la posizione del pannello mediante controllo ad anello chiuso abbiamo deciso di sfruttare due sensori: un magnetometro a 3 assi con risoluzione di 12 bit *Honeywell HMC5883L* ed un accelerometro+giroscopio entrambi a 3 assi, a 16 bit *TDK MPU6050*. Entrambi i sensori sfruttano l'interfaccia *I2C* ad una velocità massima di 400KHz.

Questi, saranno posizionati **direttamente sul pannello** e provvederanno a restituire all'algoritmo la sua posizione rispetto ai due assi di movimento (Azimuth ed Elevazione).

Tuttavia, non avendo avuto a disposizione l'accelerometro durante la fase di assemblaggio, in questa release non è fisicamente presente.

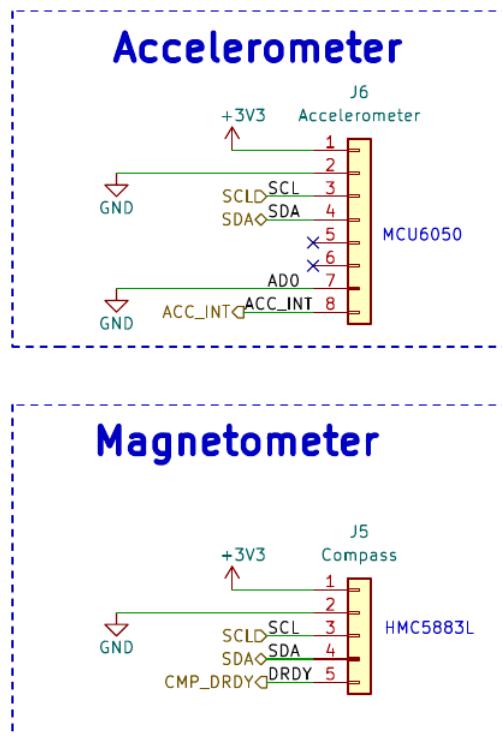


Figure 2.20: Connettori Accelerometro e Magnetometro

Connettore display OLED

Abbiamo previsto un connettore per un display *OLED* 128 x 64 con controller integrato *SSD1306* al fine di presentare all'utente una immediata visualizzazione dei dati della scheda e del pannello solare. Tuttavia, non avendo avuto a disposizione il componente durante la fase di assemblaggio, in questa release non è fisicamente presente.

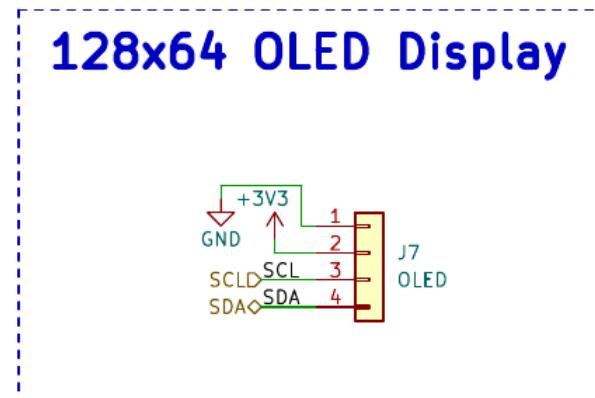


Figure 2.21: Connettore display OLED 128x64

LED RGB

Per quanto riguarda il circuito del led *RGB* 1210 abbiamo commesso un errore in fase di progettazione: considerandone la posizione, tra collettore ed emettitore del transistor cadranno circa 12-(3.3-0.7)=9.4V!

Conseguentemente ai capi dei diodi led cadrà una tensione molto bassa, probabilmente nemmeno sufficiente per portarli in conduzione.

Purtroppo il problema non è sistemabile, se non modificando il circuito ed il layout della PCB.

Considerando che il componente è del tutto ausiliario abbiamo potuto testare comunque la scheda senza alcun problema.

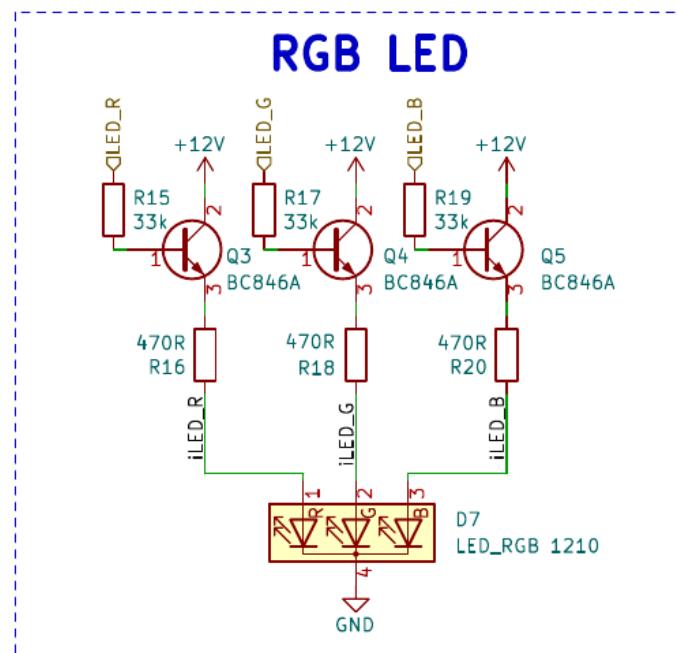


Figure 2.22: Circuito led RGB 1210

Buzzer

Abbiamo previsto un buzzer passivo di tipo elettromagnetico per poter segnalare eventuali stati all'utente e/o errori. Per amplificare il segnale proveniente dall'RP2040 abbiamo utilizzato un transistor NPN con rete di polarizzazione integrata ed un resistore in parallelo al cicalino per poter scaricare la tensione imposta dall'induttanza del buzzer, che altrimenti danneggierebbe il transistor.

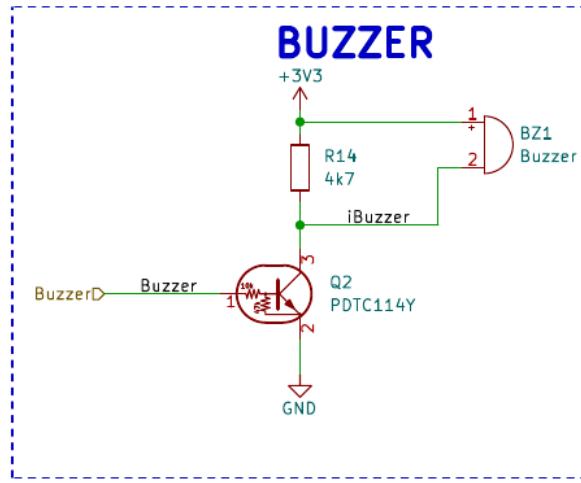


Figure 2.23: Circuito buzzer

2.4. Panel sensing

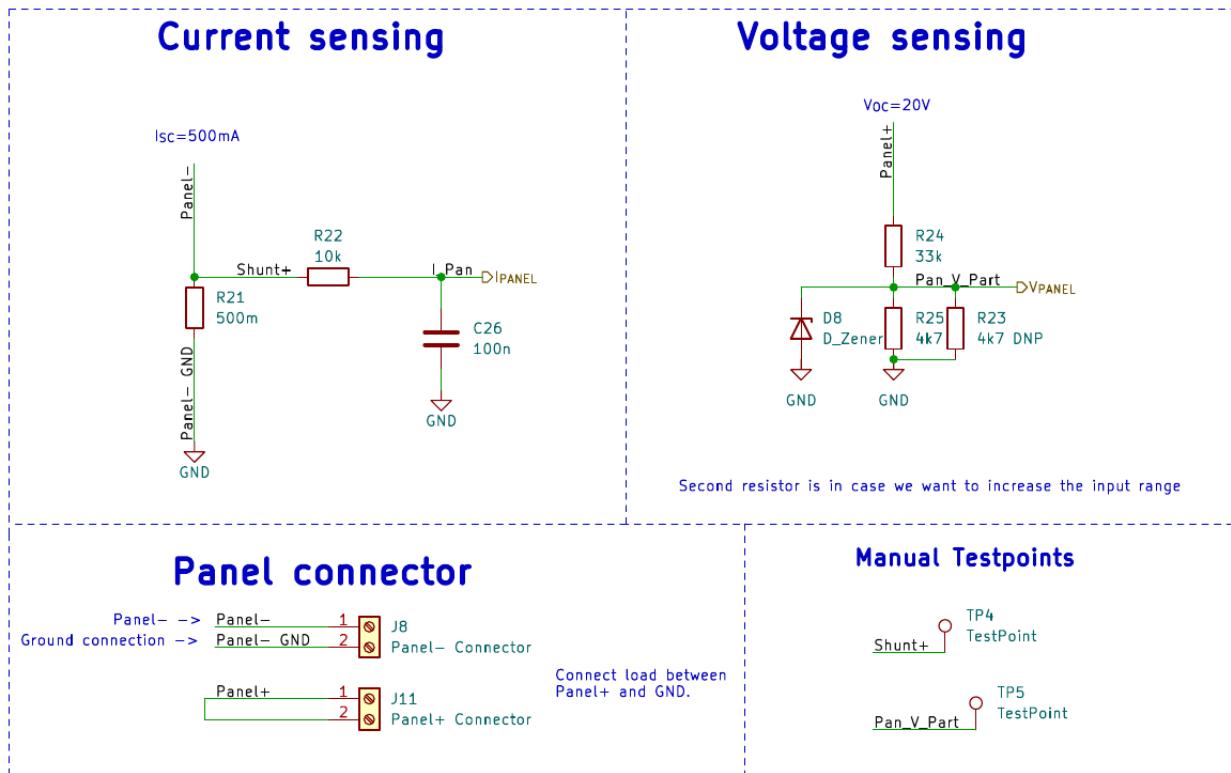


Figure 2.24: Foglio "Panel Sensing" SALMO

Il panel sensing è stato progettato per la misura dei parametri di tensione e di corrente del pannello solare. La misura di corrente del pannello tiene conto di una corrente di cortocircuito (I_{sc}) pari a 500 mA. Per effettuare la misura è stata posta una resistenza di shunt pari a 500 mΩ per ottenere una misura in configurazione *low-side*, ottenendo così una caduta di tensione ai suoi capi massima di 0.25V che può essere letta senza problemi dall'ADC dell'RP2040 (GPIO27, canale 2 dell'ADC).

Nella scelta della resistenza di shunt, condizionata dal fatto che non volevamo complicare inutilmente il circuito con degli amplificatori operazionali, è stato dato maggior peso alla potenza dissipata che allo sfruttare in maniera ottimale il range dell'ADC.

La tensione misurata dall'ADC andrà infatti da 0V ($I_{panel}=0$ A) a 0.25 V ($I_{panel}=I_{sc}=500$ mA) e la potenza dissipata massima sarà quindi $R * I^2 = 0.5 * 0.5^2 = 0.125 W$.

Avendo un ADC a 12 bit, è possibile ottenere una sensibilità di $\frac{(V_{ref}-V_{ss})}{2^{bit}} = \frac{3.3}{2^{12}} = 805.66 \mu V/bit$, corrispondente a $\frac{(V_{ref}-V_{ss})}{2^{bit} * 0.5} = \frac{3.3}{2^{11}} = 1.6 mA/bit$. Data la Vmax di segnale pari a 0.25 V è possibile sfruttare solamente il 7.6% dell'intero range disponibile ($\frac{V_{ref}}{V_{max}} = \frac{3.3}{0.25} = 7.26\%$)

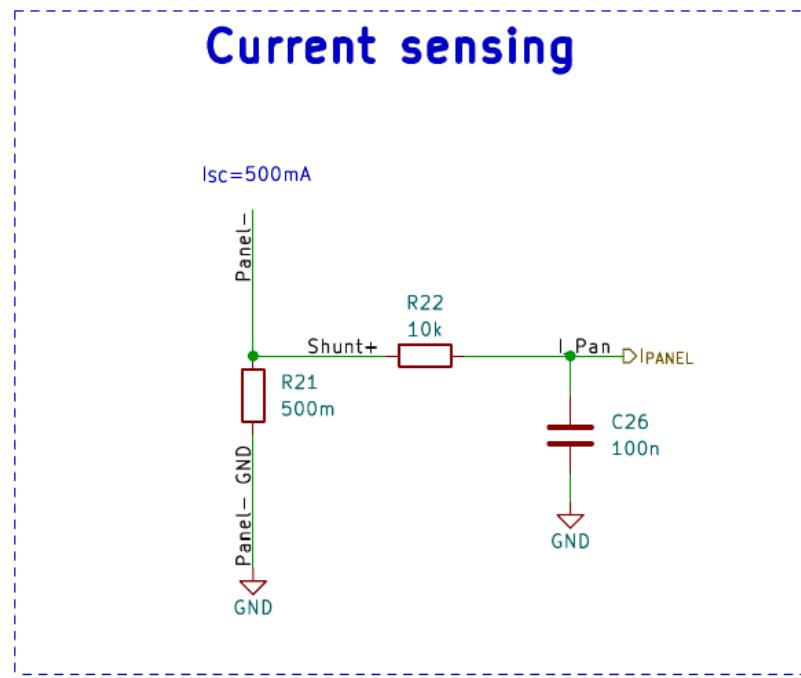
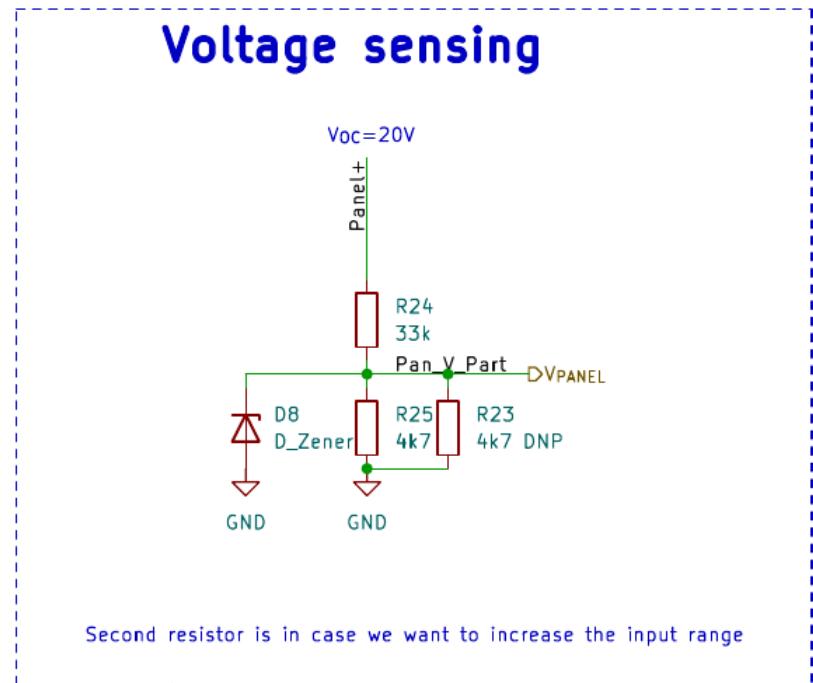


Figure 2.25: Circuito Current Sensing SALMO

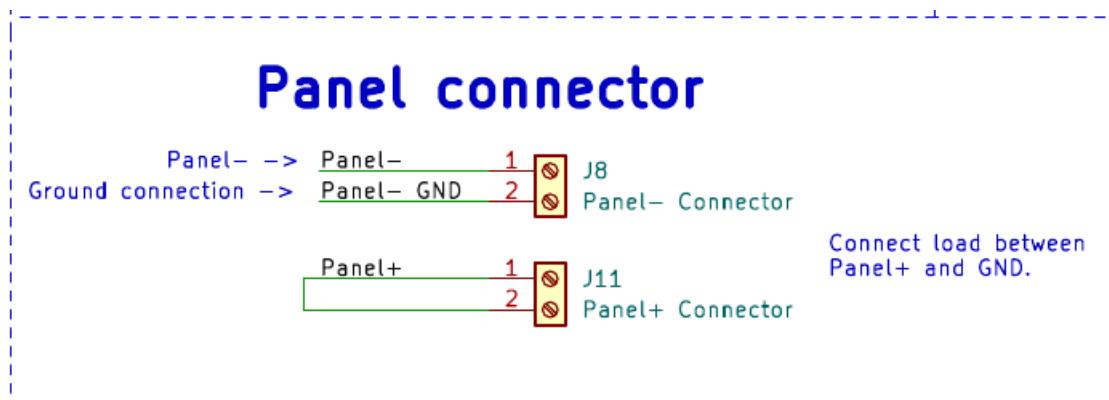
Per la misura di tensione del pannello, non volendo nuovamente utilizzare amplificatori operazionali o integrati specifici, abbiamo scelto di utilizzare il metodo del partitore di tensione, posizionando due resistenze da 33k e 47k per avere un rapporto di partizione di circa $\frac{4.7}{4.7+33} \simeq 0.125$ e quindi un consumo pari a

se ipotizziamo una tensione di open circuit V_{oc} di 20V. Assumendo le stesse ipotesi, i valori in ingresso al GPIO 26 (canale numero 1 dell'ADC) andranno da 0V a 2.5V (la tensione massima è comunque fissata dal diodo zener di protezione posto in parallelo).

