MARCO BETTI

1057017

PROGETTAZIONE E SVILUPPO DI UN ILLUMINATORE LED PER LA DEGRADAZIONE LID DI CELLE SOLARI

TESI DI LAUREA MAGISTRALE IN INGEGNERIA ELETTRONICA

Relatore:
Ch.mo Prof. Gaudenzio Meneghesso

Correlatore:
Dott. Ing. Matteo Meneghini

Università degli Studi di Padova
Dipartimento di Ingegneria dell'Informazione
Anno Accademico 2013/2014

Questa tesi è stata realizzata con LaTeX 2_ε.

INTRODUZIONE

La **degradazione indotta dalla luce, o Light Induced Degradation (LID)** è un fenomeno che si osserva nei primi mesi di vita di un pannello fotovoltaico, che comporta un continuo calo di efficienza fino al raggiungimento di un livello stabile.

L'industria fotovoltaica mondiale sta spingendo da tempo la ricerca su questa problematica che ad oggi non sembra essere del tutto compresa, a causa dei numerosi fattori legati alla chimica e alla fisica dello stato solido che regolano i meccanismi di degradazione.

Poiché la LID è strettamente correlata all'intensità luminosa incidente sulle celle solari, per condurre degli esperimenti di degradazione è necessario disporre di un **sistema di illuminazione** in grado di simulare la luce solare con una potenza pari o superiore.

Scopo di questo lavoro di tesi è la progettazione e lo sviluppo di un **illuminatore a stato solido (LED)** ad elevata potenza (fino a 3 volte l'intensità della radiazione solare), scalabile e integrabile in un sistema di misura e caratterizzazione automatizzato.

Desidero ringraziare il Professor Gaudenzio Meneghesso
per essere stato un costante punto di riferimento
e per avermi seguito con pazienza
durante tutto il mio percorso di studi.

Un sentito ringraziamento a Matteo Meneghini,
Marco Barbato e Fabiana Rampazzo,
con i quali ho condiviso questi mesi di lavoro,
dalle prime riunioni di progetto
fino al completamento del lavoro di tesi.

Un abbraccio alla mia famiglia,
a chi c'è e ci sarà sempre
e gli amici più cari
per essermi vicino.

INDICE

ELENCO DELLE FIGURE

ELENCO DELLE TABELLE

ELENCO DEI CODICI

1 LIGHT INDUCED DEGRADATION

INDICE

1.1 CELLE IN SILICIO MONOCRISTALLINO

La **degradazione indotta dalla luce, o Light Induced Degradation (LID)** è un fenomeno che si osserva nei primi mesi di vita di un pannello fotovoltaico, che comporta un continuo calo di efficienza fino al raggiungimento di un livello stabile.

1.1.1 Descrizione e prime osservazioni

I primi ad osservare il fenomeno della LID furono Fischer e Pschunder nel 1973 [5] su delle celle solari fabbricate su un substrato di **Silicio Czochralski (Cz-Si) drogato con Boro** [23]. La figura 1 mostra l'andamento della potenza erogata P_m, della corrente di cortocircuito I_{sc} e della tensione a circuito aperto V_{oc} della cella in esame, come trattato nella pubblicazione originale [5]. Dopo averne misurato questi parametri in seguito alla fabbricazione, la cella è stata esposta alla luce per un periodo di qualche ora: ciò che si è osservato è stata una **degradazione di tutti i parametri** fino al raggiungimento di un livello stabile, individuato da B nella figura 1. Inoltre si è visto che in seguito ad un **annealing** a bassa temperatura (200°) le caratteristiche della cella tornavano ai valori originali (punto A fig. 1).

Grazie a delle misure a livello microscopico è stata individuata la causa del fenomeno LID nella **variazione del tempo di vita dei portatori** nel bulk tra due valori, corrispondenti ai punti A, con un tempo di vita più lungo, e B, con un tempo di vita inferiore a causa della radiazione luminosa.

Negli anni a seguire furono proposti diversi modelli per spiegare quanto osservato. Le variazioni del tempo di vita non erano dovute alla creazione diretta di difetti da parte della luce, ma dalla **dissociazione di coppie di difetti donore-accettore** a causa di un eccesso di portatori e ad un meccanismo che favoriva la ricombinazione (Recombination-Enhanced Defect Reaction, REDR). Questa tesi fu con-

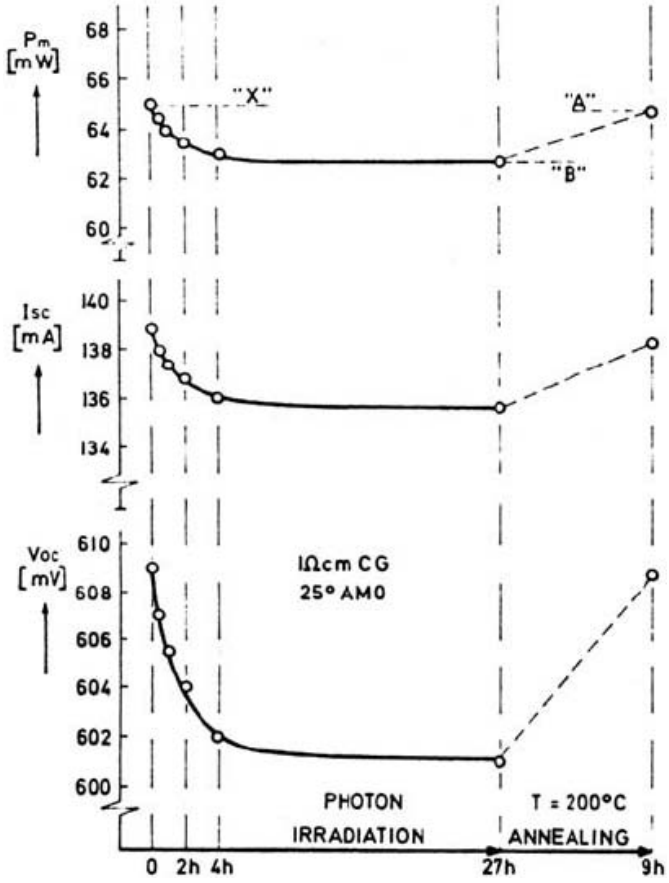


Figura 1: Prima osservazione della light-induced degradation in celle solari Cz-Si drogate con Boro (1973) [5].

fermata dal fatto che la degradazione del tempo di vita si osservava anche con la cella al buio applicando una tensione diretta.

Uno dei complessi donore-accettore più studiati nel Silicio cristallino è la **coppia Ferro-Boro** la quale si dissocia sotto l'effetto della luce formando un difetto interstiziale del Ferro che, in condizioni di bassa iniezione, è un centro di ricombinazione più efficace rispetto alla coppia Ferro-Boro, e quindi comporta una degradazione significativa del tempo di vita dei portatori [18].

1.1.2 Struttura e trasformazione dei difetti metastabili

Nel 1997 venne proposto per la prima volta un modello [20] in cui i principali difetti responsabili della degradazione non erano le impurità metalliche, ma la formazione di una **coppia di difetti composta da un Boro interstiziale e un Ossigeno interstiziale** B_iO_i durante l'esposizione alla luce del Silicio Cz. Studi successivi hanno confermato la forte correlazione tra la LID in Cz-Si e le concentrazioni di Boro e Ossigeno [6]. In particolare si è osservato che la degradazione del tempo

di vita aveva una dipendenza lineare dalla concentrazione di Boro e una dipendenza alla quinta potenza dalla concentrazione di Ossigeno interstiziale. Questo risultato ha evidenziato la presenza di un difetto più complesso rispetto alla coppia B_iO_i.

Misure spettroscopiche più accurate [19, 17], grazie a metodi come l'Injection-Dependent Lifetime Spectroscopy (IDLS), hanno individuato il livello energetico del centro di ricombinazione responsabile della LID **vicino al centro del bandgap**, molto diverso dal livello energetico della coppia B_iO_i (fig.2).

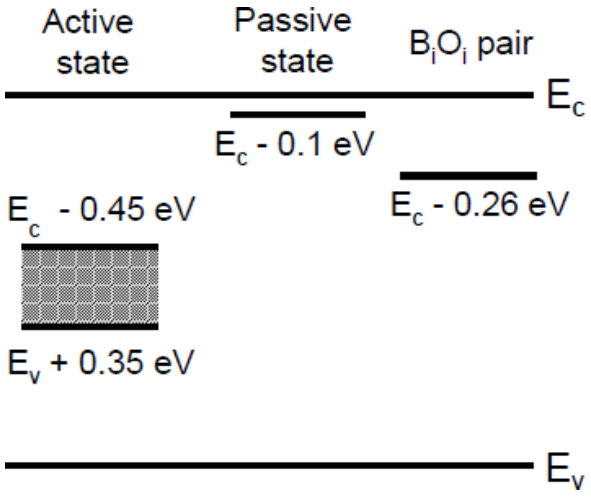
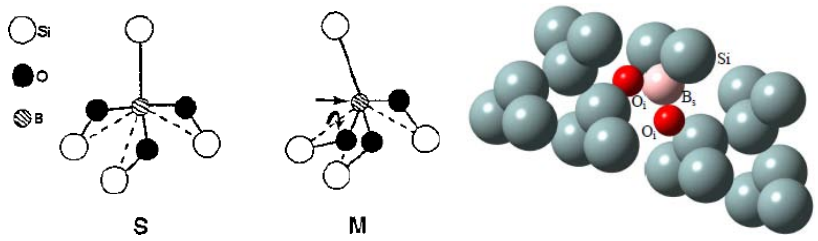


Figura 2: Livelli di energia del difetto metastabile del Cz-Si nel suo stato attivo e passivo determinato con spettroscopia del tempo di vita [23].

Sulla base di queste misure fu proposta una nuova struttura composta da un atomo di **Boro sostituzionale** B_s **circondato da tre atomi di Ossigeno interstiziali (fig.3)**. La generazione di tali difetti è un processo ad *attivazione termica* con una barriera di energia relativamente bassa [21] pari a $E_{gen} = 0.4\,eV$, che segue una relazione del tipo:

$$R_{gen}(T) = R_{gen}(T \mapsto \infty) \exp\left(-\frac{E_{gen}}{kT}\right) \tag{1}$$

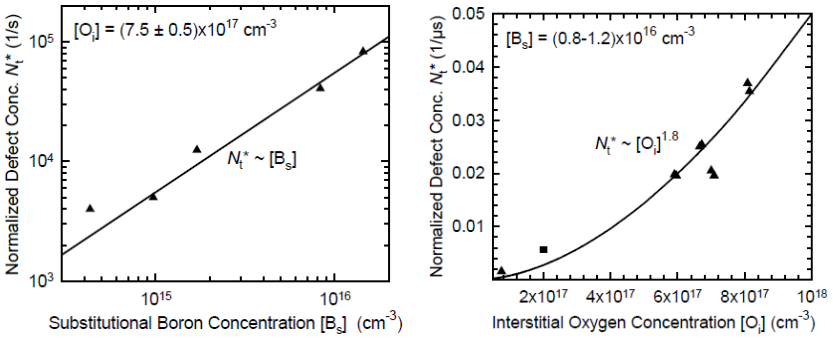
Allo stesso modo anche l'annichilazione di tali difetti dipende dalla temperatura: esperimenti differenti hanno rilavato delle barriere di energia comprese tra $E_{ann} = 1.3\,eV$ [17] ed $E_{ann} = 1.8\,eV$ [21]. La figura 4 mostra la concentrazione normalizzata di difetti N_t^* in funzione delle concentrazioni di Boro sostituzionale e Ossigeno interstiziale [21]: si noti la dipendenza *lineare* di N_t^* dalla concentrazione di Boro e quella *quadratica* dalla concentrazione di Ossigeno, *in contrasto* con quanto trovato in [6]. Questa discrepanza, dovuta probabilmente a due tecniche di **passivazione superficiale** differenti adottate nei due esperimenti, è indice del fatto che il difetto metastabile è composto da **un atomo di Boro sostituzionale e due atomi di Ossigeno interstiziale** B_sO_{2i} (fig.3).



(a) *Configurazione stabile (S) e metastabile (M) del complesso Boro-Ossigeno [23].*

(b) *Possibile configurazione del difetto metastabile* B_sO_{2i} *all'interno del reticolo di Silicio [22].*

Figura 3: Struttura microscopica dei composti metastabili B-O.



(a) *Concentrazione normalizzata di difetti* N_t^* *in funzione della concentrazione di Ossigeno interstiziale* O_i.

(b) *Concentrazione normalizzata di difetti* N_t^* *in funzione della concentrazione di Boro sostituzionale* B_s.

Figura 4: Concentrazione normalizzata di difetti N_t^* in funzione delle concentrazioni di Boro e Ossigeno [21].

1.1.3 Soluzioni alternative

Misure effettuate su substrati Cz-Si di tipo p drogati con Gallio, oppure di tipo n drogati con Fosforo hanno dimostrato che tali materiali non mostrano alcuna degradazione del tempo di vita dei portatori [20]. A partire da questi risultati sono stati proposti diversi metodi per ridurre la degradazione indotta dalla luce in celle Cz-si, tra cui:

- **Sostituzione del Boro** con un altro drogante, come il Gallio (Ga) o il Fosforo (P);

- **Riduzione della concentrazione di Ossigeno nel materiale Cz**. L'elevata quantità di Ossigeno nei lingotti di Silicio cresciuti con il metodo Czochralski tradizionale è dovuta alla parziale evaporazione del crogiolo di silice durante il processo di crescita. Il contenuto di Ossigeno può essere ridotto confinando i flussi di materiale con dei campi magnetici. Il cosiddetto **Silicio Cz magnetico (MCz-Si)** con una concentrazione di Ossigeno di ordini di grandezza inferiore presenta una degradazione LID quasi tra-

scurabile e ha dei tempi di vita confrontabili con quello del Cz-Si con drogaggio Ga.

La figura 5 mostra il tempo di vita dei portatori misurato per differenti tipi di substrato di tipo Cz, magnetico e non, drogati con Boro, Fosforo e Gallio. Si può notare che il substrato di tipo p drogato con Boro subisce la degradazione maggiore in seguito all'esposizione alla luce, mentre il substrato Cz magnetico, pur essendo drogato con Boro, è perfettamente allineato ai substrati che non risentono del fenomeno LID, come il Ga-dop Cz-Si e il P-dop Cz-Si.

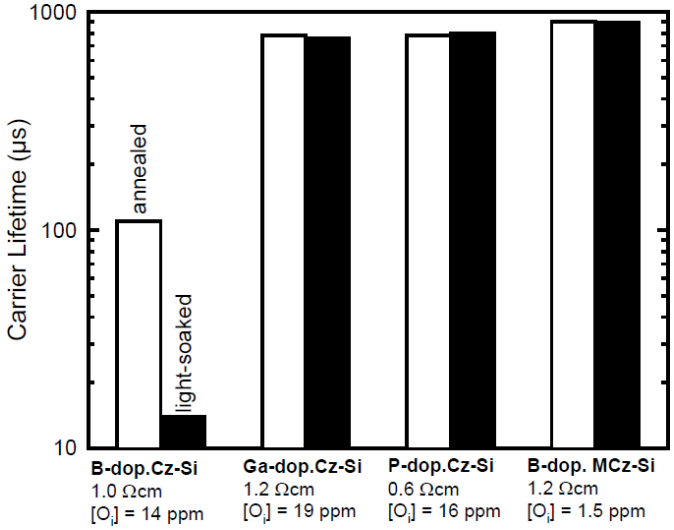


Figura 5: Tempi di vita dei portatori misurati per diversi tipi di Cz-Si (con drogaggio B, P e Ga) e di MCz-Si (con drogaggio B) dopo esposizione alla luce [23].

2 PROGETTO HARDWARE

INDICE

2.1 SPECIFICHE DI PROGETTO

Per poter condurre gli esperimenti necessari ad investigare il fenomeno della Light-Induced Degradation (LID) sulle celle solari è necessario disporre di un'adeguata **sorgente luminosa**. Scopo del presente lavoro è la *progettazione e lo sviluppo di un illuminatore a stato solido per esperimenti di degradazione LID*.

Gli obiettivi del progetto e le caratteristiche dell'illuminatore sono:

- Sorgente luminosa allo **stato solido** con **LED** bianchi ad elevata potenza;
- Flusso luminoso massimo tra **2.5 e 3 Sun**;
- Elevata **concentrazione del fascio luminoso** per mezzo di ottiche su un'area sufficiente a coprire una cella standard 15 x 15 cm;
- **Luminosità variabile**;

- **Modularità e scalabilità**;
- Interfaccia **USB** per il controllo remoto tramite **LabView**.

2.2 LED

2.2.1 Caratteristiche

I LED di potenza adottati per lo sviluppo dell'illuminatore sono degli **XLamp XB-D** ® di colore bianco, prodotti dalla **Cree** ®. Secondo il produttore si tratta della famiglia di LED con il footprint più piccolo della loro categoria: solo 2.45 X 2.45 mm, con una **corrente massima di** 1 A [3].

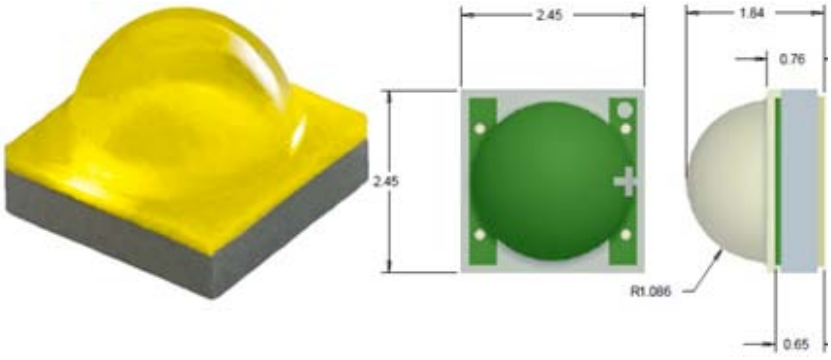


Figura 6: LED XLamp XB-D e relative dimensioni [3].

Color	CCT Range		Base Order Codes Min. Luminous Flux @ 350 mA			Calculated Minimum Luminous Flux (lm)**		Order Code
	Min.	Max.	Group	Flux (lm) @ 85 °C	Flux (lm) @ 25 °C*	700 mA	1000 mA	
Cool White	5000 K	8300 K	R3	122	139	210	271	XBDAWT-00-0000-000000F51
			R2	114	130	196	253	XBDAWT-00-0000-000000E51

Figura 7: Flusso nominale in funzione della corrente per la famiglia di LED bianchi XB-D [3].

In particolare è stato scelto il modello **XBDAWT-00-0000-000000E51** cool white, con temperatura di colore compresa tra 5000K e 8300K, e con un **flusso luminoso** calcolato pari a 196 lm con una corrente di 700 mA, e 253 lm con la corrente massima di 1 A (fig. 7).

2.2.2 Ottiche

I LED XB-D sono caratterizzati da un **angolo di emissione** di 115°, quindi, per poter concentrare la potenza luminosa su un'area ristretta come quella di una cella solare, è necessario accoppiare delle **ottiche** ai LED. La scelta è ricaduta sulle lenti plastiche in PMMA **C12607_VIRPI_S** ® prodotte dalla **Ledil** ®, organizzate per ospitare una matrice di 5

x 5 LED XB-D. Si caratterizzano per un'**efficienza ottica del 92%** e un angolo di apertura a mezzo massimo pari a **13°** (fig. 8) [12].

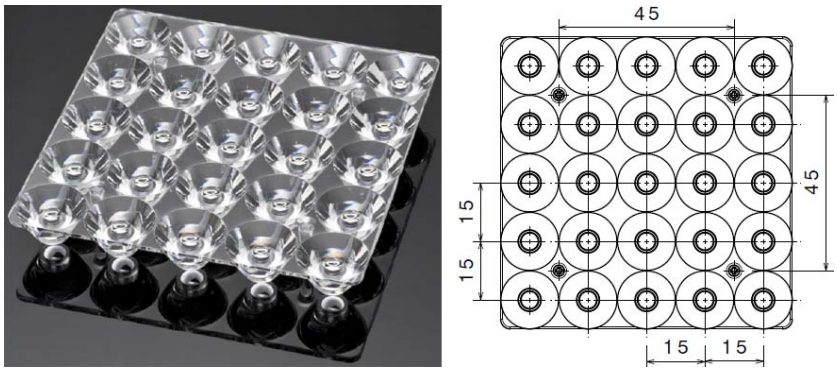


Figura 8: Ottiche Ledil VIRPI S per LED XB-D [12].

2.2.3 Dimensionamento

Secondo le specifiche la massima **tensione diretta** ai capi del LED è di 3,5 V [3]. Più LED connessi in serie formano una **stringa**, la cui tensione operativa è pari alla tensione diretta di ciascun LED per il numero di LED che la compongono. Poiché uno degli obiettivi di questo progetto è la *standardizzazione* si è optato per organizzare i LED in **stringhe da 13**, che possano essere alimentate con un alimentatore commerciale da 48 V nominali anche nel caso peggiore:

$$V_{STR} = V_{Fmax} \cdot N_{LED} = 3{,}5\,V \cdot 13 = 45{,}5\,V \tag{2}$$

Valori tipici della tensione diretta sono comunque attorno a 3,2 V per una corrente di 1 A, come si vede dalla caratteristica I-V (fig. 9).

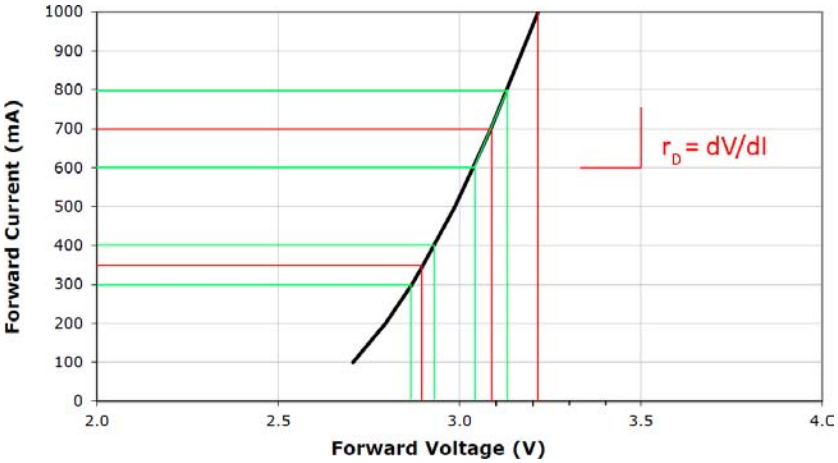


Figura 9: Caratteristica I-V di un LED XB-D bianco [3].

La **potenza dissipata** dal singolo LED è quindi pari a:

$$P_{LED} = V_F \cdot I_F = 3{,}2\,V \cdot 1\,A = 3{,}2\,W \tag{3}$$

Il LED tuttavia non è efficiente al 100% nella conversione della potenza elettrica in potenza ottica. Una stima conservativa afferma che **solo il 25% della potenza elettrica venga convertito in potenza ottica, mentre il restante 75% va dissipato in calore** [2]. Quindi la **potenza ottica** emessa da ciascun LED è:

$$P_{OPT} = V_F \cdot I_F \cdot 0.25 = 3{,}2\,V \cdot 1\,A \cdot 0{,}25 = 0{,}8\,W \tag{4}$$

Se consideriamo un'ipotetica matrice di 9 lenti, che corrisponde ad una matrice di 15 x 15 LED, ricordando che ogni stringa viene limitata a 13 LED otteniamo una matrice di 15 x 13 = 195 LED, a cui corrisponde una potenza elettrica totale pari a:

$$P_{LEDtot} = P_{LED} \cdot 195 = 3{,}2\,W \cdot 195 = 624\,W \tag{5}$$

considerando l'efficienza ottica delle lenti $\eta_L = 0.92$ si ottiene una potenza ottica totale pari a:

$$P_{OPTtot} = P_{LEDtot} \cdot \eta_L \cdot 0{,}25 = 624\,W \cdot 0{,}92 \cdot 0{,}25 \simeq 144\,W \tag{6}$$

Poiché il dato di interesse per gli esperimenti LID è la potenza ottica per unità di area, dobbiamo considerare l'area della proiezione del fascio luminoso data dall'apertura delle ottiche. È chiaro che più allontaniamo le ottiche dal punto in cui andrà collocata la cella solare e più il fascio si allargherà su un'area maggiore.

Supponendo di proiettare il fascio su un'area A = 20 x 20 cm = $400\,cm^2$ si ottiene una **irradianza** pari a:

$$I = \frac{P_{OPTtot}}{A} = \frac{144\,W}{400\,cm^2} = 0{,}36\,W/cm^2 = 3{,}6\,Sun \tag{7}$$

Se invece allontaniamo la sorgente luminosa, ad una distanza di circa 30 cm (fig. 11) e l'area diventa A = 25 x 25 cm = $625\,cm^2$ l'irradianza diventa:

$$I = \frac{P_{OPTtot}}{A} = \frac{144\,W}{625\,cm^2} = 0{,}23\,W/cm^2 = 2{,}3\,Sun \tag{8}$$

La disposizione finale della matrice di LED con le rispettive lenti è mostrata in figura 10. Le frecce rosse stanno ad indicare la *scalabilità* di questo illuminatore. Volendo è possibile affiancare su ciascun lato indicato dalle frecce delle analoghe matrici di LED aumentando l'area illuminata e la potenza ottica a piacere.

2.3 SCHEMA A BLOCCHI

Dopo aver dimensionato il sistema ottico è stato il momento di progettare il sistema elettronico di controllo e alimentazione. La figura 12 mostra lo schema a blocchi del sistema, che è composto in quattro blocchi funzionali:

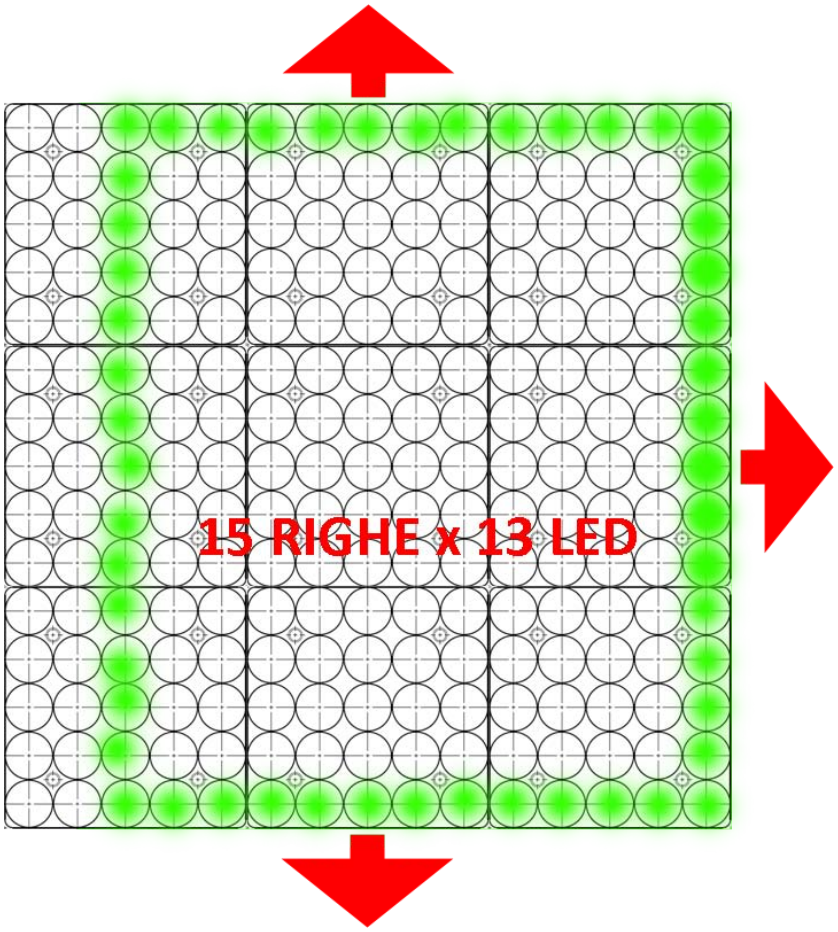


Figura 10: Disposizione delle lenti e matrice di LED.

SCHEDA DI CONTROLLO: è il cuore del sistema, basata su un potente microcontrollore. Gestisce il collegamento con il PC, è dotata di un'interfaccia utente per un eventuale funzionamento standalone e controlla la luminosità dei LED;

SCHEDA DI POTENZA: contiene i circuiti di pilotaggio dei LED di potenza, i DAC per il controllo della luminosità, i convertitori DC-DC per alimentare il circuito;

SCHEDA LED: alloggia la matrice di LED con le relative ottiche;

SENSORI DI LUCE: 4 sensori di luce per predisporre il sistema al controllo della luminosità in feedback. Vengono collegati allo stesso bus tramite un *multiplexer*.

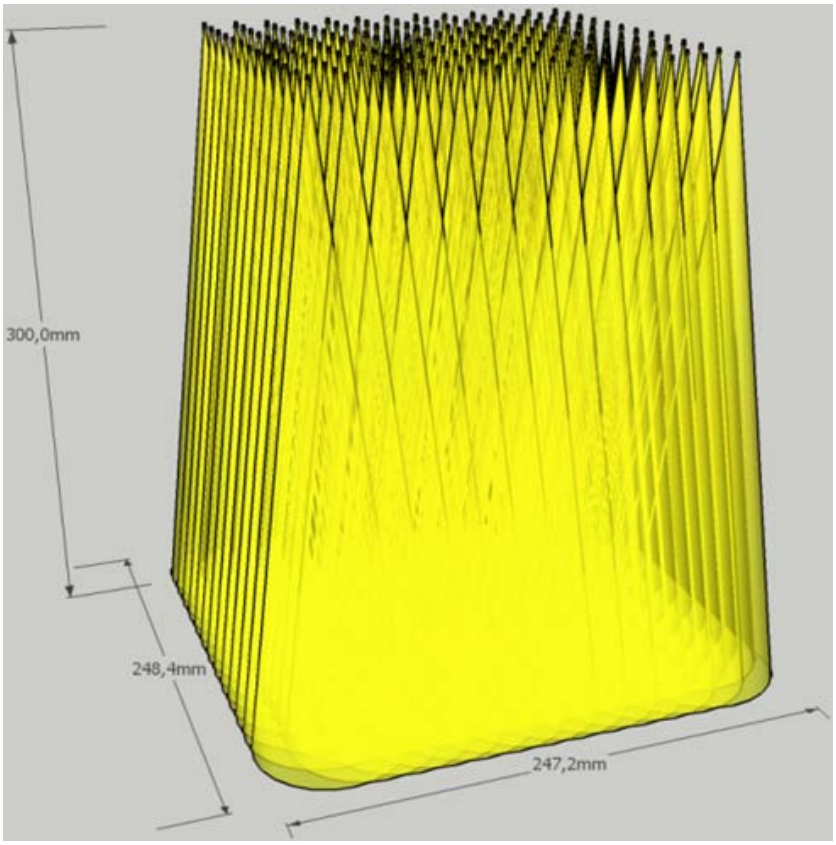


Figura 11: Rendering preliminare del fascio luminoso con ottiche da 13°poste alla distanza di 30 cm dal punto di misura.

