UNIVERSITÀ
POLITECNICA
DELLE MARCHE

UNIVERSITÀ POLITECNICA DELLE MARCHE

FACOLTÀ DI INGEGNERIA

Corso di laurea in Ingegneria Elettronica

Progettazione hardware e firmware di un nodo IoT per l'acquisizione dati da sensori e l'azionamento di attuatori da remoto a RF mediante il chip Wi-Fi ESP32-WROVER-B.

Hardware and firmware design of an IoT node for the acquisition of data from sensors and the RF remote actuators drive using the ESP32-WROVER-B Wi-Fi chip.

Relatore:

Prof. Claudio Turchetti

Correlatore:

Prof. Laura Falaschetti

LOCCIONI

Tesi di Laurea Triennale

Allegrezza Giuseppe

A.A. 2019-2020

INDICE

- INTRODUZIONE
- ESP32
  - STORIA
  - MOTIVO DELLA SCELTA
  - SPECIFICHE HARDWARE
  - ESP32 DEVKIT C
- SPECIFICHE DEL PROGETTO
  - SOFTWARE UTILIZZATI
  - PROTOTIPO GENERALE
  - PROTOTIPO TEMPERATURA
- IMPLEMENTAZIONE HARDWARE
  - HARDWARE ESP32 UTILIZZATO
  - PROTOTIPO TEMPERATURA
  - SCHEMI ELETTRICI E PCB
- IMPLEMENTAZIONE FIRMWARE
  - FUNZIONI UTILIZZATE
  - API
- IMPLEMENTAZIONE WEB
  - HTML E JAVASCRIPT
- RISULTATI E CONCLUSIONI
  - MIGLIORAMENTI POSSIBILI
  - STUDIO SU SCHEDE SIMILI
  - CONLUSIONI GENERALI

INTRODUZIONE

L’ obbiettivo principale della tesi, è stato la realizzazione di una scheda a microcontrollore basata su ESP32 per l’acquisizione di dati da sensori e l’invio di essi tramite Wi-Fi ad un PC o un'altra apparecchiatura per la successiva elaborazione. L’ idea è quella di affiancare la scheda a macchine industriali e di acquisire dati esterni ad essa, come la temperatura, il rumore, le vibrazioni, etc.

È poi possibile pilotare degli attuatori da remoto per il controllo di alcune cose, come la ventilazione o l’illuminazione.

Tutti i dati acquisiti dalla scheda devono essere accessibili in tempo reale da qualsiasi dispositivo connesso ad essa, e deve anche essere possibili visualizzare gli stati di allarme, resettarli e modificarli in base alle necessità.

ESP32

STORIA

L’ Espressif ESP32 è il degno erede del ben più famoso ESP8266, microcontrollore utilizzato moltissimo da makers e designers per la sua connettività Wi-Fi e le sue più che discrete capacità hardware dal 2014, quando l’azienda AI-Thinker ha prodotto il modulo ESP-01. All’ epoca purtroppo non era presente documentazione in inglese ma soltanto in cinese, essendo l’Espressif un’azienda situata a Shanghai, l’economicità e le capacità del microcontrollore però, attirarono molti appassionati che produssero software per un utilizzo più semplice e tradussero la documentazione cinese. Nel 2016 l’Espressif rilasciò il successore del 8266, l’ESP32 con la stessa idea di un modulo economico ma molto potente, con una ricca documentazione già tradotta.

MOTIVO DELLA SCELTA

Il motivo principale della scelta è stato la possibilità di avere un microcontrollore a buone prestazioni con connettività Wi-Fi integrata. Inoltre, il basso costo del MCU consente di abbassare notevolmente i possibili costi di produzione della scheda, le altre opzioni avevano costo simile ma prestazioni decisamente inferiori a parità di dimensioni e non avendo esigenze di periferiche ad alte prestazioni o precisione l'ESP32 è sembrata la scelta più logica.

SPECIFICHE HARDWARE

Il microcontrollore presenta un hardware di tutto rispetto:

- CPU: Xtensa dual-core LX6 a 32bit con frequenza di 160-240MHz e coprocessore ULP (ultra-low power)
- RAM: 520KB SRAM
- Wi-Fi: 802.11 b/g/n
- Bluetooth: v4.2 BR/EDR e BLE
- 12-bit SAR ADC up to 18 channels
- 2 × 8-bit DACs
- 10 × sensori touch (GPIO con sensing capacitivo)
- 4 × SPI
- 2 × I²S
- 2 × I²C
- 3 × UART
- SD/SDIO/CE-ATA/MMC/eMMC host controller
- SDIO/SPI slave controller
- Interfaccia ethernet MAC
- CAN bus 2.0
- Telecomando infrarosso (8 canali)
- PWM per il controllo di motori
- PWM per il controllo di LED

- Sensore a effetto Hall
- Pre-amplificatore a basso consumo

ESP32 DEVKIT-C

L' ESP32 DEVKIT-C è una scheda di sviluppo di fascia entry-level, che consente di sfruttare in modo semplice tutte le funzionalità del microcontrollore. Il DEVKIT-C utilizzato si basa su ESP32 WROVER-B, una delle varianti possibili dell'ESP32:

Module	Chip	Flash, MB	PSRAM, MB	Ant.	Dimensions, mm
ESP32-WROOM-32	ESP32-D0WDQ6	4	–	MIFA	18 × 25.5 × 3.1
ESP32-WROOM-32D	ESP32-D0WD	4, 8, or 16	–	MIFA	18 × 25.5 × 3.1
ESP32-WROOM-32U	ESP32-D0WD	4, 8, or 16	–	U.FL	18 × 19.2 × 3.1
ESP32-SOLO-1	ESP32-S0WD	4	–	MIFA	18 × 25.5 × 3.1
ESP32-WROVER (PCB)	ESP32-D0WDQ6	4	8	MIFA	18 × 31.4 × 3.3
ESP32-WROVER (IPEX)	ESP32-D0WDQ6	4	8	U.FL	18 × 31.4 × 3.3
ESP32-WROVER-B	ESP32-D0WD	4, 8, or 16	8	MIFA	18 × 31.4 × 3.3
ESP32-WROVER-IB	ESP32-D0WD	4, 8, or 16	8	U.FL	18 × 31.4 × 3.3

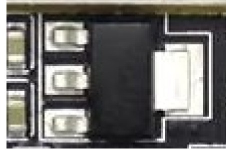
La differenza principale del WROVER-B rispetto al più comune WROOM-32 è l'aggiunta di 8MB di SPI PSRAM (pseudo-static-RAM), questo consente di avere a disposizione più RAM per applicazioni grafiche o che richiedono una buona dose di memoria.

La scheda si presenta in questa maniera:

Si può notare la presenza di diversi componenti:

Il primo è un convertitore UART-USB, utilizzato per la scrittura del firmware nella scheda o per quello che riguarda la comunicazione seriale fra scheda e PC, il chip in questione è un CP2102N prodotto dalla Silabs.

Si ha poi un regolatore di tensione:



Un AMS1117, utilizzato per convertire i 5V dell'alimentazione USB in 3.3V necessari al funzionamento della scheda.

Successivamente ci sono i due pulsanti EN e BOOT che possono essere utili nel caso non si utilizzi la porta MICRO-USB per il flash del firmware, da notare il fatto che il chip CP2102N resetta automaticamente la scheda e la mette in modalità flash nel caso si invii il firmware da scrivere.

Infine, si hanno due file di PIN utilizzabili per il collegamento con le periferiche esterne.

L' Espressif mette a disposizione lo schema elettrico nel caso si abbia bisogno di sapere come sono esattamente collegati i componenti.

Nella scheda i pin D0, D1, D2, D3, CMD e CLK sono utilizzati internamente per la comunicazione fra l'ESP32 e la memoria FLASH SPI ed è consigliato non utilizzarli nel proprio progetto a meno che non si è sicuri di disabilitare la flash esterna.

Inoltre, i pin GPIO16 e GPIO17 sono disponibili solo nelle schede basate su WROOM, nella WROVER sono utilizzati internamente e sono quindi riservati ed inutilizzabili.

La scheda consente di utilizzare diversi tipi di alimentazione, è possibile fornire 5V alla scheda utilizzando la porta micro-USB presente oppure collegando direttamente un alimentatore tra i pin 5V e GND, è anche possibile fornire direttamente i 3,3V necessari al micro attraverso i pin 3V3 e GND.

SPECIFICHE DEL PROGETTO

SOFTWARE UTILIZZATI

Durante lo svolgimento del progetto sono stati utilizzati diversi software:

ECLIPSE: per la realizzazione del firmware della scheda

ATOM: per la realizzazione della pagina WEB

KiCAD: per la realizzazione del circuito elettrico e del PCB

ECLIPSE:

Si è scelto di utilizzare Eclipse vista la necessità di un ambiente di sviluppo compatibile con il linguaggio C, gli sviluppatori dell'Espressif hanno creato un Plugin per interfacciare la scheda con Eclipse. La guida utilizzata per l' installazione è presente nel seguente link, onestamente la prima installazione ha dato del filo da torcere, ho avuto diversi problemi, primo fra tutto la necessità di una versione specifica di Java e di Python, è stato necessario disinstallare e reinstallare più volte sia Eclipse che il plugin per riuscire a far funzionare tutto correttamente.