Nel caso in cui il pannello avesse tensione di open circuit più alta o volessimo usare più pannelli in serie tra di loro è possibile montare la resistenza ausiliaria R23, che permette di avere un rapporto di partizione di 0.067.

**Figure 2.26:** Circuito Voltage Sensing SALMO

In questo foglio abbiamo inoltre deciso di posizionare i connettori per il collegamento del pannello e del relativo carico/utilizzatore.

**Figure 2.27:** Connettori per carico e pannello SALMO

Abbiamo infine inserito dei test point per consentirci di misurare manualmente la tensione e la corrente del pannello (misurando la caduta di tensione sulla resistenza di shunt si ottiene $i = \frac{V_{misurata}}{R_{21}}$).

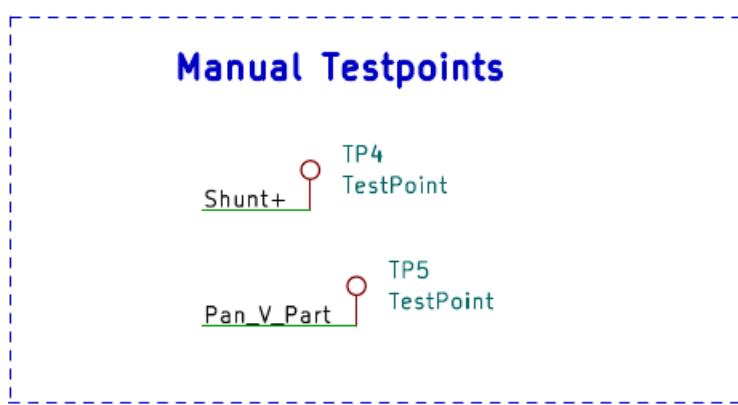


Figure 2.28: Testpoint per la misura manuale di tensione e corrente del pannello

2.5. Actuation

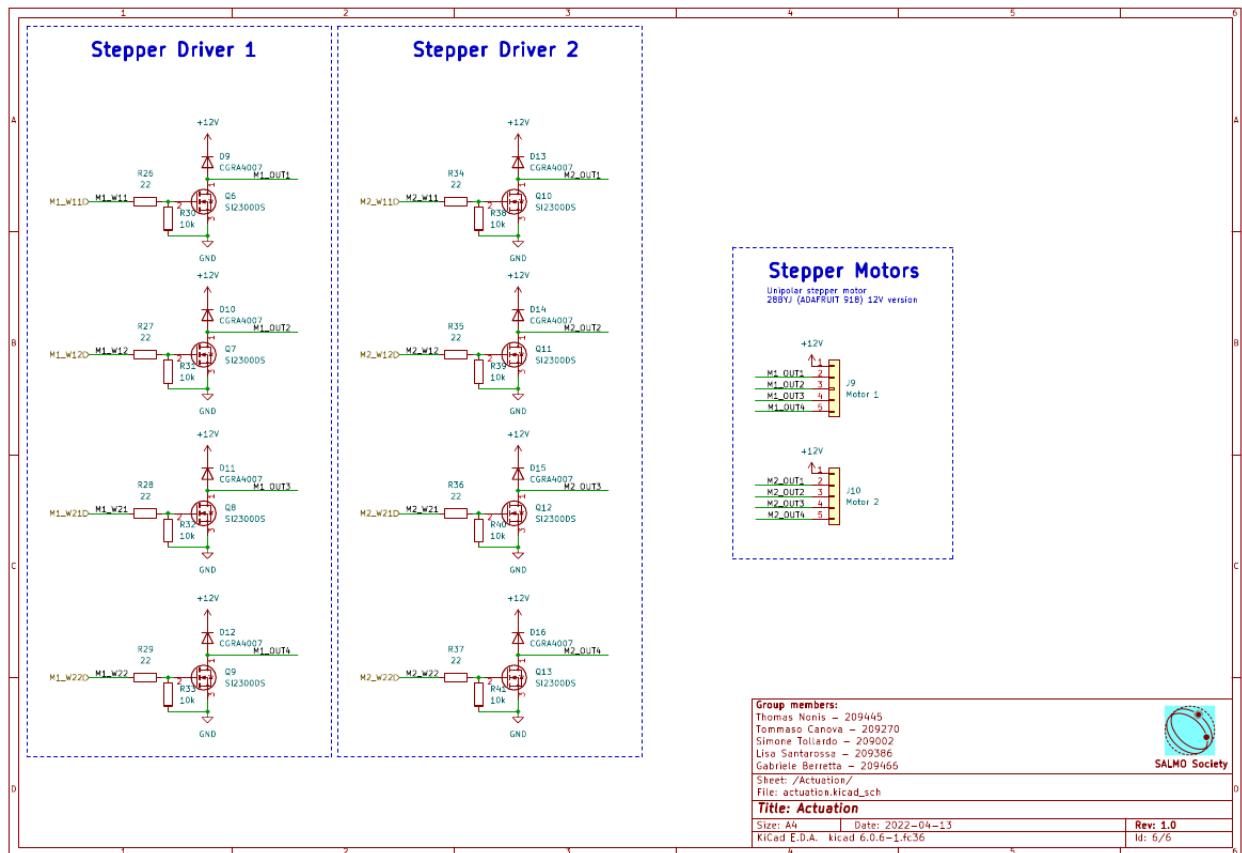


Figure 2.29: Foglio "Actuation" SALMO

Nella pagina *Actuation* abbiamo inserito i circuiti di controllo per i due motori stepper unipolari tramite driver separati a componentistica discreta e i relativi connettori. I due attuatori hanno la funzione di orientare il pannello fotovoltaico nella direzione desiderata, ruotando il pannello attorno agli assi z e y.

Abbiamo scelto i motori passo-passo Adafruit 918, con tensione e corrente nominali rispettivamente di 12V e 500mA, unipolari con motoriduttore e coppia statica di 250gr/cm.

Da evidenziare è la presenza del motoriduttore (*gearbox*) che permette di ottenere una maggiore risoluzione (da 200 step fino a ben 512) e soprattutto una *maggior coppia statica*; quest'ultima ci consente di mantenere in posizione il pannello anche quando il motore è disalimentato così da limitare

i consumi totali. Ipotizzando, infatti, il peso medio di un pannello fotovoltaico da 1W disponibile in commercio di circa 20gr, una coppia statica di 250gr/cm risulta sufficiente per consentire di ancorare meccanicamente a dovere il pannello all'asse del motore.

Per giunta, ogni motore essendo di tipo unipolare è dotato di due avvolgimenti separati con presa intermedia comune pertanto sono sufficienti solo quattro transistor per il comando del singolo. Infatti a differenza degli stepper bipolar, non è necessario un *h-bridge* per cambiare il senso di rotazione ma basta invertire l'ordine sequenziale di attivazione dei transistor. A seguito è illustrato il circuito base a confronto:

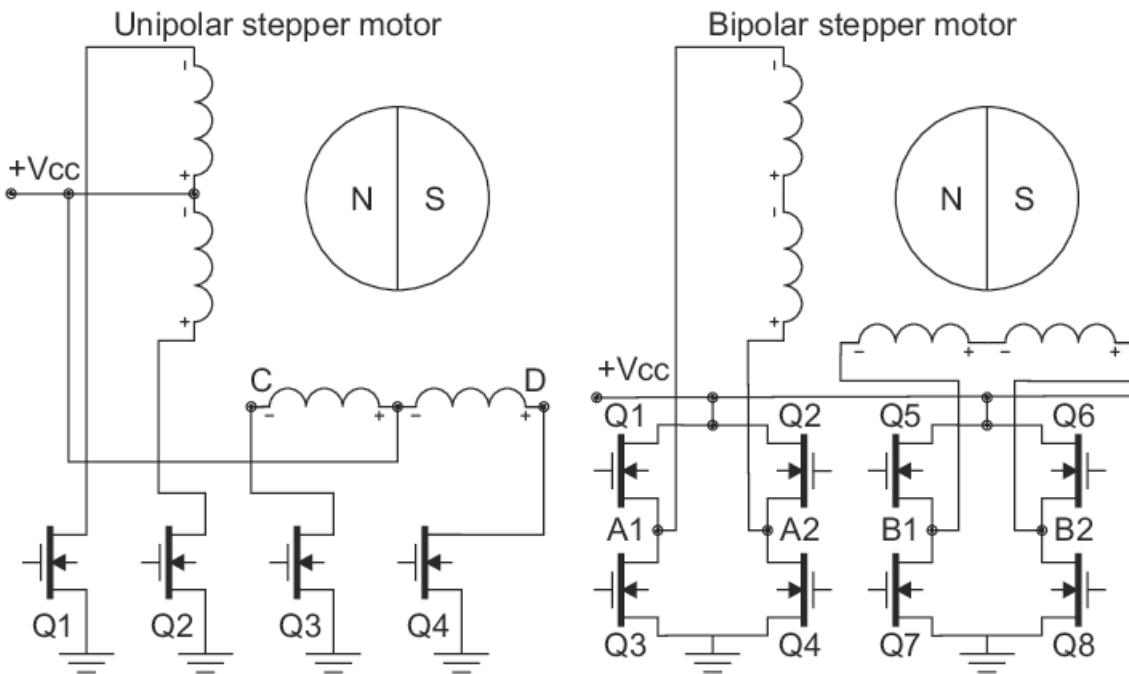


Figure 2.30: Driver per stepper unipolare vs driver per stepper bipolare

Il driver è costituito da un n-mos pilotato dal MCU, con resistenza tra gate e source da $10\text{k}\Omega$ e resistenza di limitazione tra pin del MCU e gate da 22Ω . La prima funge da pull-down nel caso in cui il pin del MCU sia flottante, al fine di evitare accensioni involontarie del transistor, pertanto abbiamo scelto il valore di $10\text{k}\Omega$ analogamente alle altre resistenze di pull-up/down già presenti nel circuito. La resistenza da 22Ω invece funge da limitatore di corrente sul ramo che connette il pin I/O al gate del mosfet, introducendo un ritardo nel fenomeno di carica e scarica della capacità di gate-source del dispositivo, evitando così di bruciare il pin del MCU dato che all'inserzione questa corrente può essere elevata. Il valore di 22Ω è stato scelto arbitrariamente e fa sì che sul gate del transistor ci sia una tensione di circa 3.3V (considerando il partitore tra R26 e R27) quando il pin di output del microcontrollore è a livello logico alto e, inoltre, determina una costante di carica/scarica della capacità di ingresso di Q6 di:

$$\tau = R_{26} \cdot C_{gate-source} = 22 \cdot \Omega \cdot 7.3nF \approx 160ns$$

Una costante di tempo di tale grandezza è considerata accettabile, perché ipotizzando una frequenza di lavoro dei mosfet di circa 20Hz, quindi con un periodo di 55ms, permette allo switch di commutare come previsto.

In parallelo alle sezioni degli avvolgimenti abbiamo inserito un diodo di flyback cosicché all'apertura dello switch non si verifichino spike di tensione critici tra drain e source dello stesso.

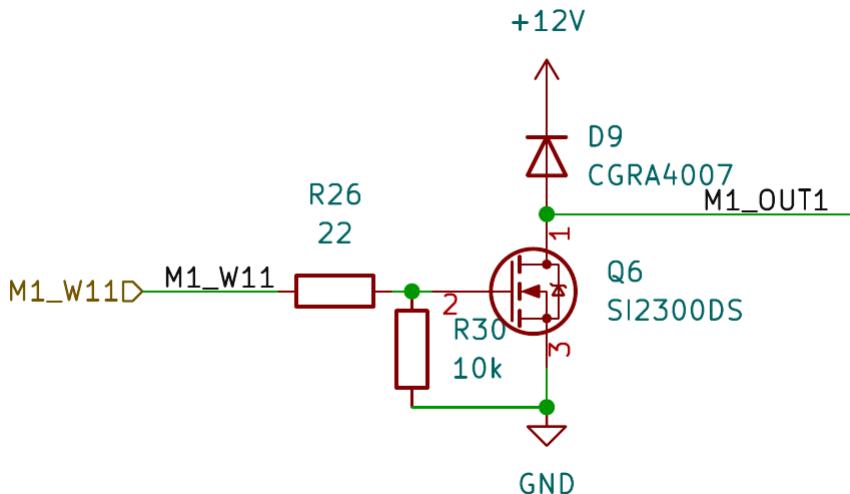


Figure 2.31: Circuito driver per un avvolgimento

Nel circuito in questione lo switch deve garantire:

- di poter essere pilotato con una tensione di gate di almeno 3.3V (livello logico alto dei GPIO del MCU scelto);
- una corrente di drain di almeno 0.5A (corrente nominale dei motori);
- una differenza di potenziale tra drain e source di almeno 12V (tensione di alimentazione dei motori).

Considerando le specifiche appena elencate abbiamo scelto dei transistor Vishay SI2300DS nel comune package SOT23, con tensione di soglia 1.5V e valori massimi supportati tra drain e source di 30V, tra gate e source di 12V e una corrente di 3A.

ABSOLUTE MAXIMUM RATINGS $T_A = 25^\circ\text{C}$, unless otherwise noted			
Parameter	Symbol	Limit	Unit
Drain-Source Voltage	V_{DS}	30	V
Gate-Source Voltage	V_{GS}	± 12	
Continuous Drain Current ($T_J = 150^\circ\text{C}$)	I_D	3.6 ^a	
		3.0	
		3.1 ^{b, c}	
		2.5 ^{b, c}	
Pulsed Drain Current	I_{DM}	15	A
Continuous Source-Drain Diode Current	I_S	1.4	
		0.9 ^{b, c}	
Maximum Power Dissipation	P_D	1.7	
		1.1	
		1.1 ^{b, c}	
		0.7 ^{b, c}	
Operating Junction and Storage Temperature Range	T_J, T_{stg}	- 55 to 150	$^\circ\text{C}$
Soldering Recommendations (Peak Temperature) ^{d, e}		260	

Figure 2.32: Estratto datasheet SI2300DS

2.6. Finalizzazione schematico

Una volta terminata la progettazione di tutti i sotto-circuiti della nostra scheda abbiamo provveduto a fare un controllo globale. Per questo abbiamo fatto sia un'analisi manuale, valutando tutte le connessioni ed i valori inseriti, sia un'analisi automatizzata tramite il DRC. Dopo aver riscontrato e risolto qualche inevitabile errore abbiamo reputato concluso il processo di progettazione elettrica. A questo punto il DRC segnala comunque 2 errori e molti warning, che però sono dovuti alle dichiarazioni di alcuni pin dei simboli e non influenzano in alcun modo il funzionamento.