2.4 SCHEDA DI CONTROLLO

2.4.1 Microcontrollore

Il controllo e la gestione dell'intero sistema sono affidate ad un potente microcontrollore **PIC24** ® con architettura a 16 bit della **Microchip Technology** ®. In particolare è stato scelto il modello **PIC24FJ256GB110**, le cui caratteristiche principali sono [34]:

- Architettura a 16 bit con performance fino a **16 MIPS**;
- 256 KB di memoria Flash;
- 16384 Bytes di memoria RAM;
- Tensione operativa tra 2 e 3.6 V;
- Periferiche per comunicazioni digitali: 4 UART, 3 SPI, 3 I2C, PMP;
- Periferiche analogiche: 1 ADC 16 canali, 10 bit 500 ksps, capacitive touch sensing 16 canali;

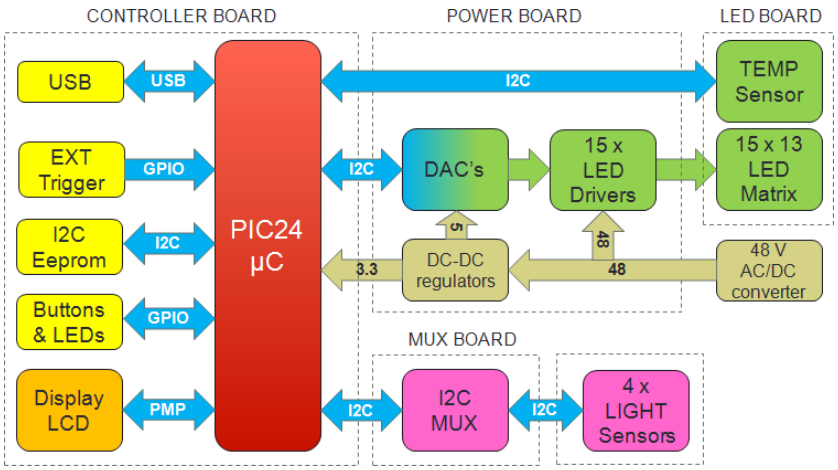


Figura 12: Schema a blocchi del sistema di illuminazione per misure LID.

- Modulo **USB** Full Speed (Device, Host, OTG);
- 9 moduli CCP/PWM da 16 bit;
- 5 Timer da 16 bit, Real Time Clock Calendar (RTCC).

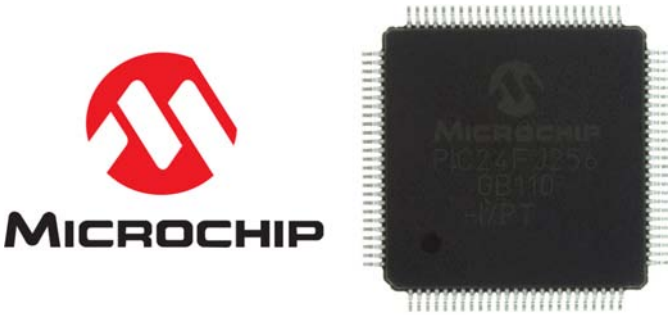


Figura 13: PIC24FJ256GB110 in package TQFP-100.

La figura 14 mostra il *pinout* del microcontrollore. Dei 100 pin di cui dispone alcuni sono dedicati all'alimentazione, altri alla sezione dell'oscillatore, altre sono ingressi/uscite di tipo general purpose ed altri ancora vengono condivisi dalle varie periferiche interne al µC.

2.4.2 Display - Interfaccia HMI

La scheda di controllo è dotata di un display LCD grafico monocromatico con risoluzione di 128 x 64 pixel che, insieme a 4 pulsanti, un encoder rotativo e 4 LED, predispone il sistema all'interazione diretta con l'utente.

Il display adottato per questo progetto è il modello **64128M-FC-BW-3** prodotto da **Displaytech** ® (fig. 15). La sua interfaccia digitale è basata sull'integrato ST7565R presente a bordo del display e può essere di tipo seriale o parallela. In questo progetto è stata usata l'interfaccia

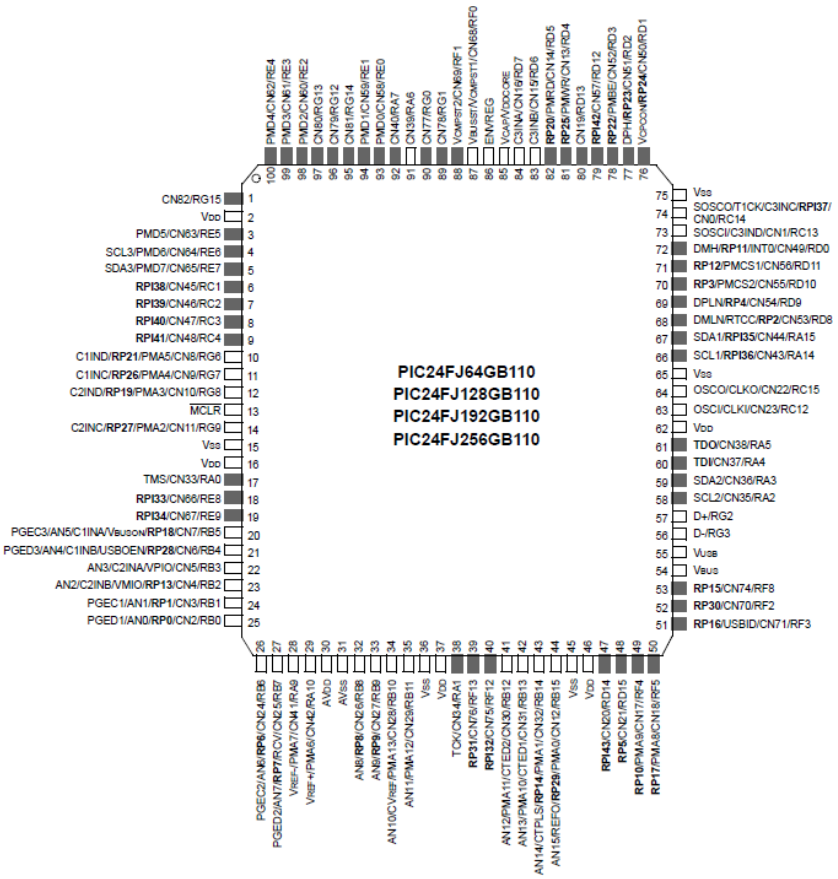


Figura 14: Pinout del PIC24FJ256GB110 in package TQFP-100 [34].

parallela, che viene collegata alla **Parallel Master Port** del microcontrollore secondo lo schema di fig. 16. Come si vede nella figura 17 l'interfaccia del display è composta da diverse linee dati e di controllo:

DB<7:0>: sono le linee del bus a 8 bit (pin da 6 a 13);

RS/A0: questa linea stabilisce se il byte del bus DB va interpretato come un comando (RS = 0) o un dato da visualizzare (RS = 1) (pin 3);

R/W: stabilisce se il display è in modalità scrittura (R/W = 0) o lettura (R/W = 1) (pin 4);

E/RD: si pone questa linea a 1 per eseguire il latch del byte presente sul bus DB e dello stato delle linee RS e R/W (pin 5).

C86 E P/S: stabiliscono la modalità di interfacciamento del display (seriale o parallela) (pin 27 e 28).

CS1: chip select in caso di presenza contemporanea di più controller a bordo del display (pin 1).

La figura 18 mostra l'andamento e la temporizzazione delle linee appena descritte in un ciclo di lettura o scrittura del display LCD. La periferica PMP del PIC24 si occuperà della corretta gestione e temporizzazione di questi segnali in maniera trasparente al codice utente, che dovrà solo configurare la periferica per mezzo di appositi registri. Gli altri pin sono dedicati all'alimentazione e al regolatore di tensione interno che, grazie a qualche condensatore esterno, genera tutte le tensioni necessarie per il funzionamento dell'LCD a partire da un'unica tensione di alimentazione di 3,3 V. Due pin sono riservati alla retroilluminazione. Secondo le specifiche [4] il display contiene un gruppo di LED da alimentare con una corrente di 80 mA.

Figura 15: Display LCD grafico 128 x 64.

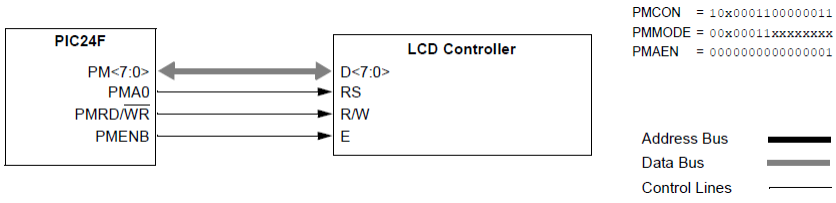


Figura 16: Connessione di un display LCD alla Parallel Master Port del PIC24 [31].

2.4.3 Schema elettrico

Le figure 20, 21, 22 mostrano lo schema elettrico della scheda di controllo, che è stato sviluppato a partire dai collegamenti minimi e dai requisiti richiesti per il corretto funzionamento del microcontrollore descritti nel relativo datasheet [34]. Lo schema può essere suddiviso in diverse aree funzionali:

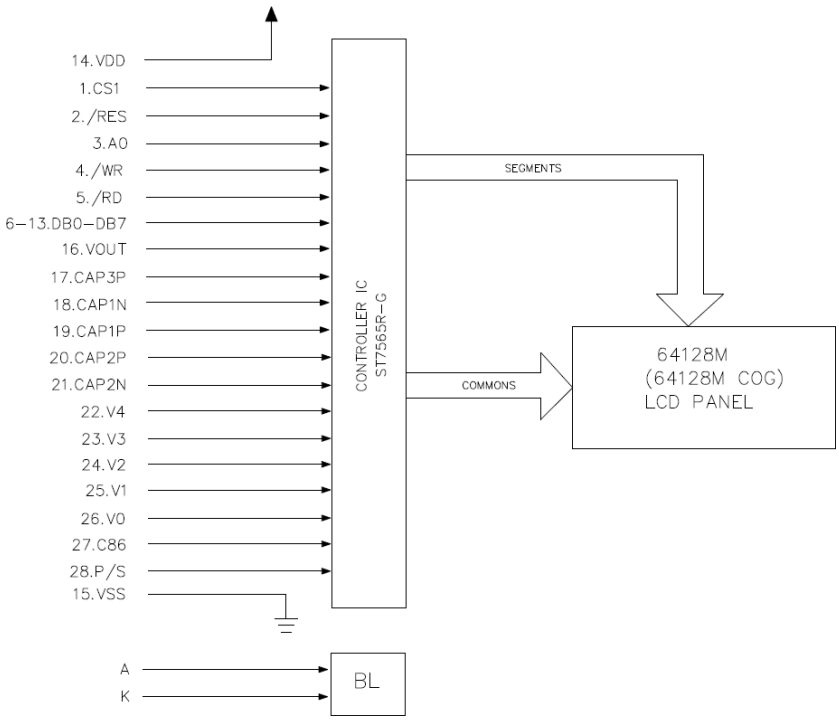


Figura 17: Pinout del display LCD grafico 64128M [4, 25].

SEZIONE OSCILLATORE: il microcontrollore necessita di un oscillatore che generi un segnale di clock per poter funzionare ed eseguire le istruzioni del programma. La sorgente di clock può essere sia un oscillatore RC interno al µC oppure un oscillatore con cristallo di quarzo, che va collegato all'esterno ma è più stabile in frequenza. Nello schema sono presenti due quarzi: Y1, con frequenza di 16 MHz, per l'oscillatore principale, e Y2, con frequenza di 32.768 kHz, per l'oscillatore secondario e l'orologio interno. I condensatori C30, C31, C32 e C33 sono delle capacità necessarie al corretto funzionamento dell'oscillatore i cui valori vengono imposti dal campo di frequenza del quarzo [34].

SEZIONE PROGRAMMAZIONE: i componenti J5, C20, C21, R37 permettono il collegamento del microcontrollore con l'interfaccia di programmazione e **In-Circuit Debug ICD3** ® (fig.19) per mezzo dell'interfaccia proprietaria **In-Circuit Serial Programming, ICSP** ®.

SEZIONE DISPLAY: il display grafico è collegato alla periferica PMP del PIC24, che gestisce autonomamente il protocollo di comunicazione. I condensatori da C5 a C13 servono al regolatore di tensione interno per generare tutte le tensioni necessarie al pilotaggio del display LCD. L'integrato U2 è un regolatore a *pompa di carica* utilizzato per mantenere costante la corrente dei LED della retroilluminazione. Tali LED presentano una tensione diretta di circa 3,2 V con una corrente nominale di 80 mA. Tenendo conto

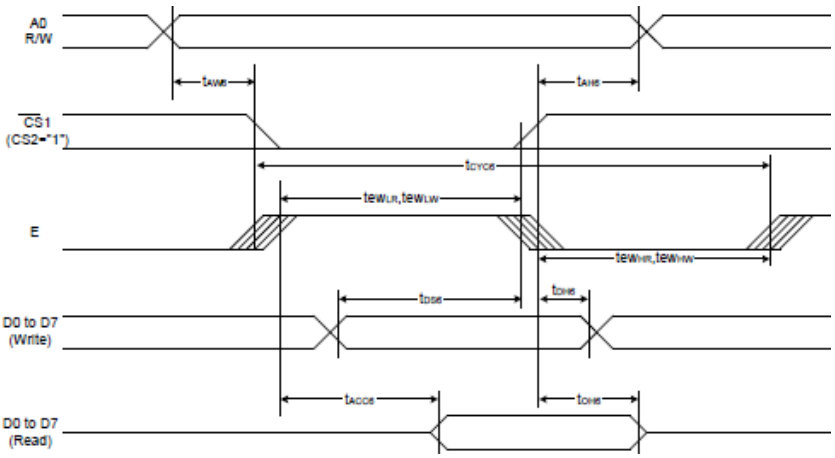


Figura 18: Timing dell'interfaccia parallela del controller del display LCD [25].

che l'integrato **MCP1252** ha un comparatore con riferimento interno a 1,21 V [28], per imporre una corrente costante ai LED è sufficiente inserire una resistenza serie (R17) il cui valore è:

$$R_{ext} = R_{17} = 1{,}21\,V/I_{LED} = 1{,}21\,V/80\,mA \simeq 15\,\Omega \qquad (9)$$

SEZIONE LED-PULSANTI: i pulsanti da SW1 a SW4, insieme all'encoder S1, e ai LED da D1 a D4 completano l'interfaccia con l'utente. Encoder e pulsanti sono collegati a ingressi general purpose del micro aventi la possibilità di generare un *interrupt* in seguito ad una variazione di livello logico (es. pressione del tasto). Sia nel caso dell'encoder che dei pulsanti sarà necessario gestire il fenomeno del *bouncing*, ovvero di fluttuazioni del livello logico dovute a rimbalzi meccanici dei contatti. I LED sono di tipo bicolore dimensionati per una corrente di 20 mA ciascuno.

SEZIONE I2C: l'intero sistema di controllo è stato pensato attorno al bus I2C per quanto riguarda la comunicazione tra i vari blocchi funzionali. Secondo le specifiche del bus I2C [24] sono necessari dei resistori di pull-up sulla linea dati SDA e sulla linea clock SCL, in quanto i dispositivi collegati sono di tipo *open drain*. I resistori di pull-up sono R38, R39, R40, R41. In questo progetto vengono sfruttate due delle tre periferiche I2C del microcontrollore: una per la comunicazione con i DAC della scheda di potenza, tramite il connettore J9, e una per la comunicazione con i sensori di luce, tramite J8. J8 e J9 vengono sfruttati anche per i collegamenti dell'alimentazione.

SEZIONE EEPROM: si tratta di una memoria EEPROM non volatile da 64K bits (8k Bytes) [27] con interfaccia **I2C**. Le resistenze R25, R26, R27, R30, R31, R32 permettono, se montate o meno, di configurare l'indirizzo I2C della memoria.

SEZIONE RS232: la scheda di controllo è predisposta anche per un'eventuale comunicazione seriale via RS232. Un traslatore di livello

MAX3222 (U3) [10] è collegato ad una delle periferiche **USART** del PIC24.

SEZIONE USB: le funzionalità USB sono affidate totalmente al modulo interno al microcontrollore. Sulla scheda di controllo sono presenti solo il connettore J2 e dei protettori da scariche ESD (D5, D6, D7).

Esiste anche la predisposizione per un eventuale **trigger esterno** via J10, collegato ad un pin general purpose con interrupt on-change, per comandare l'accensione dell'illuminatore.

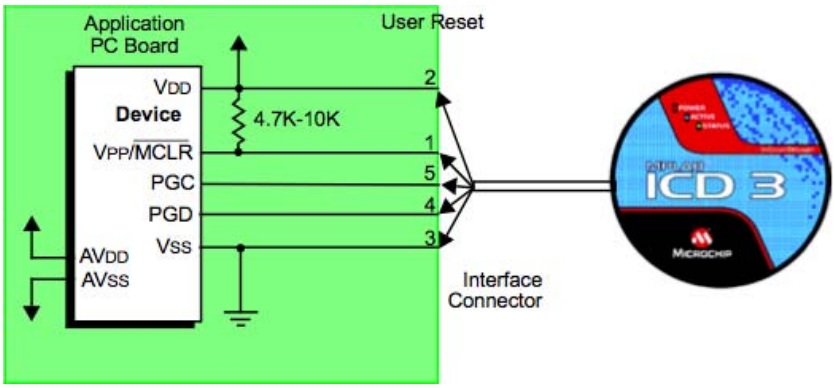


Figura 19: ICD3 debugger con relativo schema di collegamento.

2.4.4 PCB

Dopo aver sviluppato lo schema elettrico e aver associato a ciascun componente il relativo *footprint*, ovvero l'insieme dei pad fisici per la saldatura, si passa al CAD per disegnare il circuito stampato. Le figure 23 e 24 mostrano rispettivamente il *top layer* e il *bottom layer* del PCB della scheda di controllo, ciascuno con la relativa serigrafia dei componenti. Si tratta di un PCB *doppia faccia* in cui coesistono diversi piani di massa/alimentazione.

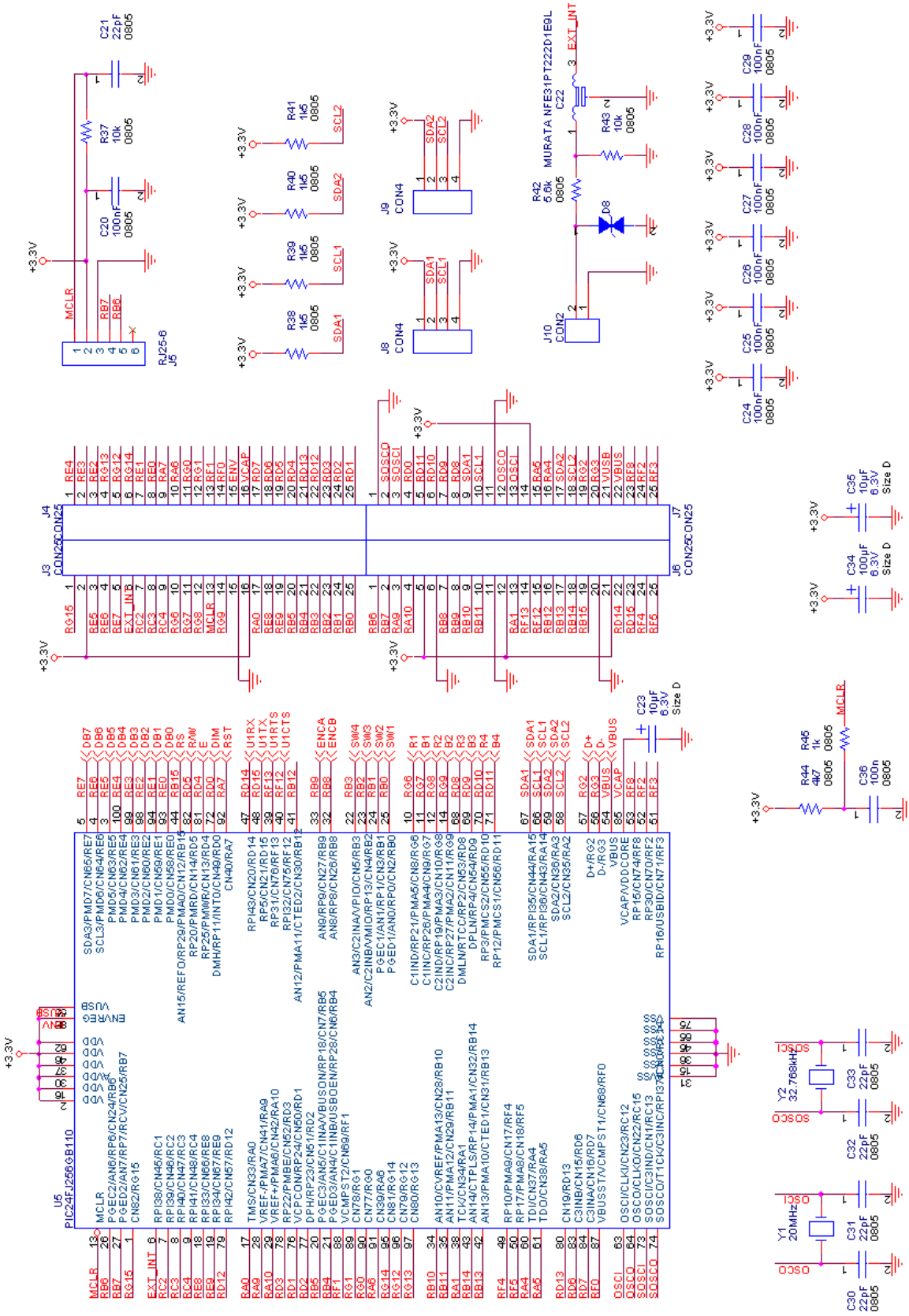


Figura 20: Schema elettrico della scheda di controllo - parte 1.

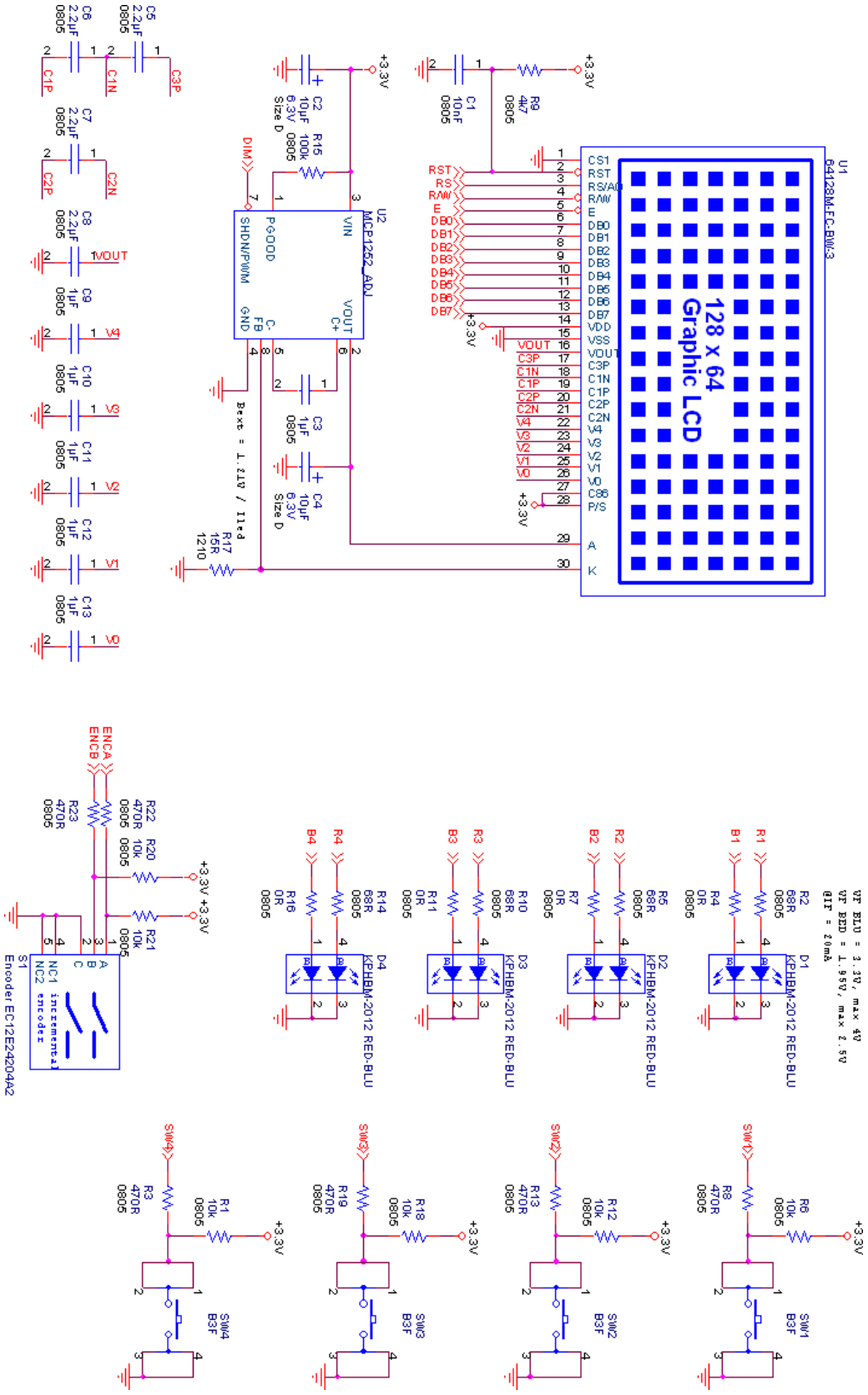


Figura 21: Schema elettrico della scheda di controllo - parte 2.

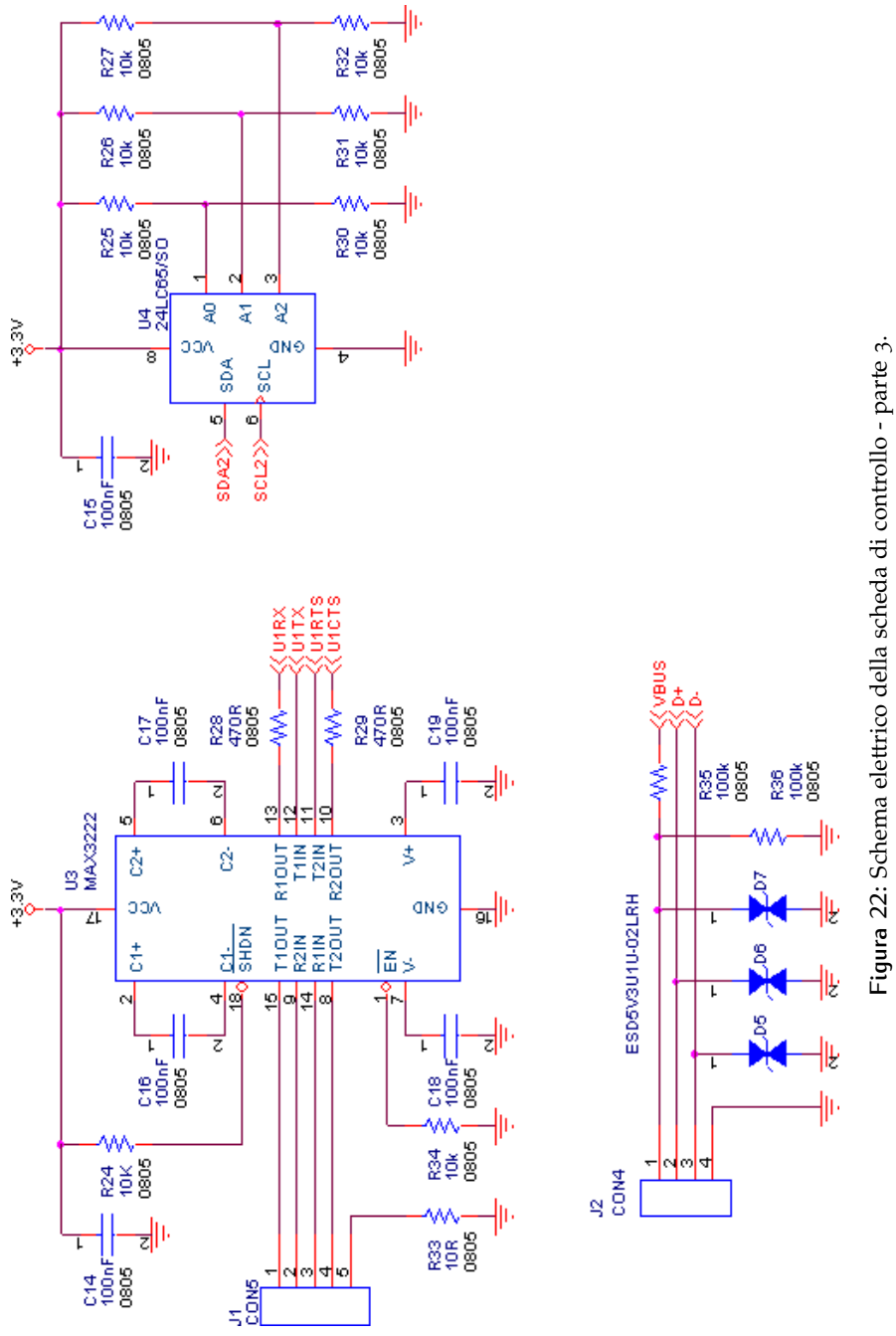


Figura 22: Schema elettrico della scheda di controllo - parte 3.

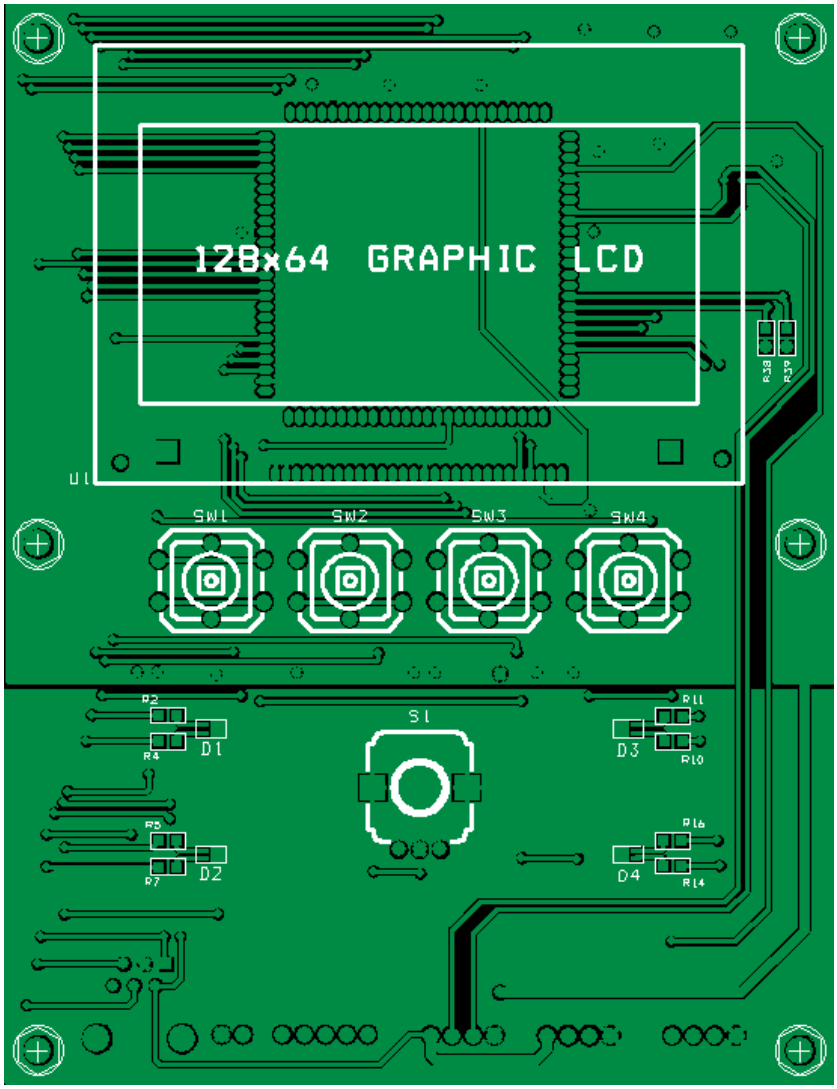


Figura 23: Visione della scheda di controllo al CAD con serigrafia - Top Layer.