ATOM: La scelta di ATOM è stata dettata dalla necessità di un software per lo sviluppo di pagine HTML e JAVASCRIPT, non essendo molto esperto in programmazione WEB, mi sono affidato alle buone recensioni del software online, mi sono comunque trovato molto bene e le varie features di formattazione automatica e di suggerimento delle funzioni mi ha aiutato molto.

KiCAD: Per la scelta di questo software ho chiesto direttamente agli ingegneri presenti nell' ufficio dove lavoravo, mi hanno consigliato KiCAD visto che è open source, e soprattutto perché loro lo hanno utilizzato per diverso tempo con buoni risultati. Nonostante fosse stata la mia prima esperienza con il software sono stato in grado di sviluppare uno schema elettrico ed un PCB nel giro di un paio di giorni, inoltre è possibile trovare online impronte e simboli della maggior parte dei componenti esistenti.

È stato utilizzato inoltre, anche il pacchetto office per la scrittura della documentazione e per il tracciamento di alcuni grafici.

PROTOTIPO GENERALE

Sono stati creati due prototipi con specifiche leggermente diverse, l'idea era creare una scheda il più modulare possibile in modo da poter essere utilizzata con un vasto numero di sensori, quindi il primo prototipo è realizzato su questa impronta, il secondo invece è un esempio applicativo ed incorpora alcune funzionalità aggiuntive.

Il prototipo generale ha il seguente funzionamento, per prima cosa si acquisiscono i valori dai vari sensori e si salvano in memoria, nel caso ci sia un dispositivo connesso alla scheda tramite Wi-Fi, si inviano in caso di richiesta i valori in modo da visualizzarli nella pagina web realizzata, ed è anche possibile modificare lo stato di uscite digitali e visualizzare invece lo stato degli ingressi, inoltre è possibile impostare diversi parametri come la risoluzione del ADC o i limiti entro i quali far scattare un allarme.

Le specifiche hardware del prototipo generale sono:

- 5 OUTPUT DIGITALI GPIO: 25 26 27 14 12
- 4 INPUT DIGITALI GPIO: 22 23 5 18
- 4 INPUT ANALOGICI GPIO:32 33 34 35
- INPUT PWM GPIO: 19
- UN OUTPUT PWM GPIO:15
- UN LED DI STATO GPIO:13
- UN LED DI ERRORE GPIO:21
- UN LED WIFI GPIO:4

Il numero di pin è stato scelto in modo da avere più diversificazione possibile nel tipo di periferiche da gestire e si è anche cercato di utilizzare tutti i pin disponibili nella scheda.

I cinque output digitali possono servire per il pilotaggio di diversi elementi come LED o ventole, gli input digitali possono servire per l'acquisizione di dati da sensori o da dispositivi esterni come fine corsa.

Gli input analogici sono utilizzati per acquisire ovviamente, segnali analogici, che devono essere condizionati prima di poter essere acquisiti onde evitare problemi o potenziali rotture nel microcontrollore.

L' input PWM è molto utile nel caso si sensori con uscita PWM o nel caso si voglia comunicare con dispositivi tramite essa.

L' OUTPUT PWM viceversa, è utile per pilotare ventole e dispositivi PWM.

Infine, i tre LED sono utilizzati per la segnalazione di eventi, quello di stato lampeggia di continuo e serve a verificare che la scheda stia funzionando correttamente e non si sia bloccata, il led di errore si accende di rosso nel caso si sia passato un limite precedentemente impostato e infine il led Wi-Fi si accende se la scheda è connessa a qualche dispositivo.

Nella scheda è stata salvata la pagina WEB che ne consente il controllo e la visione dei valori, la si riceverà ogni qualvolta ci si connette con un browser all' indirizzo IP della scheda (192.168.4.1).

Questa è come si presenta la pagina WEB del prototipo generale:

ESP32 DATA PAGE

DIGITAL

Name	Value	EN
DIGITAL OUTPUT 1	HIGH	☐
DIGITAL OUTPUT 2	HIGH	☐
DIGITAL OUTPUT 3	HIGH	☐
DIGITAL OUTPUT 4	HIGH	☐
DIGITAL OUTPUT 5	HIGH	☐

Name	Value
DIGITAL INPUT1	HIGH
DIGITAL INPUT 2	LOW
DIGITAL INPUT 3	LOW
DIGITAL INPUT 4	LOW

PWM

PWM	FREQUENCY	DC
PWM 1	0	0

ANALOG

Name	MAX	MIN	AVG	ACT
ANALOG INPUT1	0	0	0	0
ANALOG INPUT 2	0	0	0	0
ANALOG INPUT 3	0	0	0	0
ANALOG INPUT 4	0	0	0	0

ADC SET CH: 1 Bits 9 Att: 0 SET

ADC TC CH: 1 TC: SET

ADC STATUS

Attenuation:0=0dB, 1=2.5dB, 2=6dB, 3=11dB

Name	Bits	Attenuation	Limit Max	Limit Min
Analog1	LOW	LOW	LOW	LOW
Analog2	LOW	LOW	LOW	LOW
Analog3	LOW	LOW	LOW	LOW
Analog4	LOW	LOW	LOW	LOW

ADC LIMIT CH: 1 Limit Max: Limit Min: SET

ERRORS

Name	State
Analog1	LOW
Analog2	LOW
Analog3	LOW
Analog4	LOW
NULL	LOW
NULL	LOW
NULL	LOW
NULL	LOW
NULL	LOW

RESET

Nella pagina è possibile come precedentemente detto, visualizzare tutti i valori dei sensori ed effettuare modifiche ad alcuni parametri.

La parte DIGITAL consente di impostare gli OUTPUT digitale e di visualizzare lo stato degli INPUT, la parte ANALOG visualizza i valori relativi ai sensori analogici, ed è anche possibile

impostare alcuni parametri relativi all ADC. ADC STATUS è la parte in cui si visualizzano le impostazioni relative ai vari canali ed è anche impossibile impostare dei limiti massimi e minimi per ogni canale, oltre i quali viene attivato l'errore.

Infine, la parte ERRORS, serve ad indicare se si sono verificati degli errori nei canali scelti.

PROTOTIPO TEMPERATURA

Questo prototipo è una specializzazione di quello generale, il suo scopo principale è registrare la temperatura attraverso un sensore NTC e attivare una ventola se la temperatura raggiunge valori troppo elevati. In base alla misura effettuata inoltre, la ventola girerà ad un numero prefissato di RPM e la sua velocità verrà rilevata attraverso un ingresso PWM dell'ESP32. La scheda è inoltre dotata di un LED RGB ad alta intensità che può essere controllato tramite la pagina WEB, la sua utilità principale è quella di variare la temperatura misurata dalla scheda in modo da avere un prototipo che potesse variare la temperatura percepita autonomamente. È anche presente una scheda micro-SD per il log dei dati.

Le specifiche hardware del prototipo generale sono:

- MicroSD GPIO:2 4 12 13 14 15
- NTC temperatura GPIO:32
- LED DI STATO GPIO:18
- LED WIFI GPIO:22
- LED ERRORI GPIO:21
- PWM INPUT GPIO:19
- PWM OUTPUT GPIO:27
- PWM LED RGB GPIO:23 25 26
- INPUT DIGITALE GPIO:34
- OUPUT DIGITAL GPIO:33

Come si può vedere, molti pin qui sono stati utilizzati per la micro-SD, questo limita molto il numero di sensori che è possibile collegare alla scheda, ed inoltre altri tre pin sono utilizzati per i vari colori del LED RGB. L'INPUT PWM è stato utilizzato per acquisire la velocità della ventola, mentre l'OUTPUT PWM per comandarla.

Anche la pagina WEB è variata, implementando alcune funzioni aggiuntivee modificandone delle altre.

ESP32 DATA PAGE

LOCCIONI

DIGITAL

Name	Value	EN
DIGITAL OUTPUT 1	HIGH	☐

Name	Value
DIGITAL INPUT1	HIGH

PWM

PWM	FREQUENCY	DC
RPM FAN	0	0

ANALOG

Name	MAX C	MIN C	AVG C	ACT C
TEMPERATURA	0	0	0	0

ADC SET CH: 1 Bits 9 Att: 0 SET

ADC TC CH: 1 TC: SET

ERRORS

Name	State
TEMPERATURA	LOW

RESET

ADC STATUS

Attenuation:0=0dB, 1=2.5dB, 2=6dB, 3=11dB

Name	Bits	Attenuation	Limit Max	Limit Min
Analog1	LOW	LOW	LOW	LOW

ADC LIMIT CH: 1 Limit Max: Limit Min: SET

LED COLOR

LOG

MAX C	MIN C	AVG C	ACT C	DIGI OUT1	DIGI IN1	PWM F	PWM DC	ERR	DATE

La parte DIGITAL è rimasta inalterata, mentre quella PWM è stata realizzata in modo da convertire la lettura del segnale PWM direttamente in RPM. La parte ANALOG ha ora solo un sensore relativo alla temperatura la cui misura è convertita direttamente in gradi

centigradi, e anche la parte di STATUS ha solamente un canale, quello per l'appunto utilizzato di cui è possibile variare direttamente le impostazioni e i limiti sempre in gradi centigradi. È presente, inoltre, una parte per il controllo del LED RGB che ci consente di impostare il colore, e una parte di LOG che ci riporta tutti valore relativi alla scheda con tanto di data ogni secondo.