3

Analisi del Layout PCB

Una volta terminato il progetto dello schematico e verificata la sua correttezza, siamo passati allo sbroglio circuitale. Abbiamo quindi esportato la netlist da *eesschema* per poi importarla in *pcbnew*, popolando il progetto con i footprint di tutti i componenti.

3.1. Layout

Come primo passo abbiamo raggruppato i vari componenti per sezione circuitale (alimentazione, microcontrollore, controllo motori, ecc...) in modo da avere un primo ordine.

Fino a questo punto non abbiamo posto alcun riguardo verso la posizione dei componenti, ma abbiamo semplicemente effettuato una suddivisione funzionale. Con il passo successivo abbiamo invece collocato, per ciascun blocco circuitale, ogni componente nel modo che più ci è sembrato adeguato, tenendo in mente gli aspetti elettrici (per esempio il posizionamento del GPS lontano dalla sezione di potenza e dal microcontrollore), di distanziamento, di funzionalità (per esempio posizionamento di connettori e pulsanti), di comodità per l'utente finale ed infine di estetica.

Per quanto riguarda la sezione legata al microcontrollore abbiamo posizionato, in ordine di priorità:

1. il quarzo con i suoi condensatori ed il suo resistore il più vicino possibile ai pin XIN e XOUT;
2. i condensatori di decoupling più vicini possibile ai pin di alimentazione;
3. la memoria flash;
4. i resistori di terminazione della linea USB.

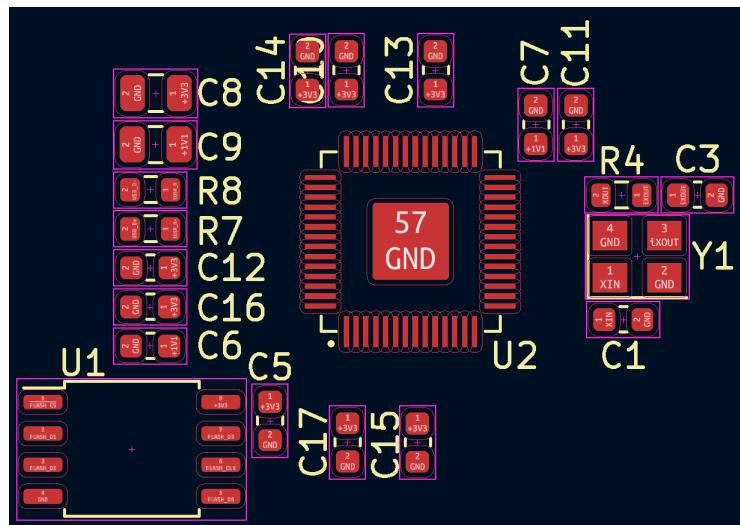


Figure 3.1: Primo posizionamento componenti principali foglio Control

Il fatto di posizionare i componenti il più vicino possibile ai relativi pin è necessario per ridurre di quanto possibile l'impedenza della linea di alimentazione, e quindi la lunghezza del percorso del segnale. Maggiore è la lunghezza del percorso tra condensatore e pin e maggiore sarà l'induttanza

parassita della traccia, che porta a problemi di integrità di segnale quando si tratta con segnali ad alta velocità. Considerando a titolo di esempio una linea di alimentazione, l'induttanza parassita è contrastata dalla presenza dei condensatori di decoupling. In questo modo siamo in grado fornire un percorso a bassa impedenza tra condensatore e pin del MCU, perciò la restante lunghezza che va dal condensatore al pin di alimentazione deve rimanere il più breve possibile. Si noti che ai fini del calcolo si deve considerare sia la linea di andata (es: alimentazione) che quella di ritorno (GND).

Per quanto riguarda la sezione di controllo dei motori abbiamo posizionato i connettori, i transistori, i diodi ed i resistori di polarizzazione in modo da ridurre al minimo le tracce necessarie e da allineare il tutto per renderlo il più esteticamente bello possibile.

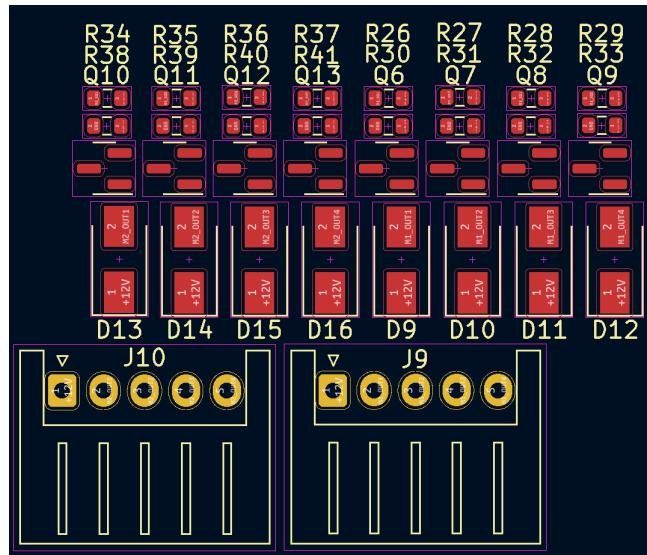


Figure 3.2: Primo posizionamento componenti driver motori stepper

Parlando dello stadio di alimentazione, abbiamo posizionato tutti i componenti in modo da minimizzare il numero di tracce necessario e da facilitare il routing di tracce di grande spessore. Abbiamo rivolto particolare attenzione al regolatore a commutazione, cercando di rendere il nodo di switching più piccolo possibile per ridurre al minimo il ripple.

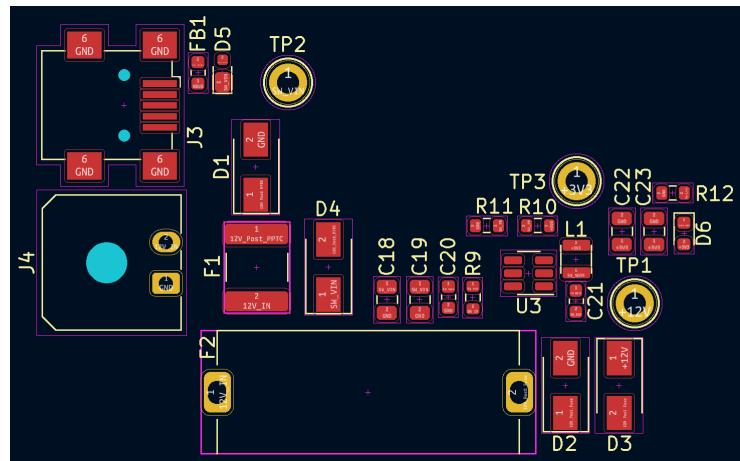


Figure 3.3: Primo posizionamento componenti foglio Power

Le restanti sezioni non includono componenti particolarmente critici, quindi il posizionamento è stato fatto seguendo il principio di semplicità di routing delle tracce e di bellezza estetica.

A questo punto, dopo aver ottenuto un insieme di diversi blocchi circuituali, abbiamo provato a posizionarli in modo da capire che dimensioni imporci come limite per la scheda ed abbiamo appurato che 100x60mm fosse una misura adeguata per contenere il tutto.

Avendo definito le dimensioni siamo passati alla creazione del disegno del contorno della scheda sul layer Edge.Cuts, introducendo un arrotondamento degli angoli per fini puramente estetici. Successivamente abbiamo disegnato i rettangoli di riempimento, assegnandoli alla net di GND e con priorità minima per entrambi i layer, in modo che coprissero la scheda intera.

Abbiamo quindi posizionato i blocchi circuitali discussi in precedenza, non prima di aver posizionato i fori di montaggio ed i fiducials, facendo in modo di avere i connettori lungo il bordo. Anche il GPS è stato posizionato in un angolo in modo da tenerlo il più lontano possibile da ogni altro componente; sotto il suo ingombro abbiamo inserito anche una *keepout zone*, come richiesto da datasheet.

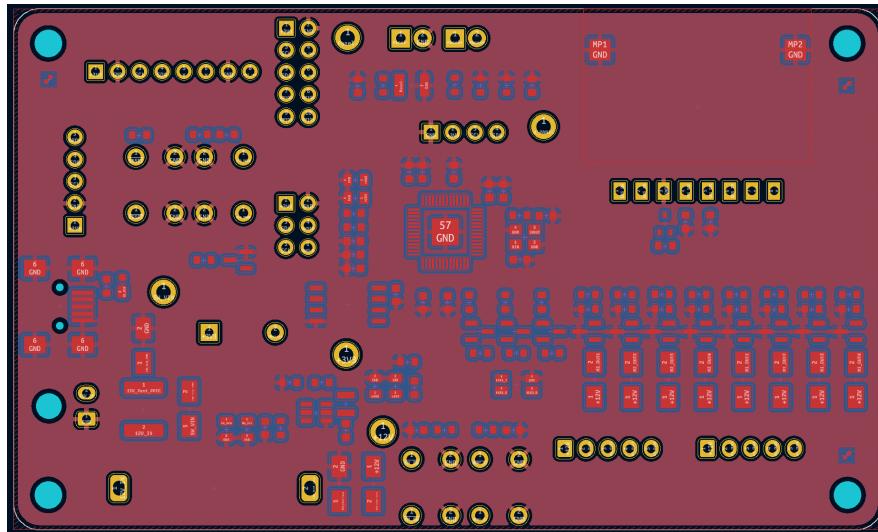


Figure 3.4: Prima bozza piazzamento componenti

3.2. Routing

Una volta definite le posizioni dei vari componenti siamo passati alla fase di routing, cominciando dalle linee particolarmente sensibili e poi quelle di alimentazione. Prima di iniziare abbiamo creato una serie di dimensioni predefinite per le tracce:

- 0.2mm per il fanout del microcontrollore
- 0.3mm per le linee di segnale in generale
- 0.5mm per le linee di alimentazione
- 0.8mm per le linee di alimentazione dei motori

Abbiamo inoltre creato due dimensioni predefinite per le vias (diametro pad/diametro foro):

- 0.55mm/0.25mm per le linee di segnale
- 0.8mm/0.4mm per le linee di alimentazione

Una volta preparato l'ambiente di lavoro abbiamo cominciato a collegare tutte le linee di alimentazione, partendo prima da quelle più corte che, per esempio, collegano un pin di un dispositivo al relativo condensatore di decoupling, per effettuare solo alla fine i collegamenti più lunghi facendo in modo di lasciare sufficiente spazio per le linee di segnale.

Per quanto riguarda il routing dell'alimentazione vogliamo sottolineare 3 accorgimenti che abbiamo preso:

- per le tracce che vanno dal connettore del pannello solare allo shunt per misurare la corrente, abbiamo deciso di utilizzare uno spessore di 1mm, in modo da ridurne al minimo la resistenza;
- per effettuare il fanout del microcontrollore abbiamo utilizzato tracce da 0.2mm per allontanarsi dai pin, per poi allargarle il prima possibile a 0.3mm;

- per il routing della linea di alimentazione a 12V per il controllo dei motori abbiamo utilizzato per quanto possibile i poligoni di riempimento, assegnando la net 12V ed impostando la priorità ad 1.

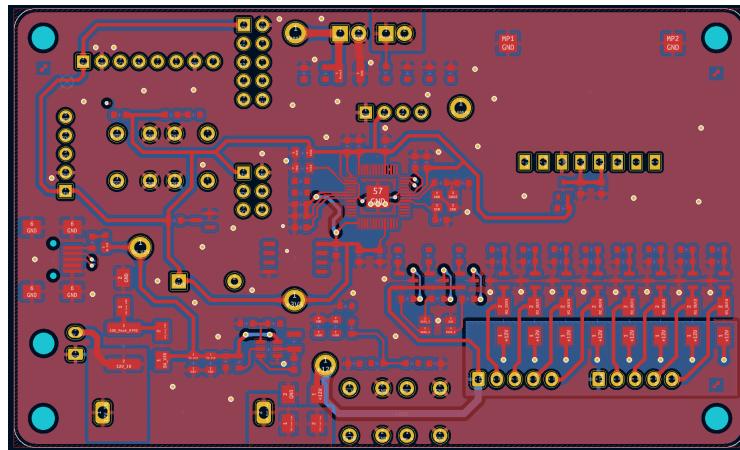


Figure 3.5: Prima bozza routing tracce

Una volta concluso il routing delle linee di alimentazione siamo passati alle restanti linee di segnale iniziando dal fanout del microcontrollore, anche in questo caso partendo con tracce da 0.2mm per poi allargarle a 0.3mm.

Abbiamo considerato per primi i segnali critici, ovvero le linee del quarzo, quelle per la trasmissione QSPI della flash e quelle differenziali per l'USB. Sebbene non sia di apprezzabile effetto dal punto di vista funzionale, abbiamo voluto provare, a titolo di esercizio personale, il tool di *length matching* per rendere le linee differenziali D+ e D- dell'USB più simili possibile in termini di lunghezza. Questo è stato fatto, perché abbiamo riscontrato alcune difficoltà nell'utilizzo del router differenziale, che avrebbe gestito il *length matching* in autonomia.

Avendo definito le tracce critiche siamo poi passati al fanout e al routing dei restanti segnali. Durante questo processo abbiamo iterato più volte spostando le assegnazioni di alcuni pin del microcontrollore, grazie alle matrici di assegnazione dei GPIO, in modo da trovare le posizioni ottimali.

Questa parte dello sbroglio è stata la più complessa a causa della necessità di avere molti dispositivi vicini al microcontrollore e alla ridotta *clearance* tra i pin.

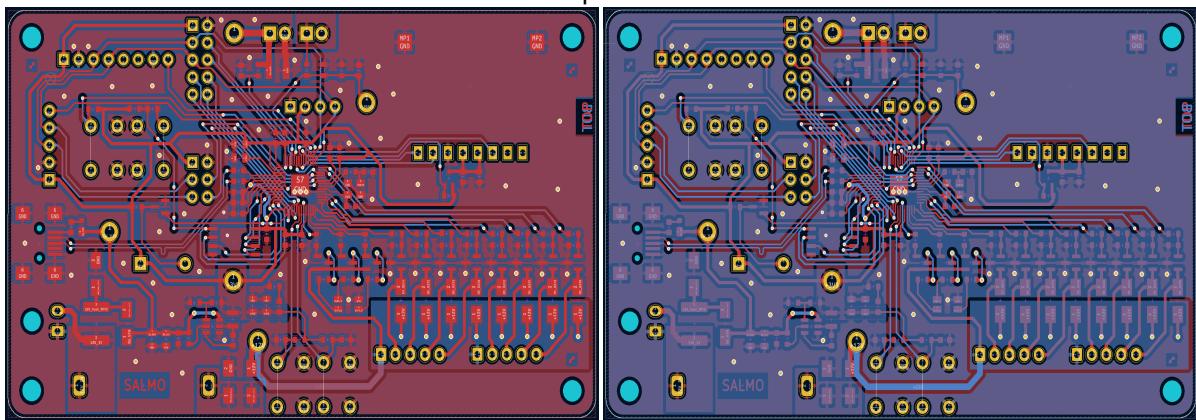


Figure 3.6: Primo risultato routing PCB

Terminato il routing di tutte le tracce abbiamo fatto un'analisi per cercare eventuali sviste o errori commessi, sia facendo un'ispezione visiva, sia utilizzando il DRC. Prevedibilmente abbiamo riscontrato qualche errore che abbiamo prontamente corretto senza troppe difficoltà.