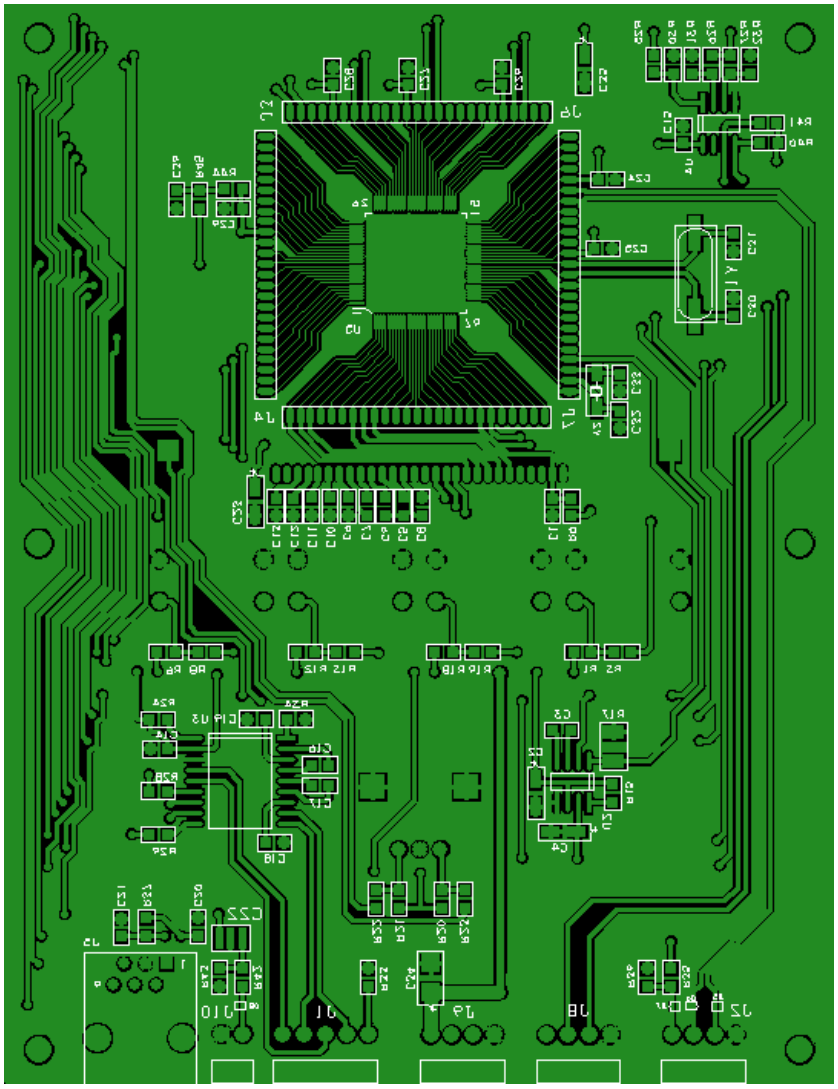


Figura 24: Visione della scheda di controllo al CAD con serigrafia - Bottom Layer.

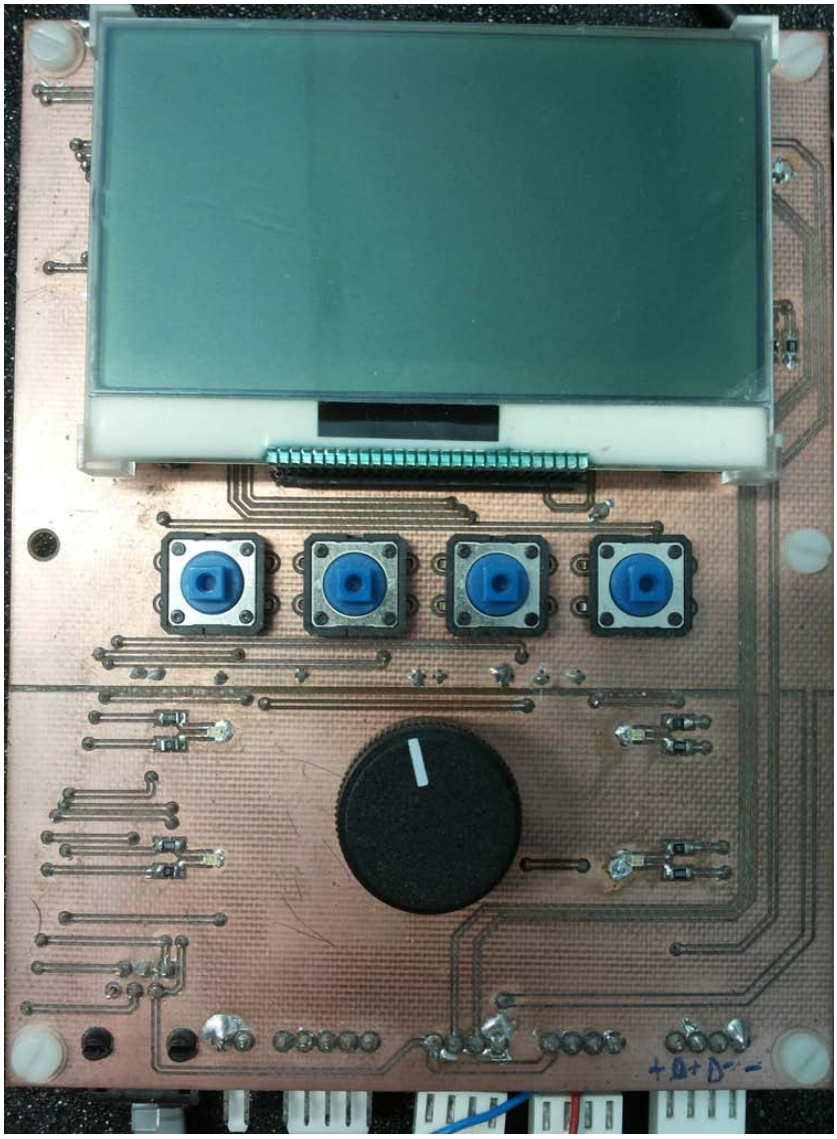


Figura 25: Scheda di controllo realizzata e con i componenti montati - Top Layer.

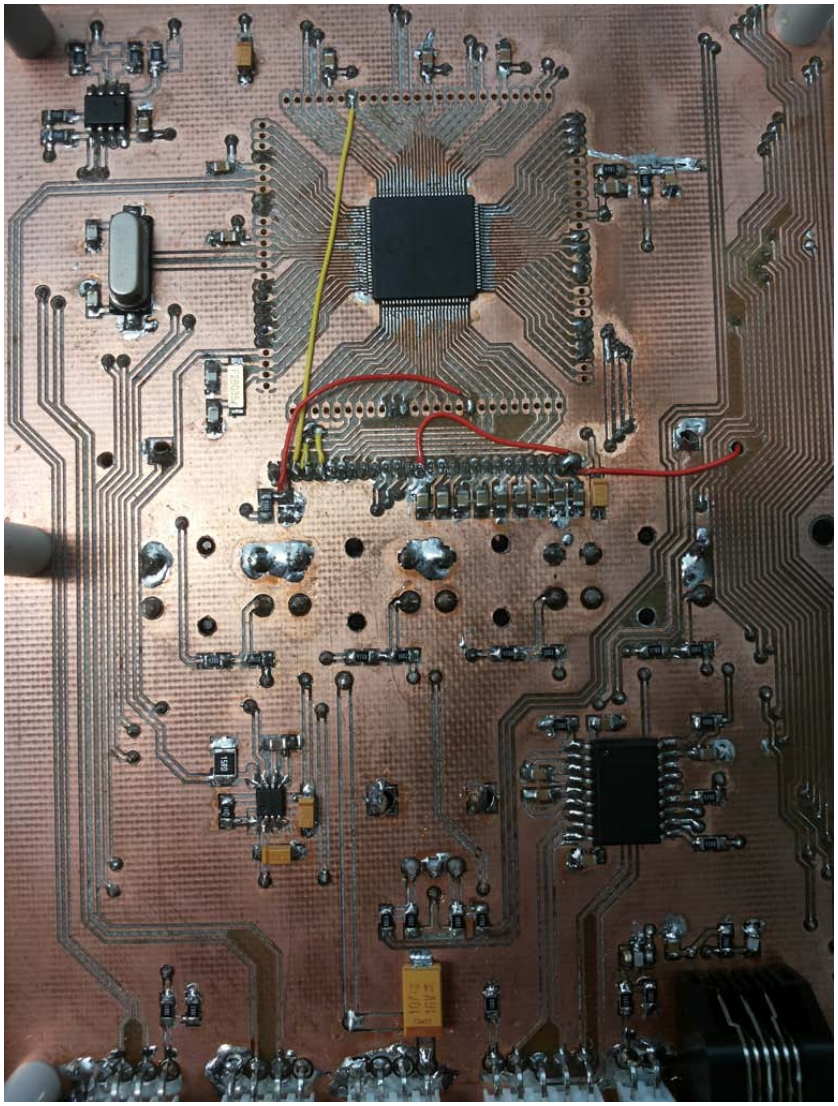


Figura 26: Scheda di controllo realizzata e con i componenti montati - Bottom Layer.

2.5 SCHEDA LED

La scheda LED è il vero e proprio corpo illuminante: svolge sia la funzione di supporto meccanico per LED e ottiche nonché da mezzo di dissipazione termica per il calore prodotto dai LED.

2.5.1 Progetto termico

L'**affidabilità** di un LED è una funzione diretta della **temperatura di giunzione**. Maggiore è la temperatura di giunzione, minore è il tempo di vita del LED. Una delle cause principali di rottura nei LED è infatti una cattiva gestione dell'aspetto termico.

Molte caratteristiche di un LED vengono influenzate notevolmente dalla sua temperatura operativa, alcune delle quali in maniera reversibile, come l'intensità del flusso luminoso, le coordinate cromatiche, la tensione operativa. Le elevate temperature invece degradano il tempo di vita in maniera irreversibile. Eccedere la massima temperatura di giunzione, che nel caso dei LED XB-D è di **150°C** può causare danni permanenti o addirittura distruttivi [2] [3].

INTENSITÀ LUMINOSA: ad elevate temperature si ha una riduzione dell'intensità luminosa emessa, che ritorna al valore originario quando il LED si raffredda. La dipendenza del flusso luminoso emesso dai LED XB-D in funzione della temperatura di giunzione è mostrata in fig.27. I dati relativi al flusso dei LED XB-D si riferiscono ad un flusso relativo unitario alla temperatura di giunzione di 85°C.

COORDINATE CROMATICHE: all'aumentare della temperatura di giunzione si ha un allargamento dello spettro di colore emesso dal LED, a causa dell'agitazione termica dei portatori;

TENSIONE OPERATIVA: la tensione diretta del LED diminuisce all'aumentare della temperatura, con un coefficiente di temperatura che varia tipicamente tra -1 e -4mV/C [2] [3]. È importante tenere conto di questa variazione nella progettazione o nella scelta del driver.

Come abbiamo visto nella descrizione dei LED, circa il 75% della potenza elettrica iniettata viene convertito in calore. Per non danneggiare i LED è necessario che l'illuminatore sia dotato di un adeguato sistema di dissipazione in grado di smaltirlo. Il calore viene generato all'interno della giunzione del LED e si propaga verso l'esterno attraversando numerose interfacce tra materiali diversi fino a raggiungere l'ambiente esterno. Ciascun materiale è caratterizzato da proprietà termiche differenti e il parametro di maggior interesse per il dimensionamento del sistema di dissipazione è la **resistenza termica** di ciascuna interfaccia.

Il percorso del calore in un sistema elettronico può essere modellizzato da una semplice *rete elettrica* composta da resistori. Ciascun resistore rappresenta una **resistenza termica**, il flusso di calore è rappresentato da una corrente elettrica e il salto di temperatura ai capi

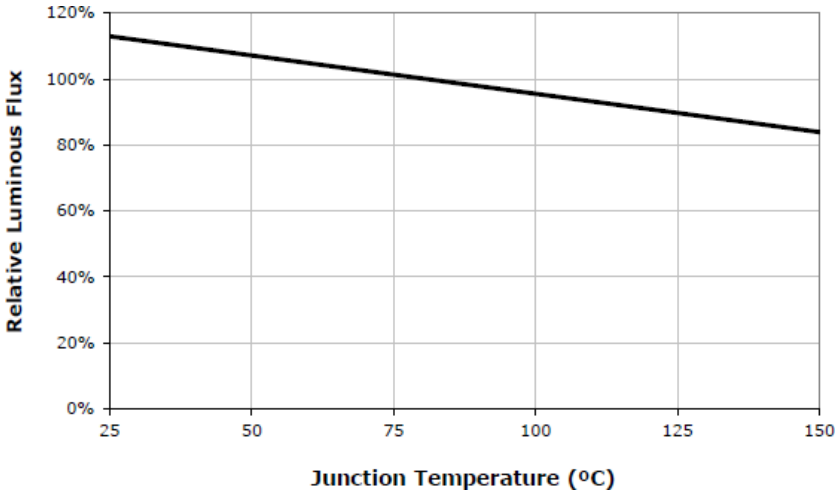


Figura 27: Flusso luminoso relativo in funzione della temperatura di giunzione per un LED Cree XB-D [3].

di una resistenza termica corrisponde ad una differenza di potenziale ai capi di un resistore. Tale modello nel caso di un sistema con più LED per un circuito stampato montato su un dissipatore metallico è rappresentato in figura 28, dove T è la temperatura (°C) nel punto corrispondente, Θ_{a-b} è la resistenza termica [°C/W] tra i punti a e b del materiale. In particolare:

Θ_{j-sp}: resistenza termica tra la giunzione del LED e il solder point (piazzola di saldatura). Secondo le specifiche del LED XB-D white tale valore è pari a 6,5 K/W [3].

Θ_{sp-pcb}: resistenza termica tra la piazzola di saldatura e il PCB. Dipende dalla **conducibilità termica** del PCB, espressa in [W/(m K)], e dall'area della piazzola di saldatura del singolo LED.

Per applicazioni con elevate densità di potenza in genera si realizza il PCB su un supporto metallico per migliorare la dissipazione termica. La figura 30 mostra la struttura di un **Metal Core PCB** [1]: uno strato metallico di base, in genere Alluminio, è ricoperto da un sottile strato di *dielettrico* ad elevata conducibilità termica (d = 50 µm) che isola lo strato di rame sovrastante. Assumiamo che la conducibilità termica del dielettrico sia pari a 2 W/(m K) [1].

Per ottenere la resistenza termica dobbiamo moltiplicare la conducibilità termica del dielettrico per l'area delle piazzole di saldatura di ciascun LED, calcolata in base alle dimensioni riportate in fig.29, ,dividere per lo spessore del dielettrico d, e farne l'inverso:

$$S_{sp} = 2 \cdot (0{,}35 \cdot 2{,}34) + (0{,}92 \cdot 2{,}34) = 3{,}8\,\text{mm}^2 = 3{,}8 \times 10^{-6}\,\text{m}^2 \tag{10}$$

$$\begin{aligned}\Theta_{sp-pcb} &= (K_{th} \cdot d \cdot S_{sp})^{-1} && (11)\\ &= (2\,W/(m\,K) \cdot 3{,}8 \times 10^{-6}\,m^2/5 \times 10^{-5}\,m)^{-1} && (12)\\ &= 6{,}58\,K/W && (13)\end{aligned}$$

$\Theta_{pcb-tim}$: resistenza termica tra il PCB e il materiale ad elevata conducibilità termica che assicura un buon contatto termico tra il PCB e il dissipatore (**Thermal Interface Material, TIM**). In questo caso dobbiamo considerare la conducibilità termica dello strato di base del PCB, ovvero dell'Alluminio, pari a 150 W/(m K) [1]. Lo spessore dello strato di alluminio è tipicamente d = 1,6 mm. Supponendo che l'area del PCB sia pari a S_{pcb} = 30 x 30 cm^2 = 0,09 m^2 la resistenza termica diventa:

$$\begin{aligned}\Theta_{pcb-tim} &= (K_{th} \cdot d \cdot S_{pcb})^{-1} && (14)\\ &= (150\,W/(m\,K) \cdot 0{,}09\,m^2/1{,}6 \times 10^{-3}\,m)^{-1} && (15)\\ &= 1{,}2 \times 10^{-4}\,K/W && (16)\end{aligned}$$

Θ_{tim-hs}: resistenza termica tra il TIM e il dissipatore (**heat sink**). Considero come TIM una pasta termica con spessore di 500µm, conducibilità termica pari a 5 W/(m K) [1] ed area pari a quella del PCB $S_{tim} = S_{pcb}$ = 30 x 30 cm^2 = 0,09 m^2 si ottiene:

$$\begin{aligned}\Theta_{tim-hs} &= (K_{th} \cdot d \cdot S_{tim})^{-1} && (17)\\ &= (5\,W/(m\,K) \cdot 0{,}09\,m^2/5 \times 10^{-4}\,m)^{-1} && (18)\\ &= 1{,}1 \times 10^{-3}\,K/W && (19)\end{aligned}$$

Θ_{hs-a}: resistenza termica tra il dissipatore e l'ambiente esterno. Va determinata a partire dalla potenza da dissipare per scegliere il dissipatore adeguato.

Per trovare la resistenza termica totale del sistema si applicano le regole delle reti elettriche riguardo serie e paralleli di resistori. In particolare, osservando la figura 28, si ha il parallelo di 225 resistori equivalenti Θ_{j-sp} + Θ_{sp-pcb}, tanti quanti sono i LED che la scheda può ospitare, in serie alle altre resistenze termiche:

$$\Theta_{TOT} = \Theta_{j-hs} + \Theta_{hs-a} \quad (20)$$

$$\Theta_{j-hs} = (\Theta_{j-sp} + \Theta_{sp-pcb})/225 \simeq 0{,}058\,K/W \quad (21)$$

Per ricavare la resistenza termica del dissipatore devo conoscere la potenza termica generata dai LED, la temperatura di giunzione T_J e la temperatura ambiente T_A.

$$T_J = P_{TOT} \cdot (\Theta_{j-hs} + \Theta_{hs-a}) + T_A \quad (22)$$

da cui, invertendo, si ottiene:

$$\Theta_{hs-a} = \frac{T_J - T_A}{P_{TOT}} - \Theta_{j-hs} \tag{23}$$

Imponendo $T_J = 85°$ per un corretto funzionamento dei LED, $T_A = 30°$ e ricordando che la potenza totale dissipata in calore vale:

$$P_{TOT} = V_F \cdot I_F \cdot N_{LED} \cdot 0.75 = 3{,}2\,V \cdot 1\,A \cdot 225 \cdot 0{,}75 = 540\,W \tag{24}$$

si ottiene unaa resistenza termica pari a

$$\Theta_{hs-a} = \frac{85 - 30}{540} - 0{,}058\,K/W = 0{,}043\,K/W \tag{25}$$

La scelta è ricaduta su un dissipatore in Alluminio ad alta efficienza con ventilazione forzata.

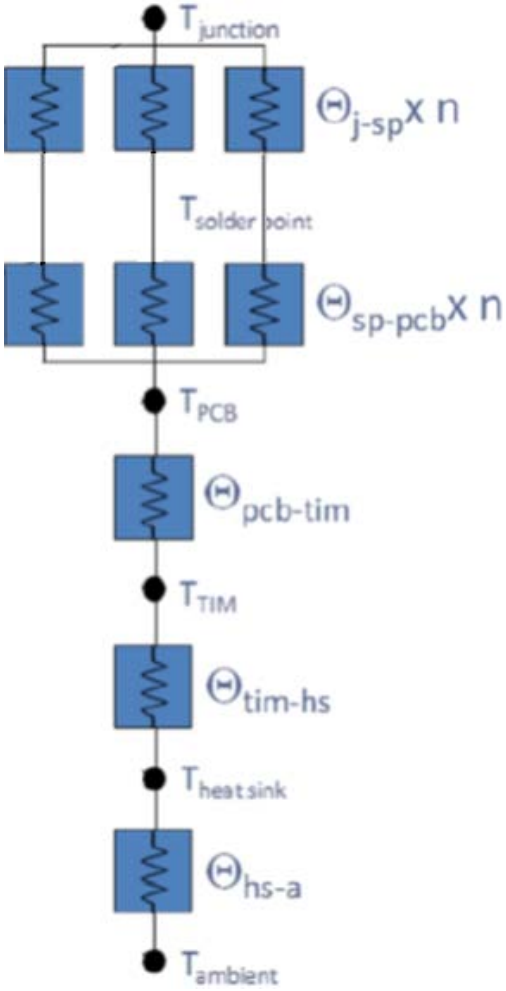


Figura 28: Modello termico della scheda LED [2].

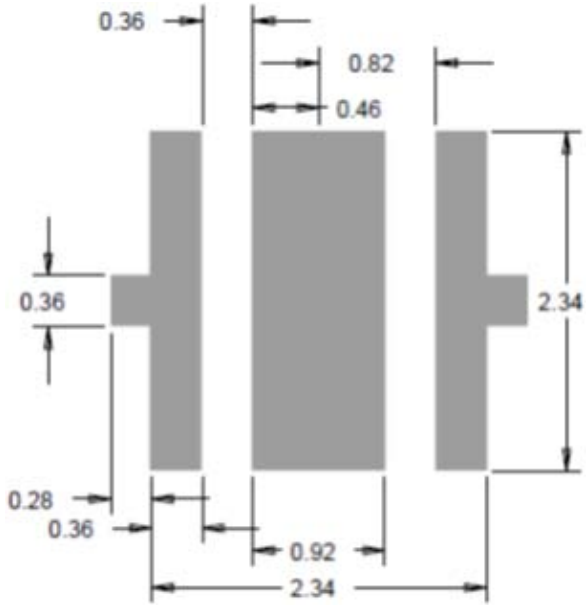


Figura 29: Footprint di saldatura raccomandato per il LED XB-D [3].

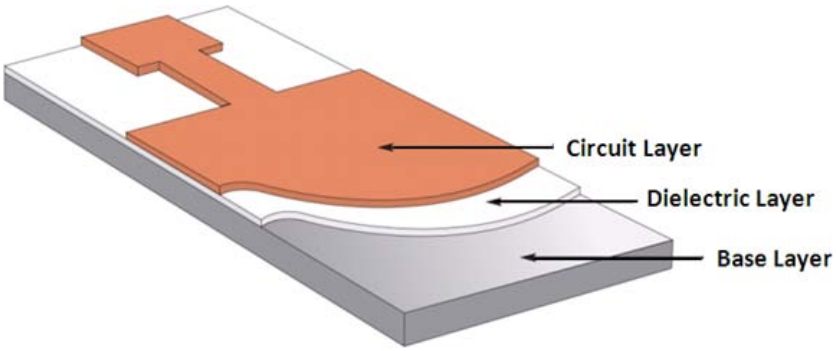


Figura 30: Stackup del PCB metallico T-Clad ® [1].

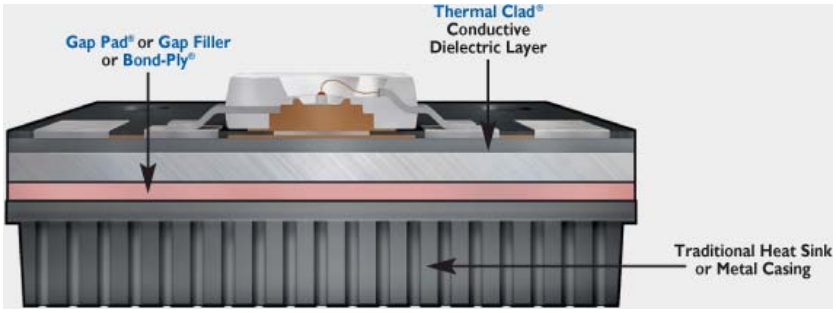


Figura 31: Sezione dell'insieme LED, PCB, TIM e dissipatore [1].

2.5.2 Schema elettrico

Lo schema elettrico della scheda LED è visibile nelle figure 33 e 34. Essa è predisposta per ospitare 15 stringhe indipendenti da 15 LED XB-D ciascuna, per un totale di 225 LED con le corrispondenti ottiche. Per questioni legate alla tensione di alimentazione in questo illuminatore vengono montati solo 13 LED per ciascuna stringa, cortocircuitando le rimanenti due piazzole. La scheda LED ospita anche i connettori per collegare le 15 stringhe alla scheda di potenza nonché un sensore di temperatura digitale **MCP9843** collegato su uno dei bus I2C del sistema [29].

2.5.3 PCB

Come abbiamo visto nel paragrafo relativo al dimensionamento termico, il PCB della scheda LED è realizzato su uno speciale supporto metallico per una dissipazione ottimale del calore generato dai LED. La figura 32 mostra un primo rendering della scheda LED completa mentre le figure 35, 36, 37, 38, 39 mostrano il risultato del PCB realizzato e con 195 LED montati assieme alle ottiche.

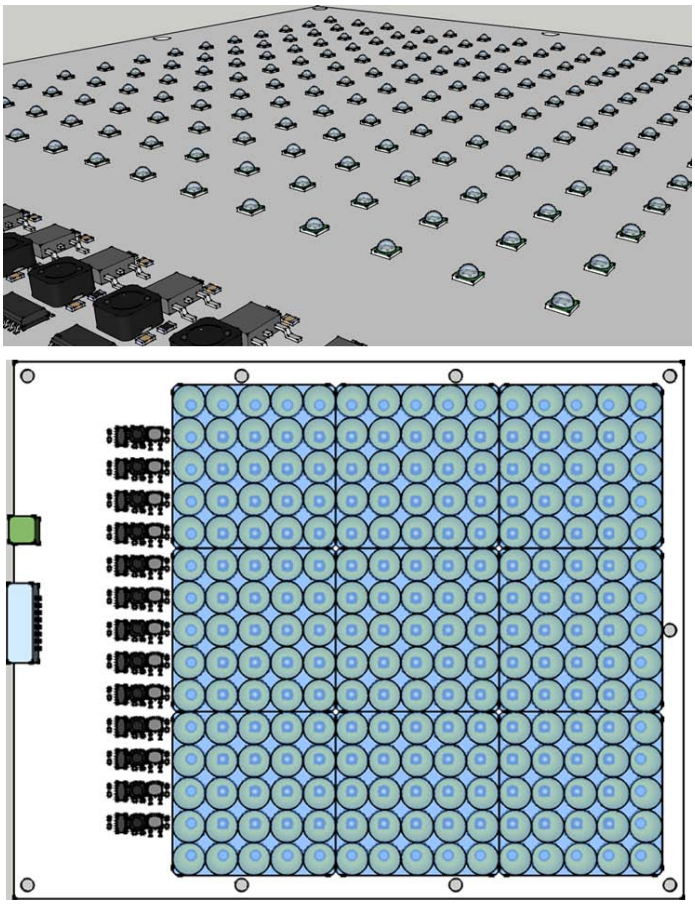


Figura 32: Scheda LED - rendering preliminari.

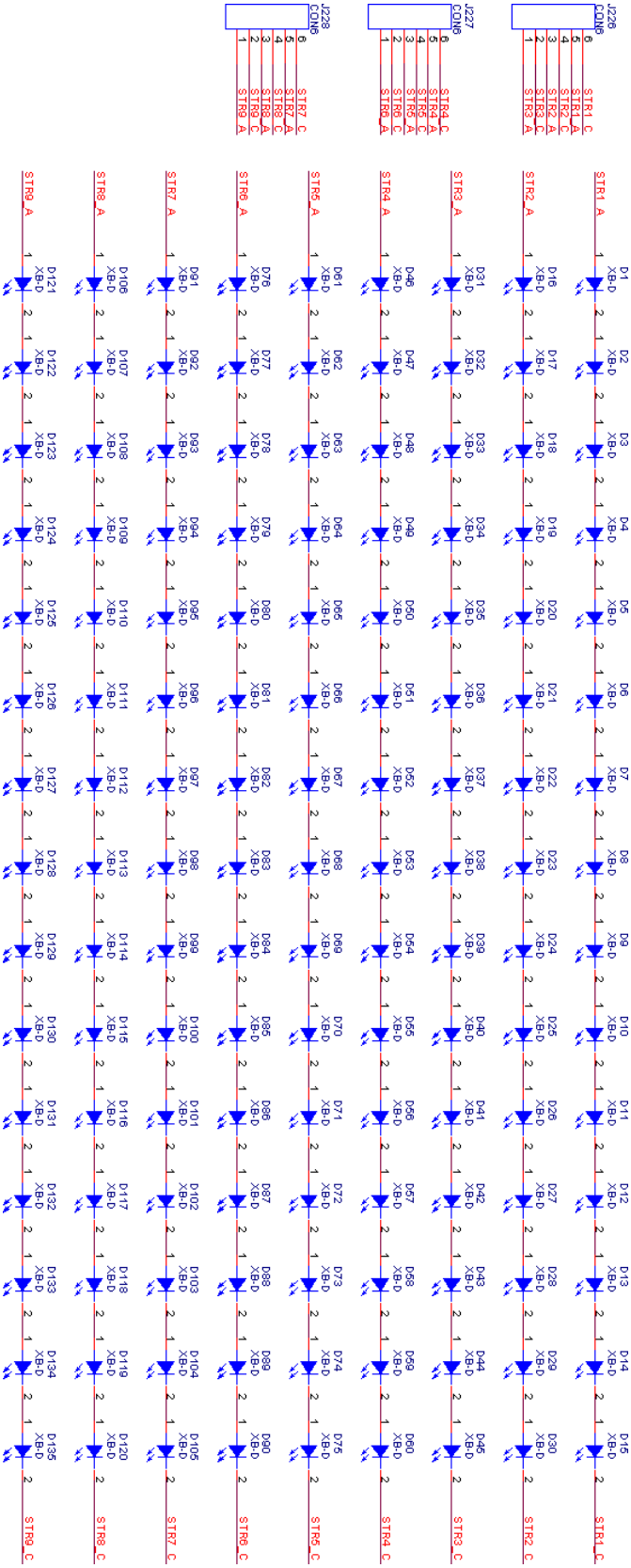


Figura 33: Schema elettrico della scheda LED - parte 1.