Con questa implementazione si è cercato di realizzare un esempio applicativo della scheda in modo da poterne testare le varie potenzialità e limiti.

HARDWARE UTILIZZATO

ADC

Il WROVER-B contiene 2 unità ADC a 12-bit, che hanno un totale di 18 canali, divisi tra ADC1 e ADC2. Il primo contiene otto canali, mentre il secondo ne contiene dieci.

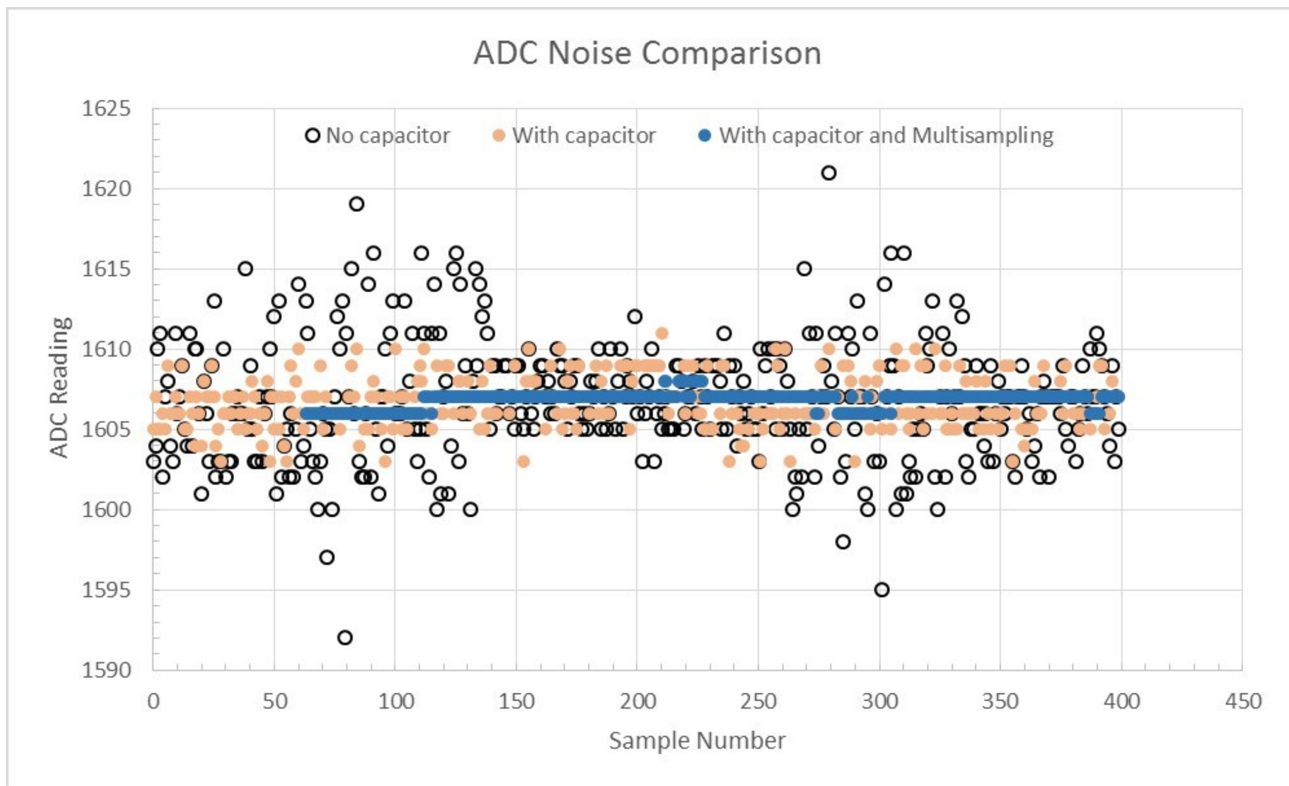
ADC1: GPIO 32-39

ADC2: GPIO 0,2,4,12-15,25-27

La prima unità è libera, mentre la seconda è in comune con il driver Wi-Fi e non è possibile utilizzare contemporaneamente entrambi i dispositivi. Nel caso si richiedono entrambe le funzionalità è necessario disabilitare il driver e riattivarlo dopo aver effettuato la campionatura. Inoltre, alcuni canali dell'ADC2 sono utilizzati da altre periferiche della demo board, quindi si è deciso di utilizzare soltanto il primo modulo che contiene comunque, sufficienti canali.

Purtroppo, l'ADC è abbastanza sensibile al rumore e questo porta a una grande diversificazione fra le letture, tuttavia è possibile sopperire a questo problema utilizzando due metodi. Il primo consiste nell' collegare un condensatore da 0.1uF vicino all' ingresso del canale dell'ADC, il secondo invece è un semplice multi-sampling, che può essere effettuato con diversi numeri di campioni a discapito però della velocita di campionamento.

La seguente immagine[1] mostra come variano le letture con i vari metodi utilizzati:



[1]

Si vede appunto come le letture siano non molto precise normalmente mentre migliorano notevolmente con i due metodi descritti sopra.

GPIO

L' ESP32 ha un totale di 40 GPIO ma nella demo board molte sono utilizzate da periferiche interne o componenti. Ad esempio, i pin tra 6 e 11 sono utilizzati per l'SPI flash presente nella scheda e i GPIO da 34 a 39 possono essere utilizzati soltanto come input e non presentano pull-up o pull-down software.

LED CONTROL

Il LED control è una periferica il cui scopo primario è il controllo dell'intensità dei LED, ma può anche essere utilizzata per generare segnali PWM di frequenza e duty-cycle variabili in tempo reale. I canali del LEDC sono divisi in due gruppi e sono otto in totale. Un gruppo lavora in HIGH SPEED MODE e offre una variazione di duty-cycle automatica e glitch-free, l'altro lavora in SLOW SPEED MODE e non può variare il suo DC in tempo reale ma è

necessario arrestare il driver e farlo ripartire tra un cambio e l'altro tramite una funzione di aggiornamento nel firmware.

PULSE COUNTER

Il pulse counter è una periferica che permette il conteggio di fronti di salita o discesa di un segnale applicato in ingresso. Ha a disposizione un registro a 16bit e due canali che possono essere configurati per aumentare o decrementare il conteggio nel registro, inoltre ogni canale ha a disposizione un segnale di controllo che permette di disabilitare o abilitare l'ingresso.

SDMMC Host

L' ESP32 ha un host SDMMC con due slot:

- Slot 0 a 8bit
- Slot 1 a 4bit

Inoltre, la mappatura dei pin è impostata di default secondo questo schema:

Signal	Slot 0	Slot 1
CMD	GPIO11	GPIO15
CLK	GPIO6	GPIO14
D0	GPIO7	GPIO2
D1	GPIO8	GPIO4
D2	GPIO9	GPIO12
D3	GPIO10	GPIO13
D4	GPIO16	
D5	GPIO17	
D6	GPIO5	
D7	GPIO18	
CD	any input via GPIO matrix	
WP	any input via GPIO matrix	

I pin di Card Detect e Write Protect possono essere indirizzati a qualunque altra GPIO.

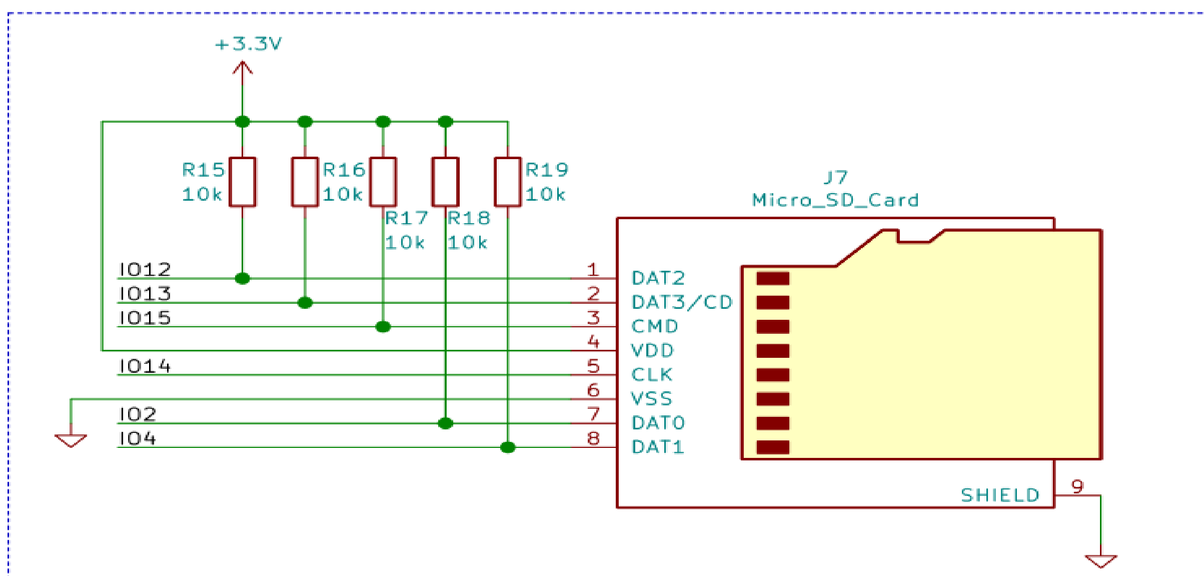
L’ Host supporta le seguenti modalità:

- Default Speed (20MHz)
- High Speed(40MHz)
- High Speed DDR (40MHz)

Inoltre, nei chip WROVER e WROOM lo slot 0 è condiviso con l’SPI flash e non può quindi essere normalmente utilizzato per la SD. Per poter accedere allo slot, bisogna cambiare i pin dell’SPI e scriverli nell’ eFuse.