Anche dopo queste correzioni il DRC restituisce alcuni errori, che però sono previsti e non causano problemi nella realizzazione. Questi sono di due tipi, il primo è il *Courtyards overlap* per i componenti che stanno vicino al GPS, mentre il secondo è l' *Items not allowed* ed è dovuto a come sono stati realizzati i footprint dei fiducials.

Il primo lo si può ignorare, perché in realtà il GPS sarà montato su un header rialzato e quindi la superficie del PCB sarà libera da occlusioni. Il secondo errore invece è dovuto al fatto che il footprint dei fiducials è stato realizzato con due pad che sovrappongono una *keepout zone*, questo però è comunque un effetto desiderato, quindi possiamo ignorarlo.

3.3. Finitura

Arrivati a questo punto abbiamo appurato che l'aspetto elettrico della scheda era corretto, quindi ci siamo concentrati sugli aspetti estetico e utilitario.

Prima di tutto abbiamo aggiunto un pad di GND a lato scheda per poter collegare una pinza a coccodrillo in modo da semplificare le misurazioni per il debug hardware. Poi abbiamo fatto in modo di posizionare, sul layer silkscreen tutti i refdes dei componenti, in modo tale che fossero ordinati e ben visibili, facendo particolare attenzione, per quanto possibile, a non posizionarli sopra vias che avrebbero causato potenziali problemi di lettura.

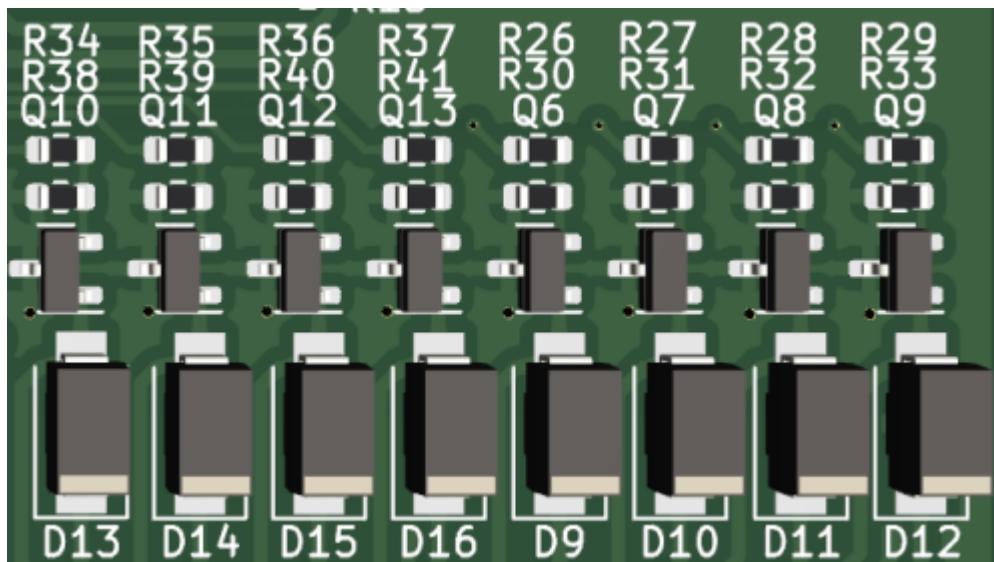


Figure 3.7: Esempio Silkscreen SALMO

Sistemati tutti i refdes abbiamo aggiunto sui layer di silkscreen ulteriori descrizioni per chiarire le funzioni dei connettori e per aggiungere informazioni utili. In particolare abbiamo aggiunto l'indicazione di polarità sul connettore di alimentazione e, ad ogni connettore il suo nome e, sul retro, la descrizione di ogni pin. Per i connettori di espansione e debug non c'era spazio a sufficienza, quindi abbiamo creato sul retro una grafica apposita composta da una tabella, che rappresenta la funzione di ogni pin, e da un segmento di retta che riconduce la tabella al connettore a cui fa riferimento.

Abbiamo successivamente aggiunto, sempre sul retro, il titolo del progetto con il nome del corso, nomi, cognomi e matricole dei membri del gruppo e data e numero di revisione.

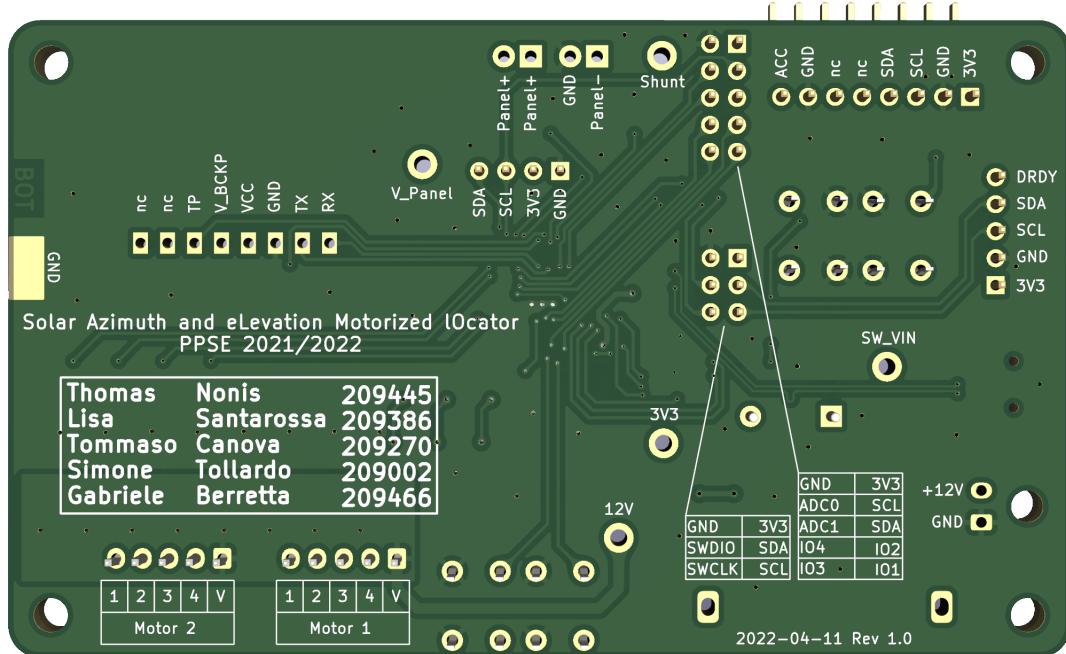


Figure 3.8: Vista del retro della PCB SALMO

Per poter facilmente identificare i layer top e bottom abbiamo aggiunto, a lato scheda, i testi TOP e BOT sui layer corrispondenti.

A questo punto la scheda si può considerare terminata e non è rimasto altro da fare che generare i teardrops, tramite un plugin dedicato, ed esportare i file gerber e drill (che riportiamo negli allegati della presente relazione).

A questo punto abbiamo importato i file gerber in *gerbview* per un controllo finale. Grazie a quest'ultima ispezione abbiamo identificato qualche piccolo errore che abbiamo prontamente corretto. Abbiamo infine mandato tutti i file al Professore, che ha provveduto all'invio di questi al produttore.

4

Firmware

4.1. Ambiente di sviluppo

Come detto inizialmente, l'intero codice del progetto è open-source ed è visionabile al seguente link: <https://github.com/thomasnonis/ppse-2021>.

Per lo sviluppo del codice abbiamo scelto di lavorare con l'ambiente *Microsoft Visual Studio Code*, in quanto molto versatile e continuamente supportato dagli sviluppatori.

Per poter utilizzare le librerie del microcontrollore *RP2040* è stato necessario includere nel progetto il *pico-sdk*, un kit di sviluppo che permette di implementare le istruzioni *HAL* (Hardware Abstraction Level) per potersi interfacciare all'hardware del micro ed alle sue relative periferiche. L'SDK non è l'unica libreria di cui abbiamo fatto affidamento, infatti abbiamo aggiunto al progetto anche *pico-examples* e *picotool*. Con *pico-examples* siamo riusciti a testare delle funzioni base del microcontrollore ed allo stesso tempo a capire il processo di compilazione via *Makefile*, in quanto la libreria era dotata di molteplici esempi base per ogni periferica offerta dal micro, mentre *picotool* è stata utilizzata per poter automatizzare il processo di flashing del codice.

Le librerie citate appaiono nel progetto come *submodules*, in quanto, grazie al software di controllo versione *git* è possibile tenerle aggiornate senza che siano presenti vincoli legati a delle specifiche versioni.

4.2. Compilazione

Durante una fase iniziale di testing abbiamo trovato delle difficoltà per compilare il codice, in quanto nella documentazione ufficiale non era ben chiaro come si dovessero scrivere e disporre in modo corretto i file CMakeLists. Con un po' di lavoro abbiamo deciso di strutturare la compilazione nel seguente modo:

```
SALMO_pico_fw
├── CMakeLists.txt (Project cmake file)
├── build
└── src
    ├── CMakeLists.txt (SALMO.C cmake file)
    ├── your_lib
    │   ├── docs
    │   ├── CMakeLists.txt (lib cmake file)
    │   ├── your_lib.c
    │   └── your_lib.h
```

Figure 4.1: Struttura del codice di SALMO

Ogni libreria contiene un proprio CMakeLists in cui viene specificato il nome della libreria (linkando i file .c e .h) e linkando altre librerie esterne (come *pico_stlplib*).

```
#add all files required to build the library
add_library(sun_tracker sun_tracker.c sun_tracker.h)
#link necessary lib to use printf
target_link_libraries(sun_tracker pico_stdlib)
```

Figure 4.2: CMakeLists della libreria per il calcolo della posizione del sole

Dopodichè nel main principale in cui verrà eseguito il codice (presente in *SALMO.c*), viene definito l'eseguibile (ovvero si linka il file .c attribuendolo ad un nome), dopodiché si linkano le librerie definite precedentemente per poi includere anche le cartelle in cui queste sono presenti, per poi opzionalmente decidere se abilitare altri parametri come *usb output* o *uart output*.

```
if (TARGET tinyusb_device)
    add_executable(SALMO SALMO.c)

    target_link_libraries(SALMO PRIVATE pico_stdlib sun_tracker stepper MPU6050 hardware_i2c hardware_uart hardware_pwm pico_timer HMC5883L SSD1306 lwgps)

    include_directories("../tracking-algorithms")
    include_directories("../timer")
    include_directories("../MPU6050")
    include_directories("../HMC5883L")
    include_directories("../SSD1306")
    include_directories("../stepper")
    include_directories("../nmea-parser")

    # enable usb output, disable uart output
    pico_enable_stdio_usb(SALMO 1)
    pico_enable_stdio_uart(SALMO 1)

    # create map/bin/hex/uf2 file etc.
    pico_add_extra_outputs(SALMO)

elseif(PICO_ON_DEVICE)
    message(WARNING "not building hello_usb because TinyUSB submodule is not initialized in the SDK")
endif()
```

Figure 4.3: CMakeLists di SALMO.c

I file CMakeLists vengono “eseguiti” in ordine gerarchico, per cui è necessario avere un CMakeLists esterno che ha il compito di tenere traccia di tutti gli altri *CMakeLists* e di definire i parametri relativi alla versione del *CMAKE* e della versione del compilatore C e C++. Per compilare il progetto SALMO è sufficiente quindi eseguire il comando CMake sul file CMakeLists più esterno, producendo così un Makefile che, una volta eseguito, compilerà autonomamente l'intero progetto restituendo i file eseguibili necessari per flashare il microcontrollore.

```

cmake_minimum_required(VERSION 3.12)

# Pull in SDK (must be before project)
include(pico_sdk_import.cmake)

project(SALMO C CXX ASM)
set(CMAKE_C_STANDARD 11)
set(CMAKE_CXX_STANDARD 17)

if (PICO_SDK_VERSION_STRING VERSION_LESS "1.3.0")
    message(FATAL_ERROR "Raspberry Pi Pico SDK version 1.3.0 (or later) required. Your version is ${PICO_SDK_VERSION_STRING}")
endif()

set(SALMO_PATH ${PROJECT_SOURCE_DIR})

# Initialize the SDK
pico_sdk_init()

add_compile_options(-Wall
    -Wno-format           # int != int32_t as far as the compiler is concerned because gcc has int32_t as long int
    -Wno-unused-function # we have some for the docs that aren't called
    -Wno-maybe-uninitialized
    )

# Hardware-specific examples in subdirectories:
# declare first library folders and then src folder
add_subdirectory(tracking-algorithm)
add_subdirectory(nmea-parser)
add_subdirectory(SSD1306)
add_subdirectory(HMC5883L)
add_subdirectory(MPU6050)
add_subdirectory(stepper)
add_subdirectory(timer)
add_subdirectory(src)

```

Figure 4.4: CMakeLists del progetto

Per rendere più veloce la fase di compilazione abbiamo deciso di scrivere tre piccoli script *bash*, situati nella cartella *src*: *build.sh*, *flash.sh*, *build_and_flash.sh*: lo script di build utilizza *cmake* per compilare i *Makefile* e salvare i relativi eseguibili dentro la cartella *build*, mentre quello di flash utilizza *picotool* per caricare l'eseguibile in formato *.uf2* direttamente nella *flash* del microcontrollore, anche se il RP2040 non si trova in bootloader mode (*picotool* riavvia il MCU in bootloader mode). Infatti se non usassimo *picotool* dovremmo resettare la scheda tenendo premuto il pulsante di boot per entrare in bootloader mode, successivamente trascinare il file eseguibile all'interno della partizione Mass Storage del micro, per poi resettarlo via pulsante. Lo script di *build* è inoltre dotato di un *flag* opzionale attivabile scrivendo *./build.sh all*, con il quale viene rimossa l'intera directory di build ed il progetto viene ricompilato da capo (utile in caso di errori del compilatore non realmente presenti nel codice). Abbiamo scelto di flashare mediante l'eseguibile con estensione *.uf2* in quanto, utilizzando questo tipo di file, è sufficiente copiare l'eseguibile nella partizione di tipo Mass Storage che il RP2040 offre una volta avviato in bootloader mode.

Come intuibile, il file *build_and_flash.sh* esegue sia le operazioni di compilazione sia l'operazione di flashing.

4.3. Algoritmo per ottenere la posizione del sole

Al fine di poter pilotare correttamente i motori verso il sole abbiamo dovuto trovare un modo per convertire i dati ottenuti dal GPS in elevazione ed azimuth del sole (espressi in gradi). Per approcciare correttamente il problema abbiamo iniziato a leggere vari *papers* e relazioni tecniche riguardo ad algoritmi per la rilevazione della posizione del sole, il primo, forse tra i più datati, è stato *The astronomical almanac's algorithm for approximate solar position* di Joseph J. Michalsky. Dopo alcune letture abbiamo capito che era necessario passare tra vari sistemi di riferimento per ottenere le coordinate del sole, in particolare ci siamo imbattuti nel concetto di *Julian Date* (giorni passati dalle 12 del 1 Gennaio 4713 B.C.), coordinate celesti, *GMST* (Greenwich Mean Sidereal Time, un giorno siderale è 4 minuti più corto di un giorno solare) e *LMST* (Local Mean Sidereal Time). Nonostante nel paper di *Michalsky* fosse presente del codice *FORTRAN* per l'implementazione dell'algoritmo, non lo abbiamo ritenuto

abbastanza “solido” ed efficiente per il nostro tipo di applicazione.