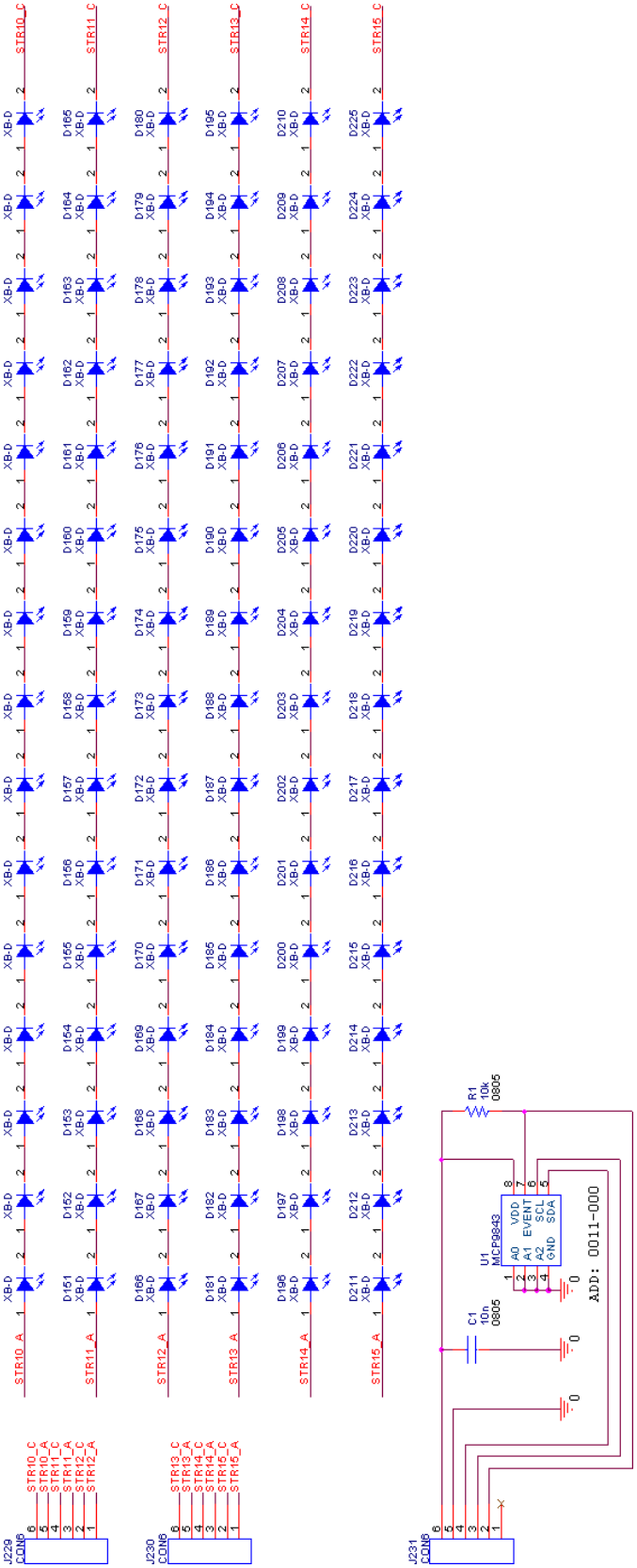


Figura 34: Schema elettrico della scheda LED - parte 2.

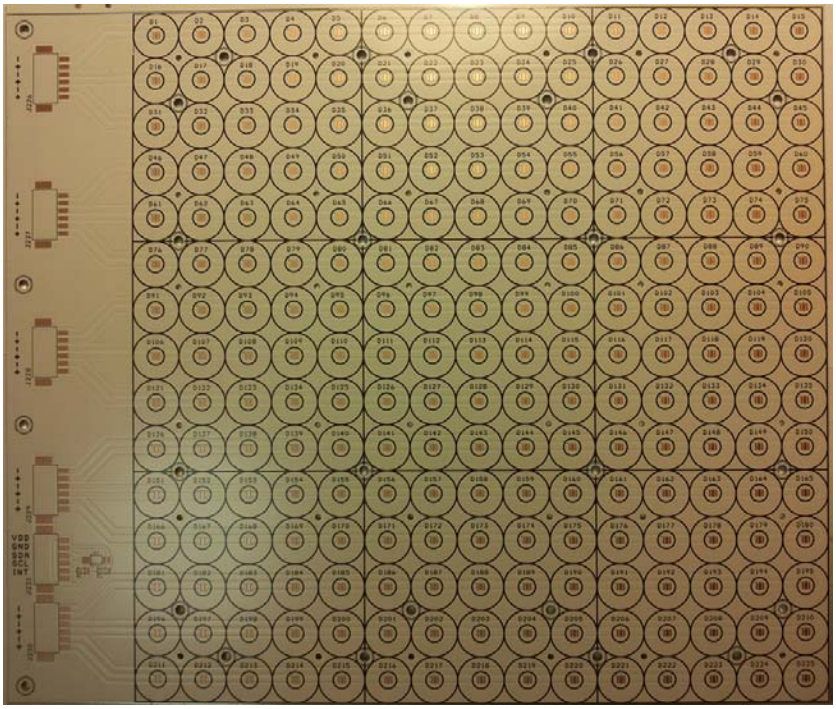


Figura 35: Scheda LED, PCB realizzato.

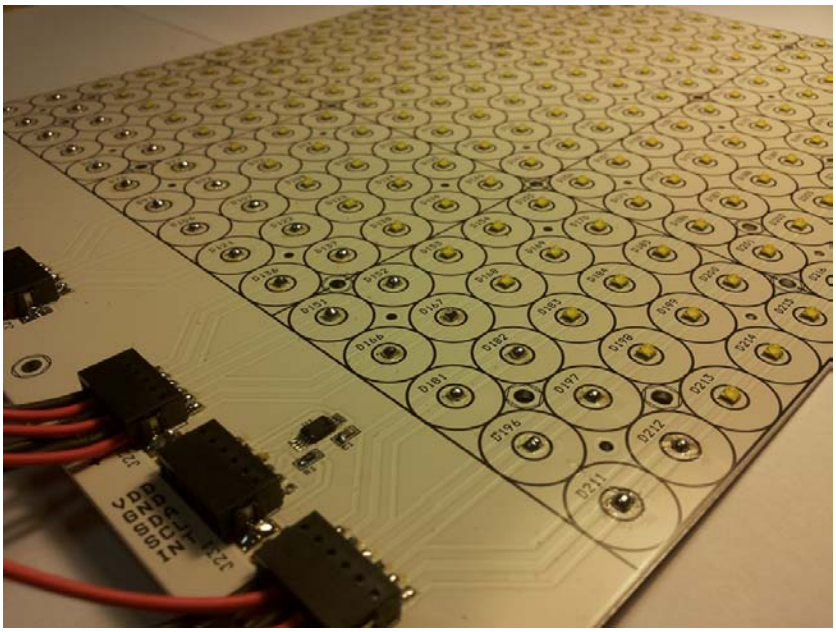


Figura 36: Scheda LED - particolare dei LED e dei connettori. In basso il sensore di temperatura.

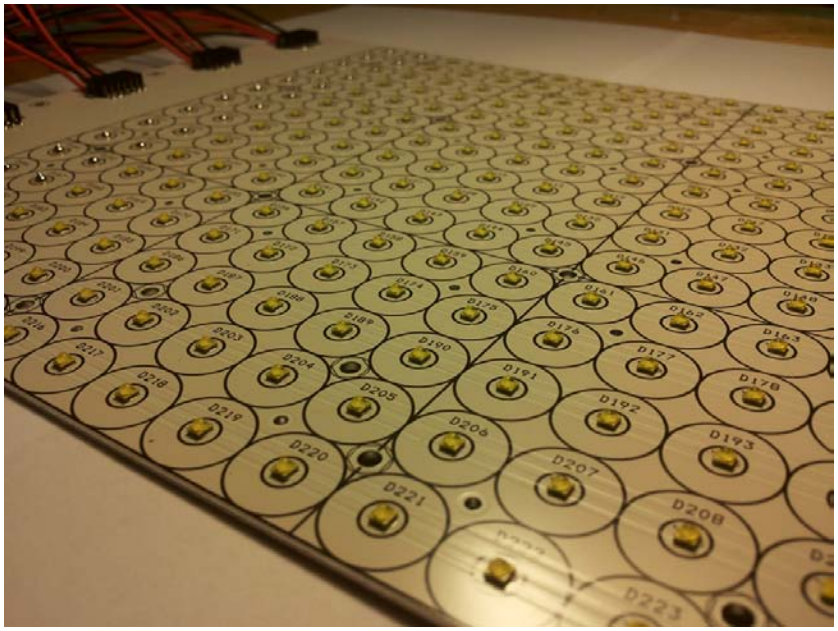


Figura 37: Scheda LED - particolare dei LED.

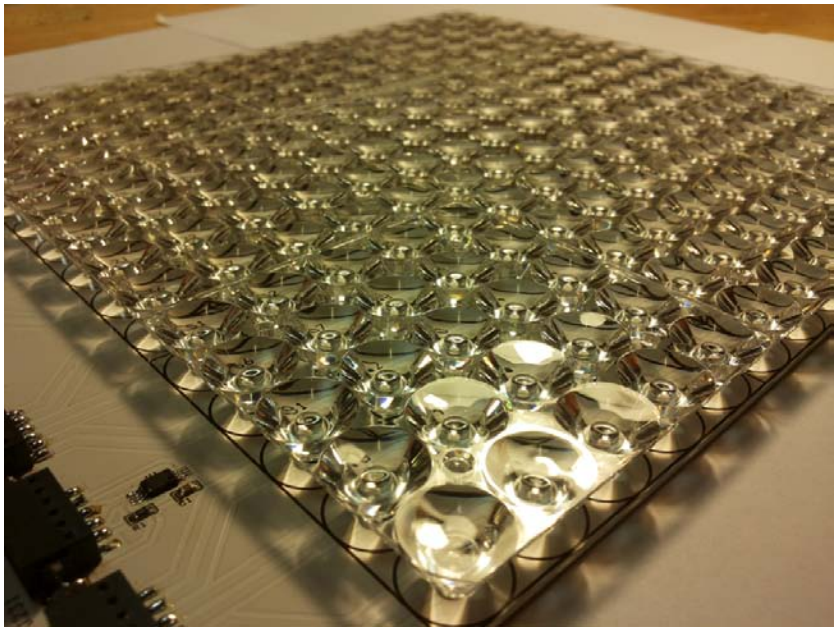


Figura 38: Scheda LED con ottiche montate.

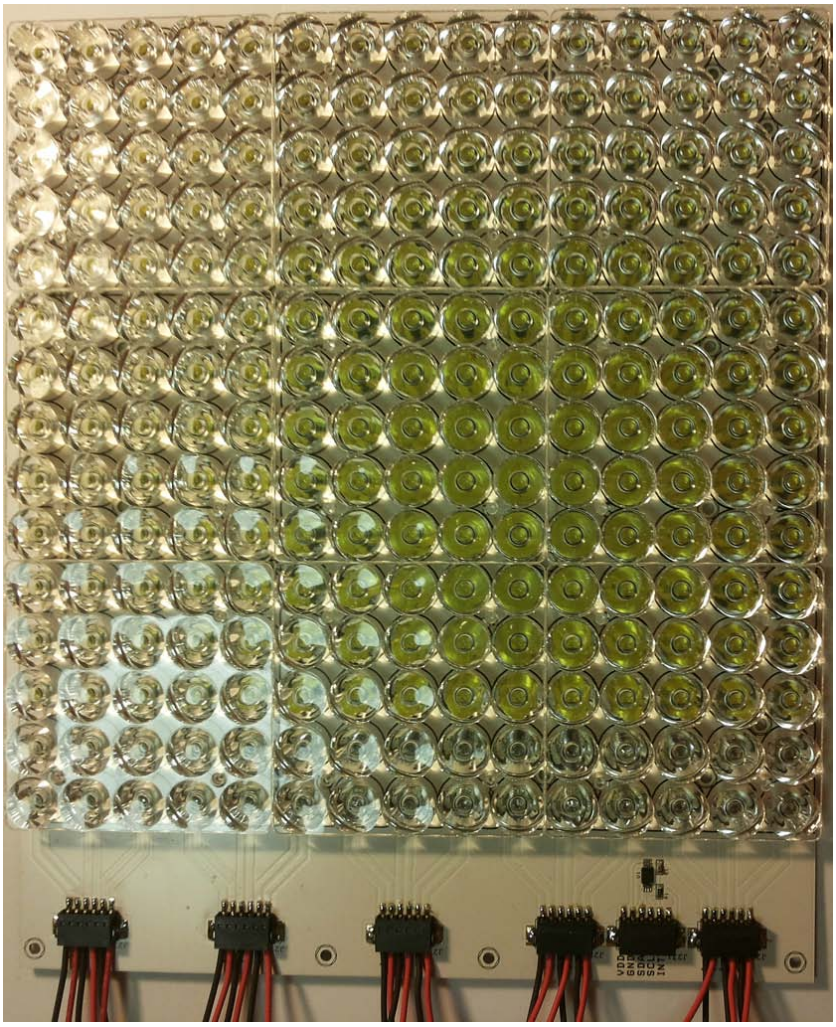


Figura 39: Scheda LED - visione completa con ottiche montate. In basso il sensore di temperatura.

2.6 SCHEDA DI POTENZA

La scheda di potenza contiene i componenti necessari ad alimentare tutti i circuiti digitali del sistema di controllo e i driver per le stringhe di LED.

2.6.1 Schema elettrico

Lo schema elettrico della scheda di potenza è mostrato nelle figure 42 e 43. Poiché le stringhe di LED vengono pilotate a **corrente costante** è necessario che ogni stringa sia alimentata da un apposito **driver**.

La scelta del driver è ricaduta su dei moduli commerciali pronti all'uso: in particolare il modello **RCD-48-1.20/M** prodotto dalla **Recom** ®. Si tratta di un convertitore DC-DC di tipo **buck** con tensione di ingresso compresa tra 9 e 60 VDC e una corrente massima di uscita pari a 1,2 A con un'efficienza tipica a pieno carico del 96%.

La luminosità può essere variata mediante un segnale PWM a bassa frequenza oppure con una tensione analogica di controllo, compresa tra 0 e 5 V (fig.40). Una tensione di 5 V sul pin di controllo corrisponde a spegnere i LED mentre una tensione decrescente corrisponde ad un aumento lineare della corrente, fino alla corrente massima che corrisponde ad una tensione di controllo pari a 0 V. In questo progetto viene utilizzato il metodo analogico in quanto la modulazione PWM non è percepita dall'occhio umano ma influenzerebbe pesantemente la risposta della cella solare.

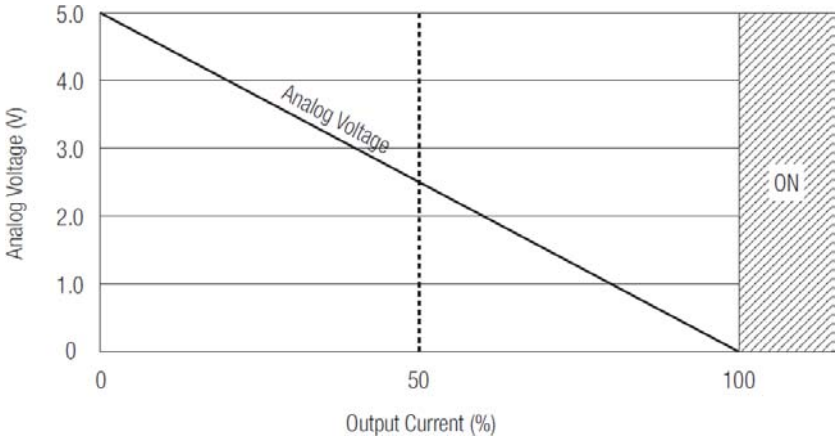


Figura 40: Caratteristica corrente-tensione di controllo del driver RCD-48-1.20/M [15].

Per poter variare la luminosità in maniera digitale vengono utilizzati dei convertitori digitale-analogico (**DAC**) controllati tramite il bus I2C. Si tratta dei **MAX5815** prodotti dalla **Maxim** ® con quattro DAC a 12 bit al loro interno (fig.41) [11].

Ogni uscita dei DAC è collegata all'ingresso di controllo di ciascun driver Recom, e la tensione di riferimento di 5 V per ciascuno dei quat-

tro DAC U16, U17, U18, U19 viene prelevata dal pin Vref dei driver (fig.43).

Ogni DAC ha un indirizzo diverso in modo da poter essere individuato univocamente tra tutti i dispositivi collegati sul bus I2C. La differenziazione degli indirizzi avviene tramite il collegamento a massa o all'alimentazione di alcuni pin dedicati.

Tutto il sistema digitale è alimentato a 3,3 V mentre la parte analogica dei driver Recom necessita di un range di tensione fino a 5 V. Per questo motivo è stato necessario prevedere un piccolo convertitore DC-DC (U21) che a partire da 3,3 V ricava una tensione di 5,5 V per alimentare la parte analogica dei DAC (**MAX1797**) [9].

La tensione di alimentazione generale a 3,3 V invece viene ricavata a partire da 48 V grazie a un modulo DC-DC (U20) della **Traco Power** ®, in particolare il modello TEN8-4810WI da 8 Watt [16], visibile in figura 44 in alto a sinistra.

Il dettaglio della sezione di alimentazione è visibile in figura 42.

I connettori da J1 a J5 sono vanno collegati ai corrispondenti connettori della scheda LED, J6 è collegato al sensore di temperatura sulla scheda LED mentre J7 alimenta la scheda di controllo e collega il bus I2C.

2.6.2 PCB

Le figure 44 e 45 mostrano rispettivamente il Top Leyer e il Bottom Layer del PCB della scheda di potenza al CAD, mentre la figura 46 mostra la scheda realizzata con i componenti montati: sono chiaramente visibili la sezione di alimentazione in alto a sinistra, i driver Recom e i quattro DAC.

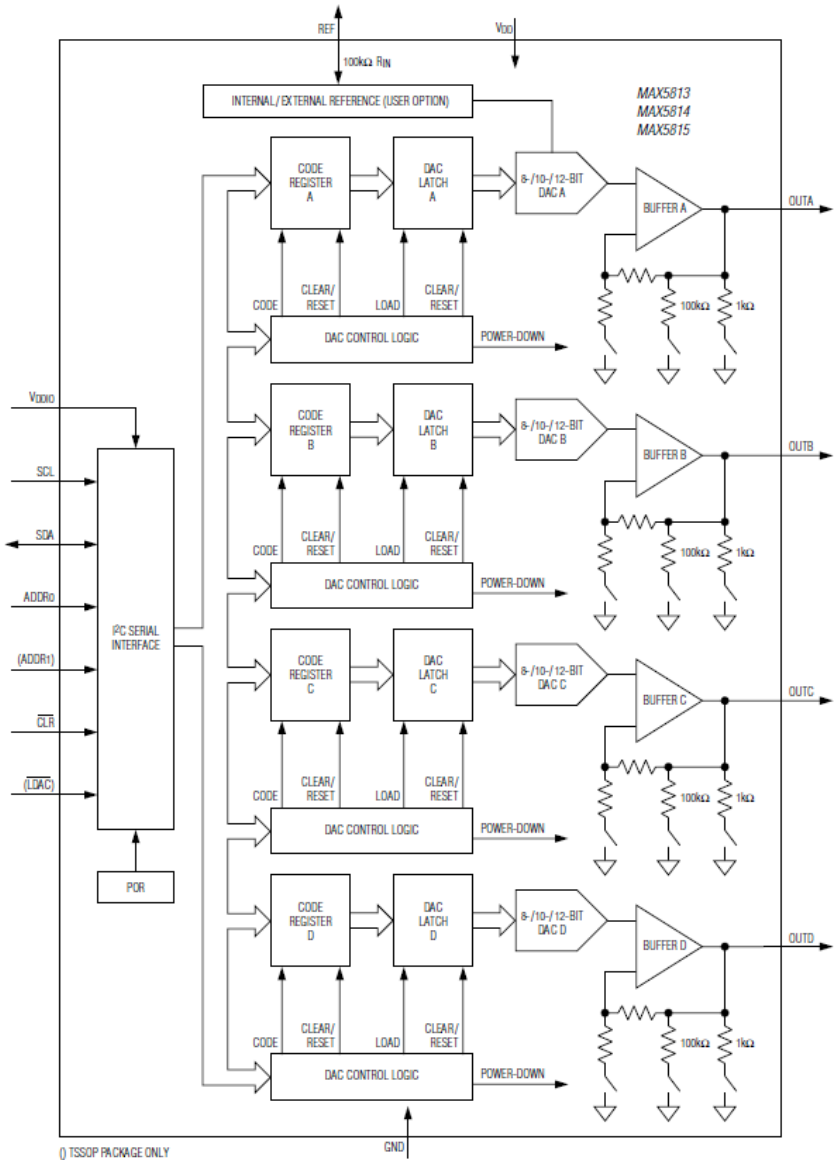


Figura 41: Schema a blocchi interno del DAC MAX5815 [11].

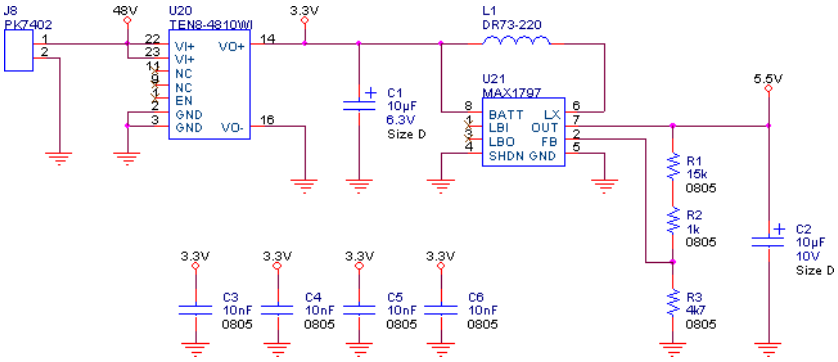


Figura 42: Schema elettrico della scheda di potenza - parte di alimentazione.

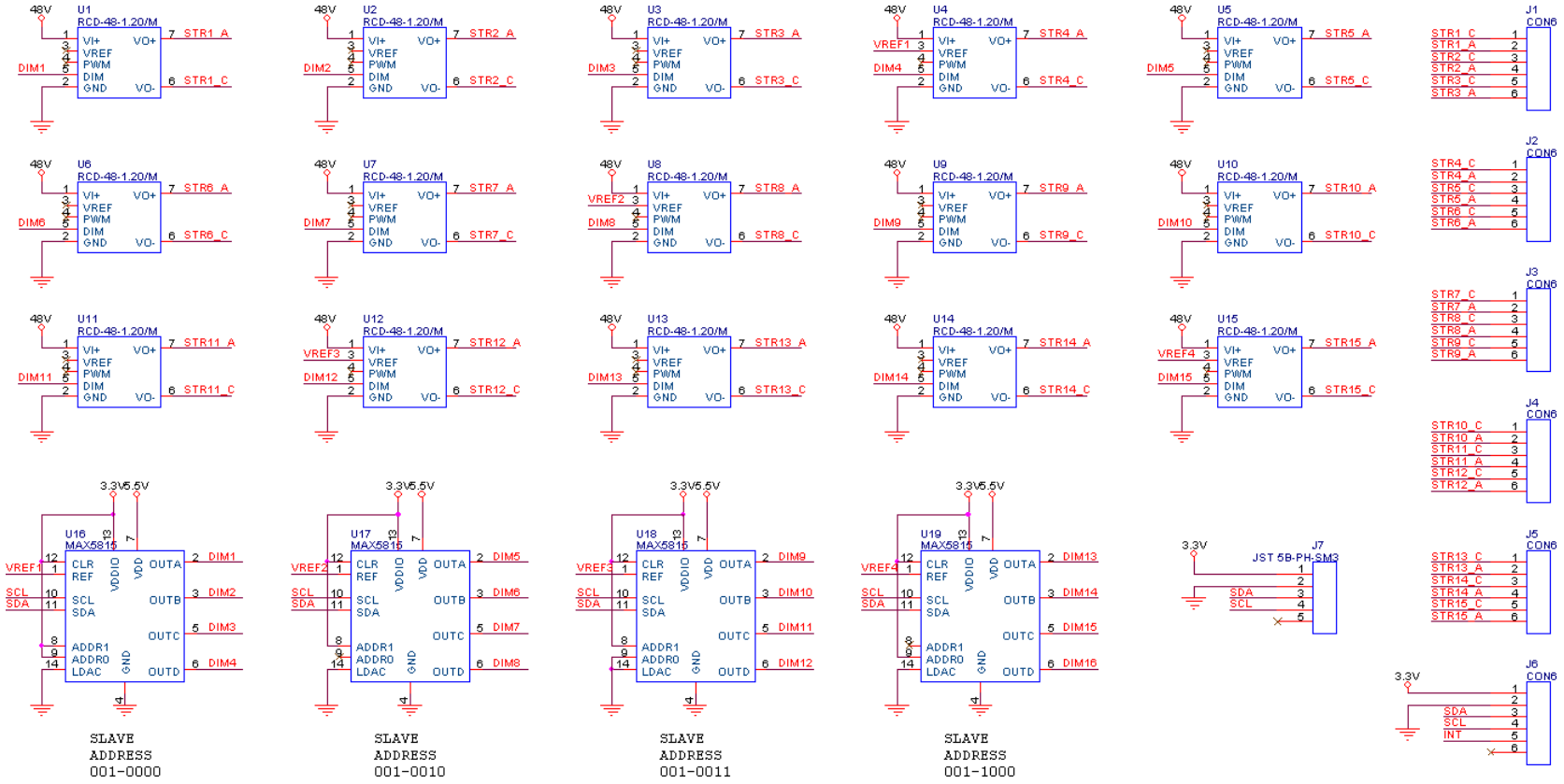


Figura 43: Schema elettrico della scheda di potenza - parte driver e DAC.

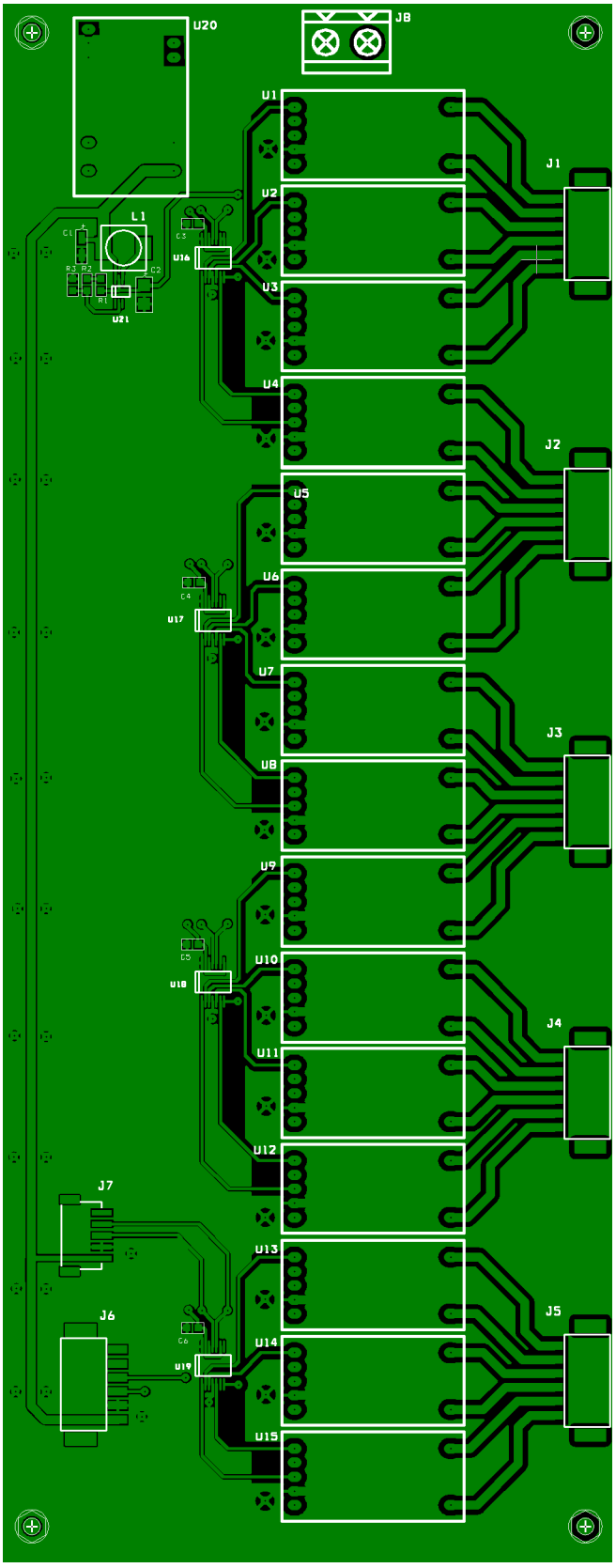


Figura 44: Visione della scheda di potenza al CAD con serigrafia - Top Layer.

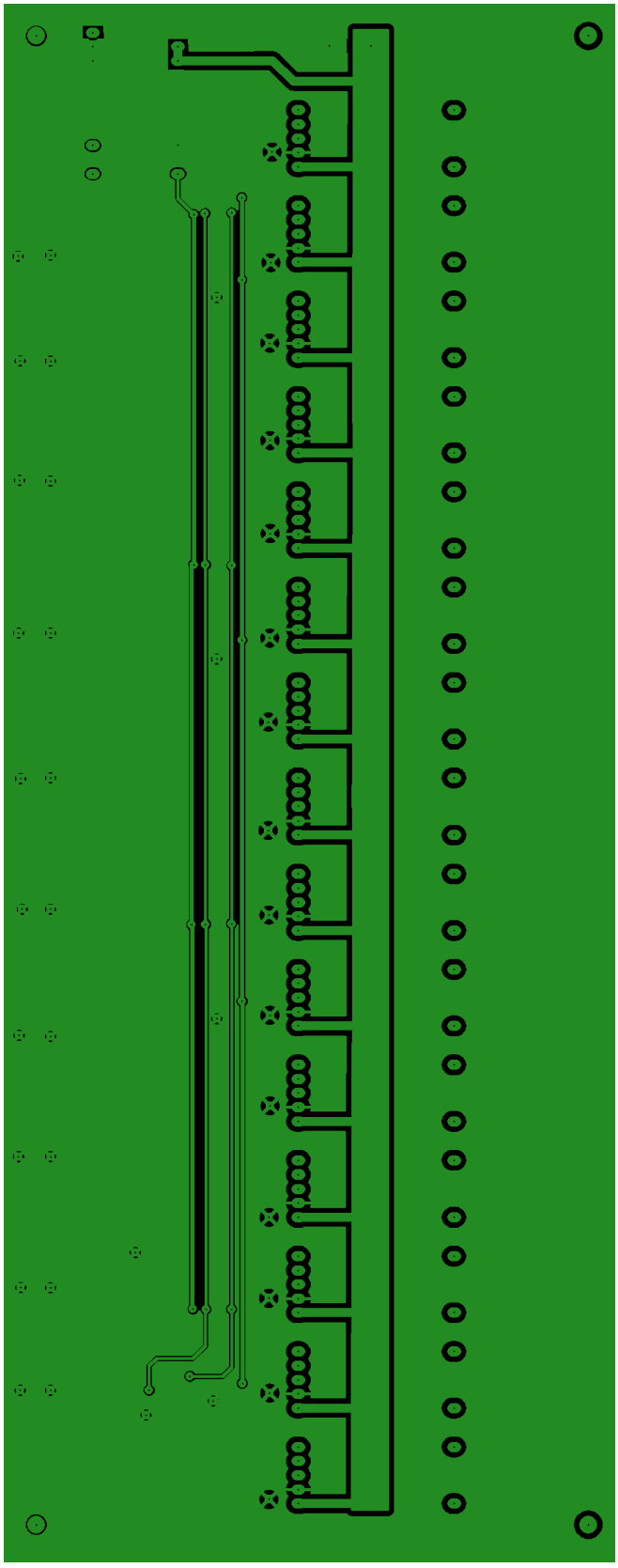


Figura 45: Visione della scheda di potenza al CAD con serigrafia - Bottom Layer.