È anche possibile utilizzare la SD in modalità SPI e 1bit nel caso ci sia il bisogno di risparmiare dei pin.

Lo schema di utilizzo tipico è il seguente:



TIMERS

L’ ESP32 ha due gruppi di timers hardware, ogni gruppo contiene 2 general-purpose timers. Sono tutti a 64bit basati su prescaler a 16bit e dotati di contatori up e down a 64bit con capacità di auto ricarica.

Ovviamente tutti i timer possono generare interrupt nel caso si abbia bisogno di eventi a tempi prefissati.

Il clock utilizzato è a 80MHz.

Wi-Fi

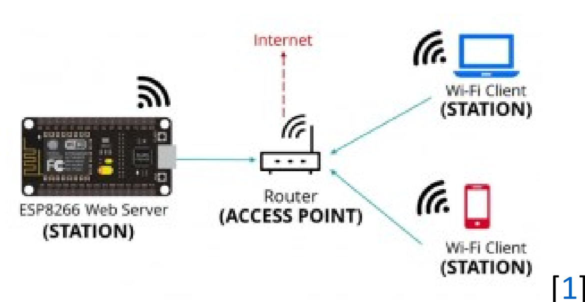
L'ESP32 comprende un modulo Wi-Fi con antenna integrata, le principali features sono:

- 802.11 b/g/n
- 802.11 n (2.4 GHz), up to 150 Mbps
- WMM
- TX/RX A-MPDU, RX A-MSDU
- Immediate Block ACK
- Deframmentazione
- Monitor automatico del Beacon
- 4 interfacce Wi-Fi virtuali

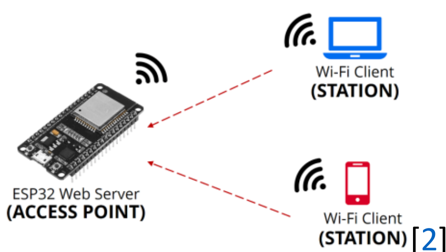
Può essere utilizzato in tre modi:

- Station
- Access Point
- Station e Access Point contemporaneamente

La doppia modalità è molto utile nel caso si voglia utilizzare il microcontrollore come ripetitore Wi-Fi, per estendere una rete già esistente. In modalità STATION la scheda si connette ad un access point come se fosse un normale smartphone o PC [1], mentre in modalità ACCESS POINT sono gli altri dispositivi che si connettono alla scheda [2].



[1]

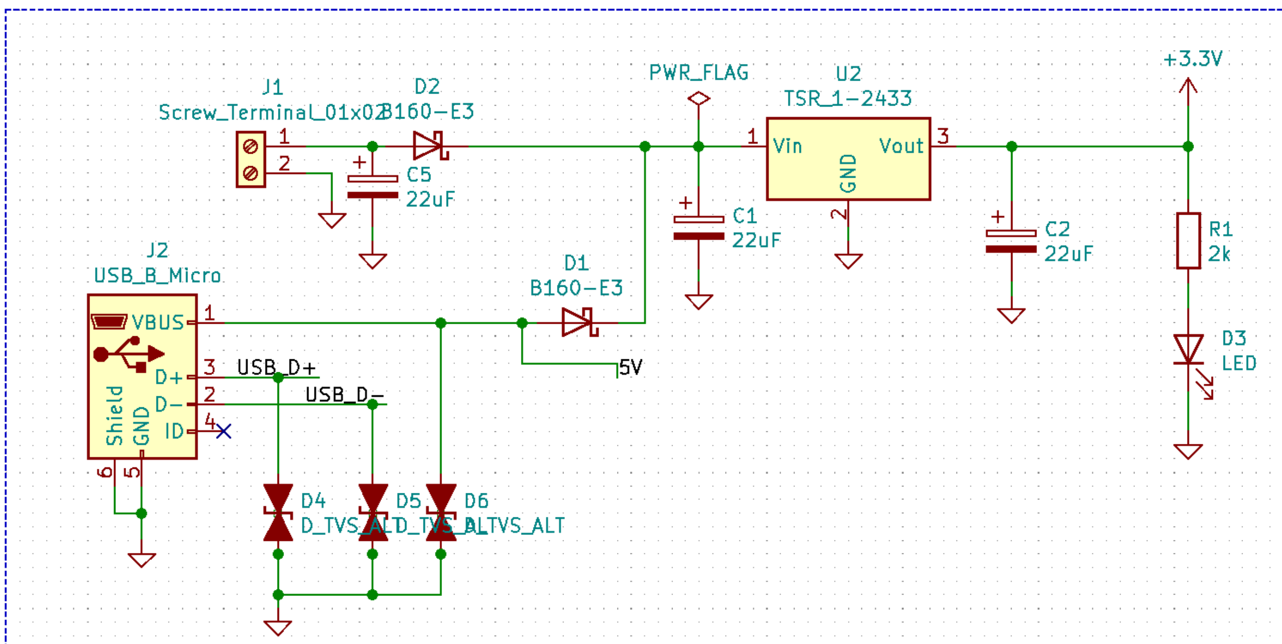


[2]

SCHEMI ELETTRICI E PCB

Dopo lo sviluppo del firmware e della pagina WEB, è stato anche sviluppato un prototipo di scheda non basato su DEVKIT-C, si è quindi preso un modulo ESP-32, e si è creata una nuova scheda intorno ad esso.

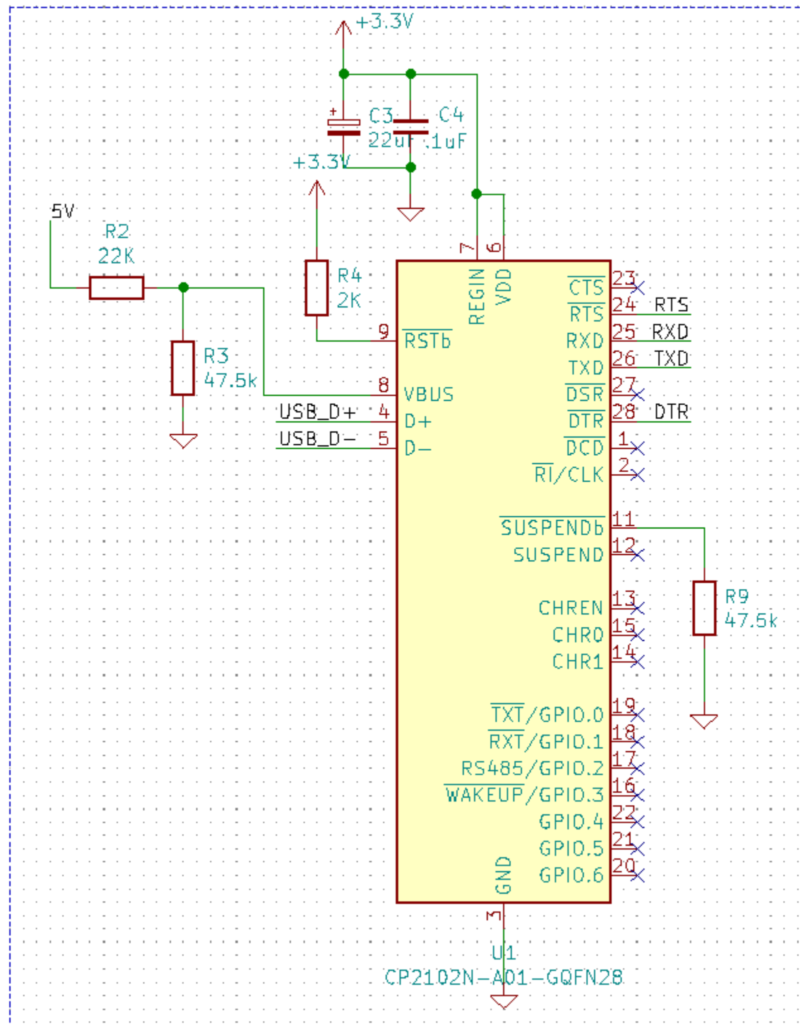
Si discuterà parte per parte le scelte progettuali legate ad essa.