Per poter verificare la corretta posizione del sole abbiamo utilizzato questo sito come contro prova dei nostri calcoli: <https://gml.noaa.gov/grad/solcalc/>.

The screenshot shows the NOAA solcalc web interface. In the 'Location' section, the latitude is 41.9, longitude is 12.48, and time zone is Europe/Rome. In the 'Date' section, the day is 10, month is May, and year is 2022. The 'Result' section displays the following data:

Equation of Time (minutes):	Solar Declination (in °):	Solar Noon (hh:mm:ss):	Apparent Sunrise (hh:mm):	Apparent Sunset (hh:mm):	Az/El (in °) at Local Time:
3.58	17.65	13:06:31	05:55	20:18	109.25 45.41

Checkboxes for 'Show Sunrise', 'Show Sunset', and 'Show Azimuth' are present.

Figure 4.5: Interfaccia web NOAA solcalc

Fortunatamente la NOAA (National Oceanic and Atmospheric Administration) oltre a fornire questa interfaccia web, condivide anche i file excel con cui vengono eseguiti i calcoli al fine di ottenere gli stessi risultati presenti sul loro sito. Di conseguenza, essendo questo tool uno dei pochi affidabili in rete con cui poter verificare i nostri calcoli, abbiamo deciso di riportare le formule del file excel in codice C per ottenere gli stessi risultati. I calcoli intermedi non verranno riportati nel dettaglio in quanto sono computazioni astronomiche che non rientrano nelle tematiche trattate nel corso.

Per la computazione abbiamo fatto riferimento a due strutture dati, una denominata *Place* in cui viene salvata la data, l’orario e latitudine e longitudine “parsate” dal GPS (mediante una libreria open source), e l’altra, *Position*, in cui vengono salvati tutti i parametri intermedi ed i risultati finali per il calcolo della posizione del sole (basato sull’algoritmo della NOAA).

```
typedef struct Place{
    int year, month, day, hour, minute;
    double second, latitude, longitude;
} Place;
```

Figure 4.6: Struct "Place"

```
typedef struct Position{
    double jd, julian_days_since_epoch, julian_centuries_since_epoch;
    double mean_longitude, mean_anomaly;
    double eccentricity, sun_eq_of_center;
    double sun_true_longitude, sun_true_anomaly;
    double sun_rad_vector, sun_app_long;
    double obliquity_corr, mean_obliquity_ecliptic;
    double right_ascension, declination;
    double gmst, lmst, eq_of_time, hour_angle, elevation, azimuth, refraction;
} Position;
```

Figure 4.7: Struct "Position"

Per calcolare la posizione esatta del sole infatti i dati contenuti in *Place* ven gono utilizzati per ottenere una *Julian Date*, una *Julian Date* conteggiata a partire dalle 12 del primo Gennaio 2000 e successivamente tutti i parametri intermedi per poi ottenere elevazione e azimuth del sole. In seguito ad alcuni test abbiamo riscontrato che siamo in grado di ottenere un'ottima precisione, addirittura nell'ordine del grado!

4.4. Implementazione software delle periferiche

Tutte le periferiche, a meno di GPS, utilizzano interfaccia di comunicazione I2C.

Il protocollo hardware I2C prevede due linee seriali di comunicazione: SDA (Serial DAta) per i dati e SCL (Serial CLock) per il clock (la comunicazione I2C è di tipo sincrono).

Siccome le linee SDA ed SCL sono di tipo Open-Drain (o Open-Collector a seconda della tecnologia utilizzata) sono necessari dei resistori di pull-up per portare a livello logico alto le linee. Nel progetto SALMO, siccome accelerometro e magnetometro vengono acquistati sotto forma di moduli commerciali, troviamo già i resistori di pull-up su di essi, per cui, questi ultimi, verranno dimensionati e montati direttamente in solo uno dei moduli (verranno quindi rimossi dagli altri).

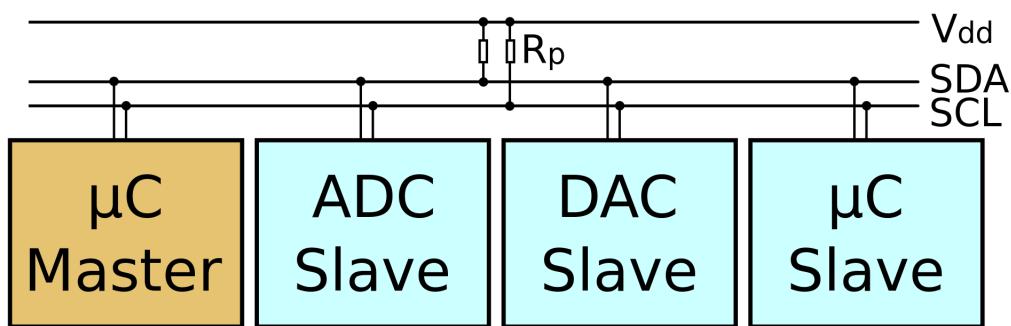


Figure 4.8: Esempio di bus I2C

La struttura dei driver di HMC5883L e MPU6050 segue una logica ben precisa, in modo da rendere particolarmente semplice e versatile il codice:

- <drivername>_defines.h
- <drivername>_I2C.h
- <drivername>_I2C.c
- <drivername>.c
- <drivername>.h

Utilizzando questa struttura è possibile innanzitutto separare tutte le definizioni di registri ed indirizzi in un file header separato (<drivername>_defines.h) ed in secondo luogo, forse vantaggio principale, è possibile dichiarare e definire le funzioni di lettura e scrittura nel bus I2C in due soli file (<drivername>_I2C.h e <drivername>_I2C.c) in modo che il vero e proprio driver (<drivername>.h e <drivername>.c) rimanga completamente indipendente dalle funzioni I2C. Questo, è particolarmente vantaggioso perché le funzioni I2C sono dipendenti dal HAL offerto dall'SDK (e quindi in certo senso dall'architettura), per cui nel caso in cui volessimo ri-utilizzare i driver per un'altra architettura (es. STM32) sarebbe sufficiente modificare solamente le due funzioni di lettura e scrittura su bus I2C nel file <drivername>_I2C.c.

Il microcontrollore offre due controller I2C: I2C0 ed I2C1. Nel progetto SALMO viene utilizzato solo uno dei due controller, che viene condiviso da accelerometro, magnetometro e display.

Tutte e 3 le periferiche utilizzano il bus in modalità fast, ovvero a 400KHz.

Per effettuare una lettura o la scrittura di un registro di una periferica nel bus è necessario che il master invii un messaggio specifico determinato dal protocollo I2C: bit di start, 7, 8 oppure 10 bit di indirizzo (HMC5883L utilizza 8 bit, mentre MPU6050 e SSD1306 utilizzano 7 bit di indirizzo), un bit che indica lettura o scrittura, un bit di acknowledge da parte dello slave, e successivamente 8 bit di dato (ad esempio il registro da leggere o il registro in cui si vuole scrivere, seguito di nuovo da un bit di acknowledge).

È possibile inviare N blocchi da 8 bit seguiti dal bit di acknowledge dopo il bit di read/write (anch'esso seguito da un bit di acknowledge).

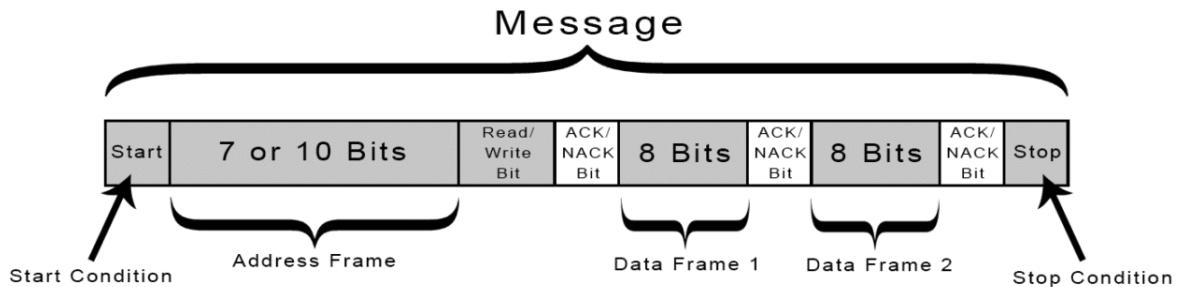


Figure 4.9: Data frame protocollo I2C

Nel caso, ad esempio, in cui si voglia scrivere il contenuto di un registro del MPU6050 è sufficiente inviare: bit di start, 7 bit di indirizzo (LSB configurabile), bit di read (es. 0 1101000 1) che saranno seguiti da un bit di acknowledge ed infine invieremo gli 8 bit dell'indirizzo su cui vogliamo scrivere (es. 00111011). Successivamente invieremo, dopo aver ricevuto l'acknowledge, i dati che vogliamo scrivere.

Nel caso della lettura invece, è necessario ripetere la procedura illustrata precedentemente fino all'invio dell'indirizzo del registro (in questo caso da leggere) ed inviare però ora un bit di start seguito dall'indirizzo dello slave ed infine un bit di read. La periferica slave invierà finalmente, sempre dopo il bit di acknowledge, il contenuto del registro richiesto.

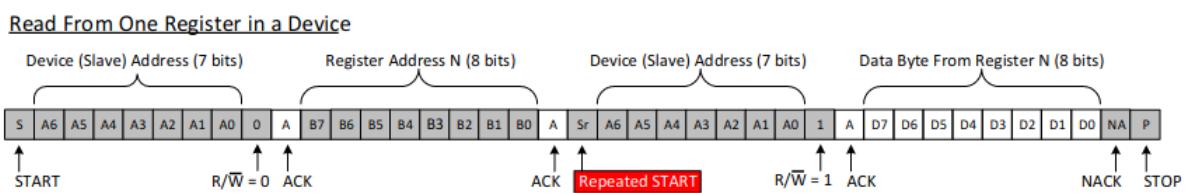


Figure 9. Example I²C Read from Slave Device's Register

Figure 4.10: Data frame esempio lettura I2C

```

void MPU6050_I2C_ByteWrite(uint8_t slaveAddr, uint8_t* pBuffer, uint8_t writeAddr){
    uint8_t buffer[2];
    buffer[0]=writeAddr;
    buffer[1]=*pBuffer;

    int ret=0;
    ret=i2c_write_timeout_us(I2C_PERIPHERAL, slaveAddr, buffer, 2, false, 1000); //1ms timeout max
    if(ret==PICO_ERROR_GENERIC||ret==PICO_ERROR_TIMEOUT){
        printf("MPU6050_I2C_ByteWrite: device not present or timeout reached\n");
    }
};

void MPU6050_I2C_BufferRead(uint8_t slaveAddr,uint8_t* pBuffer, uint8_t readAddr, uint16_t NumByteToRead){
    int ret=0;
    // before read the protocol needs to send the register address to the slave (see i2c protocol)
    ret=i2c_write_timeout_us(I2C_PERIPHERAL, slaveAddr, &readAddr, 1, true, 1000); //1ms timeout max
    if(ret==PICO_ERROR_GENERIC||ret==PICO_ERROR_TIMEOUT){
        printf("MPU6050_I2C_BufferRead: device not present or timeout reached\n");
    }
    ret=i2c_read_timeout_us(I2C_PERIPHERAL, slaveAddr, pBuffer, NumByteToRead, false, 1000); //1ms timeout max
};

```

Figure 4.11: Funzioni di write e read I2C (driver MPU6050)

Per quanto concerne l'implementazione software del display OLED, abbiamo deciso di utilizzare la libreria già presente negli SDK di Raspberry Pico (seppur per nulla completa). Sottolineiamo che attualmente il driver di questa periferica non è stato completato poiché, non avendola sotto mano, abbiamo deciso di prioritizzare il resto del lavoro.

Un altro protocollo di comunicazione utilizzato nel progetto è *UART (Universal Asynchronous Receiver-Transmitter)*, come visto nel primo modulo del corso, permette il trasferimento e ricezione di dati in seriale ad un tasso di velocità bit/s fissato, detto baud rate. Abbiamo inserito la configurazione *UART* nel modulo *PAM7Q.c*, file in cui vengono inizializzati i pin RX e TX, e vengono settati tutti i parametri necessari a comporre il data frame, assumendo la seguente configurazione:

- Hardware flow: no
- Baud rate: 9600 bit/s
- Bit di dato: 8
- Bit di stop: 1
- Bit di parità: no

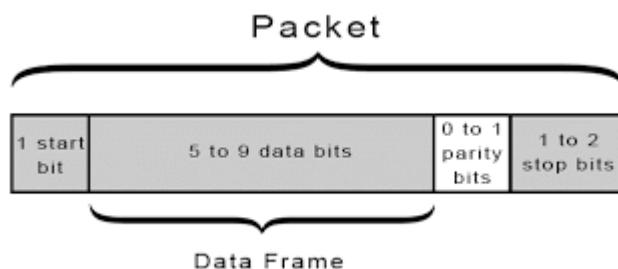


Figure 4.12: Pacchetto gestito con comunicazione UART

Il modulo GPS comunica in *UART* con il microcontrollore mediante protocollo a 9600 bit/s, codificando i messaggi con standard NMEA 1803 (National Marine Electronics Association). Per la lettura dei dati viene popolato un buffer, di lunghezza 100 bytes, con ogni carattere ricevuto. All'occorrenza del carattere “\n” la stringa ricevuta viene salvata in un array di supporto, il buffer viene successivamente svuotato e ricomincia la lettura per un numero pari a *GPS_MAX_SENTENCES*. Abbiamo inoltre deciso di comunicare con il GPS in polling, anziché utilizzare un interrupt, poiché interagendo con il modulo ad intervalli ben definiti la seconda opzione avrebbe portato ad un inutile overhead a causa della continua interruzione del microcontrollore rispetto alle sue principali task.

Per tradurre le stringhe NMEA in informazioni utili alla struttura *Place*, definita nel modulo dell'algoritmo, è stata utilizzata una libreria open source in grado di analizzare questi messaggi (vedasi bibliografia). Tipicamente una stringa NMEA è composta nel seguente modo:

*\$PREFISSO, dato1, dato2 ... datoN – 1, datoN * CHECKSUM*

In base al tipo di prefisso vengono trasferiti diversi tipi di dati, per la nostra applicazione abbiamo fatto principale affidamento ai due messaggi:

- *\$GPRMC*: fornisce la data (Anno, Mese, Giorno)
- *\$GPGGA*: fornisce tempo (Ore, Minuti, Secondi), Latitudine, Longitudine.

Abbiamo valutato la correttezza del parsing utilizzando un tool online (vedasi bibliografia)

4.5. Implementazione driver motori

Il firmware di controllo dei motori è stato sviluppato partendo da una libreria già esistente e open source disponibile su github ed è composta da due file: *pico stepper.h* e *pico stepper.c*.

Nel primo è definita la *struct* necessaria per inizializzare ogni motore e gli *header* delle funzioni implementate poi nel secondo file.

Per interfacciarsi ai motori è stato necessario inizializzare la *struct* e i pin fisici delle 4 fasi, definire la velocità di rotazione del motore, successivamente i motori vengono comandati mediante la funzione *step()*, che implementa il metodo “wave drive”, nel quale le fasi vengono pilotate in modo sequenziale

da un'onda quadra con un duty cycle del 25%. La forma d'onda non è generata tramite la periferica *pwm* bensì in *bit banging* per semplicità di scrittura firmware dato che non erano necessarie velocità elevate di commutazione.