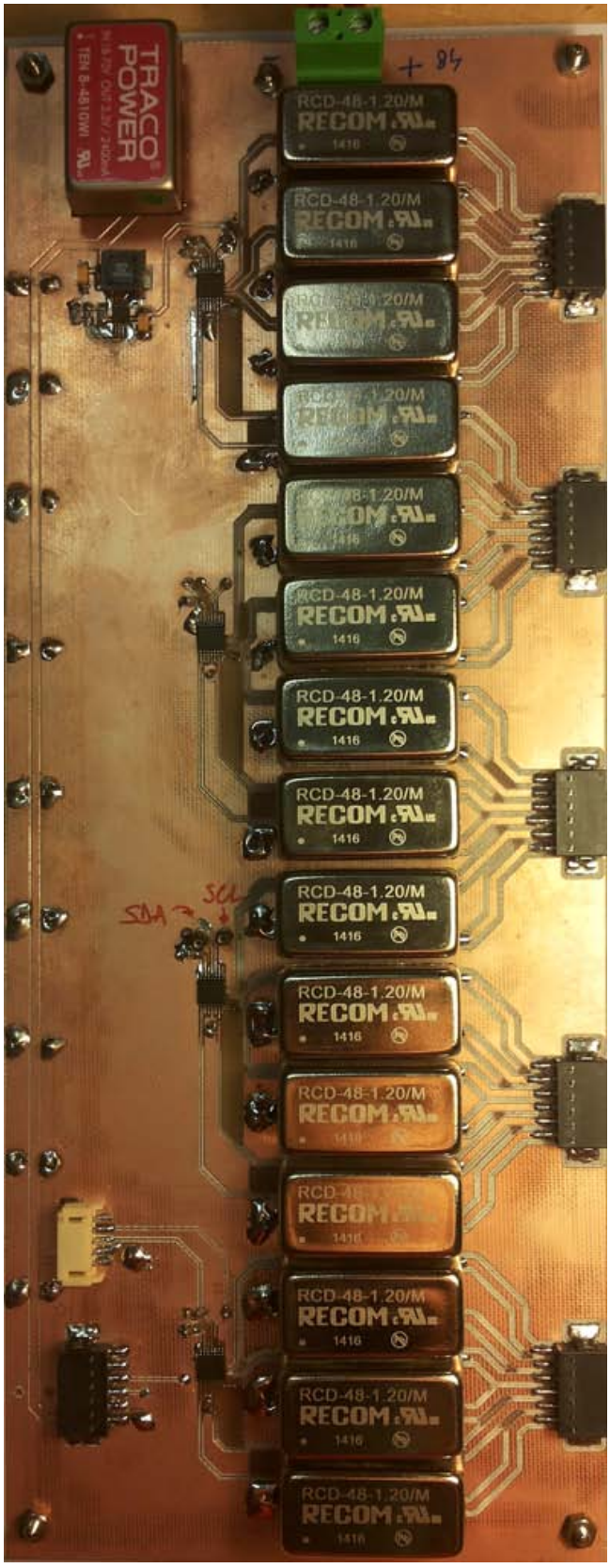


Figura 46: Scheda di potenza realizzata e con i componenti montati.

2.7 SCHEDA MUX E SENSORI DI LUCE

Uno degli scopi dell'illuminatore progettato è quello di poter impostare un valore numerico di irradianza e che tale valore venga mantenuto costante dal sistema. Per fare ciò sono stati introdotti quattro sensori di luce da porre al contorno della cella solare oggetto della misura, che permetteranno di implementare un algoritmo di controllo.

La scelta dei sensori è ricaduta sul modello **TCS34725** della **Austria Microsystems** ®. Si tratta di un **sensore di colore** con interfaccia I2C contenente quattro fotodiodi, uno per il canale rosso, uno per il canale blu, uno per il canale verde e uno per il canale di luce bianca. Lo schema a blocchi del sensore, il suo pinout, nonché la responsività dei fotodiodi sono riportate nella figura 47 [13]. La risoluzione di lettura del sensore è variabile da 10 a 16 bit a seconda del tempo di integrazione impiegato dal chip per effettuare la misura.

Data l'elevata intensità luminosa di questo illuminatore, è stato predisposto l'uso di **filtri attenuatori** di tipo **neutral density** da interporre tra la sorgente e il sensore. L'ordine di grandezza dell'attenuazione andrà calcolato in base all'irradianza massima della lampada tenendo conto della responsività di ciascun canale del sensore.

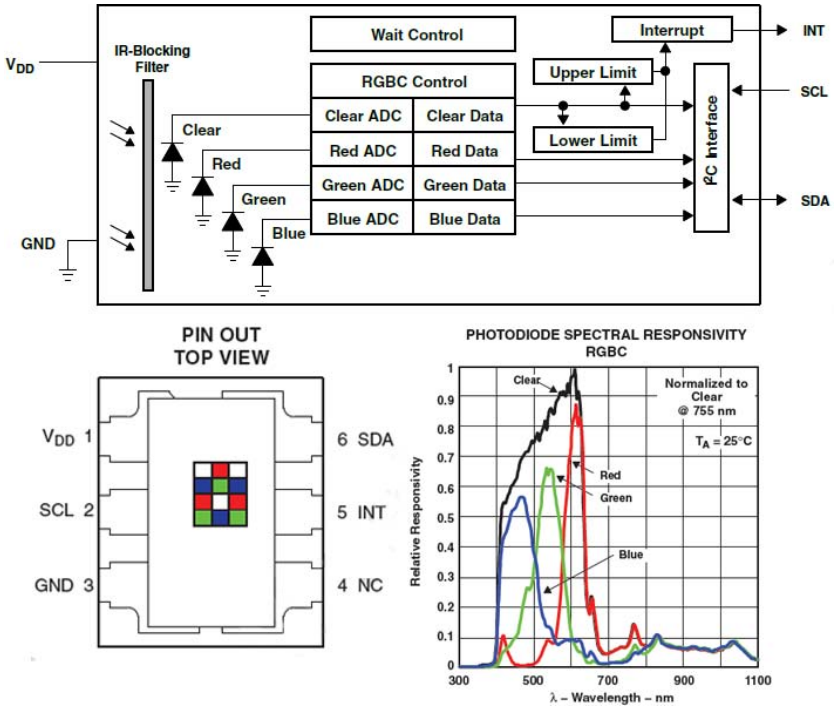


Figura 47: Schema a blocchi, pinout e responsività del sensore di colore TCS34725 [13].

Poiché questi sensori hanno un indirizzo I2C identico e non differenziabile, per poter collegare quattro sensori sullo stesso bus I2C si è reso necessario l'utilizzo di un **multiplexer I2C PCA9544A** [8] che permette di leggere un sensore alla volta in sequenza.

2.7.1 Schema elettrico

Le figura 48 mostra lo schema di collegamento del sensore di colore TCS34725. Si tratta di un semplice connettore (J1) che collega ciascun sensore alla scheda MUX, il cui schema è invece visibile in figura 49. L'integrato che funge da multiplexer (PCA9544A) è U1, SW1 è un banco di switch che permette di variare l'indirizzo I2C del multiplexer stesso, le varie resistenze sono i pull-up necessari al funzionamento del bus I2C mentre J3 è il connettore per collegare il bus I2C alla scheda di controllo e alimentare sia il MUX che i sensori.

2.7.2 PCB

Le figure 50 e 51 mostrano rispettivamente il Top Layer e il Bottom Layer della scheda multiplexer vista al CAD, mentre la figura 52 mostra la stessa scheda realizzata e con i componenti montati.

Le figure 53 e 54 invece mostrano rispettivamente il PCB del sensore visto al CAD e con i componenti montati. In figura 50 si vede chiaramente la predisposizione per il supporto delle lenti attenuatrici intercambiabili.

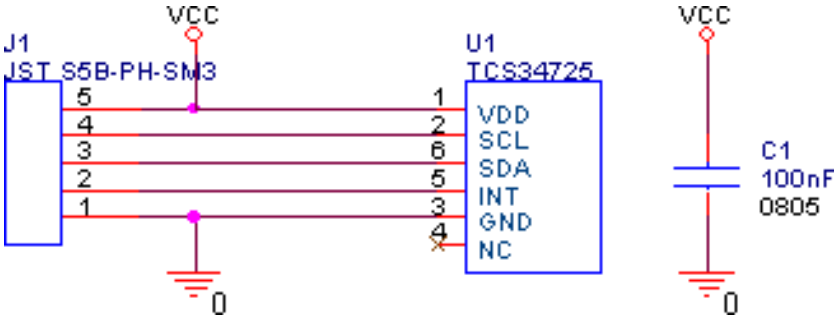


Figura 48: Schema elettrico della scheda con il sensore di colore TCS34725.

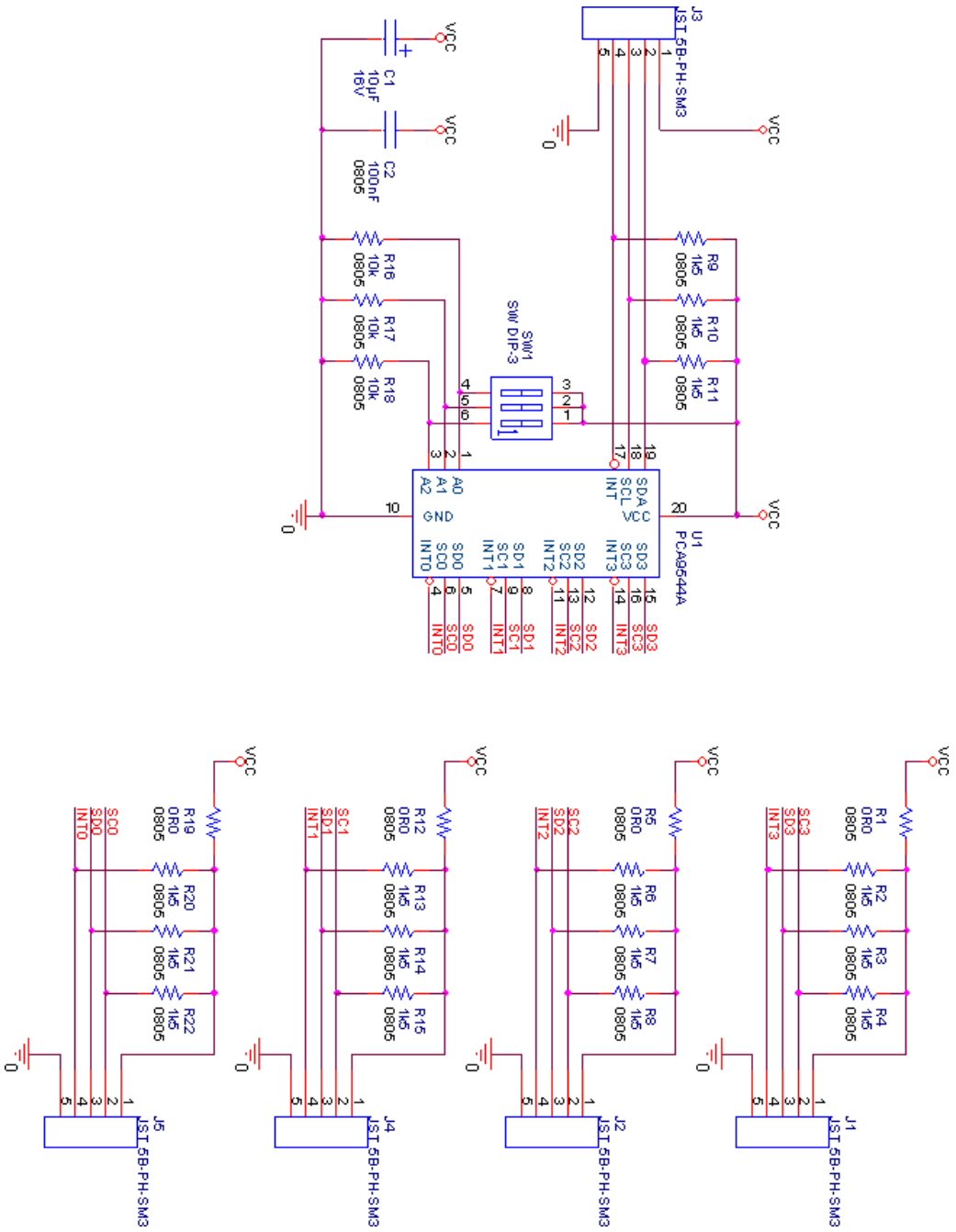


Figura 49: Schema elettrico della scheda multiplexer I2C.

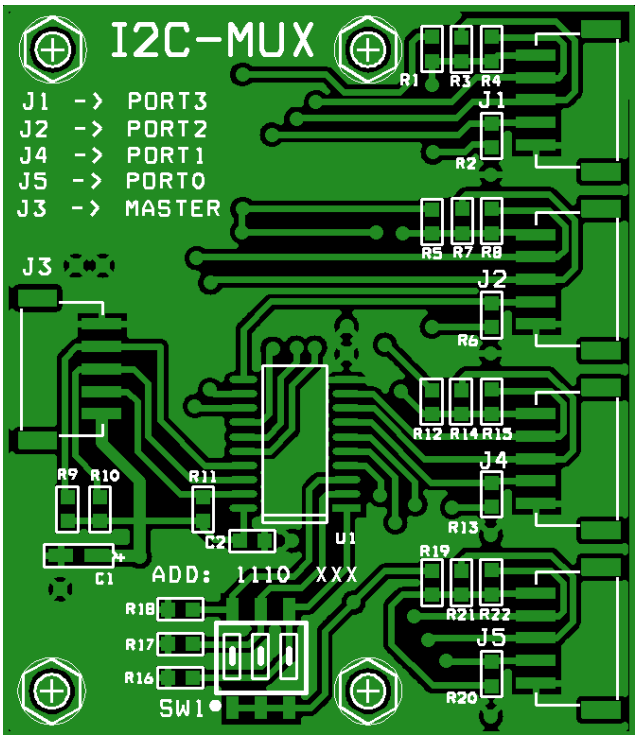


Figura 50: Visione della scheda multiplexer I2C al CAD con serigrafia - Top Layer.

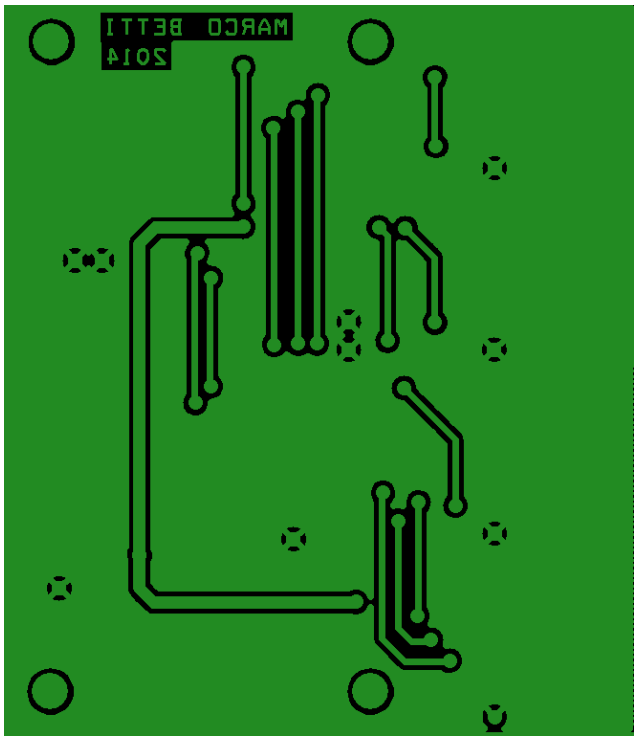


Figura 51: Visione della scheda multiplexer I2C al CAD - Bottom Layer.

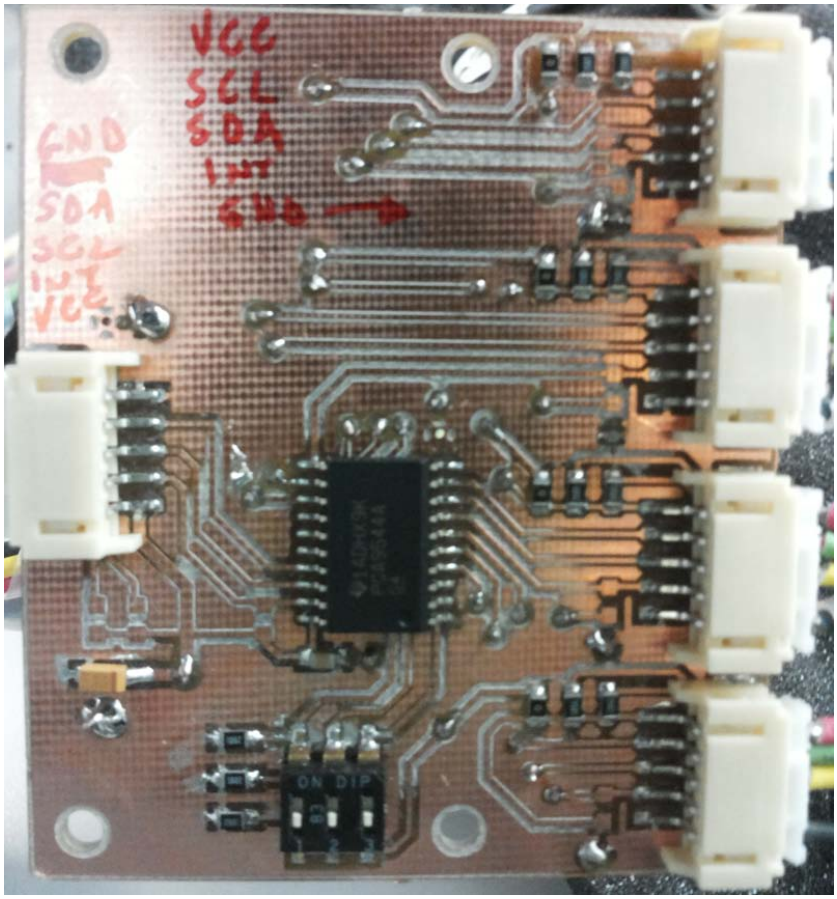


Figura 52: Scheda multiplexer I2C realizzata e con i componenti montati.

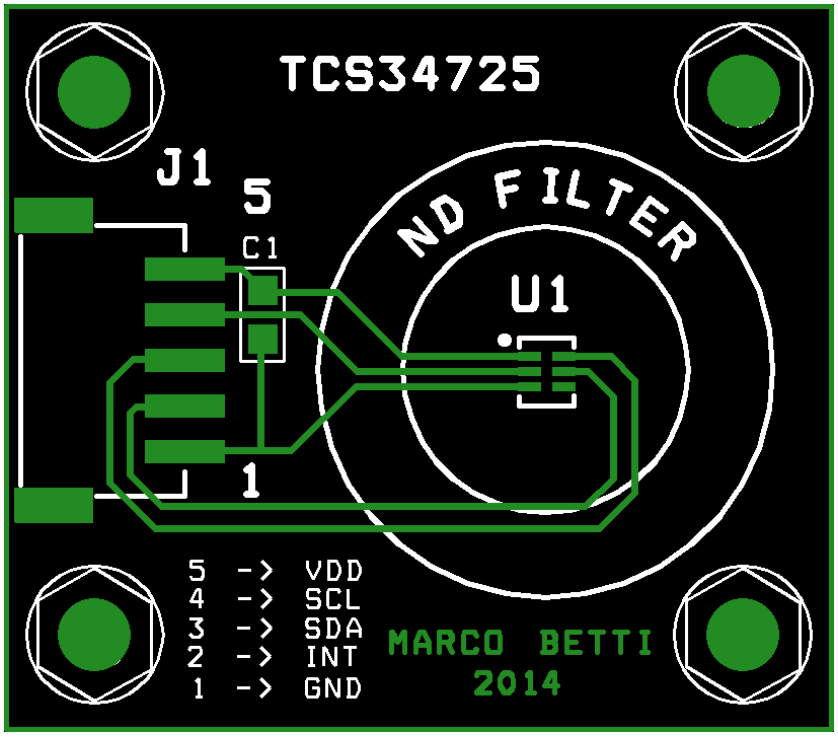


Figura 53: Visione della scheda sensore di luce al CAD con serigrafia - Top Layer.

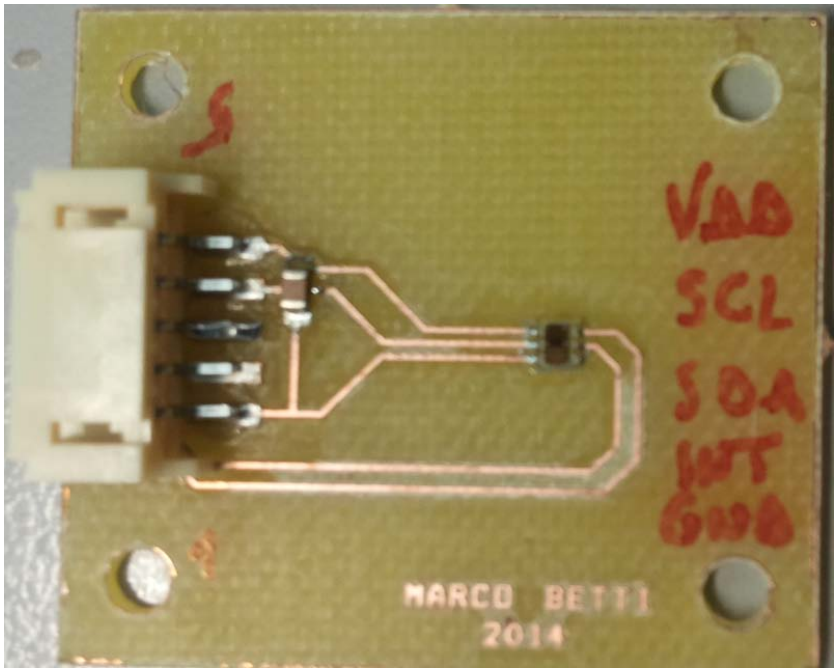


Figura 54: Scheda sensore di luce realizzata e con i componenti montati.

2.8 IL BUS I2C

Come abbiamo visto nei paragrafi precedenti, tutto il sistema di controllo e gestione dell'illuminatore è basato sul protocollo I2C.

I2C è l'acronimo di **Inter-Integrated Circuit**: si tratta di un sistema di comunicazione seriale a due fili utilizzato tra circuiti integrati, sviluppato nel 1982 dalla **Philips** ®, oggi nota come **NXP Semiconductors** ®.

Le caratteristiche principali del bus I2C sono:

- Solo due fili per la comunicazione: una linea dati **(Serial Data, SDA)** e una linea di clock **(Serial Clock, SCL)**. Ogni linea necessità di una resistenza di **pull-up**, poiché i dispositivi I2C sono di tipo **open-drain**;
- Ogni dispositivo connesso al bus è identificabile grazie ad un **indirizzo univoco**, che può essere a 7 bit o a 10 bit. L'indirizzo è diviso in un primo gruppo di bit più significativi che viene assegnato ad ogni produttore di semiconduttori, mentre i bit meno significativi possono essere variati mediante dei pin dedicati, in modo da far coesistere più dispositivi uguali sullo stesso bus;
- La comunicazione seriale avviene con pacchetti delle dimensioni di **8 bit**, in maniera bidirezionale, con velocità fino a 100 kbit/s in modalità Standard, fino a 400 kbit/s in modalità Fast e fino a 3.4 Mbit/s in modalità High-speed;
- Il controllo del bus è affidato ad un dispositivo detto **Master**, mentre gli altri dispositivi si comportano da **Slave**. Ogni Master è responsabile della generazione del clock sulla linea SCL durante le proprie trasmissioni. L'architettura del protocollo I2C prevede che sullo stesso bus possa esserci più di un dispositivo Master, grazie a delle tecniche di **collision detection** e **arbitraggio** che evitano la corruzione dei dati nel caso due Master trasmettano contemporaneamente;
- Il numero di dispositivi che possono essere connessi allo stesso bus è limitato solo dalla disponibilità di indirizzi diversi e dalla capacità massima del bus, che non deve superare i 400 pF.

La figura 55 mostra un esempio di applicazione del bus I2C.

2.8.1 Protocollo I2C

Vediamo ora più nel dettaglio come si compone un pacchetto dati I2C.

Quando il bus è libero, entrambe le linee SDA e SCL sono a livello logico alto (grazie ai pull-up). Un esempio di trasmissione I2C è riportata in fig.56. Nel caso in cui un Master desideri inviare dei dati ad uno Slave deve seguire la procedura seguente (fig:57):

1. La trasmissione inizia con una condizione di **START (S)** da parte del Master, in cui la linea SDA viene forzata a livello basso mentre la linea SCL è ancora a livello alto;

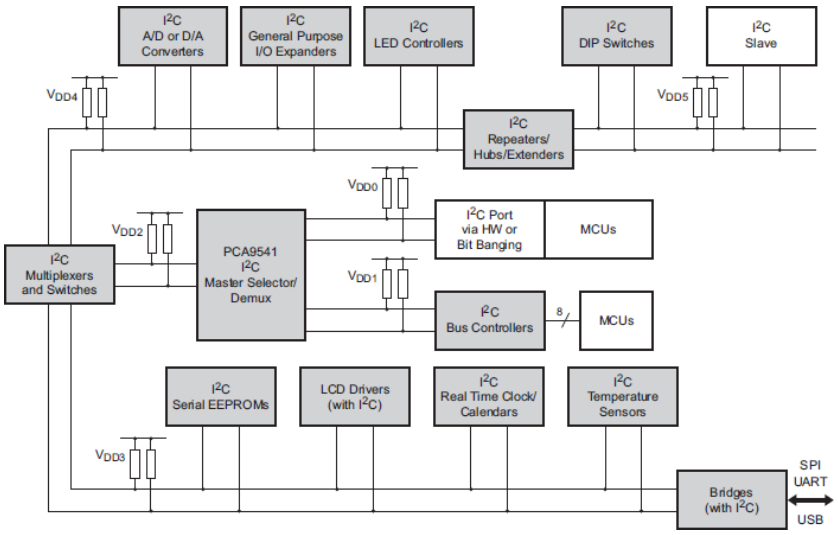


Figura 55: Esempio di applicazione del Bus I2C [24].

2. Il Master invia sul bus i 7 bit più significativi dell'indirizzo dello Slave, seguiti da un ottavo bit R/W che determina la direzione della comunicazione: R/W = 0 significa che lo Slave viene scritto, R/W = 1 significa che lo Slave viene letto;

3. Se l'indirizzo è decodificato correttamente lo Slave mantiene la linea SDA a livello basso, per segnalare il cosiddetto **Acknowledge, ACK**;

4. Il Master trasmette 8 bit di dati alla volta, ciascuno seguito da un ACK da parte dello Slave

5. Per terminare la comunicazione il Master genera una condizione di **STOP (P)**, in cui la linea SCL viene forzata a livello alto mentre la linea SDA è ancora a livello basso.

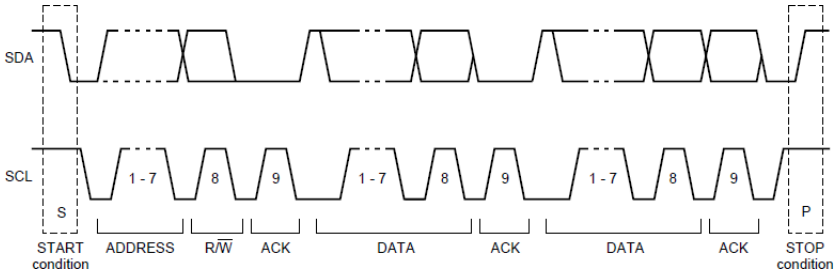


Figura 56: Esempio di trasferimento dati sul Bus I2C [24].

Nel caso in cui il Master desideri indirizzare uno Slave per poi leggerne dei dati, la procedura è la seguente (fig:58):

1. Il Master invia una condizione di **START (S)** sul bus;

2. Il Master invia i 7 bit più significativi dell'indirizzo dello Slave seguiti dal bit R/W = 0; lo Slave manda un ACK;
3. Il Master trasmette dei comandi per impostare la lettura dei dati. Tali comandi sono caratteristici del device usato come Slave; lo Slave manda un ACK;
4. Il Master, invece che generare uno STOP, genera una condizione di **repeated START (Sr)**;
5. Il Master invia i 7 bit più significativi dell'indirizzo dello Slave seguiti dal bit R/W = 1 per passare in modalità lettura; lo Slave manda un ACK;
6. Il Master genera il clock e lo Slave invia 8 bit di dati alla volta; Ogni 8 bit ricevuti il Master deve generare un ACK; quando l'ultimo byte di dati viene ricevuto il Master genera un NACK (la condizione opposta dell'Acknowledge);
7. Per terminare la comunicazione il Master genera una condizione di **STOP (P)**.

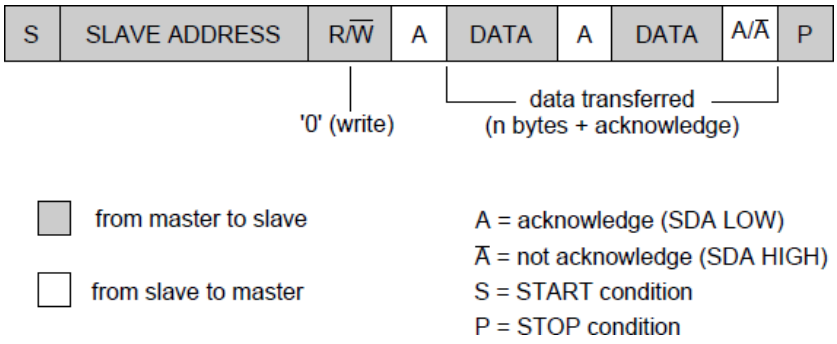


Figura 57: Esempio di scrittura sul Bus I2C da Master a Slave [24].

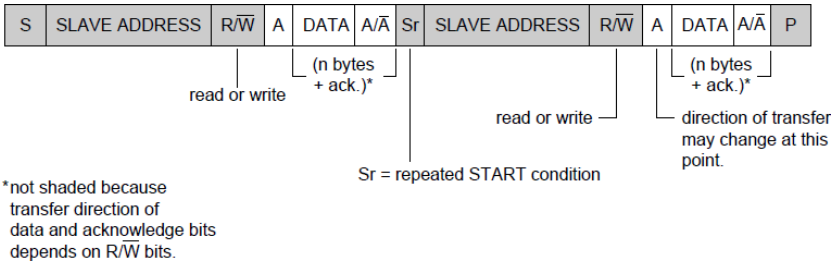


Figura 58: Esempio scrittura e lettura combinate sul Bus I2C [24].

A titolo di esempio le figure 59 e 60 mostrano due screenshot dell'oscilloscopio relative rispettivamente alla comunicazione con il DAC MAX5815, usato per variare la luminosità dei LED, e con il multiplexer I2C PCA9544A, utilizzato per poter collegare più sensori di luce con lo stesso indirizzo sullo stesso bus.

Nella figura relativa al DAC si vedono chiaramente le condizioni di START e STOP, il primo byte di dati, che corrisponde all'indirizzo dello Slave, e i byte successivi, che in questo caso modificano il valore di tensione in uscita al DAC. Tale valore è rappresentato dal segnale di colore viola che, come si può vedere, varia non appena i comandi I2C sono stati interpretati correttamente dal DAC.