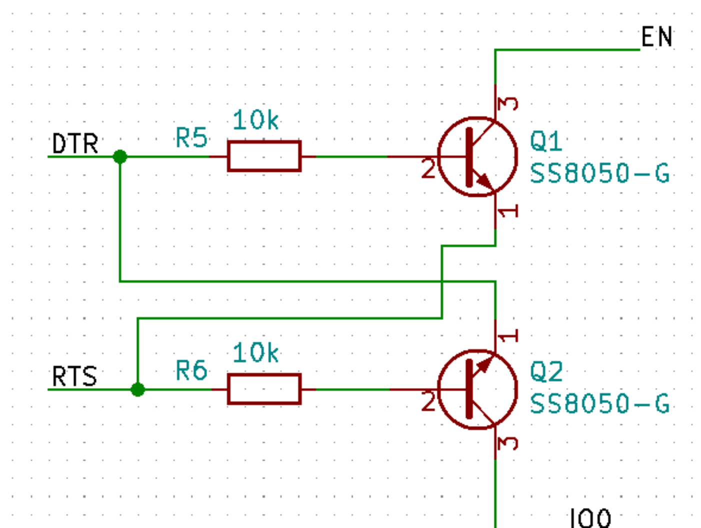
Questa è la parte di alimentazione che genera la 3.3V necessaria alla scheda, è possibile alimentare la scheda da USB oppure attraverso una tensione applicata al terminale J1, l'integrato TSR_1_2433 è un regolatore di tensione a 3.3V, la scelta è ricaduta su di esso perché spesso in ambito industriale si ha disponibile la 24V e non sono molti i regolatori in grado di convertire una tensione cosi "alta" in 3.3V.

Nella seconda parte è presente il chip CP2102N che è un convertitore da UART a USB, è utilizzato per flashare il firmware e comunicare in seriale con altri dispositivi.

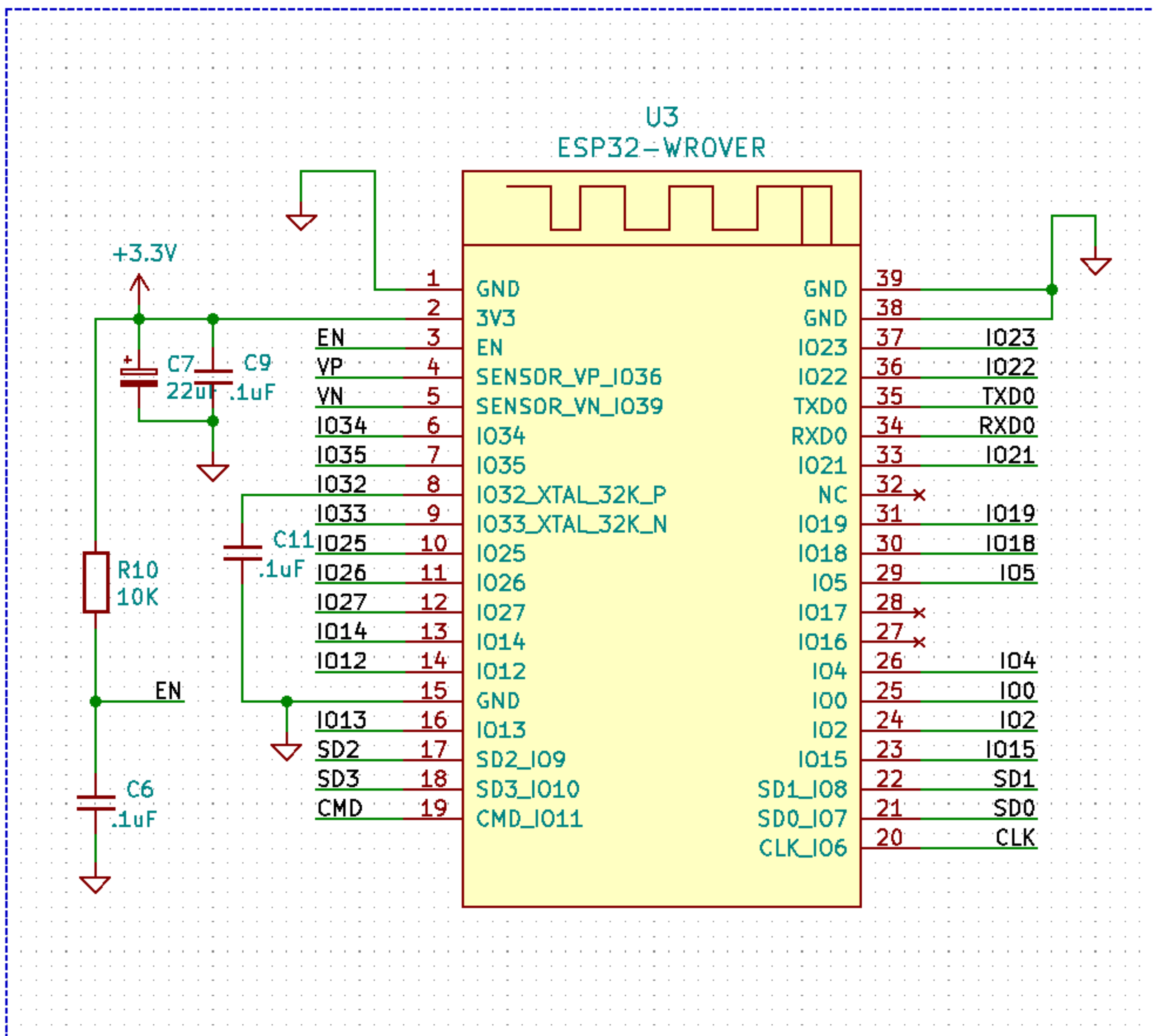
Si è scelto di utilizzare un connettore micro USB visto l'attuale popolarità, ma non si esclude una futura implementazione che sfrutta il connettore Type-C vista la sua incredibile modularità sia dal punto di vista di tensioni che di velocità di comunicazione.



Per le connessioni si è seguito semplicemente il datasheet, un'ottima particolarità è la possibilità del chip di mettere automaticamente il chip in modalità di scrittura, attraverso la seguente rete di transistor:

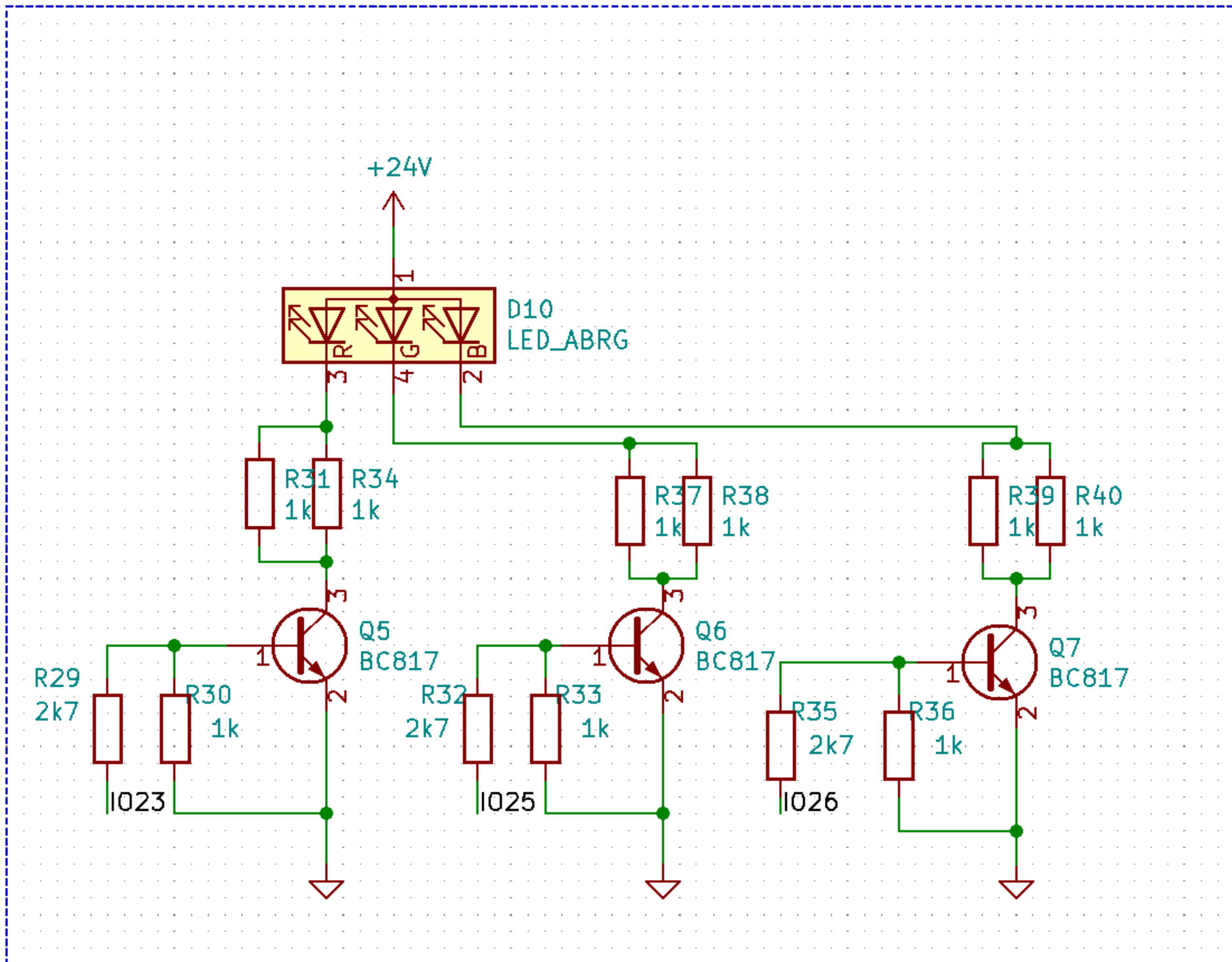


Si va poi a inserire la parte relativa al microcontrollore vero e proprio:



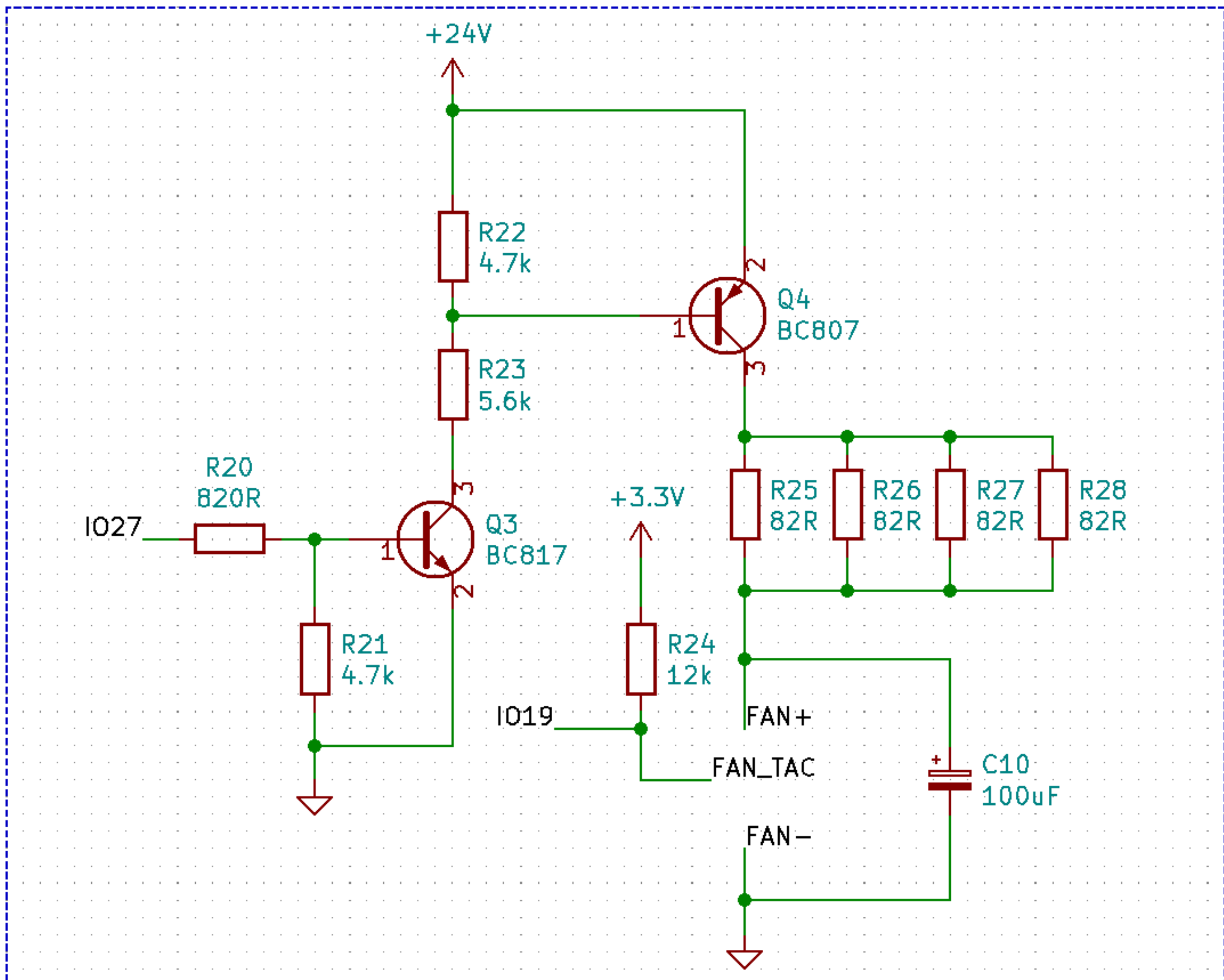
Qui non c'è molto da dire, si inseriscono semplicemente i rimandi ad altre parti del circuito e si porta l'alimentazione alla scheda, inoltre si inserisce anche un pull-up nel pin di ENABLE della scheda.

La parte successiva implementa l’hardware per il controllo del LED RGB:



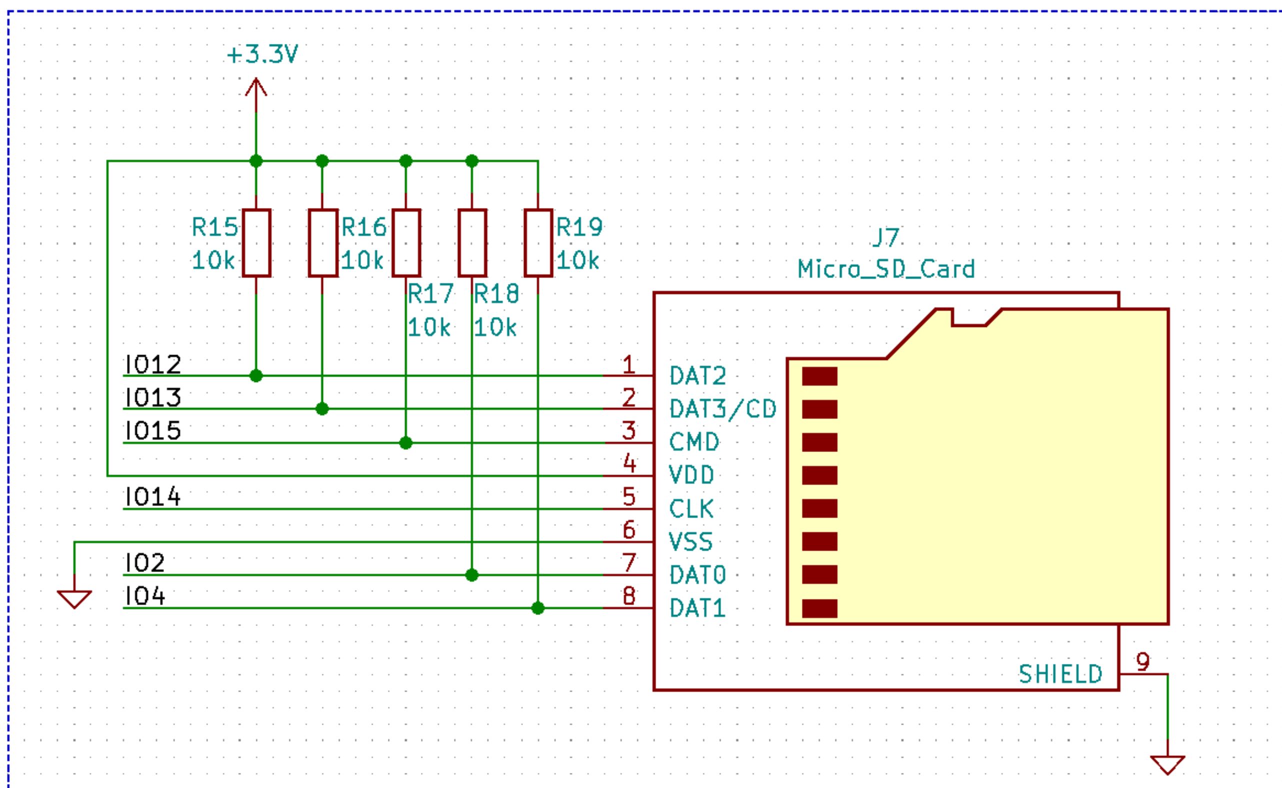
Si hanno tre canali, uno per colore del LED e ognuno è collegato a ad un transistor e ad un banco di resistenze che ne limitano la corrente, il banco è necessario visto che la corrente di ogni canale è decisamente elevata. Ogni transistor, inoltre, presenta una resistenza di pull-down per assicurarsi che la base sia a massa in caso di uscita ad alta impedenza. Da notare che il LED è alimentato a 24V cosa che porta su notevolmente il valore di resistenza da utilizzare per limitare la corrente.

La penultima parte descrive il pilotaggio della ventola:



Qui sono presenti due transistor in cascata il cui compito è quello di pilotare la ventola tramite un segnale PWM generato dal microcontrollore, i due transistor uno NPN e l'altro PNP sono posti in cascata ed è presente un grosso banco di resistenze per la limitazione di corrente, inoltre, è stato inserito un condensatore in parallelo ai terminali della ventola per aiutare il pilotaggio di ventole che non funzionano in PWM. L' IO19 è l'ingresso PWM della scheda che consente la misurazione della velocità di rotazione, è necessario un pull-up per il corretto funzionamento.