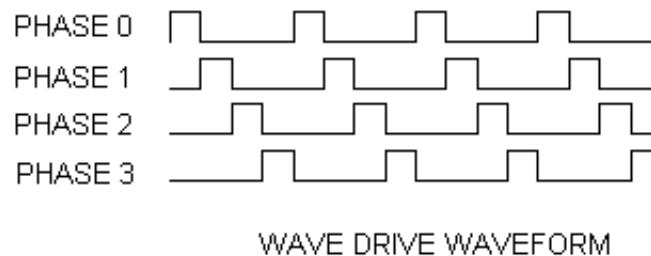


Figure 4.13: Fasi per pilotaggio stepper unipolare



Figure 4.14: Output SALMO su oscilloscopio

4.6. Funzionamento generale del codice

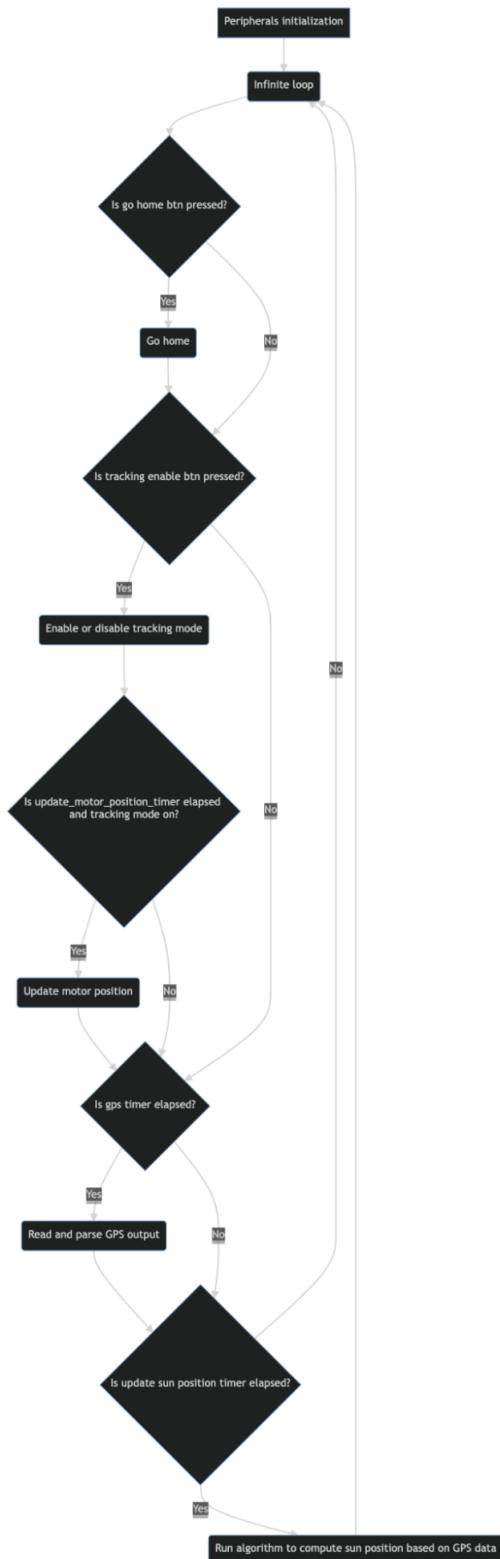


Figure 4.15: Flowchart del funzionamento del codice

Come prima cosa vengono inizializzate le varie periferiche e moduli ovvero:

- UART

- USB CDC
- I₂C
- Accelerometro
- Magnetometro
- Motore stepper

Dopodiché il microcontrollore entra in loop infinito in cui vengono controllati degli input e delle variabili interne impostate dai timer. Sono presenti tre timer che scandiscono:

- Lettura input gps (ogni 4 secondi)
- Aggiornamento della posizione del motore (ogni 10 secondi)
- Aggiornamento posizione del sole mediante algoritmo (ogni 4 secondi)

Entrati nel loop infinito viene controllato, tramite interrupt, se il pulsante di “Go home” (SW3) viene premuto, qualora succeda, i motori vengono azionati per dirigere il pannello verso il Nord ottenuto dalla bussola ed a 45° di elevazione (tilt). Successivamente viene verificato, sempre tramite interrupt, se il pulsante di “Tracking enable/disable” (SW4) viene premuto, in questo caso il flag di tracking viene abilitato o disabilitato in base al numero di iterazioni col pulsante, infatti ad ogni pressione il flag viene negato, pertanto dopo averlo premuto una volta il motore rimarrà in tracking mode e se il pulsante verrà azionato di nuovo il flag di tracking verrà disabilitato. Alla scadenza del timer relativo al tracking se il flag dovuto al pulsante è abilitato viene aggiornata la posizione del motore. Dopodichè viene controllato se il timer relativo alla lettura del gps è terminato, in caso affermativo viene eseguita la lettura via uart ed il relativo parsing, infine viene controllato se il timer relativo alla posizione del sole è scaduto, e di conseguenza viene o meno eseguito il calcolo della posizione del sole basandosi sui dati ottenuti dal GPS. Il programma poi ritorna nel loop principale e continua il controllo delle variabili.

A causa della mancanza di accelerometro, pannello e struttura di supporto per i motori, ulteriori feature verranno implementate in futuro.

5

Assemblaggio della scheda

Come già accennato in precedenza, una volta finita la fase di progetto della scheda, abbiamo inviato al professore tutti i file relativi alla produzione (Gerber, Drill e BOM), il quale ha poi provveduto ad inviare le PCB in produzione, ordinando anche uno stencil, e ad effettuare gli ordini per i componenti necessari.

Dopo circa due settimane dall'invio eravamo finalmente in possesso delle schede e della maggior parte dei componenti necessari, quindi abbiamo potuto cominciare l'assemblaggio, che è stato svolto sotto la supervisione del Professore presso il laboratorio di elettronica del FabLab.

Prima di qualsiasi cosa abbiamo effettuato un'ispezione visiva delle PCB sotto una lente per verificarne la correttezza; grazie a questa abbiamo identificato un solo problema: il marker sul layer silkscreen per il pin 1 del microcontrollore non è stato stampato, probabilmente perché era troppo piccolo.

Abbiamo poi ispezionato anche lo stencil per la pasta saldante ed abbiamo notato che il produttore non ha usato come riferimento il layer apposito, ma ha usato quello per la soldermask. Per questo motivo sullo stencil erano presenti aperture anche in corrispondenza dei pad through-hole del gps (che abbiamo provveduto a coprire con del nastro kapton) e le aperture dei fiducials sono risultate più grandi del previsto, anche se questo non ha alcuna importanza per la realizzazione, quindi siamo passati allo step successivo.

Per iniziare l'assemblaggio abbiamo fissato una PCB con altre 4 utilizzando del nastro ed abbiamo allineato lo stencil fissandolo da un lato con altro nastro per poterlo alzare una volta finito, come in figura.

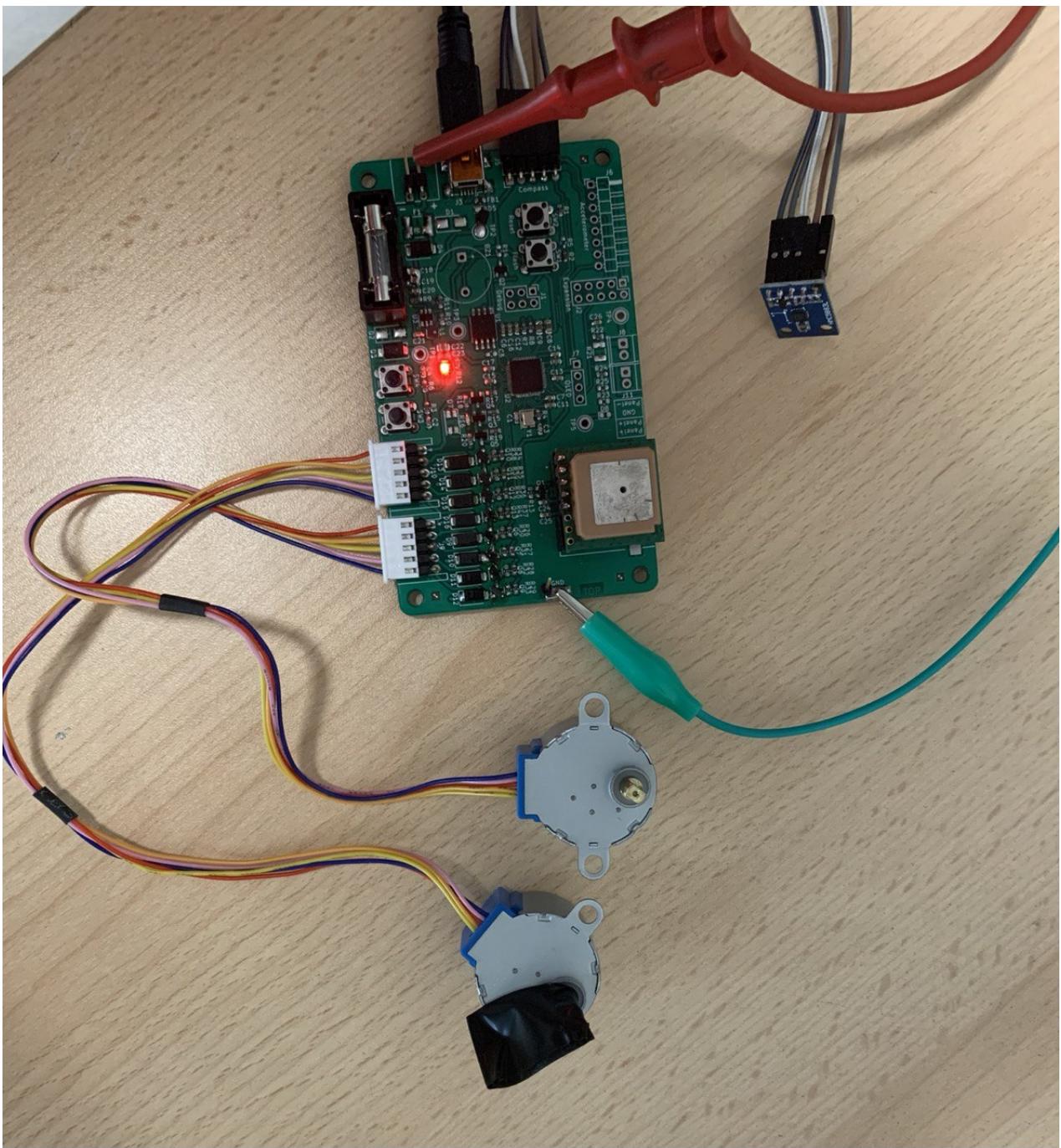


Figure 5.1: Scheda SALMO completa di componenti essenziali

Abbiamo quindi depositato, con l'aiuto di una spatola, la pasta saldante, controllando di riempire ogni apertura con una quantità adeguata. Verificata la corretta applicazione abbiamo tolto la scheda dal supporto da noi creato ed abbiamo cominciato ad applicare i componenti SMD con l'aiuto di pinzette di precisione e controllando continuamente il layout.

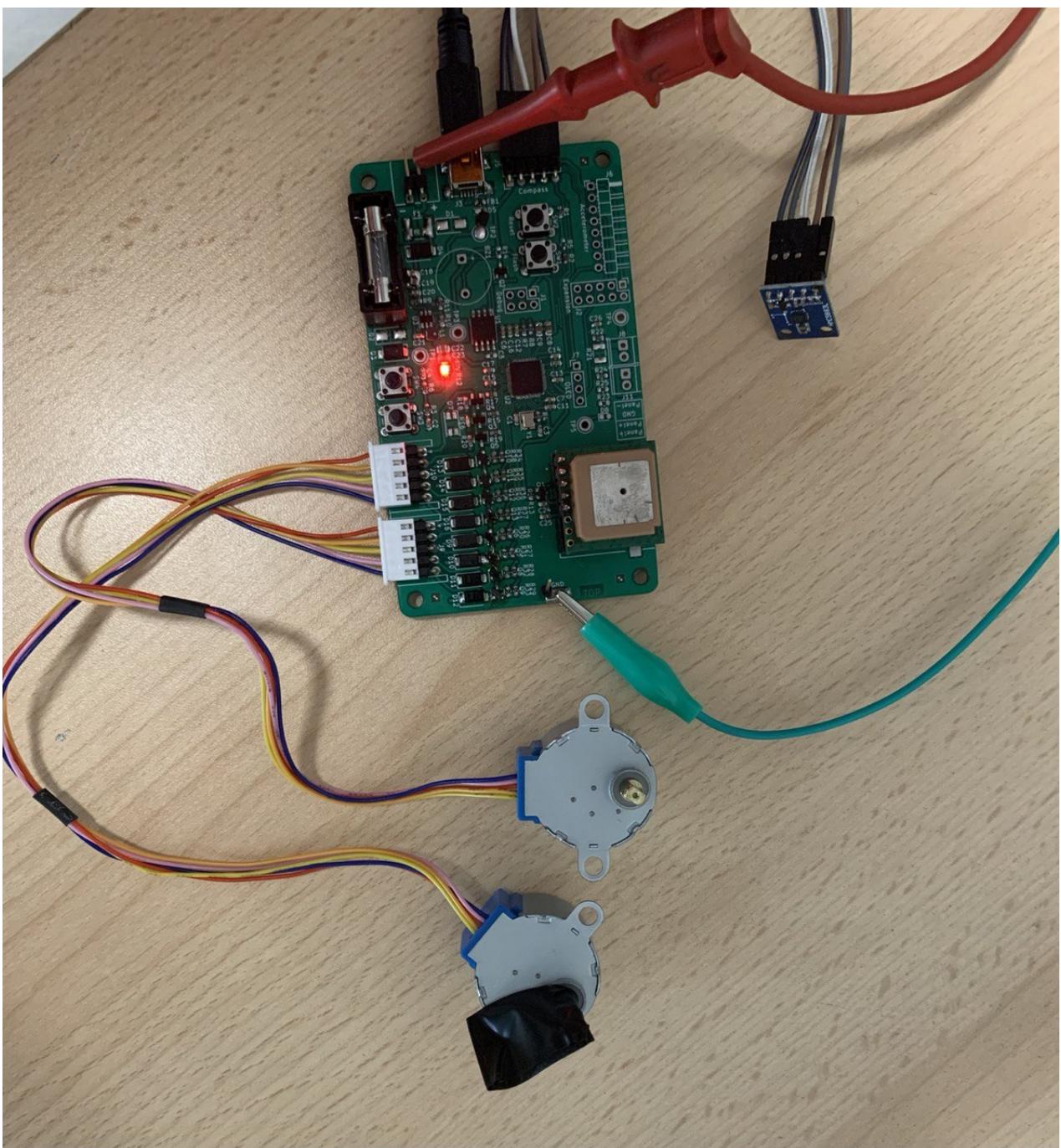


Figure 5.2: Scheda SALMO completa di componenti essenziali

Per tenere traccia del progresso abbiamo usato estensivamente un utilissimo plugin per KiCad chiamato iBOM, che permette di visualizzare il layout del PCB, associando ad ogni componente la sua posizione in una grafica molto intuitiva, e di contrassegnare ogni componente come piazzato.