Nella transazione relativa al MUX, invece, i due byte di dati sono rispettivamente l'indirizzo del MUX e la porta di uscita selezionata.

La figura 61 mostra uno schema dei device collegati ai due bus I2C utilizzati nel sistema dell'illuminatore LID, con i relativi indirizzi. I bit contrassegnati da una 'X' sono modificabili a discrezione del progettista, collegando degli appositi pin a massa o all'alimentazione.

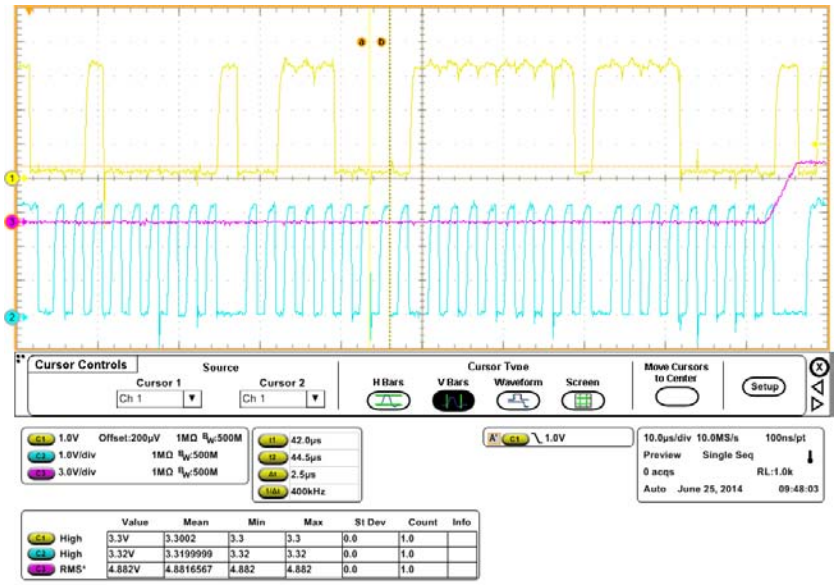


Figura 59: Screenshot dall'oscilloscopio della scrittura di un DAC MAX5815 sul Bus I2C.

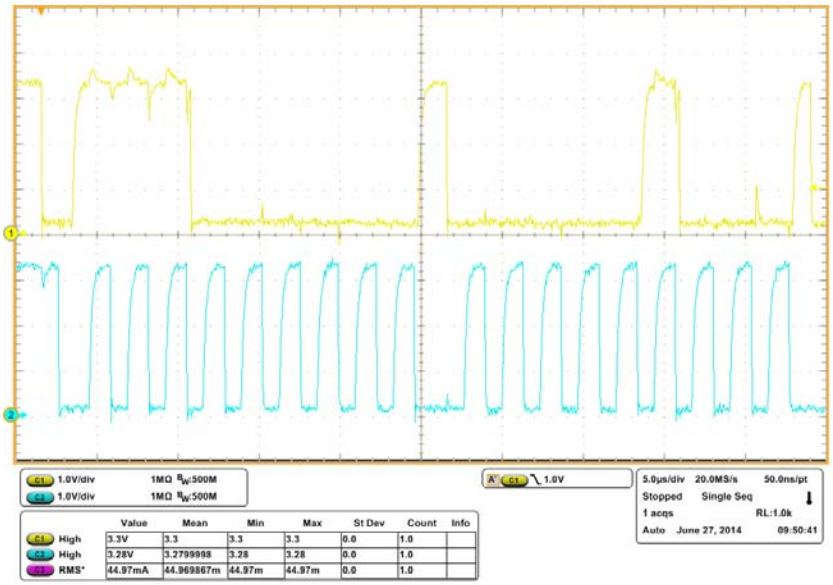


Figura 60: Screenshot dall'oscilloscopio della scrittura di un MUX PCA9544A sul Bus I2C.

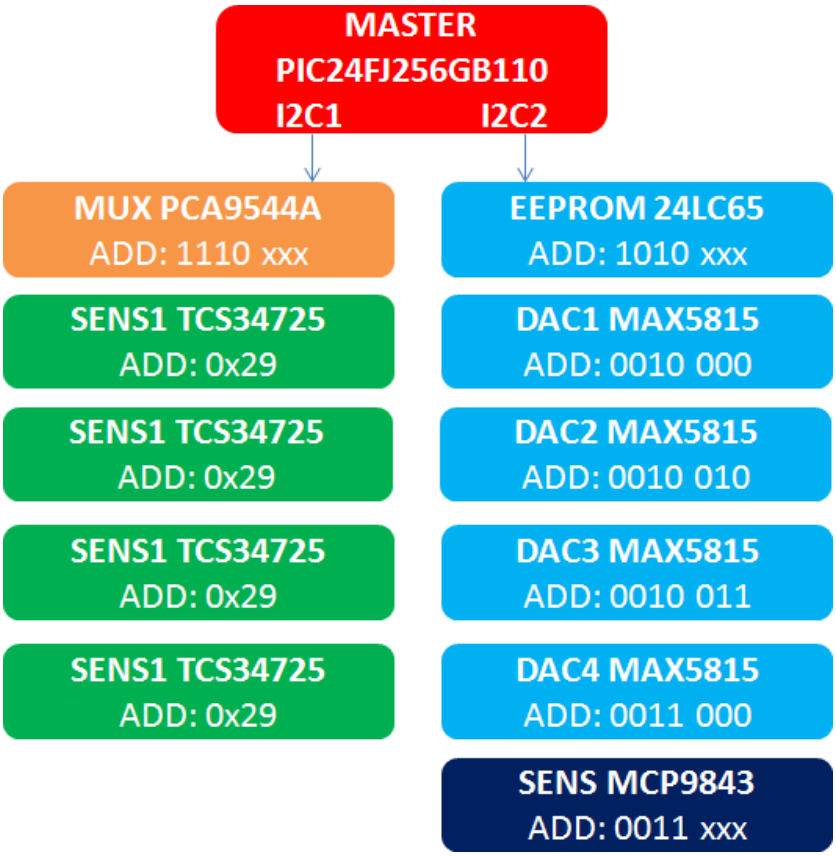


Figura 61: Schema del bus I2C dell'illuminatore LID.

3 PROGETTO SOFTWARE

INDICE

3.1 FIRMWARE DEL MICROCONTROLLORE

Il **firmware** è il programma che gira sul microcontrollore e che permette il funzionamento di tutto il sistema dell'illuminatore LID. Il codice è scritto interamente in linguaggio **C** all'interno dell'ambiente di sviluppo della **Microchip ® MPLAB IDE ®**.

Il compilatore utilizzato è quello fornito dalla Microchip in versione gratuita per studenti, l'**MPLAB C30 Compiler for PIC24 MCUs and dsPIC DSCs ®**.

In questo progetto sono state usate anche le **librerie** fornite da Microchip sia per l'uso delle periferiche del microcontrollore che per quanto riguarda lo **stack USB**. Tutti i riferimenti sono accessibili nella documentazione messa a disposizione nel sito del produttore [30, 26].

Il codice principale del microcontrollore, il cosiddetto *main*, è riportato nel listato 1, di cui analizziamo nel dettaglio alcuni particolari.

Le righe da 17 a 19 sono le cosiddette **configuration words**, che configurano i parametri principali per l'operatività del microcontrollore, come la sorgente di clock, la configurazione dell'oscillatore, le porte per il debug e la programmazione e, in questo caso, la presenza o meno del **PLL**. Il PLL è necessario in applicazioni che fanno uso del modulo USB poiché permette di ricavare una frequenza di 48 MHz necessaria per il corretto funzionamento dell'USB a partire da sorgenti di clock a frequenza inferiore, come un quarzo esterno (nel nostro caso da 16 MHz).

Le righe da 53 a 89 contengono il ciclo principale, che viene eseguito continuamente, dopo aver chiamato le funzioni di inizializzazione dell'USB, dei DAC, dei sensori di colore e di altre periferiche necessarie. In particolare l'inizializzazione dei DAC è visibile alla riga 174, in cui viene scritto il valore $(\texttt{0} \times \texttt{0FFF})_{\texttt{H}}$ nel registro a 12 bit di ciascuno dei DAC presenti sul bus I2C. Tale valore corrisponde ad una tensione di 5 V necessaria per mantenere i driver Recom spenti.

In seguito (righe da 176 a 183) vengono inizializzati i sensori di colore; poiché questi hanno tutti lo stesso indirizzo I2C è necessario sele-

zionare l'uscita del MUX I2C corrispondente prima di comunicare con ogni sesore.

Il codice USB utilizzato, basato su un codice dimostrativo delle librerie Microchip, implementa la classe di dispositivo **Human Interface Device, HID**; nonostante la classe HID venga impiegata appunto per dispositivi di interfaccia uomo macchina come mouse e tastiere, è sufficientemente flessibile da permettere lo scambio di dati di tipo generico senza richiedere la scrittura di un driver proprietario. La comunicazione avviene attraverso due **interrupt endpoint**, rispettivamente di tipo IN e OUT, che nel firmware vengono assegnati a due array da 64 byte ciascuno.

Tutti i device USB vengono identificati da due valori numerici univoci detti **Vendor ID (VID)** e **Product ID (PID)**. Il VID è diverso per ciascun produttore di device USB e viene rilasciato dietro pagamento ai soli membri della *USB Alliance*, mentre il PID identifica i diversi device dello stesso produttore. Nel nostro progetto manteniamo i valori assegnati nel codice demo della Microchip, in particolare VID = $(0 \times 04D8)_H$ e PID = $(0 \times 003F)_H$.

Una volta inizializzato il sistema, la funzione *ProcessIO()* viene chiamata con una certa regolarità. Essa legge i dati ricevuti dal PC nel buffer dei dati in ingresso e decide quale funzione chiamare a seconda del comando ricevuto. Allo stato delle cose il codice implementa dei semplici comandi con un protocollo di comunicazione elementare:

- **Selezione di una o più uscite dei DAC e scrittura di un valore compreso tra 0 e 4095 ($(0 \times 0FFF)_H$ 12 bit)**;
  questo comando è identificato dal valore esadecimale $(0 \times 10)_H$ come primo dato ricevuto nel buffer di ingresso. Il secondo e il terzo byte ricevuti possono assumere solo valori compresi tra 1 e 5, e identificano rispettivamente i DAC dal primo al quarto, oppure tutti, e le rispettive uscite dalla prima alla quarta, oppure tutte. Il quarto e il quinto byte contengono i 12 bit da inviare al DAC selezionato, formattati opportunamente (fig.62).

- **Selezione di una delle uscite del MUX I2C e lettura dei registri del sensore di colore corrispondente**;
  questo comando è identificato dal valore esadecimale $(0 \times 11)_H$ come primo dato ricevuto nel buffer di ingresso. Il secondo byte ricevuto può assumere solo valori compresi tra 1 e 4, e identifica le uscite del MUX I2C dalla prima alla quarta, e quindi il sensore di colore che vi è collegato (fig.62).

  Una volta completata la lettura del sensore, il firmware invia l'eco del comando seguito dai dati relativi ai canali di luce bianca, rossa, verde e blu tramite il buffer di uscita dell'USB (fig.62).

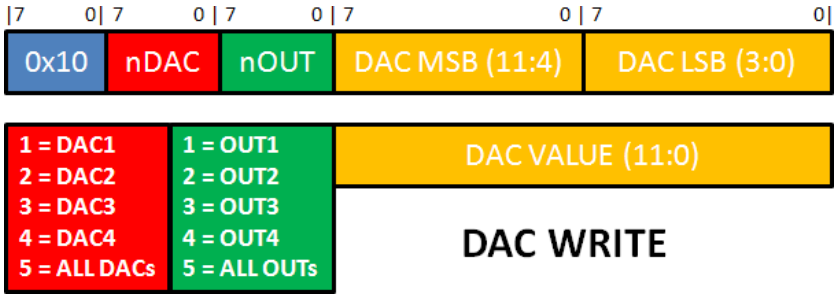


Figura 62: Struttura del comando di scrittura del DAC implementato nel firmware del microcontrollore.

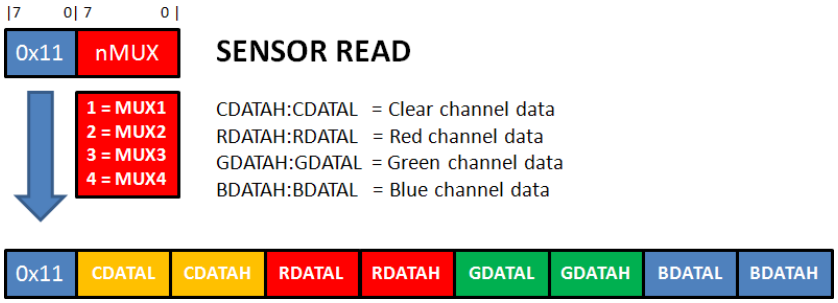


Figura 63: Struttura del comando di lettura dei sensori di colore implementato nel firmware del microcontrollore.

3.2 CODICE LABVIEW

3.2.1 Driver VISA

Come detto in precedenza, un dispositivo USB HID non necessita di un driver proprietario in quanto si tratta di una classe standard, implementata di default in tutti i sistemi operativi. Tuttavia, per poter comunicare con LabView è necessario associare un driver **VISA** al device HID, in modo che LabView possa accedere agli endpoint del device e quindi utilizzarli per la comunicazione.

VISA (Virtual Instrument Software Architecture) è una interfaccia di programmazione standard (API) largamente utilizzata nell'ambito delle misure elettroniche e dell'automazione industriale che permette di comunicare con numerosi strumenti di misura di produttori anche diversi attraverso interfacce come USB, GPIB, Ethernet, VXI e altre ancora.

LabView ha previsto un'applicazione apposita per la creazione dei driver VISA, il cosiddetto *Driver Wizard* [7]. Una volta aperta l'applicazione ci viene chiesto per quale tipo di interfaccia vogliamo creare un driver (fig.64): nel nostro caso selezioniamo USB e procediamo.

La maschera successiva elenca tutti i dispositivi di tipo USB collegati al PC: scegliamo quello con i valori di VID e PID corrispondenti a quelli del firmware, come visibile in figura 65 (VID = $(0 \times 04D8)_H$ e PID = $(0 \times 003F)_H$).

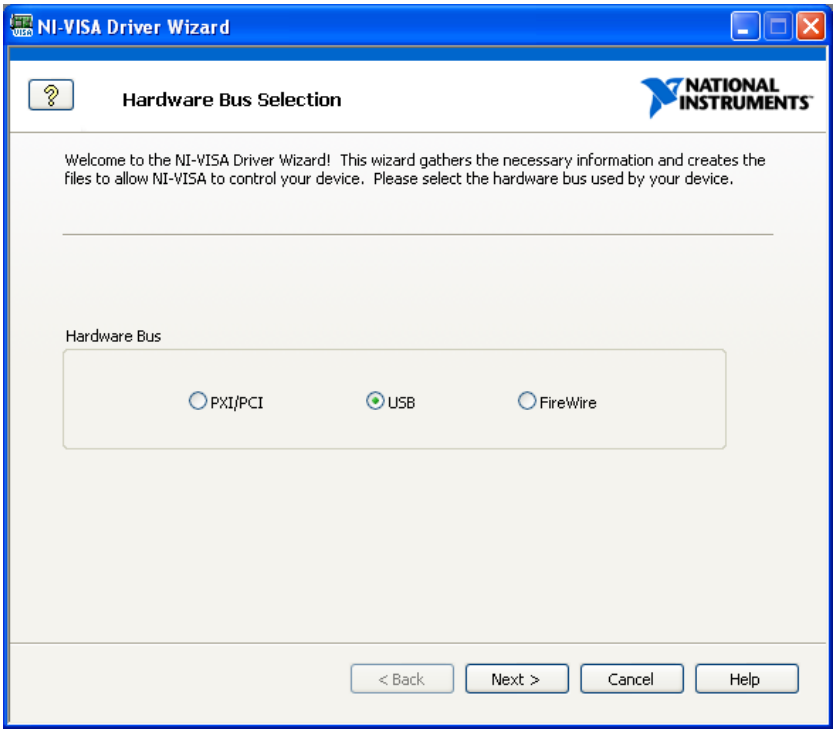


Figura 64: Creazione del driver VISA tramite l'applicazione *Driver Wizard* [7].

Una volta creato il driver, l'applicazione provvede a registrarlo nel sistema operativo. A questo punto, tramite il programma **National Instrument Measurement & Automation Explorer** sarà possibile individuare il dispositivo installato, visualizzarne le caratteristiche e, cosa più importante, associarvi una cosiddetta **VISA Resource** che ci permetterà di individuare univocamente il device all'interno dei programmi LabView. Nel nostro caso è stata creata una VISA Resource di nome **LID** (fig.66).

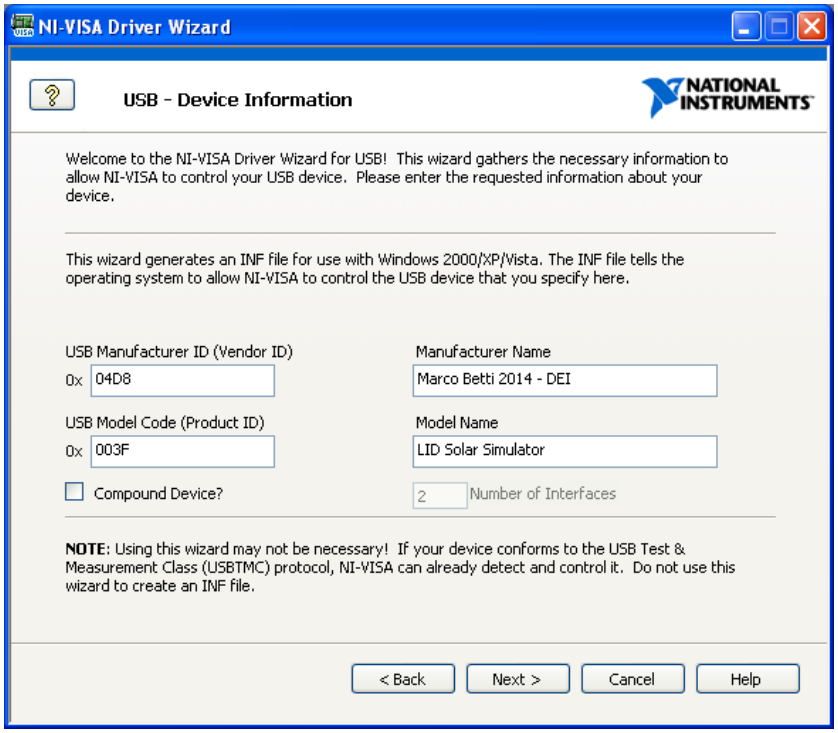


Figura 65: Creazione del driver VISA tramite l'applicazione *Driver Wizard* [7].

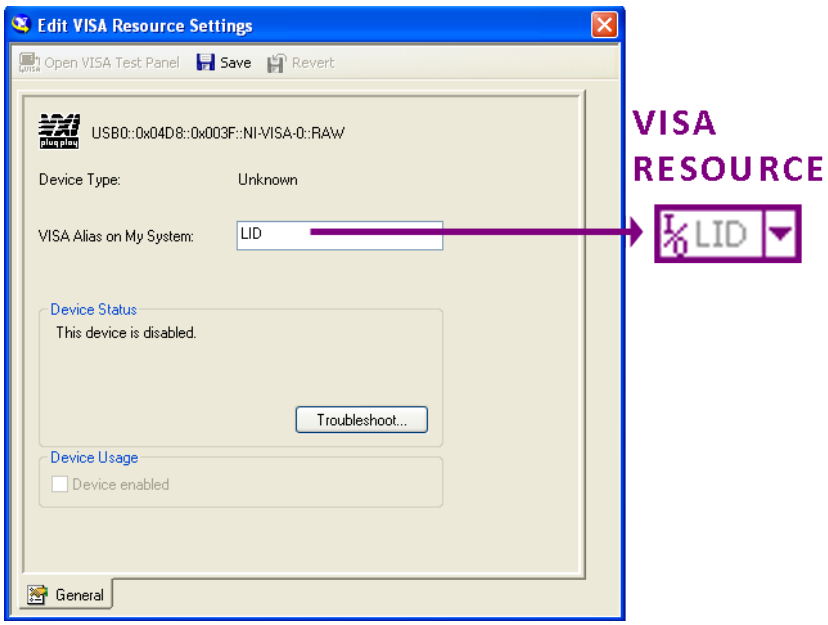


Figura 66: Assegnazione di un *alias* al driver VISA e corrispondente VISA Resource in LabView.

3.2.2 VI LabView

Dopo aver creato il driver VISA è stato possibile creare un **VI (Virtual Instrument)** in LabView tramite il quale comunicare con il microcontrollore dell'illuminatore LID.

Il *pannello frontale* del VI è visibile in figura 67. Si tratta di un primo VI elementare per verificare le funzionalità del sistema. Esso è diviso in vari pannelli:

PROPRIETÀ DEL DEVICE USB: vengono letti dal driver e visualizzati i valori di Vendor ID (VID), Product ID (PID), Manufacturer name, Product name, il numero seriale della risorsa VISA e alcune caratteristiche dei buffer di comunicazione;

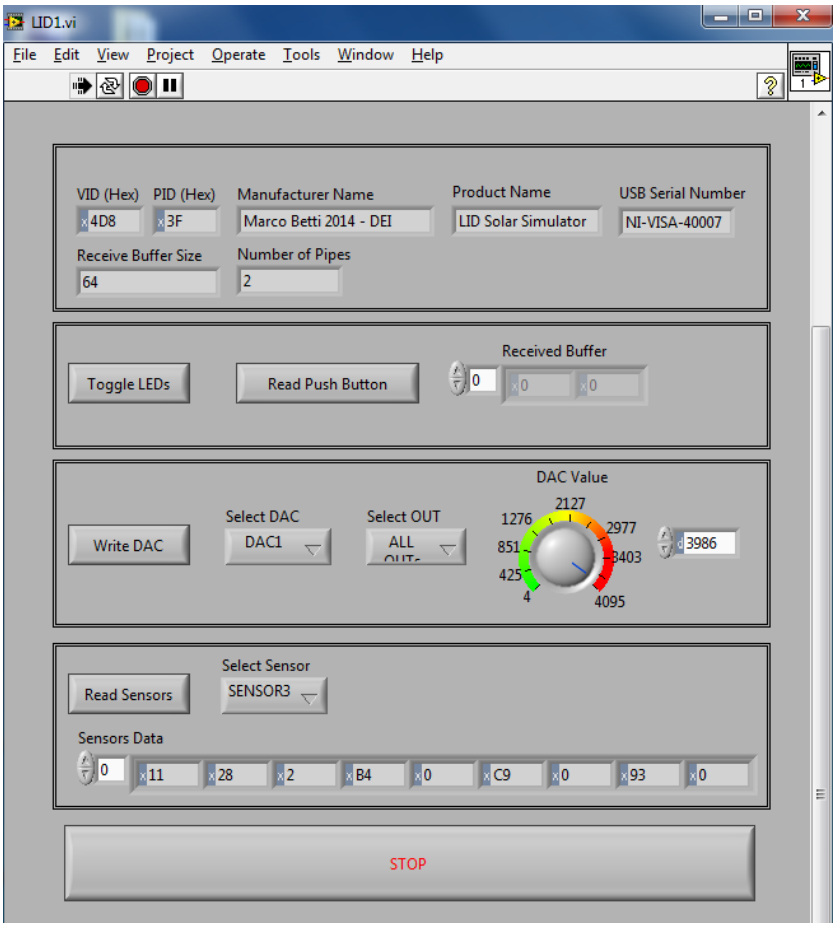


Figura 67: Pannello frontale del VI LabView utilizzato per comunicare con l'illuminatore LID.

COMUNICAZIONE CON L'INTERFACCIA HMI DELLA SCHEDA DI CONTROLLO: questa sezione implementa delle semplici routine per inviare e ricevere comandi dall'interfaccia HMI (LED, pulsanti) predisposta sulla scheda di controllo;

SELEZIONE E SCRITTURA DEI DAC PER IL CONTROLLO DELLA LUMINOSITÀ: in questa sezione sono stati predisposti dei menu a tendina con i valori preimpostati secondo il formato del comando di scrittura dei DAC (fig.62): tali menu permettono di selezionare una o più uscite di uno o più DAC, e inviare il valore numerico impostato tramite una manopola rotativa o inserito direttamente nella casella a fianco;

SELEZIONE E LETTURA DEI SENSORI DI COLORE: anche qui un menu con dei valori preimpostati (fig.63) permette di selezionare una delle quattro uscite del MUX I2C e di leggere il sensore di colore che vi è collegato. I dati che vengono letti sono due byte per ciascun canale di colore (bianco, rosso, verde e blu).

Tutto il VI è basato su i blocchetti messi a disposizione da LabView per la comunicazione VISA. I blocchetti utilizzati sono visibili in figura 68, ciascuno con la relativa descrizione.

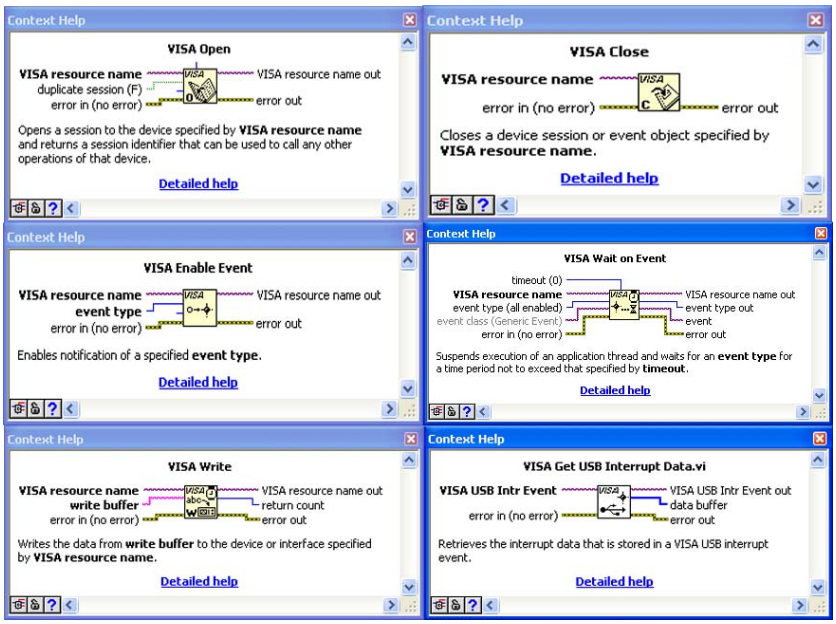


Figura 68: Blocchetti LabView per la comunicazione VISA e relativa descrizione.

Lo schema a blocchi del VI è mostrato in figura 71: a partire dalla VISA Resource denominata LID esce un filo che si collega al blocchetto *VISA Open* e a un blocco *USB Raw* che restituisce le proprietà della risorsa desiderata. Un ciclo while mantiene il VI attivo finché non viene premuto il pulsante STOP. All'interno del ciclo while sono presenti delle costanti numeriche esadecimali che vengono passate come parametro ad una struttura di tipo *case* più interna a seconda che vengano premuti o meno dei pulsanti sul pannello frontale.

Tali costanti rappresentano un determinato comando implementato nel firmware del microcontrollore. Ad esempio, se viene premuto il

pulsante *Write DAC* viene passato il valore $(\texttt{0} \times \texttt{10})_{\texttt{H}}$, altrimenti passa il valore di default 0.

La struttura *case* contiene tutti i casi possibili al suo interno, uno per ciascun valore del comando inviato. Le figure 69 e 70 mostrano rispettivamente il contenuto della struttura *case* relativo alla scrittura dei DAC ($(\texttt{0} \times \texttt{10})_{\texttt{H}}$) e alla lettura dei sensori di colore ($(\texttt{0} \times \texttt{11})_{\texttt{H}}$).

Per quanto riguarda la scrittura dei DAC (fig.69) il valore del comando viene concatenato in un array numerico insieme ai valori dei menu a tendina relativi alla selezione del DAC e delle uscite. Il valore del DAC viene trattato come un intero a 16 bit in LabView: per separarlo in due byte vengono mascherati solo 12 dei 16 bit utilizzati per poi essere divisi per 16 (si ricordi che la base è esadecimale).

L'array così creato, rispettando la struttura di fig.62, viene convertito in una stringa e passato al blocchetto *VISA Write*.

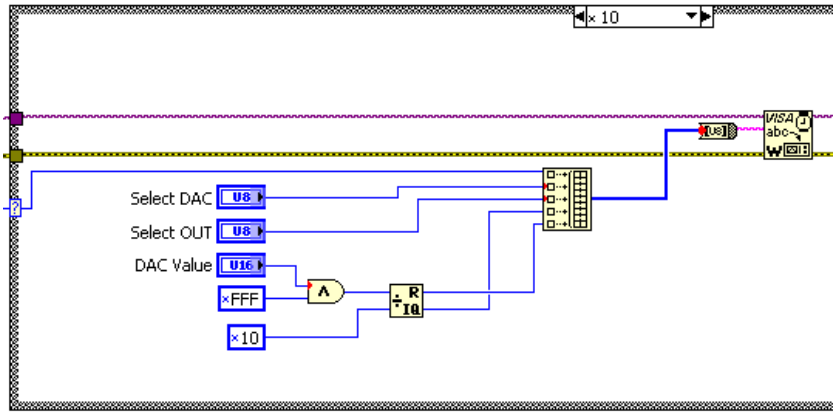


Figura 69: Schema a blocchi per la scrittura dei DAC tramite VISA.

La lettura dei sensori di colore è leggermente più laboriosa: in questo caso è necessario abilitare la risorsa VISA alla ricezione di dati in risposta ad un interrupt USB; per fare ciò si passa la risorsa VISA insieme ad una costante che identifica l'evento da abilitare (USB Interrupt) al blocchetto *VISA Enable Event* (fig.70).

Il codice esadecimale del comando viene concatenato in un array numerico insieme al valore del menu che permette di selezionare l'uscita del MUX I2C (e quindi il sensore desiderato) (fig.63), array che a sua volta viene convertito in stringa e passato al blocchetto *VISA Write*.

A questo punto si abilita un timeout entro il quale il device deve rispondere, pena la generazione di un errore da parte dell'API VISA. Il valore del timer espresso in millisecondi viene passato al blocchetto *VISA Wait on Event*.

Se la lettura va a buon fine entro il tempo impostato è possibile estrarre i dati ricevuti nel buffer USB tramite il blocchetto *VISA Get USB Interrupt Data* sottoforma di array di interi.

A partire da queste semplici routine che implementano una comunicazione di base è possibile creare dei nuovi VI da inserire in algoritmi di controllo della luminosità più avanzati, nonché creare dei siste-

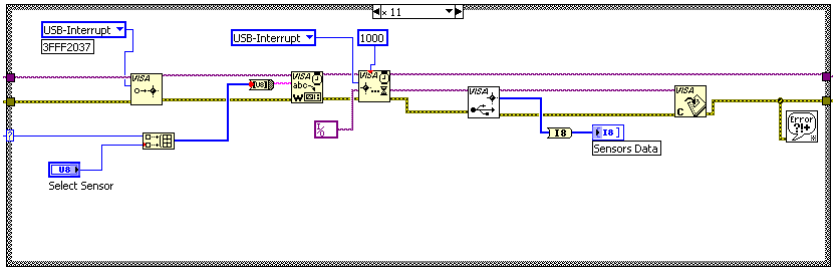


Figura 70: Schema a blocchi per la lettura dei sensori di colore tramite VISA.

mi di misura per celle solari totalmente automatizzati, che prevedono l'interazione con altri strumenti di misura e caratterizzazione tramite VISA.