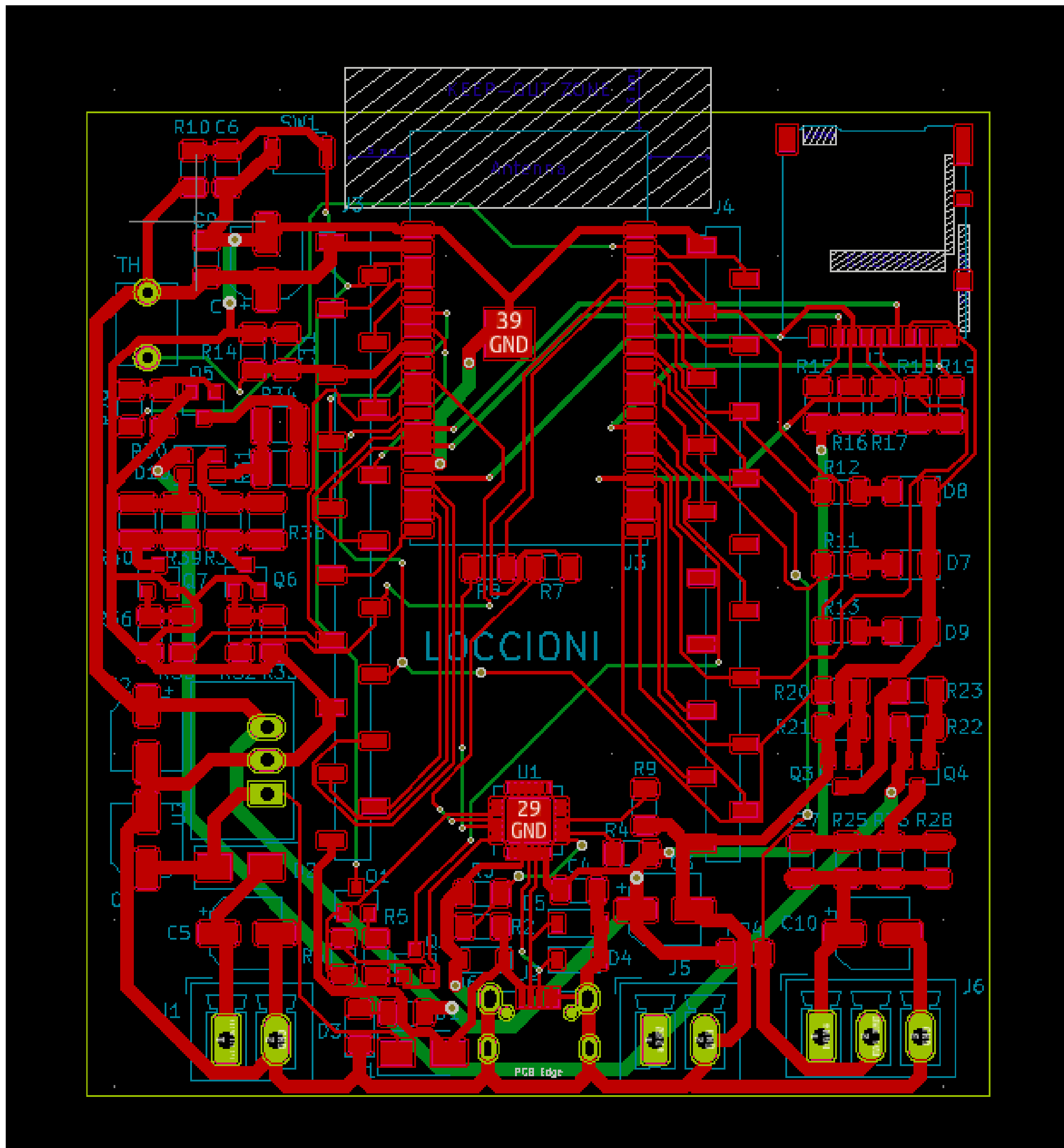
L’ ultima parte riguarda la scheda micro-SD:



La scheda viene connessa allo slot direttamente, e si utilizzano dei resistori di pull-up per aiutare la comunicazione.

PCB

Con il precedente schema elettrico è stato realizzato un PCB, sempre attraverso KiCAD, che si presenta nella seguente maniera:

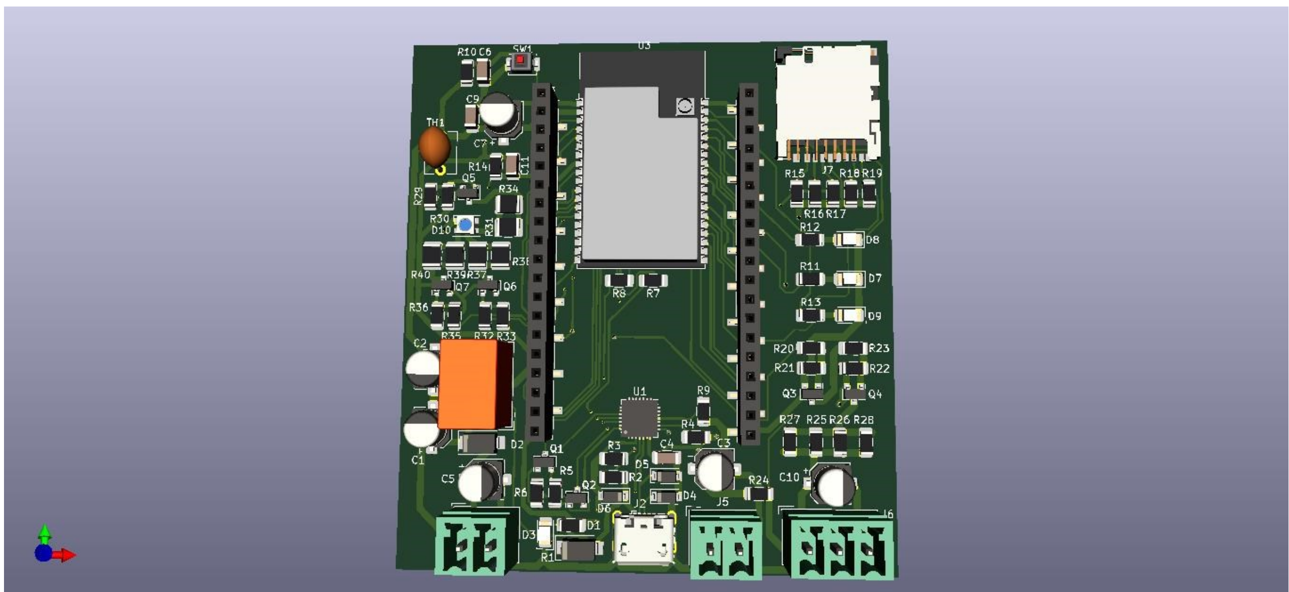


Il PCB presenta le dimensioni di 69mm x 75mm si è cercato di rendere tutti il più compatto possibile anche in modo di aumentare la velocità di scrittura sulla SD nel caso in una futura implementazione si richiesto di salvare una quantità di dati maggiore, ovviamente le

dimensioni delle piste sono state scelte in base all' ammontare di corrente che devono trasportare, quelle di alimentazione presentano una larghezza di 1mm o di 1.5mm mentre quelle di segnale una larghezza di 0.3mm.

Il PCB è un dual-layer, quindi sono state create piste in entrambe le facce ed è stato inoltre necessario lasciare spazio nella parte superiore per evitare disturbi con la comunicazione Wi-Fi del microcontrollore.

Il PCB renderizzato si presenta così:



Inoltre, è stato effettuato uno studio riguardante la possibilità di alimentare la scheda tramite batteria, aimè l'assorbimento della scheda in IDLE si aggira sui 80-100mAh che scaricherebbero una batteria da 2000mA nel giro di una giornata, inoltre si raggiungono picchi di quasi 400mAh quando la comunicazione Wi-Fi è attiva che fanno scaricare ancor più velocemente la batteria.

È possibile implementare una funzionalità di deep-sleep che porterebbe il consumo della scheda nell'ordine microampere, purtroppo però non è possibile risvegliare la scheda tramite Wi-Fi, quindi, è impossibile comunicare in tempo reale in caso di necessità. Si lascia quindi questo tipo di alimentazione solo ad implementazioni particolari che non richiedono la comunicazione in qualsiasi momento.