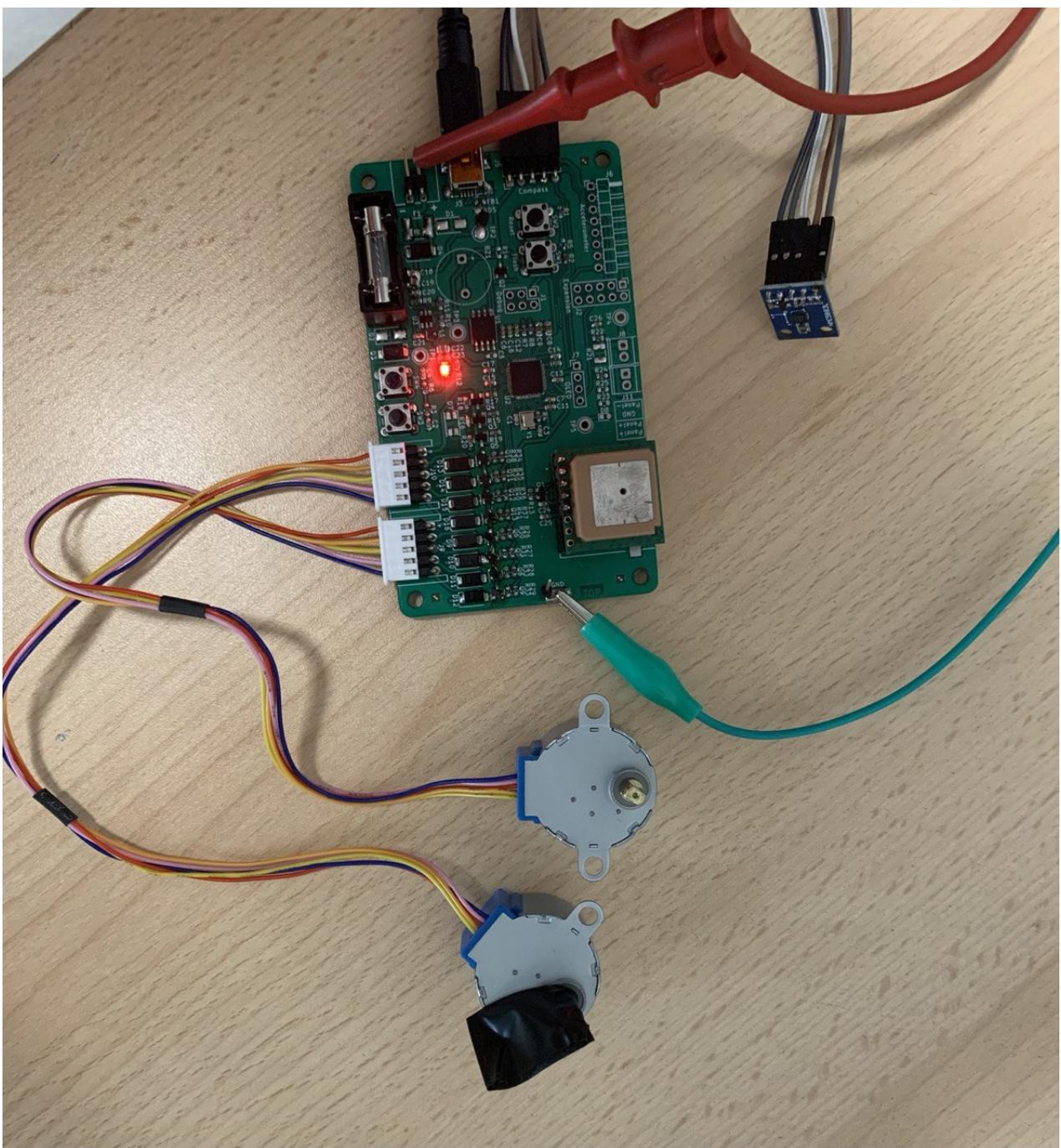


Figure 5.3: Scheda SALMO completa di componenti essenziali

Alcuni dei componenti utilizzati presentavano un modello diverso da quello previsto in fase di progettazione a seconda delle disponibilità in-house del professore ma, essendo una scheda di prototipazione e non una scheda di produzione, sono risultati ugualmente adatti, senza alterare il funzionamento della scheda. Infine, abbiamo posto la scheda su di una piastra elettrica riscaldante controllata per fare il reflow della pasta e saldare definitivamente i componenti SMD. A questo punto abbiamo saldato manualmente, con un classico saldatore a stilo, i restanti componenti THT, ovvero buzzer, pulsanti, portafusibile e connettori.

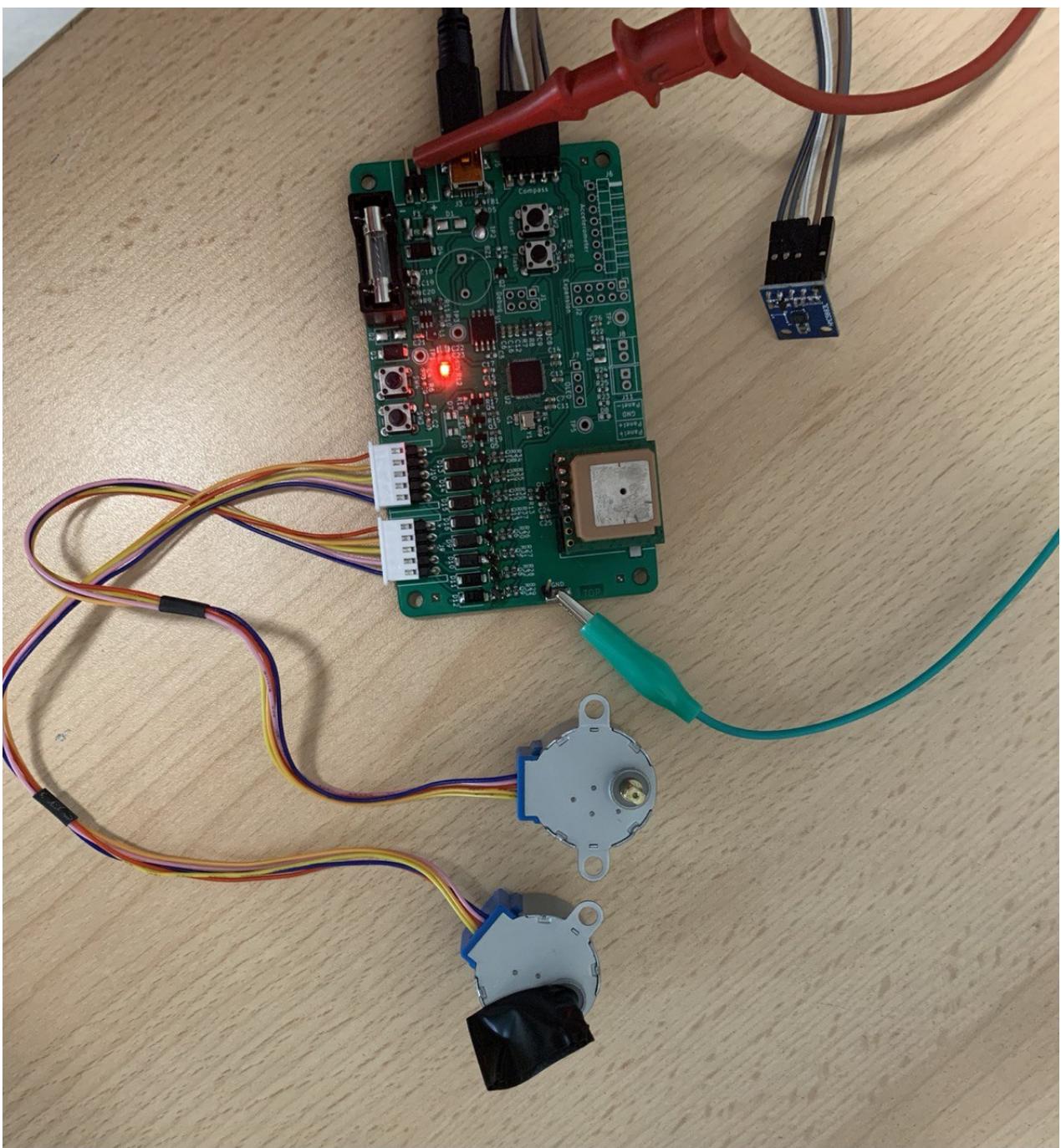


Figure 5.4: Scheda SALMO completa di componenti essenziali

Abbiamo ripetuto questi passaggi per sei volte, realizzando così sei PCBA, uno per ogni membro del gruppo più uno per il professore.

Finito l'assemblaggio, ci siamo accorti che per una particolare scheda i pin del microcontrollore non sembravano essere fissati a dovere ed infatti testando la sua alimentazione abbiamo potuto constatare il surriscaldamento eccessivo del componente. Perciò per rimediare all'errore il microcontrollore è stato dissaldato, ma nel farlo, le tre piste che collegavano i rispettivi pin si sono alzate, non consentendo così l'utilizzo della scheda. Un altro errore è sorto durante la fase di testing, infatti, misurando il segnale di comando in input ai motori, ci siamo accorti che al transistor a canale N montato sulla scheda era stata assegnata la piedinatura scorretta. Pertanto, abbiamo dovuto dissaldare i transistor atti al pilotaggio dei motori per poi saldarli nuovamente nel verso corretto.

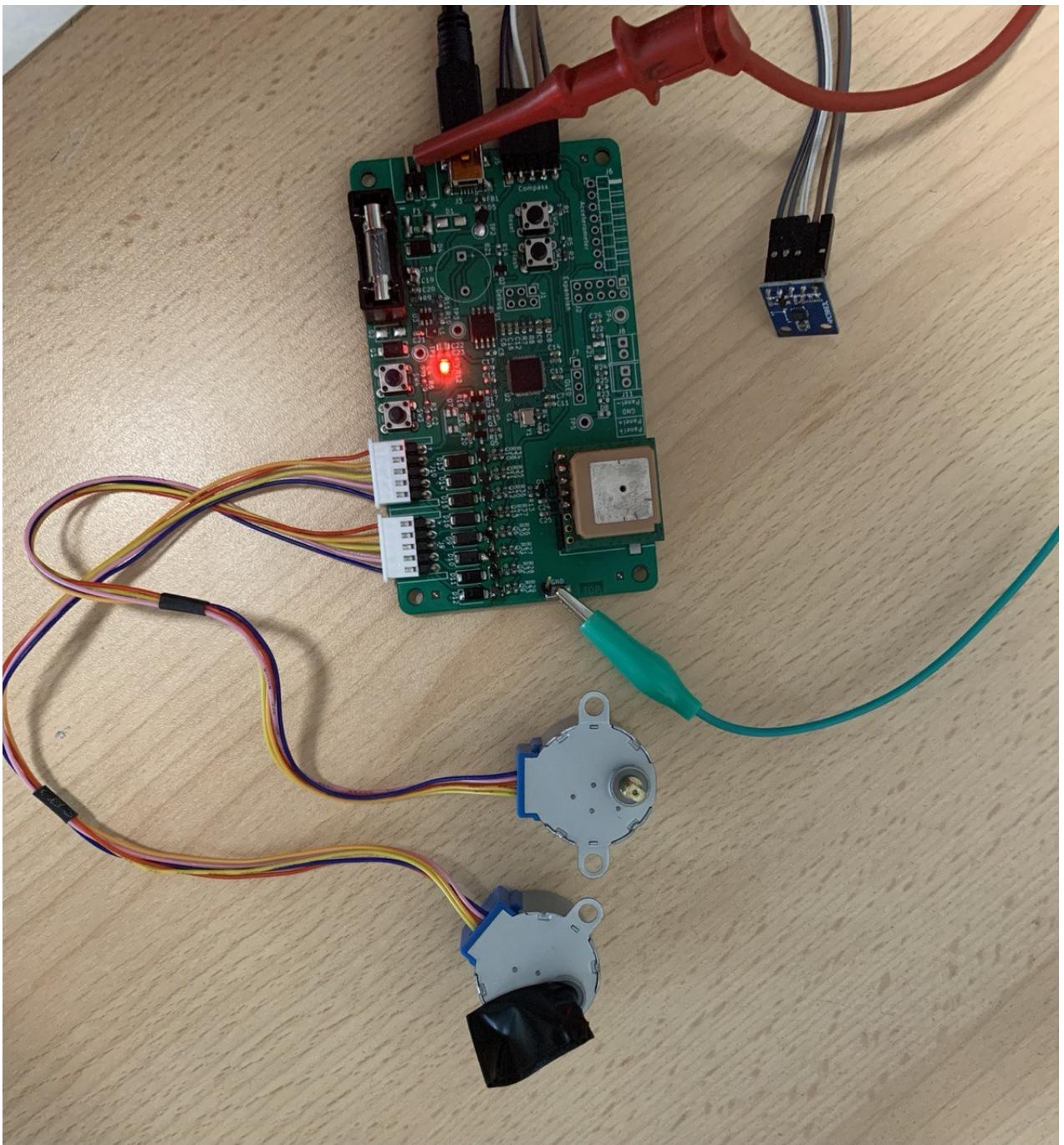


Figure 5.5: Scheda SALMO completa di componenti essenziali

6

Testing della scheda

Testando con l'oscilloscopio un segnale di prova di pilotaggio dei motori ci siamo accorti che in uscita non si aveva il segnale desiderato. Perciò prima abbiamo testato il segnale PWM generato dal microcontrollore, il quale è risultato corretto.

Poi siamo passati al test sul MOSFET di pilotaggio.

Qui ci siamo accorti che il segnale era sbagliato, quindi abbiamo intuito che il problema fossero i transistor. Il professore ci ha poi aiutati a capire nel dettaglio il problema misurando la tensione ai capi del diodo di protezione tra drain e source del MOSFET. Risultando un cortocircuito tra drain e source abbiamo capito che quelli non erano realmente il drain e il source del MOSFET e che quindi il transistor era stato saldato in modo errato. Effettuando la misurazione di tensione tra i vari pin del transistor abbiamo capito quale fosse il gate. Successivamente abbiamo girato i transistor come descritto nell'assemblaggio e, dopo altri test, il sistema di pilotaggio degli stepper è risultato funzionante.