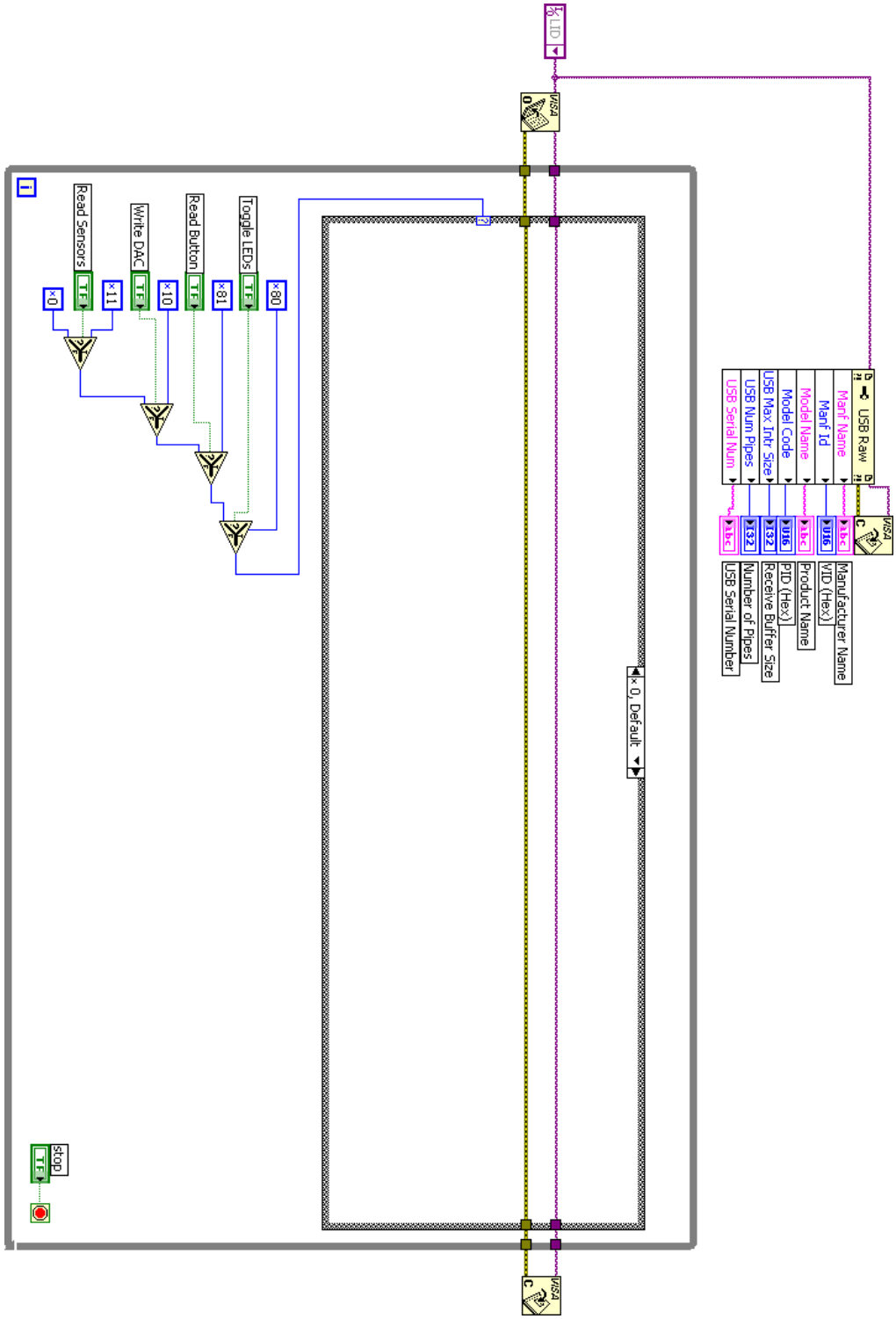


Figura 71: Diagramma a blocchi del VI LabView utilizzato per comunicare con l'illuminatore LID.

3.3 CODICI SORGENTE

Codice 1: main.c

```
/********************************************************************/
 FileName:        main.c
/********************************************************************/

#ifndef MAIN_C
#define MAIN_C

/************************ INCLUDES ******************************/
#include "./USB/usb.h"
#include "HardwareProfile.h"
#include "./USB/usb_function_hid.h"
#include "DAC_MAX5815.h"
#include "MUX_PCA9544.h"
#include "TCS34725.h"

// Configuration Words
_CONFIG1(JTAGEN_OFF & GCP_OFF & GWRP_OFF & FWDTEN_OFF & ICS_PGx2)
_CONFIG2(PLL_96MHZ_ON & IESO_OFF & FCKSM_CSDCMD & OSCIOFNC_OFF
            & POSCMOD_HS & FNOSC_PRIPLL & PLLDIV_DIV4 & IOL1WAY_ON)

/************************ VARIABLES ******************************/

#define RX_DATA_BUFFER_ADDRESS
#define TX_DATA_BUFFER_ADDRESS

unsigned char ReceivedDataBuffer[64] RX_DATA_BUFFER_ADDRESS;
unsigned char ToSendDataBuffer[64] TX_DATA_BUFFER_ADDRESS;

//USB handles.  Must be initialized to 0 at startup.
USB_HANDLE USBOutHandle = 0;
USB_HANDLE USBInHandle = 0;
BOOL blinkStatusValid = TRUE;

// Variables for DACs and I2C MUX
unsigned char nDac, nOut, nMux;
unsigned char TCS34725Val[8];
unsigned char *rdptr;

unsigned int i = 0;



/******************* PRIVATE PROTOTYPES ***************************/
void BlinkUSBStatus(void);
static void InitializeSystem(void);
void ProcessIO(void);
void UserInit(void);
void USBCBSendResume(void);

/********************************************************************
 * Function:        void main(void)
 ********************************************************************/
int main(void)
{
    InitializeSystem();

    #if defined(USB_INTERRUPT)
        USBDeviceAttach();
    #endif

    while(1)
    {
        #if defined(USB_POLLING)
                // Check bus status and service USB interrupts.
                // Interrupt or polling method. If using polling,
                // must call this function periodically.
                // This function will take care of processing
```

```
68                  // and responding to SETUP transactions (such as
69                  // during the enumeration process when you first
70                  // plug in). USB hosts require that USB devices
71                  // should accept and process SETUP packets in a
72                  // timely fashion. Therefore, when using polling,
73                  // this function should be called regularly
74                  // (such as once every 1.8ms or faster** [see
75                  // inline code comments in usb_device.c for
76                  // explanation when "or faster" applies]).
77                  // In most cases, the USBDeviceTasks() function
78                  // does not take very long to execute (ex: <100
79              // instruction cycles) before it returns.
80          USBDeviceTasks();
81
82          #endif
83
84                  // Application-specific tasks.
85                  // Application related code may be added here,
86                  // or in the ProcessIO() function.
87          ProcessIO();
88      }
89  }
90
91
92  /********************************************************************
93   * Function:        static void InitializeSystem(void)
94   *
95   * Overview:        InitializeSystem is a centralize initialization
96   *                  routine. All required USB initialization routines
97   *                  are called from here.
98   *
99   *                  User application initialization routine should
100  *                  also be called from here.
101  *******************************************************************/
102 static void InitializeSystem(void)
103 {
104 //      The USB specifications require that USB peripheral devices
105 //      must never source current onto the Vbus pin.  Additionally,
106 //    USB peripherals should not source current on D+ or D- when
107 //      the host/hub is not actively powering the Vbus line.
108 //      When designing a self powered (as opposed to bus powered)
109 //    USB peripheral device, the firmware should make sure not to
110 //      turn on the USB module and D+ or D- pull up resistor unless
111 //      Vbus is actively powered. Therefore, the firmware needs some
112 //      means to detect when Vbus is being powered by the host.
113 //      A 5V tolerant I/O pin can be connected to Vbus (through a
114 //      resistor), and can be used to detect when Vbus is high (host
115 //      actively powering), or low (host is shut down or otherwise
116 //      not supplying power). The USB firmware can then periodically
117 //      poll this I/O pin to know when it is okay to turn on the USB
118 //      module/D+/D- pull up resistor. When designing a purely bus
119 //      powered peripheral device, it is not possible to source
120 //      current on D+ or D- when the host is not actively providing
121 //      power on Vbus. Therefore, implementing this bus sense feature
122 //      is optional. This firmware can be made to use this bus sense
123 //      feature by making sure "USE_USB_BUS_SENSE_IO" has been
124 //      defined in the HardwareProfile.h file.
125     #if defined(USE_USB_BUS_SENSE_IO)
126     tris_usb_bus_sense = INPUT_PIN; // See HardwareProfile.h
127     #endif
128
129 //      If the host PC sends a GetStatus (device) request,
130 //      the firmware must respond and let the host know if the USB
131 //      peripheral device is currently bus powered or self powered.
132 //      See chapter 9 in the official USB specifications for details
133 //      regarding this request.  If the peripheral device is capable
134 //      of being both self and bus powered, it should not return
135 //      a hard coded value for this request. Instead, firmware should
136 //      check if it is currently self or bus powered, and respond
137 //      accordingly. An I/O pin can be polled to determine the
138 //      currently selected power source. If using this feature, make
139 //      sure "USE_SELF_POWER_SENSE_IO" has been defined in
140 //      HardwareProfile.h, and that an appropriate I/O pin
141 //      has been mapped to it.
```

```
142     #if defined(USE_SELF_POWER_SENSE_IO)
143     tris_self_power = INPUT_PIN; // See HardwareProfile.h
144     #endif
145
146     UserInit();         // User initialization routines
147
148     USBDeviceInit();    // usb_device.c.
149                         //Initializes USB module SFRs and firmware
150                         //variables to known states.
151 }
152
153
154
155 /********************************************************************/
156  * Function:        void UserInit(void)
157  *
158  * Overview:        This routine should take care of all the
159  *                  initialization that is required.
160  *
161  * Note:
162  *
163 /********************************************************************/
164 void UserInit(void)
165 {
166     //Initialize all of the LED pins
167     mInitAllLEDs();
168     //Initialize all of the push buttons
169     mInitAllSwitches();
170     // Initialize I2C modules
171     InitI2C();
172     // Initializzo all DAC's out to the max value
173     // to keep off the Recom LED drivers
174     WriteDac(DAC1,DAC_OUT_ALL,0x0FFF);
175     // Select a MUX out and inizialize the color sensor
176     SetMux(MUX_OUT1);
177     TCS34725Init();
178     SetMux(MUX_OUT2);
179     TCS34725Init();
180     SetMux(MUX_OUT3);
181     TCS34725Init();
182     SetMux(MUX_OUT4);
183     TCS34725Init();
184
185     //initialize the variable holding the handle for the last
186     // transmission
187     USBOutHandle = 0;
188     USBInHandle = 0;
189
190     blinkStatusValid = TRUE;
191 }
192
193 /********************************************************************
194  * Function:        void ProcessIO(void)
195  *
196  * Overview:        This function is a place holder for other user
197  *                  routines. It is a mixture of both USB and
198  *                  non-USB tasks.
199  ********************************************************************/
200 void ProcessIO(void)
201 {
202     //Blink the LEDs according to the USB device status
203     if(blinkStatusValid)
204     {
205         BlinkUSBStatus();
206     }
207
208     // User Application USB tasks
209     if((USBDeviceState < CONFIGURED_STATE)||(USBSuspendControl==1)) return;
210
211     //Check if we have received an OUT data packet from the host
212     if(!HIDRxHandleBusy(USBOutHandle))
213     {
214         // We just received a packet of data from the USB host.
215         // Check the first byte of the packet to see what command the host
```

```
216             // application software wants us to fulfill.
217             // Look at the data the host sent, to see what kind of
218             // application specific command it sent.
219             switch(ReceivedDataBuffer[0])
220             {
221                 case 0x80:  //Toggle LEDs command
222                         // Stop blinking the LEDs automatically,
223                         // going to manually control them now.
224                             blinkStatusValid = FALSE;
225                     if(mGetLED_1() == mGetLED_2())
226                     {
227                         mLED_1_Toggle();
228                         mLED_2_Toggle();
229                     }
230                     else
231                     {
232                         if(mGetLED_1())
233                         {
234                             mLED_2_On();
235                         }
236                         else
237                         {
238                             mLED_2_Off();
239                         }
240                     }
241                     break;
242                 case 0x10:  //Write DAC
243                             switch(ReceivedDataBuffer[1])
244                         {
245                             case 1:
246                                 nDac = DAC1;
247                                 break;
248                             case 2:
249                                 nDac = DAC2;
250                                 break;
251                             case 3:
252                                 nDac = DAC3;
253                                 break;
254                             case 4:
255                                 nDac = DAC4;
256                                 break;
257                             case 5:
258                                 nDac = MAX5815_BROADCAST_ADD;
259                                 break;
260                         }
261                         switch(ReceivedDataBuffer[2])
262                         {
263                             case 1:
264                                 nOut = DAC_OUT1;
265                                 break;
266                             case 2:
267                                 nOut = DAC_OUT2;
268                                 break;
269                             case 3:
270                                 nOut = DAC_OUT3;
271                                 break;
272                             case 4:
273                                 nOut = DAC_OUT4;
274                                 break;
275                             case 5:
276                                 nOut = DAC_OUT_ALL;
277                                 break;
278                             }
279                             WriteDac(nDac,nOut,((ReceivedDataBuffer[3] << 4)
280                                     | ReceivedDataBuffer[4]));
281                     break;
282                 case 0x11:  //Read Sensors
283                     switch(ReceivedDataBuffer[1])
284                     {
285                         case 1:
286                             nMux = MUX_OUT1;
287                             break;
288                         case 2:
289                             nMux = MUX_OUT2;
```

```
290                         break;
291                     case 3:
292                         nMux = MUX_OUT3;
293                         break;
294                     case 4:
295                         nMux = MUX_OUT4;
296                         break;
297                 }
298 
299                 SetMux(nMux);
300                 rdptr = TCS34725Val;
301                 TCS34725Read(rdptr);
302 
303                 //Check to make sure the endpoint/buffer is free
304                 //before we modify the contents
305                 if(!HIDTxHandleBusy(USBInHandle))
306                 {
307                     //Echo back to the host the command
308                     ToSendDataBuffer[0] = 0x11;
309                     ToSendDataBuffer[1] = TCS34725Val[0];
310                     ToSendDataBuffer[2] = TCS34725Val[1];
311                     ToSendDataBuffer[3] = TCS34725Val[2];
312                     ToSendDataBuffer[4] = TCS34725Val[3];
313                     ToSendDataBuffer[5] = TCS34725Val[4];
314                     ToSendDataBuffer[6] = TCS34725Val[5];
315                     ToSendDataBuffer[7] = TCS34725Val[6];
316                     ToSendDataBuffer[8] = TCS34725Val[7];
317 
318                     //Prepare the USB module to send the data packet to the host
319                     USBInHandle = HIDTxPacket(HID_EP,(BYTE*)&ToSendDataBuffer[0],64);
320                 }
321                 break;
322             case 0x81:  //Get push button state
323                 //Check to make sure the endpoint/buffer is free
324                 // before we modify the contents
325                 // Long wait
326                 for(i=0;i<0xFFFF;i++)
327                     Nop();
328 
329                 if(!HIDTxHandleBusy(USBInHandle))
330                 {
331                     //Echo back to the host PC the command
332                     ToSendDataBuffer[0] = 0x81;
333                         if(sw3 == 1) //pushbutton not pressed (pull-up resistor)
334                         {
335                             ToSendDataBuffer[1] = 0x01;
336                         }
337                         else        //sw3 must be == 0, pushbutton is pressed
338                         {
339                             ToSendDataBuffer[1] = 0x00;
340                         }
341                         //Prepare the USB module to send the data packet to the host
342                         USBInHandle = HIDTxPacket(HID_EP,(BYTE*)&ToSendDataBuffer[0],64);
343                 }
344                 break;
345                 break;
346         }
347         //Re-arm the OUT endpoint, so we can receive the next OUT data packet
348         //that the host may try to send us.
349         USBOutHandle = HIDRxPacket(HID_EP, (BYTE*)&ReceivedDataBuffer, 64);
350     }
351 }
352 
353 /********************************************************************
354  * Function:        void BlinkUSBStatus(void)
355  *
356  * Overview:        BlinkUSBStatus turns on and off LEDs
357  *                  corresponding to the USB device state.
358  *
359  * Note:            mLED macros can be found in HardwareProfile.h
360  *                  USBDeviceState is declared and updated in
361  *                  usb_device.c.
362  *******************************************************************/
363 void BlinkUSBStatus(void)
```

```
364 {
365     // No need to clear UIRbits.SOFIF to 0 here.
366     // Callback caller is already doing that.
367     static WORD led_count=0;
368 
369     if(led_count == 0)led_count = 100000U;
370     led_count--;
371 
372     #define mLED_Both_Off()         {mLED_1_Off();mLED_2_Off();}
373     #define mLED_Both_On()          {mLED_1_On();mLED_2_On();}
374     #define mLED_Only_1_On()        {mLED_1_On();mLED_2_Off();}
375     #define mLED_Only_2_On()        {mLED_1_Off();mLED_2_On();}
376 
377     if(USBSuspendControl == 1)
378     {
379         if(led_count==0)
380         {
381             mLED_1_Toggle();
382             if(mGetLED_1())
383             {
384                 mLED_2_On();
385             }
386             else
387             {
388                 mLED_2_Off();
389             }
390         }//end if
391     }
392     else
393     {
394         if(USBDeviceState == DETACHED_STATE)
395         {
396             mLED_Both_Off();
397         }
398         else if(USBDeviceState == ATTACHED_STATE)
399         {
400             mLED_Both_On();
401         }
402         else if(USBDeviceState == POWERED_STATE)
403         {
404             mLED_Only_1_On();
405         }
406         else if(USBDeviceState == DEFAULT_STATE)
407         {
408             mLED_Only_2_On();
409         }
410         else if(USBDeviceState == ADDRESS_STATE)
411         {
412             if(led_count == 0)
413             {
414                 mLED_1_Toggle();
415                 mLED_2_Off();
416             }//end if
417         }
418         else if(USBDeviceState == CONFIGURED_STATE)
419         {
420             if(led_count==0)
421             {
422                 mLED_1_Toggle();
423                 if(mGetLED_1())
424                 {
425                     mLED_2_Off();
426                 }
427                 else
428                 {
429                     mLED_2_On();
430                 }
431             }//end if
432         }
433     }
434 }
435 
436 #endif
```

Codice 2: usb_descriptors.c

```
/********************************************************************
 FileName:      usb_descriptors.c
 Dependencies:  See INCLUDES section
 Processor:             PIC18 or PIC24 USB Microcontrollers
********************************************************************
-------------------------------------------------------------------
Filling in the descriptor values in the usb_descriptors.c file:
-------------------------------------------------------------------

[Device Descriptors]
The device descriptor is defined as a USB_DEVICE_DESCRIPTOR type.
This type is defined in usb_ch9.h  Each entry into this structure
needs to be the correct length for the data type of the entry.

[Configuration Descriptors]
The configuration descriptor was changed in v2.x from a structure
to a BYTE array.  Given that the configuration is now a byte array
each byte of multi-byte fields must be listed individually.  This
means that for fields like the total size of the configuration where
the field is a 16-bit value "64,0," is the correct entry for a
configuration that is only 64 bytes long and not "64," which is one
too few bytes.

The configuration attribute must always have the _DEFAULT
definition at the minimum. Additional options can be ORed
to the _DEFAULT attribute. Available options are _SELF and _RWU.
These definitions are defined in the usb_device.h file. The
_SELF tells the USB host that this device is self-powered. The
_RWU tells the USB host that this device supports Remote Wakeup.

[Endpoint Descriptors]
Like the configuration descriptor, the endpoint descriptors were
changed in v2.x of the stack from a structure to a BYTE array.  As
endpoint descriptors also has a field that are multi-byte entities,
please be sure to specify both bytes of the field.  For example, for
the endpoint size an endpoint that is 64 bytes needs to have the size
defined as "64,0," instead of "64,"

Take the following example:
    // Endpoint Descriptor //
    0x07,                       //the size of this descriptor //
    USB_DESCRIPTOR_ENDPOINT,    //Endpoint Descriptor
    _EP02_IN,                   //EndpointAddress
    _INT,                       //Attributes
    0x08,0x00,                  //size (note: 2 bytes)
    0x02,                       //Interval

The first two parameters are self-explanatory. They specify the
length of this endpoint descriptor (7) and the descriptor type.
The next parameter identifies the endpoint, the definitions are
defined in usb_device.h and has the following naming
convention:
_EP<##>_<dir>
where ## is the endpoint number and dir is the direction of
transfer. The dir has the value of either 'OUT' or 'IN'.
The next parameter identifies the type of the endpoint. Available
options are _BULK, _INT, _ISO, and _CTRL. The _CTRL is not
typically used because the default control transfer endpoint is
not defined in the USB descriptors. When _ISO option is used,
addition options can be ORed to _ISO. Example:
_ISO|_AD|_FE
This describes the endpoint as an isochronous pipe with adaptive
and feedback attributes. See usb_device.h and the USB
specification for details. The next parameter defines the size of
the endpoint. The last parameter in the polling interval.

-------------------------------------------------------------------
Adding a USB String
-------------------------------------------------------------------
A string descriptor array should have the following format:

rom struct{byte bLength;byte bDscType;word string[size];}sdxxx={
```

```
sizeof(sdxxx),DSC_STR,<text>};

The above structure provides a means for the C compiler to
calculate the length of string descriptor sdxxx, where xxx is the
index number. The first two bytes of the descriptor are descriptor
length and type. The rest <text> are string texts which must be
in the unicode format. The unicode format is achieved by declaring
each character as a word type. The whole text string is declared
as a word array with the number of characters equals to <size>.
<size> has to be manually counted and entered into the array
declaration. Let's study this through an example:
if the string is "USB", then the string descriptor should be:
(Using index 02)
rom struct{byte bLength;byte bDscType;word string[3];}sd002={
sizeof(sd002),DSC_STR,'U','S','B'};

A USB project may have multiple strings and the firmware supports
the management of multiple strings through a look-up table.
The look-up table is defined as:
rom const unsigned char *rom USB_SD_Ptr[]={&sd000,&sd001,&sd002};

The above declaration has 3 strings, sd000, sd001, and sd002.
Strings can be removed or added. sd000 is a specialized string
descriptor. It defines the language code, usually this is
US English (0x0409). The index of the string must match the index
position of the USB_SD_Ptr array, &sd000 must be in position
USB_SD_Ptr[0], &sd001 must be in position USB_SD_Ptr[1] and so on.
The look-up table USB_SD_Ptr is used by the get string handler
function.

---

The look-up table scheme also applies to the configuration
descriptor. A USB device may have multiple configuration
descriptors, i.e. CFG01, CFG02, etc. To add a configuration
descriptor, user must implement a structure similar to CFG01.
The next step is to add the configuration descriptor name, i.e.
cfg01, cfg02,.., to the look-up table USB_CD_Ptr. USB_CD_Ptr[0]
is a dummy place holder since configuration 0 is the un-configured
state according to the definition in the USB specification.

*******************************************************************/

/** DEFINITIONS ****************************************************/
 * Descriptor specific type definitions are defined in:
 * usb_device.h
 *
 * Configuration options are defined in:
 * usb_config.h
 *******************************************************************/
#ifndef __USB_DESCRIPTORS_C
#define __USB_DESCRIPTORS_C

/** INCLUDES *******************************************************/
#include "./USB/usb.h"
#include "./USB/usb_function_hid.h"

/** CONSTANTS ******************************************************/
#if defined(__18CXX)
#pragma romdata
#endif

/* Device Descriptor */
ROM USB_DEVICE_DESCRIPTOR device_dsc=
{
    0x12,                   // Size of this descriptor in bytes
    USB_DESCRIPTOR_DEVICE,  // DEVICE descriptor type
    0x0200,                 // USB Spec Release Number in BCD format
    0x00,                   // Class Code
    0x00,                   // Subclass code
    0x00,                   // Protocol code
    USB_EP0_BUFF_SIZE,      // Max packet size for EP0, see usb_config.h
    0x04D8,                 // Vendor ID
    0x003F,                 // Product ID: Custom HID device demo
```

```
583         0x0002,                 // Device release number in BCD format
584         0x01,                   // Manufacturer string index
585         0x02,                   // Product string index
586         0x00,                   // Device serial number string index
587         0x01                    // Number of possible configurations
588     };
589
590     /* Configuration 1 Descriptor */
591     ROM BYTE configDescriptor1[]={
592         /* Configuration Descriptor */
593         0x09,//sizeof(USB_CFG_DSC),    // Size of this descriptor in bytes
594         USB_DESCRIPTOR_CONFIGURATION,   // CONFIGURATION descriptor type
595         0x29,0x00,                      // Total length of data for this cfg
596         1,                              // Number of interfaces in this cfg
597         1,                              // Index value of this configuration
598         0,                              // Configuration string index
599         _DEFAULT | _SELF,               // Attributes, see usb_device.h
600         50,                             // Max power consumption (2X mA)
601
602         /* Interface Descriptor */
603         0x09,//sizeof(USB_INTF_DSC),   // Size of this descriptor in bytes
604         USB_DESCRIPTOR_INTERFACE,       // INTERFACE descriptor type
605         0,                              // Interface Number
606         0,                              // Alternate Setting Number
607         2,                              // Number of endpoints in this intf
608         HID_INTF,                       // Class code
609         0,                              // Subclass code
610         0,                              // Protocol code
611         0,                              // Interface string index
612
613         /* HID Class-Specific Descriptor */
614         0x09,//sizeof(USB_HID_DSC)+3,  // Size of this descriptor in bytes
615         DSC_HID,                        // HID descriptor type
616         0x11,0x01,                      // HID Spec Release Number in BCD format (1.11)
617         0x00,                           // Country Code (0x00 for Not supported)
618         HID_NUM_OF_DSC,                 // Number of class descriptors, see usbcfg.h
619         DSC_RPT,                        // Report descriptor type
620         HID_RPT01_SIZE,0x00,//sizeof(hid_rpt01),  // Size of the report descriptor
621
622         /* Endpoint Descriptor */
623         0x07,/*sizeof(USB_EP_DSC)*/
624         USB_DESCRIPTOR_ENDPOINT,        //Endpoint Descriptor
625         HID_EP | _EP_IN,                //EndpointAddress
626         _INTERRUPT,                     //Attributes
627         0x40,0x00,                      //size
628         0x01,                           //Interval
629
630         /* Endpoint Descriptor */
631         0x07,/*sizeof(USB_EP_DSC)*/
632         USB_DESCRIPTOR_ENDPOINT,       //Endpoint Descriptor
633         HID_EP | _EP_OUT,              //EndpointAddress
634         _INTERRUPT,                    //Attributes
635         0x40,0x00,                     //size
636         0x01                           //Interval
637     };
638
639     //Language code string descriptor
640     ROM struct{BYTE bLength;BYTE bDscType;WORD string[1];}sd000={
641     sizeof(sd000),USB_DESCRIPTOR_STRING,{0x0409
642     }};
643
644     //Manufacturer string descriptor
645     ROM struct{BYTE bLength;BYTE bDscType;WORD string[28];}sd001={
646     sizeof(sd001),USB_DESCRIPTOR_STRING,
647     {'M','a','r','c','o',' ','B','e','t','t','i',' ','2','0','1','4',
648     ' ','-',' ','D','E','I',' ','U','n','i','p','d'}};
649
650     //Product string descriptor
651     ROM struct{BYTE bLength;BYTE bDscType;WORD string[19];}sd002={
652     sizeof(sd002),USB_DESCRIPTOR_STRING,
653     {'L','I','D',' ','S','o','l','a','r',' ','S','i','m','u','l','a','t','o','r'}};
654
655     //Class specific descriptor - HID
656     ROM struct{BYTE report[HID_RPT01_SIZE];}hid_rpt01={
```

```
{
    0x06, 0x00, 0xFF,  // Usage Page = 0xFF00 (Vendor Defined Page 1)
    0x09, 0x01,        // Usage (Vendor Usage 1)
    0xA1, 0x01,        // Collection (Application)
    0x19, 0x01,        // Usage Minimum
    0x29, 0x40,        // Usage Maximum
                       // 64 input usages total (0x01 to 0x40)
    0x15, 0x01,        // Logical Minimum (data bytes in the report
                       // may have minimum value = 0x00)
    0x25, 0x40,        // Logical Maximum (data bytes in the report
                       // may have maximum value = 0x00FF = unsigned 255)
    0x75, 0x08,        // Report Size: 8-bit field size
    0x95, 0x40,        // Report Count: Make sixty-four 8-bit fields
                       // (the next time the parser hits an "Input",
                       // "Output", or "Feature" item)
    0x81, 0x00,        // Input (Data, Array, Abs): Instantiates input
                       // packet fields based on the above report size,
                       // count, logical min/max, and usage.
    0x19, 0x01,        // Usage Minimum
    0x29, 0x40,        // Usage Maximum
                       // 64 output usages total (0x01 to 0x40)
    0x91, 0x00,        // Output (Data, Array, Abs): Instantiates output
                       // packet fields.  Uses same report size and count
                       // as "Input" fields, since nothing new/different
                       // was specified to the parser since the "Input" item.
    0xC0}              // End Collection
};


//Array of configuration descriptors
ROM BYTE *ROM USB_CD_Ptr[]=
{
    (ROM BYTE *ROM)&configDescriptor1
};

//Array of string descriptors
ROM BYTE *ROM USB_SD_Ptr[]=
{
    (ROM BYTE *ROM)&sd000,
    (ROM BYTE *ROM)&sd001,
    (ROM BYTE *ROM)&sd002
};

#endif
```

Codice 3: HardwareProfile.h

```
/********************************************************************
 FileName:      HardwareProfile.h
 Dependencies:  See INCLUDES section
  *******************************************************************/

#ifndef HARDWARE_PROFILE_H
#define HARDWARE_PROFILE_H

    /*******************************************************************/
    /******** USB stack hardware selection options *********************/
    /*******************************************************************/
    //This section is the set of definitions required by the MCHPFSUSB
    //  framework.  These definitions tell the firmware what mode it is
    //  running in, and where it can find the results to some information
    //  that the stack needs.
    //These definitions are required by every application developed with
    //  this revision of the MCHPFSUSB framework.  Please review each
    //  option carefully and determine which options are desired/required
    //  for your application.

    //#define USE_SELF_POWER_SENSE_IO
    #define tris_self_power     TRISAbits.TRISA2    // Input
    #define self_power          1

    //#define USE_USB_BUS_SENSE_IO
```

```
726     #define tris_usb_bus_sense  TRISBbits.TRISB5    // Input
727     #define USB_BUS_SENSE       1
728
729     //Uncomment this to make the output HEX of this project
730     //   to be able to be bootloaded using the HID bootloader
731     //#define PROGRAMMABLE_WITH_USB_HID_BOOTLOADER
732
733     //If the application is going to be used with the HID bootloader
734     //  then this will provide a function for the application to
735     //  enter the bootloader from the application (optional)
736     #if defined(PROGRAMMABLE_WITH_USB_HID_BOOTLOADER)
737         #define EnterBootloader() __asm__("goto 0x400")
738     #endif
739
740     /*******************************************************************/
741     /******** Application specific definitions *************************/
742     /*******************************************************************/
743
744     #define CLOCK_FREQ 32000000
745
746     /** LED ************************************************************/
747     #define mInitAllLEDs()      LATG &= 0xFC3F; TRISG &= 0xFC3F;
748
749     #define mLED_1              LATGbits.LATG6
750     #define mLED_2              LATGbits.LATG7
751     #define mLED_3              LATGbits.LATG8
752     #define mLED_4              LATGbits.LATG9
753
754     #define mGetLED_1()         mLED_1
755     #define mGetLED_2()         mLED_2
756     #define mGetLED_3()         mLED_3
757     #define mGetLED_4()         mLED_4
758
759     #define mLED_1_On()         mLED_1 = 1;
760     #define mLED_2_On()         mLED_2 = 1;
761     #define mLED_3_On()         mLED_3 = 1;
762     #define mLED_4_On()         mLED_4 = 1;
763
764     #define mLED_1_Off()        mLED_1 = 0;
765     #define mLED_2_Off()        mLED_2 = 0;
766     #define mLED_3_Off()        mLED_3 = 0;
767     #define mLED_4_Off()        mLED_4 = 0;
768
769     #define mLED_1_Toggle()     mLED_1 = !mLED_1;
770     #define mLED_2_Toggle()     mLED_2 = !mLED_2;
771     #define mLED_3_Toggle()     mLED_3 = !mLED_3;
772     #define mLED_4_Toggle()     mLED_4 = !mLED_4;
773
774     /** SWITCH *********************************************************/
775     #define mInitSwitch2()      TRISBbits.TRISB0=1;
776     #define mInitSwitch3()      TRISBbits.TRISB1=1;
777     #define mInitAllSwitches()  mInitSwitch2();mInitSwitch3();AD1PCFGL |= 0x03;
778     #define sw2                 PORTBbits.RB0
779     #define sw3                 PORTBbits.RB1
780
781     #define USE_AND_OR
782
783 #endif  //HARDWARE_PROFILE_H
```

Codice 4: DAC_MAX5815.h

```
784
785 #ifndef DAC_MAX5815_H
786 #define DAC_MAX5815_H
787
788 // DAC I2C ADDRESS
789 #define MAX5185_BASE_ADD 0b00100000
790 #define DAC1 MAX5185_BASE_ADD
791 #define DAC2 MAX5185_BASE_ADD | 0b00000100  // DAC's Addresses
792 #define DAC3 MAX5185_BASE_ADD | 0b00000110
793 #define DAC4 MAX5185_BASE_ADD | 0b00010000
794
```

```
795 // MAX5815 broadcast address
796 #define MAX5815_BROADCAST_ADD 0b00010000
797
798 // DAC OUTPUT SELECTION MASK
799 #define DAC_OUT1 0b00000000
800 #define DAC_OUT2 0b00000001
801 #define DAC_OUT3 0b00000010
802 #define DAC_OUT4 0b00000011
803 #define DAC_OUT_ALL 0b00001000
804
805 // DAC's Commands
806
807 // DAC CODEn_LOADn
808 // Simultaneously writes data to the selected CODE register(s)
809 // while updating selected DAC register(s)
810 // |        BYTE 1        |           BYTE 2          |           BYTE 3
    |
811 // |00011 DAC SELECTION|CODE REGISTER DATA [11:4]|CODE REGISTER DATA [3:0]xxxx|
812 #define DAC_CODE_LOAD 0b00110000
813
814 // Functions Prototypes
815 void WriteDac(unsigned char nDac, unsigned char nOut, unsigned int val);
816
817 #endif // DAC_5815_H
```

Codice 5: MUX_PCA9544.hh

```
818
819 #ifndef MUX_PCA9544_H
820 #define MUX_PCA9544_H
821
822 // MUX I2C ADDRESS
823 #define PCA9544_BASE_ADD 0b11100000
824
825 // MUX OUTPUT SELECTION MASK
826 #define MUX_OUT1 0b00000100
827 #define MUX_OUT2 0b00000101
828 #define MUX_OUT3 0b00000110
829 #define MUX_OUT4 0b00000111
830
831 // Functions Prototypes
832 void SetMux(unsigned char nOut);
833
834 #endif // MUX_PCA9544_H
```

Codice 6: TCS34725.h

```
835
836 #ifndef TCS34725_H
837 #define TCS34725_H
838
839 // COLOR SENSOR I2C ADDRESS
840 // |0x29|R/W|
841 #define TCS34725_WRITE 0b01010010
842 #define TCS34725_READ 0b01010011
843
844 // TCS34725 COMMAND
845 // | 7 | 6  5 | 4 3 2 1 0|
846 // |CMD| TYPE |    ADDR    |
847 // CMD = 1
848 // TYPE = 00 ---> Repeated byte protocol transaction
849 //      = 01 ---> Auto increment protocol transaction
850 // ADDR = Address of the register to access
851 #define COMMAND_REPEATED_BYTE 0x80
852 #define COMMAND_AUTO_INCREMENT 0xA0
853
854 // TCS34725 REGISTER MAP
855 #define ENABLE_REG   0x00        // Enables states and interrupts
856 #define ATIME_REG    0x01        // RGBC time
857 #define WTIME_REG    0x03        // Wait time
```

```
858 #define AILTL_REG     0x04        // Clear interrupt low threshold low byte
859 #define AILTH_REG     0x05        // Clear interrupt low threshold high byte
860 #define AIHTL_REG     0x06        // Clear interrupt high threshold low byte
861 #define AIHTH_REG     0x07        // Clear interrupt high threshold high byte
862 #define PERS_REG      0x0C        // Interrupt persistence filter
863 #define CONFIG_REG    0x0D        // Configuration
864 #define CONTROL_REG   0x0D        // Control
865 #define ID_REG        0x12        // Device ID
866 #define STATUS_REG    0x13        // Device status
867 #define CDATAL_REG    0x14        // Clear data low byte
868 #define CDATAH_REG    0x15        // Clear data high byte
869 #define RDATAL_REG    0x16        // Red data low byte
870 #define RDATAH_REG    0x17        // Red data high byte
871 #define GDATAL_REG    0x18        // Green data low byte
872 #define GDATAH_REG    0x19        // Green data high byte
873 #define BDATAL_REG    0x1A        // Blue data low byte
874 #define BDATAH_REG    0x1B        // Blue data high byte
875
876 // CONFIG. CONSTANTS
877 // Access the ENABLE Register
878 // Interrupt and Wait cycle disabled, RGBC and oscillator enabled
879 // First of all set the PON bit
880 #define TCS34725_CMD_PON COMMAND_REPEATED_BYTE | ENABLE_REG
881 #define TCS34725_PON 0b00000001
882
883 // Set bit AEN
884 #define TCS34725_CMD_ON COMMAND_REPEATED_BYTE | ENABLE_REG
885 #define TCS34725_ON 0b00000011
886
887 // Set ATIME integration time
888 // ATIME = 256 - integration_time/2.4ms
889 // integration_time = 2.4ms x (256 - ATIME)
890 // ATIME = 0xC0 --> 64 INTEG_CYCLES, 154ms, MAX_COUNT = 65535
891 #define TCS34725_CMD_ATIME COMMAND_REPEATED_BYTE | ATIME_REG
892 #define TCS34725_ATIME 0xC0
893
894 // Set GAIN in CONTROL Register
895 // |RESERVED|GAIN|
896 // | xxxxxx | 10 |
897 // 00 --> 1x
898 // 01 --> 4x
899 // 10 --> 16x
900 // 11 --> 60x
901 #define TCS34725_GAIN COMMAND_REPEATED_BYTE | CONTROL_REG
902 #define TCS34725_GAIN_1 0x00
903 #define TCS34725_GAIN_4 0x01
904 #define TCS34725_GAIN_16 0x02
905 #define TCS34725_GAIN_60 0x03
906
907 // Auto increment registers read from CDATAL to BDATAH
908 #define TCS34725_READ_ADD COMMAND_AUTO_INCREMENT | CDATAL_REG
909
910
911 // Functions Prototypes
912 void TCS34725Init();
913 void TCS34725PowerOn();
914 void TCS34725EnableRGBC();
915 void TCS34725SetGain(unsigned char gain);
916 void TCS34725SetIntegTime(unsigned char intTime);
917
918 #endif // TCS34725_H
```

4 MISURE E CONCLUSIONI

INDICE

4.1 CARATTERIZZAZIONE DELL'ILLUMINATORE

4.1.1 Setup sperimentale

Dopo aver montato tutte le schede elettroniche che compongono il sistema di controllo e alimentazione dell'illuminatore LID, è stato allestito un setup di misura per la caratterizzazione delle prestazioni ottiche.

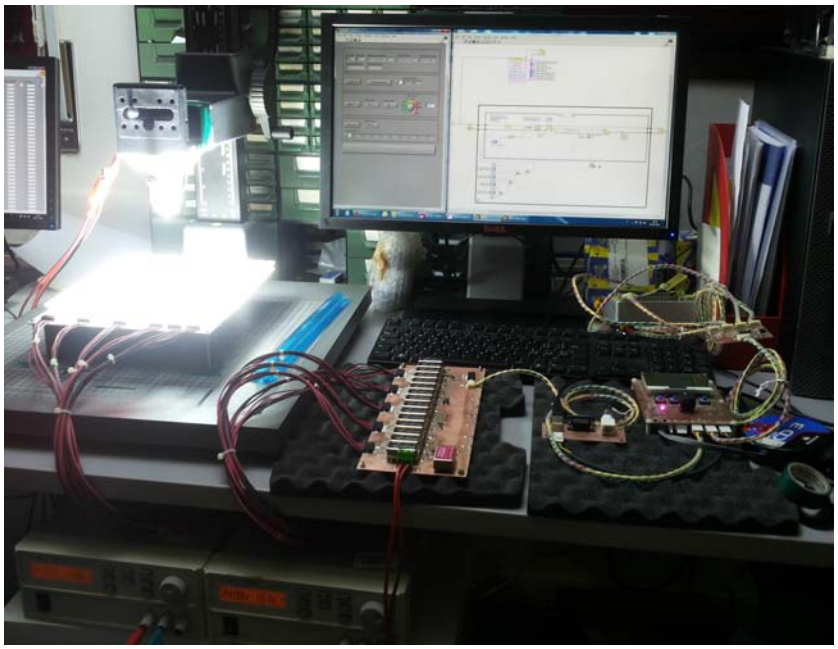


Figura 72: Setup di misura per la caratterizzazione dell'illuminatore.

Come si può vedere nella figura 72 è stata utilizzata una postazione con un PC sul quale far girare il VI LabView. Si vedono inoltre la scheda di controllo, collegata all'USB del PC, la scheda di potenza con

tutti i driver Recom montati, le schede con i sensori di luce e la scheda LED montata su uno *stativo* ad altezza variabile.

Il tutto è alimentato da due alimentatori **Agilent** ® collegati in serie, in modo da avere una tensione di 48 V ed una corrente massima di 3 A. A regime sarà necessario un alimentatore con una corrente di almeno 16 A, tuttavia i due alimentatori utilizzati sono stati sufficienti per effettuare la caratterizzazione ottica.

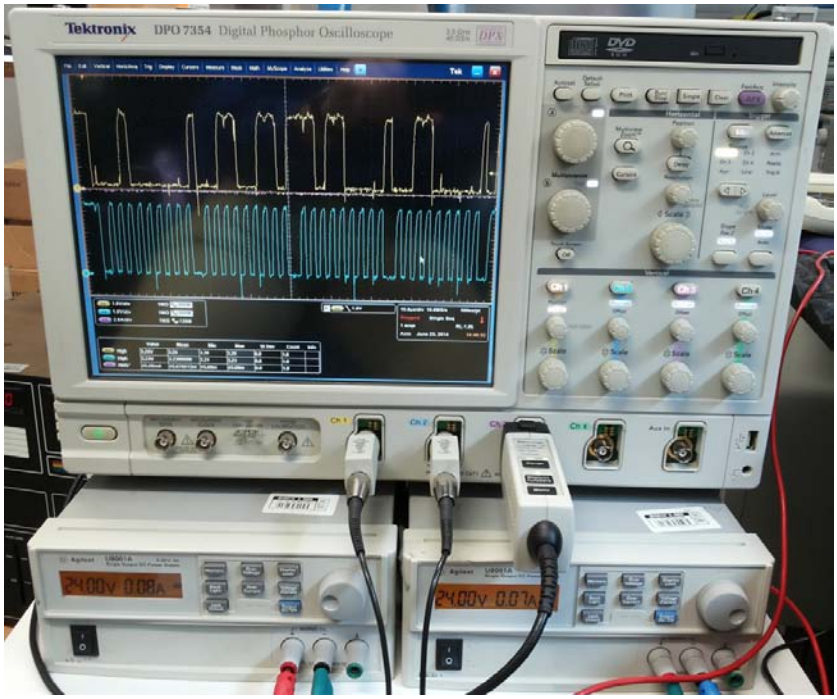


Figura 73: Oscilloscopio **Tektronix** ® DPO7354.

Il setup è completato da un potente oscilloscopio **Tektronix** ® DPO7354 con 4 canali, banda di 3,5 GHz e frequenza di campionamento massima pari a 40 Gsps, dotato di una sonda di corrente che permette di visualizzarne direttamente la forma d'onda sullo schermo (fig.73).

Prima di effettuare qualunque misura è stato necessario individuare, con l'aiuto della sonda di corrente, un valore dei DAC tale per cui la corrente nelle stringhe di LED fosse pari a 100 mA. Tale valore di corrente è stato preso come riferimento per le misure successive in quanto i risultati possono essere facilmente scalati per valori di corrente più elevati.

4.1.2 Misura di irradianza

Per effettuare la misura di irradianza è stato utilizzato un **piranometro**, ovvero uno strumento che fornisce un'indicazione della potenza ottica direttamente in W/m^2. Il modello utilizzato è il **LP PYRA 08** della **Delta Ohm** ®, le cui caratteristiche sono riportate nel datasheet del produttore [14].

Si tratta di un oggetto che, una volta alimentato con una tensione continua compresa tra 10 e 20 V, fornisce in uscita una tensione compresa tra 0 e 5 V proporzionale al valore di irradianza (5 V = 2000 W/m^2).

Il piranometro utilizzato è visibile nella figura 75. In questo caso, per questioni logistiche, la misura è stata condotta con la sorgente luminosa rivolta verso l'alto e il piranometro rivolto verso il basso.

Una volta installato il piranometro sullo stativo sono stati registrati i valori di tensione ottenuti posizionando il rivelatore a diverse altezze rispetto al piano dell'illuminatore. Applicando un fattore moltiplicativo ai valori di tensione si ottengono i seguenti valori di irradianza, graficati in figura 74:

Tabella 1: Valori di irradianza in funzione della distanza dal piano dell'illuminatore.

Distanza [cm]	Tensione Piranometro [V]	Irradianza [W/m^2]
5 cm	1,691	676,4
6 cm	1,673	669,2
7 cm	1,645	658
8 cm	1,626	650,4
9 cm	1,603	641,2
10 cm	1,59	636
15 cm	1,525	610
20 cm	1,482	592,8
25 cm	1,42	568
30 cm	1,37	548

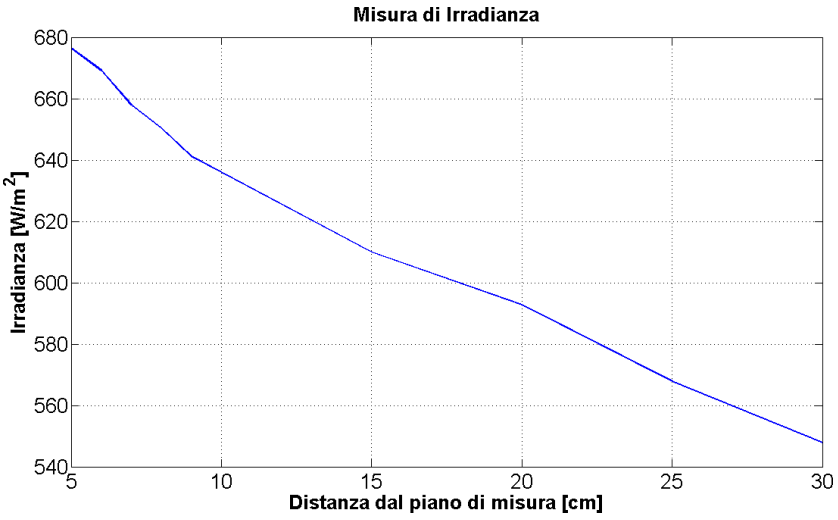


Figura 74: Irradianza in W/m^2 al variare della distanza dal piano dell'illuminatore.

Si può notare che *già a un decimo della corrente massima prevista per le stringhe di LED il valore di irradianza alla distanza di* 30 cm *è pari a* 548 W/m^2, *che corrisponde ad un valore di* 0,55 Sun.

Figura 75: Piranometro **Delta Ohm ® LP PYRA 08** [14].

4.1.3 Misura di uniformità

La misura di uniformità è stata eseguita con un **foto-radiometro HD2102.2** della **Delta Ohm ®** (77).

Anche questa volta il sensore è stato fissato allo stativo, ma invece che variare l'altezza, come nella misura precedente, viene variata la posizione in X e Y dell'illuminatore. Il sensore rimane fisso ad una distanza dalla sorgente pari a 30 cm.

A partire dalla posizione centrale l'illuminatore è stato spostato di 2 cm alla volta sia in X che in Y fino a coprire un quarto dell'intera zona coperta dalle ottiche. Gli altri tre quarti della sorgente emettono con la stessa simmetria.

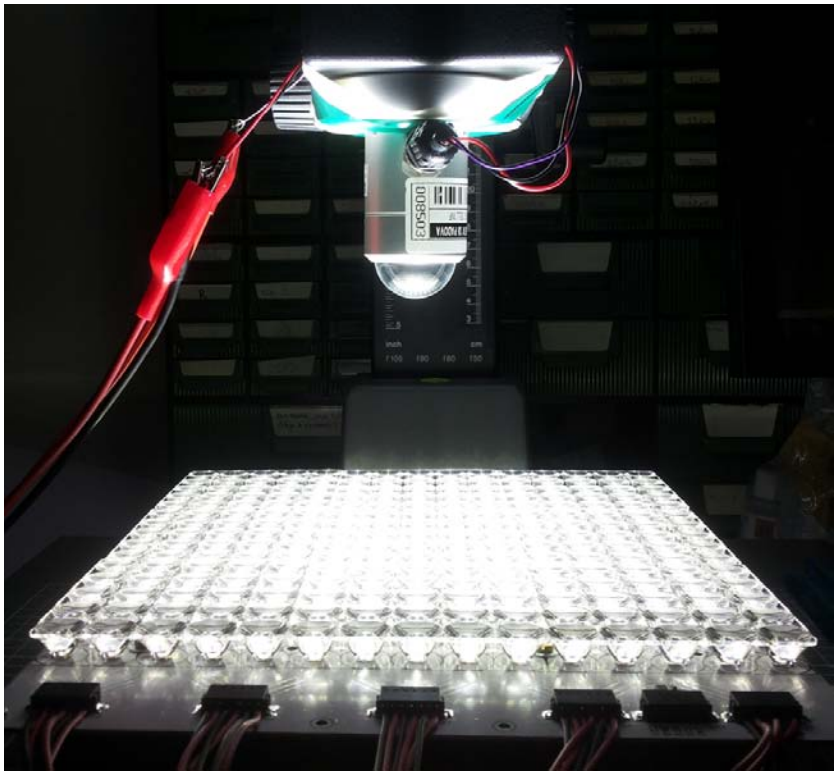


Figura 76: Illuminatore in funzione sotto il piranometro durante la caratterizzazione.

Il risultato della misura, dopo una breve elaborazione dei dati, è riportato nelle figure 78, 79 e 80, che mostrano rispettivamente i profili di irradianza lungo gli assi X ed Y, nonché la mappa di irradianza bidimensionale alla distanza di 30 cm. Si noti che i valori sono espressi in 10^4 LUX.

Il valore massimo ottenuto al centro dell'illuminatore alla distanza di 30 cm con una corrente dei LED pari a 100 mA è nell'ordine dei 160.000 LUX.

Figura 77: **Foto-radiometro HD2102.2** della **Delta Ohm** ®.

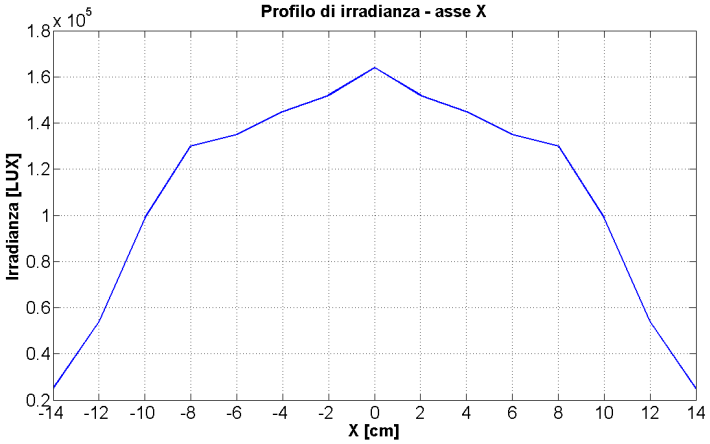


Figura 78: Profilo di irradianza lungo l'asse X dell'illuminatore.

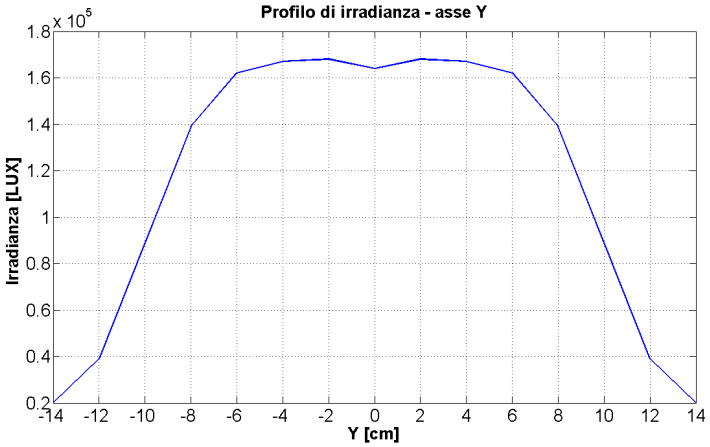


Figura 79: Profilo di irradianza lungo l'asse Y dell'illuminatore.

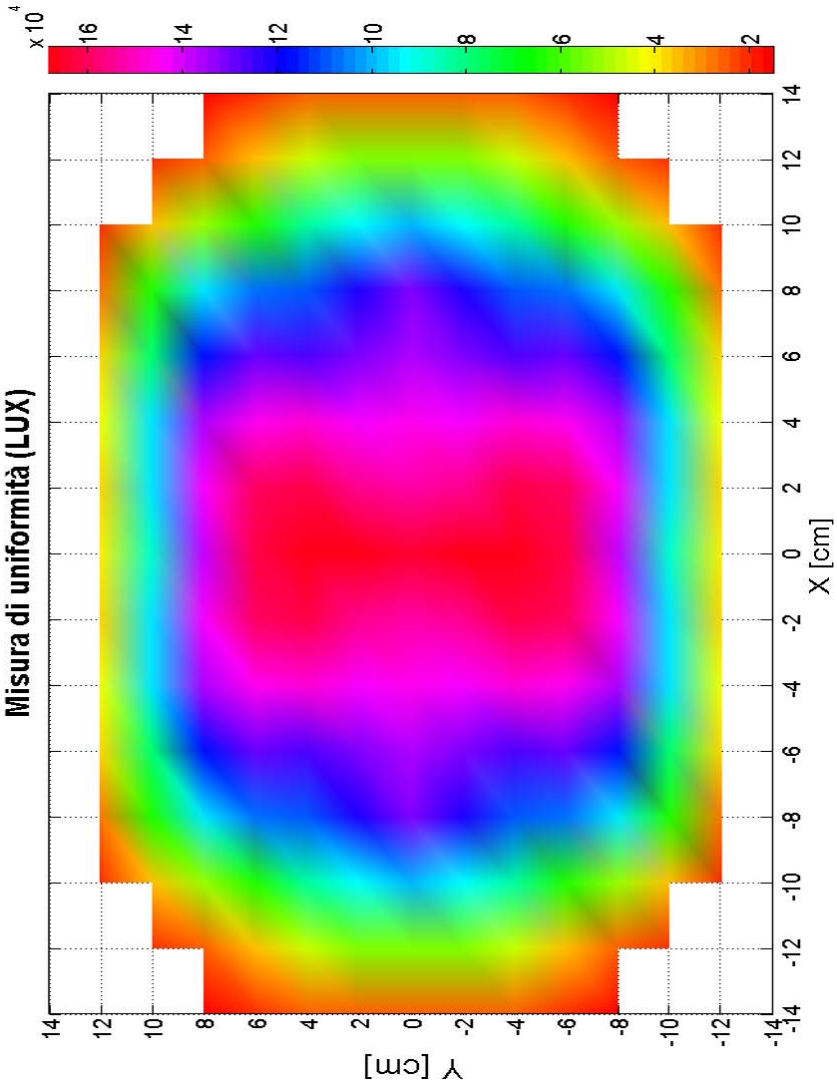


Figura 80: Mappa di irradianza dell'illuminatore alla distanza di 30 cm. I valori sulla scala colorata sono espressi in 10^4 LUX.

BIBLIOGRAFIA

[1] Bergquist. *Thermal Clad Selection Guide*. http://www.bergquistcompany.com/pdfs/techLibrary/tclad_2013_web_fullguide.pdf.

[2] Cree. *Thermal Management of Cree XLamp LEDs*. http://www.cree.com/xlamp_app_notes/thermal_management.

[3] Cree. *XLamp XB-D LEDs Product Family Data Sheet*. www.cree.com/xlamp_data_sheets/xbd.

[4] Displaytech. *64128M Series LCD Module Data Sheet*. https://www.displaytech-us.com/sites/default/files/display-data-sheet/64128Mseries.pdf.

[5] H. Fischer e W. Pschunder. «Investigation of photon and thermal induced changes in silicon solar cells». In: *Proc. 10th IEEE PVSC* 404 (1973).

[6] S.W. Glunz et al. «On the degradation of Cz-silicon solar cells». In: *Proc. 2nd WCPSEC* (1998).

[7] National Instruments. *USB Instrument Control Tutorial*. http://www.ni.com/white-paper/4478/en/pdf.

[8] Texas Instruments. *PCA9544A 4-Channel I2C And SMBus Multiplexer With Interrupt Logic*. http://www.ti.com/lit/gpn/pca9544a.

[9] Maxim Integrated. *MAX1797 Low-Supply Current Step-Up DC-DC Converter*. http://datasheets.maximintegrated.com/en/ds/MAX1795-MAX1797.pdf.

[10] Maxim Integrated. *MAX3222 Low-Power RS-232 Transceiver Data Sheet*. http://datasheets.maximintegrated.com/en/ds/MAX3222-MAX3241.pdf.

[11] Maxim Integrated. *MAX5815 Ultra-Small Quad-Channel 8-10-12-Bit Buffered Output DACs with Internal Reference and I2C Interface*. http://datasheets.maximintegrated.com/en/ds/MAX5813-MAX5815.pdf.

[12] Ledil. *Virpi XB-D Product Data Sheet*. http://www.ledil.com/luopdf3.php?s=54&t=7471.

[13] Austria Microsystems. *TCS34725 Color light-to-digital converter Vdd I2C and IR filter*. http://www.ams.com/eng/content/download/319364/1117183/file/TCS3472_Datasheet_EN_v1.pdf.

[14] Delta Ohm. *Piranometro LP PYRA 08 Data Sheet*. http://www.deltaohm.com/ver2012/download/LP_PYRA08_it.pdf.

[15] Recom Power. *RCD-48 Constant Current Buck LED Driver*. http://www.recom-power.com/pdf/Lightline/RCD-48.pdf.

[16] Traco Power. *TEN 8WI Series DC-DC Converters*. http://www.tracopower.com/products/ten8wi.pdf.

[17] S. Rein et al. «Electrical and thermal properties of the metastable defect in boron-doped Czochralski silicon». In: *Proc. 17th EU PVSEC* (2001).

[18] J.H. Reiss, R.R. King e K.W. Mitchell. «Characterization of diffusion length degradation in Czochralski silicon solar cells». In: *Applied Physics Letters* Volume:68, Issue: 23 (1996).

[19] J Schmidt e A. Cuevas. «Electronic properties of light-induced recombination centers in boron-doped Czochralski silicon». In: *J. Appl. Phys. 86, 3175* (1999).

[20] Jan Schmidt, Armin Gerhard Aberle e Rudolf Hezel. «Investigation of carrier lifetime instabilities in Cz-grown silicon». In: *Proc. 26th IEEE PVSC* (1997).

[21] Jan Schmidt, Karsten Bothe e Rudolf Hezel. «Formation and annihilation of the metastable defect in boron-doped Czochralski silicon». In: *Proc. 29th IEEE PVSC* (2002).

[22] Jan Schmidt, Karsten Bothe e Rudolf Hezel. «Structure and transformation of the metastable centre in Cz-silicon solar cells». In: *Proc. 3rd WCPSEC* (2003).

[23] Jan Schmidt e Rudolf Hezel. «Light-Induced Degradation in Cz Silicon solar cells: fundamental understanding and strategies for its avoidance». In: *12th Workshop on Crystalline Silicon Solar Cell Materials and Processes, Colorado* (2002).

[24] NXP Semiconductors. *I2C Bus Specification and User Manual - Rev. 6*. http://www.nxp.com/documents/user_manual/UM10204.pdf. Apr. 2014.

[25] Sitronix. *ST7565R 65 x 132 Dot Matrix LCD Controller/Driver*. https://www.displaytech-us.com/sites/default/files/driver-ic-data-sheet/ST7565R_1.pdf.

[26] Microchip Technology. *16-bit Language Tools Libraries*. http://www.microchip.com/downloads/en/DeviceDoc/16bit_Language_Tool_Libraries_51456c.pdf.

[27] Microchip Technology. *24LC652 64K I2C Serial EEPROM Data Sheet*. http://ww1.microchip.com/downloads/en/DeviceDoc/21073K.pdf.

[28] Microchip Technology. *MCP1252 Data Sheet*. http://ww1.microchip.com/downloads/en/DeviceDoc/21752B.pdf.

[29] Microchip Technology. *MCP9843/98243 DS - Memory Module Temperature Sensor Data Sheet*. http://ww1.microchip.com/downloads/en/DeviceDoc/22153c.pdf.

[30] Microchip Technology. *Microchib Libraries For Applications (MLA)*. http://www.microchip.com/pagehandler/en-us/devtools/mla/f.

[31] Microchip Technology. *PIC24F Reference Manual - Section 13: Parallel Master Port*. http://ww1.microchip.com/downloads/en/DeviceDoc/39713b.pdf.

[32] Microchip Technology. *PIC24F Reference Manual - Section 27: USB On-The-Go (OTG)*. http://ww1.microchip.com/downloads/en/DeviceDoc/39721b.pdff.

[33] Microchip Technology. *PIC24F Reference Manual - Section 6: Oscillator*. http://ww1.microchip.com/downloads/en/DeviceDoc/39700c.pdf.

[34] Microchip Technology. *PIC24FJ256GB110 Family Data Sheet*. http://ww1.microchip.com/downloads/en/DeviceDoc/39897c.pdf.