Il consumo del prototipo è stato misurato all'avviamento dei motori e in modalità idle, dai quali abbiamo ricavato le seguenti misure:

- consumo istantaneo a 1/1.5 W
- consumo in idle 0.2/0.4 W

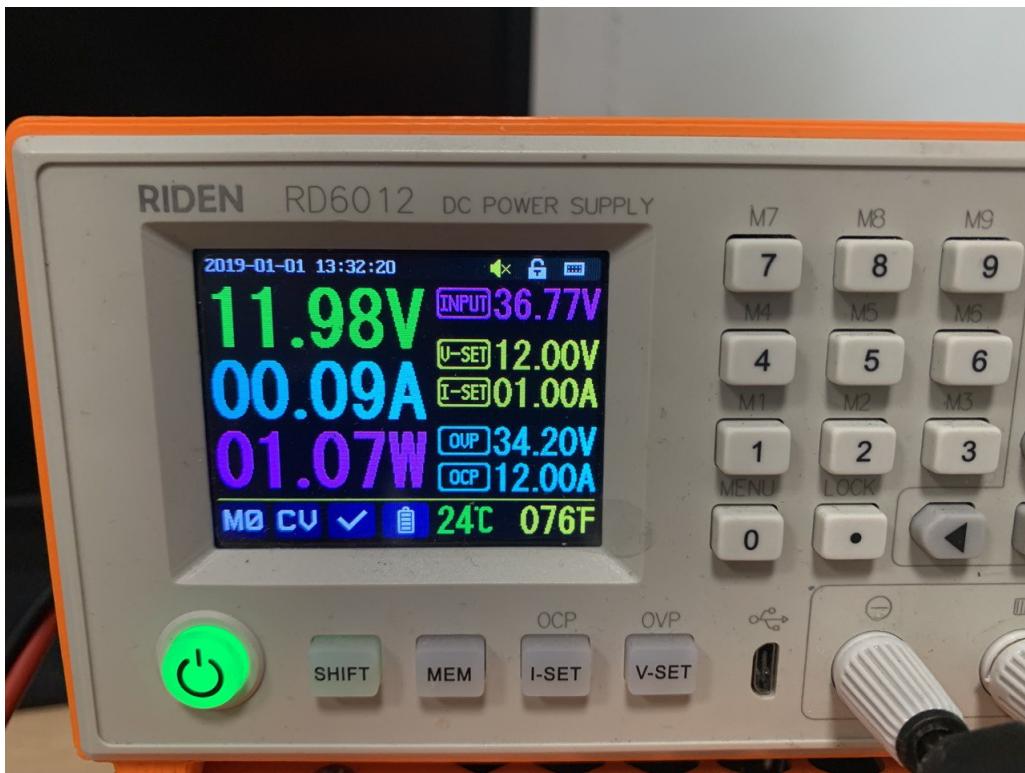


Figure 6.1: Consumi durante il movimento dei motori

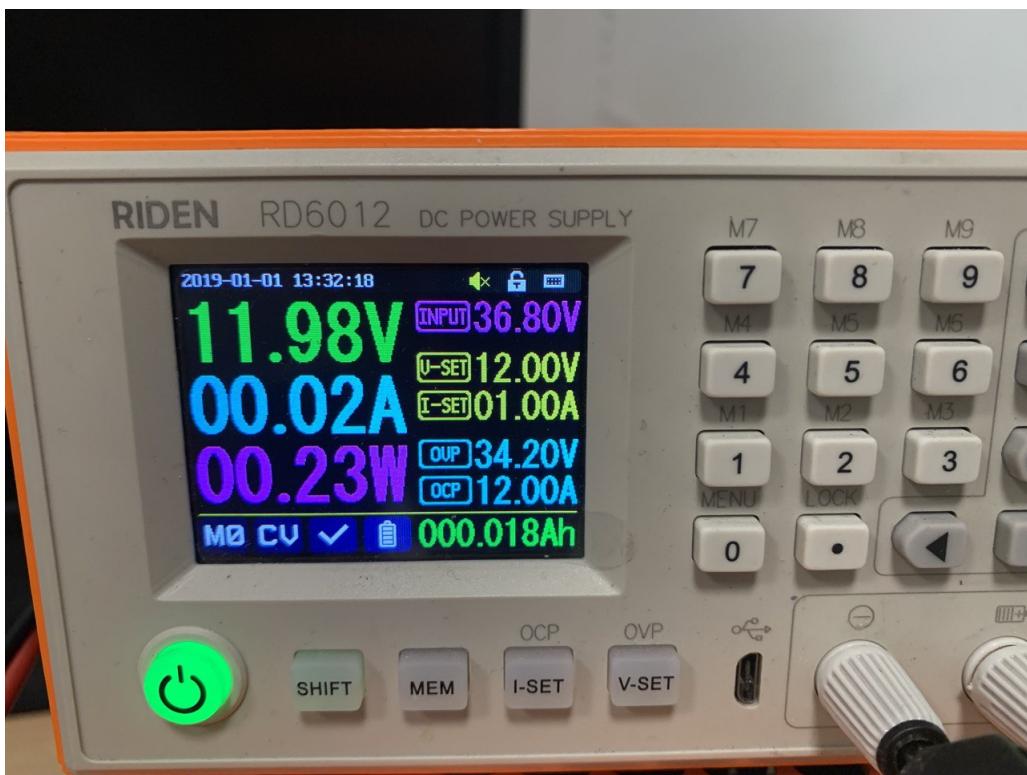


Figure 6.2: Consumi durante lo stato di idle

I consumi dei motori sono complessivamente bassi poiché, grazie alla loro coppia statica elevata, non è necessario alimentarli per far sì che mantengano la posizione.

7

Implementazioni future

Considerando i componenti mancanti, le funzionalità che vorremmo implementare in un prossimo futuro sarebbero le seguenti:

- Interfaccia utente attraverso il display OLED
- Sensing di tensione e corrente prodotti dal pannello
- Struttura per pannello, motori e sensori
- Interfaccia Python via COM port con visualizzazione valori, statistiche e controllo manuale motori

L'interfaccia utente nel display OLED mostrerà: tensione pannello e corrente pannello, posizione del sole e posizione pannello (angolo di Azimuth e angolo di Elevazione), orario e posizione geografica.

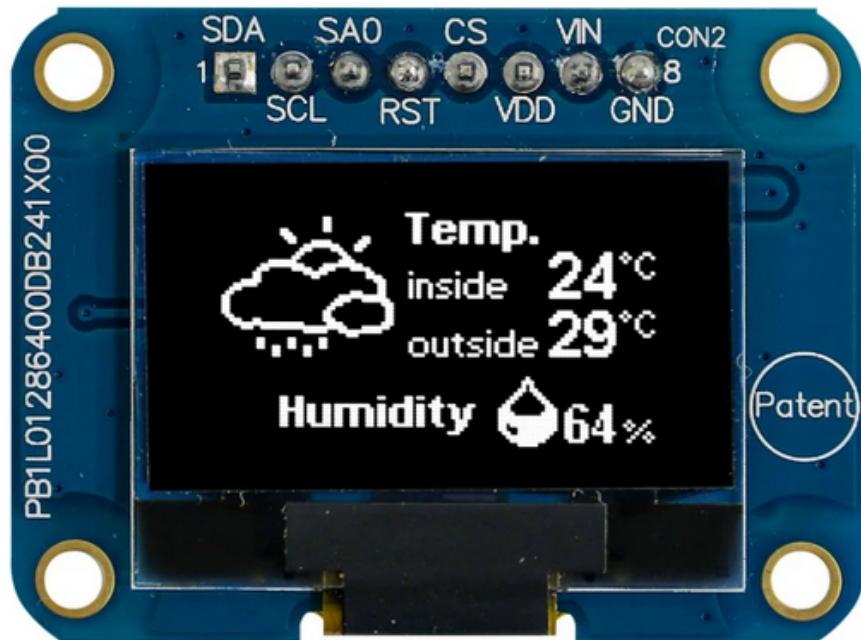


Figure 7.1: Esempio di interfaccia utente su display OLED

La struttura di supporto per il pannello servirà prima di tutto a consentire al pannello di muoversi nei due gradi di libertà (rotazione intorno all'asse Z e rotazione intorno all'asse X) e poi per contenere magnetometro ed accelerometro in modo che questi rilevino esattamente la posizione del pannello (dovranno quindi essere fisicamente attaccati al pannello e dovranno muoversi con esso).

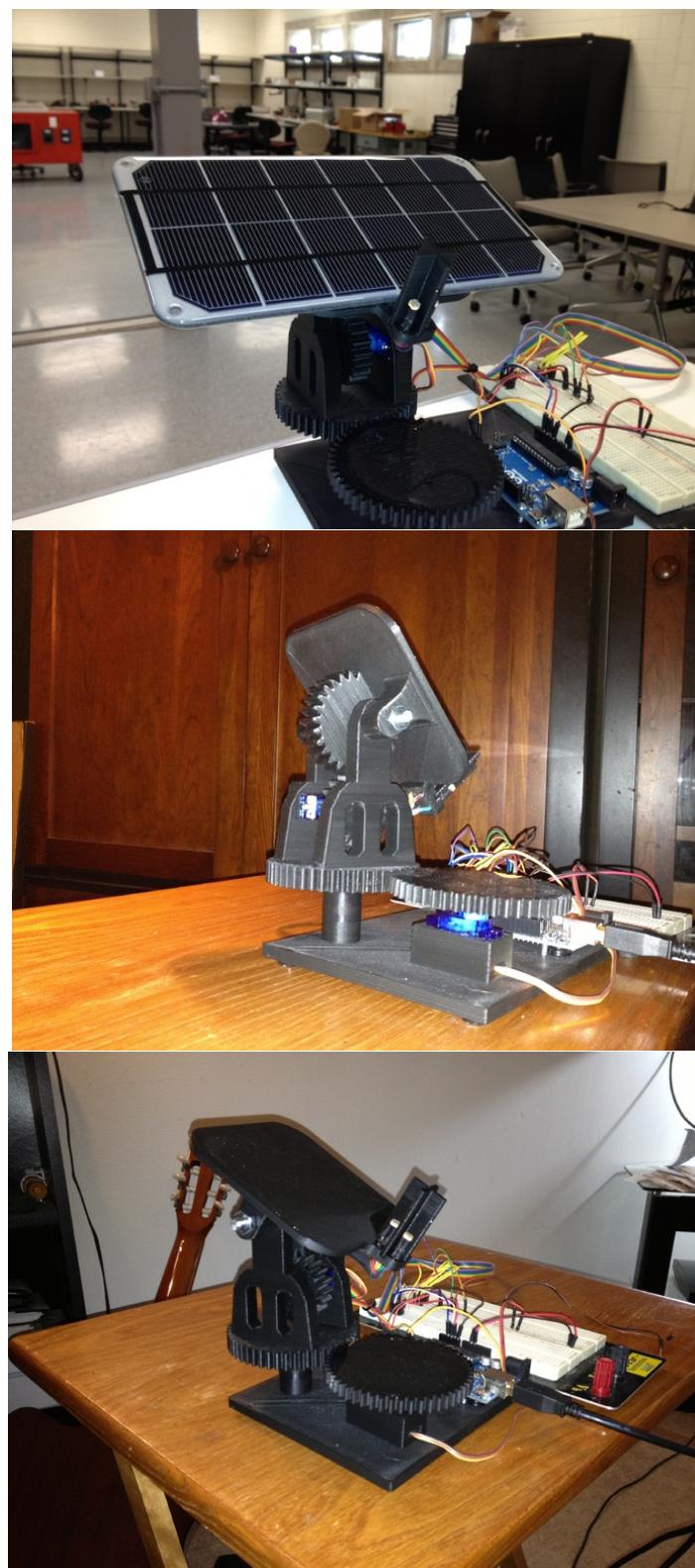


Figure 7.2: Esempio di struttura di supporto per pannello e motori di tipo servo
(Credits:<https://www.thingiverse.com/thing:53321>)

Per quanto riguarda l'interfaccia grafica per la connessione seriale via USB, si pensava di sviluppare in GUI (Graphical User Interface) in linguaggio Python che consentisse di controllare manualmente le posizioni dei motori, di abilitare il tracking o il return to home e che, attraverso l'uso di piccoli grafici, mostrasse delle semplici statistiche sulla potenza e l'energia prodotte dal pannello fotovoltaico.

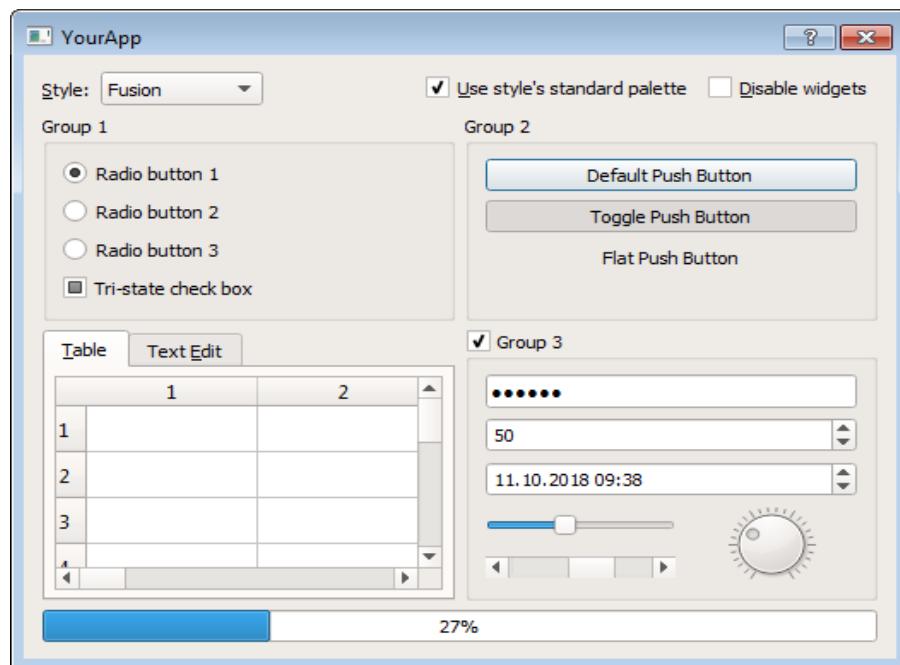


Figure 7.3: Esempio di GUI sviluppata in python

8

Conclusioni

8.1. Resoconto esperienza

Il corso è stato formativo sotto tutti i punti di vista poiché abbiamo potuto realizzare una scheda partendo da zero, cosa che negli altri corsi non avevamo mai fatto. Per alcuni quindi è stata la prima PCB progettata, perciò una tappa molto importante per il loro lavoro o studio futuro. Inoltre il fatto di aver programmato il microcontrollore con le poche basi di programmazione che avevamo dai corsi precedenti ci ha permesso di imparare molto a livello firmware. Siccome il progetto non è ancora completo, l'intenzione è quella di aggiungere man mano i componenti mancanti al fine di avere la scheda più simile possibile a quanto era stato programmato. Per concludere, i concetti imparati nel modulo 1 del corso sono stati di fondamentale importanza per la realizzazione della scheda, senza i quali sarebbe stato molto difficile avere un'idea di come impostare il lavoro. L'obiettivo di progettare in modo consono una scheda elettronica è dunque stato raggiunto.

8.2. Lezioni imparate

Ragionando a posteriori su quanto abbiamo fatto, abbiamo stilato un elenco di cosa avremmo potuto fare meglio e che cercheremo di applicare nei progetti futuri:

- Il fanout del microcontrollore lo avremmo potuto fare esclusivamente con tracce da 0.2mm, senza ricorrere ad espansioni a 0.3mm, almeno per le tracce più corte. Questo avrebbe semplificato di non poco il routing di quella sezione;
- Considerare fin da subito i return paths per i condensatori di decoupling. Nella realizzazione non abbiamo pensato a collegare alcuna traccia per il GND, perché ci siamo affidati ai poligoni di riempimento. Questo si è rivelato un problema, perché alla fine del progetto abbiamo riscontrato che molti riempimenti risultavano scollegati da GND e, soprattutto, che i percorsi effettivi delle linee di GND dei condensatori di bypass del microcontrollore sono risultati molto lunghi, il che rende quasi inutile il fatto di posizionarli vicini ai pin;
- Controllare meglio il posizionamento dei refdes sul layer di silkscreen. Abbiamo realizzato solo a scheda prodotta che il refdes di C10 è stato posizionato sopra C14, rendendolo invisibile;
- Controllare meglio i componenti scelti. Abbiamo infatti scelto di utilizzare un led rgb di un formato che in realtà non si trova sul mercato. Anche questo è stato scoperto solo in fase di assemblaggio

9

Bibliografia

- [Repository Pico SDK](#)
- [Repository Pico Examples](#)
- [Repository Picotool](#)
- [Documentazione Pico SDK](#)
- [Raspberry Pi Pico C/C++ SDK](#)
- [Hardware design with RP2040](#)
- [The astronomical almanac's algorithm for approximate solar position](#)
- [Solar Position Algorithm for Solar Radiation Applications](#)
- [Repository driver HMC5883L STM32](#)
- [Repository driver MPU6050 STM32](#)
- [Repository NMEA parser minmea](#)
- [Repository pico stepper driver](#)
- [Tool online per analisi stringhe NMEA](#)
- [Thingiverse dual axis solar tracker](#)

10

Allegati

10.1. Gerber