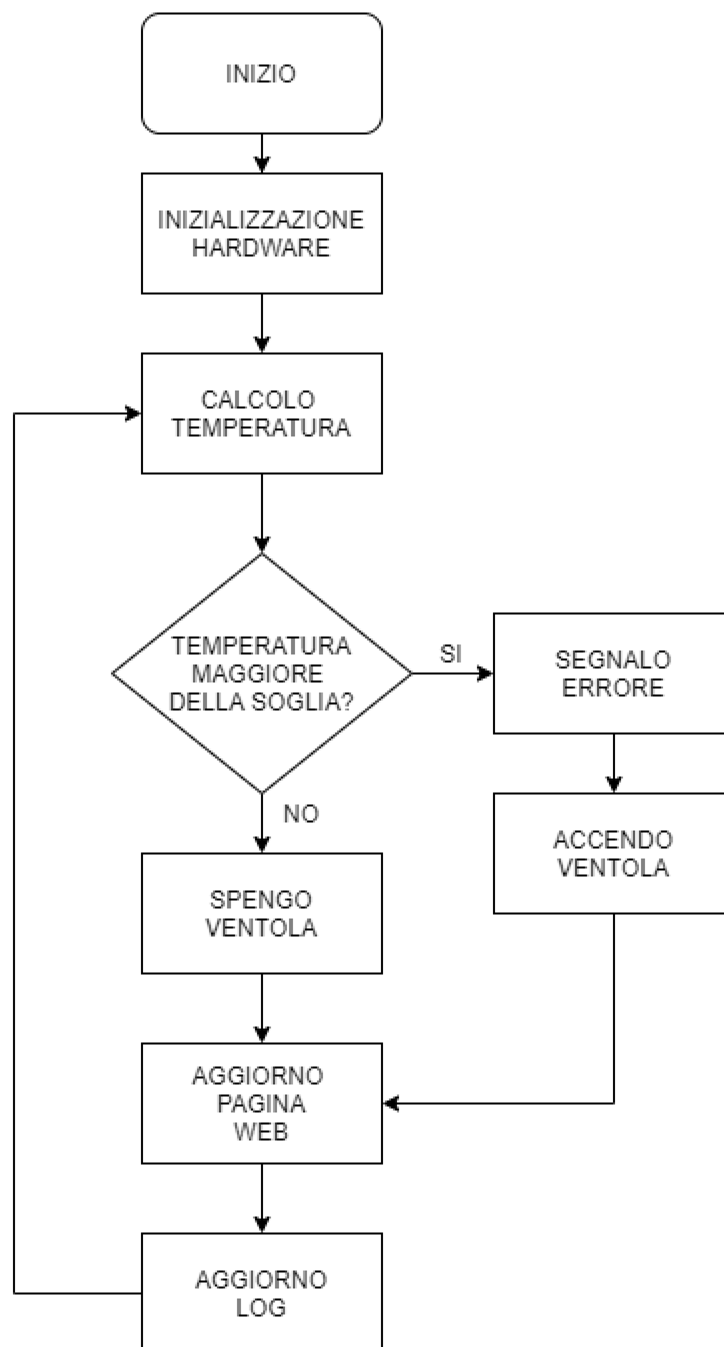
IMPLEMENTAZIONE FIRMWARE

Il firmware è stato scritto utilizzando Eclipse, e quasi tutte le funzioni utilizzate sono state prelevate dalla API ufficiale realizzata da Espressif.

Il programma è strutturato su diversi file C che implementano le varie funzioni utilizzate poi nel main del programma, i file sono:

- ADC_timers : implementa le funzionalità del timer utilizzato nel ADC.
- ADC1: implementa le funzionalità relative all' ADC come multisampling, calibrazione e inizializzazione.
- cJSON : implementa tutte le funzioni per la gestione di strutture JSON.
- COUNTER: implementa le funzionalità del pulse counter come l'inizializzazione dei registri e il clear.
- MAIN: implementa tutta la logica del programma compreso il server http e l'inizializzazione di tutte le periferiche.
- PWM: implementa tutte le funzionalità relative ai led control e al pilotaggio di ventola tramite PWM.

Dalla seguente flowchart possiamo vedere come il programma si comporta, la prima cosa che accade è ovviamente l'inizializzazione dell'hardware seguita dal calcolo della temperatura, in base alla temperatura rilevata accendo o spengo la ventola e successivamente aggiorno la pagina web e il file di LOG.



ADC_timers.c

```c
#include "driver/timer.h"

void adc_timers()
{
	timer_config_t config;
	config.alarm_en=1;
	config.auto_reload=1;
	config.counter_dir=1;
	config.counter_en=0;
	config.divider=16;
	config.intr_type=0;

	timer_init(0,1,&config);
	timer_set_counter_value(0,1,0x00000000ULL);
	timer_set_alarm_value(0,1,5000);
	timer_enable_intr(0,1);
}

void adc_timers_start()
{


	timer_start(0,1);
}


void DC_timers()
{
	timer_config_t DC_config;
	DC_config.alarm_en=0;
	DC_config.auto_reload=0;
	DC_config.counter_dir=1;
	DC_config.counter_en=0;
	DC_config.divider=32;
	DC_config.intr_type=0;

	timer_init(1,1,&DC_config);
	timer_set_counter_value(1,1,0x00000000ULL);

}
```

In questo primo file vengono definite le configurazioni per tutti i timer che si useranno in seguito, adc_timers() configura il timer relativo all' ADC, avendo un clock a 80MHz si è scelto un prescaler di 16 quindi si avrà $\frac{80.000.000}{16} = 5.000.000$ questo significa che si avranno 5 milioni di tick in un secondo, l' obbiettivo di questa funzione è quello di impostare ogni quanto campionare il segnale in ingresso all' ADC, quindi visto che da specifiche si vuole campionare il segnale ogni millisecondo andremo ad impostare un

alarm ogni 5000 tick, la funzione dell' alarm è quella di generare un interrupt ogni volta che si raggiunge il valore assegnato, nella cui routine si effettueranno tutte le operazioni di campionamento.

Si ha poi la funzione adc_timers_start() che serve a far partire i timers e la funzione DC_timers() che configura un timer utilizzato nel calcolo del duty-cycle di un segnale in ingresso.

ADC1.c

```
#include "adc1.h"

int multisample(int n , int channel)          //creo una funzione per effettuare il multisample
       {

       int ms_val=0;

       for(int i=0;i<n;i++)
              {
                     ms_val=ms_val + adc1_get_raw(channel);
              }
       int result=ms_val/n;
       return result;
       }

void adc_init(int bits, int channel, int atten,esp_adc_cal_characteristics_t *calib )
       //configuro l' adc con la relativa calibrazione
{

       esp_adc_cal_characterize(ADC_UNIT_1, atten,bits, 1093, calib);
}

uint32_t get_mv(esp_adc_cal_characteristics_t *calib, int raw)         // ottengo i mV con la calibrazione effettuata
{
       uint32_t cval= esp_adc_cal_raw_to_voltage(raw,calib);
       return cval;
}
```

Questo file definisce le funzioni utilizzate nella configurazione e nell' utilizzo dell'ADC, **multisample** effettua il multisampling di un numero dato di campioni e restituisce il risultato. **adc_init** invece configura l'ADC con i parametri dati e lo calibra con la curva di calibrazione passata. Infine, **get_mv** restituisce la lettura in millivolt in base al valore "raw" che gli viene passato.

cJSON.c

Il prossimo file è cJSON.c che contiene una libreria per la gestione di strutture JSON, la libreria è stata presa da una repository di Github. Contiene un grande numero di funzioni per gestire le strutture JSON ed ha semplificato molto l'invio e la lettura di dati da altre fonti.

Counter.c

```
#include "counter.h"

void counter_init()
{
	pcnt_config_t pulse_config;
	pulse_config.pulse_gpio_num=19;
	pulse_config.pos_mode=PCNT_COUNT_INC;
	pulse_config.channel=PCNT_CHANNEL_0;
	pulse_config.unit=PCNT_UNIT_0;
	pulse_config.neg_mode=PCNT_COUNT_DIS;
	pulse_config.lctrl_mode=PCNT_MODE_KEEP;
	pulse_config.hctrl_mode=PCNT_MODE_KEEP;
	pulse_config.ctrl_gpio_num=PCNT_PIN_NOT_USED;
	pulse_config.counter_h_lim=32000;
	pulse_config.counter_l_lim=-32000;

	pcnt_unit_config(&pulse_config);

}

void counter_clear()
{
	pcnt_counter_pause(PCNT_UNIT_0);
	pcnt_counter_clear(PCNT_UNIT_0);
	pcnt_counter_resume(PCNT_UNIT_0);
}
```

Questo file è utilizzato per inizializzare il pulse counter che serve a contare la frequenza del segnale PWM in ingresso alla scheda. La funzione **counter_init** inizializza il counter prendendo come ingresso il GPIO 19, il registro che tiene memoria degli impulsi ha possibilità di generare interrupt arrivato ad una certa soglia. La funzione **counter_clear** azzera il contatore quando necessario.

PWM.c

```c
#include "pwm.h"

void init_PWM()
{
	ledc_timer_config_t PWM_t_config;
	PWM_t_config.timer_num=LEDC_TIMER_0;
	PWM_t_config.speed_mode =LEDC_HIGH_SPEED_MODE;
	PWM_t_config.freq_hz=4000;
	PWM_t_config.duty_resolution=LEDC_TIMER_8_BIT;
	PWM_t_config.clk_cfg = LEDC_AUTO_CLK;
	ledc_timer_config(&PWM_t_config);

       ……

	ledc_channel_config_t red_config;
	red_config.channel=LEDC_CHANNEL_3;
	red_config.timer_sel=LEDC_TIMER_0;
	red_config.duty=0;
	red_config.gpio_num=23;
	red_config.hpoint=0;
	red_config.speed_mode=LEDC_HIGH_SPEED_MODE;
	ledc_channel_config(&red_config);
	ledc_fade_func_install(0);
}

void PWM_STOP()
{
	ledc_stop(LEDC_HIGH_SPEED_MODE,LEDC_CHANNEL_1,0);
}

void PWM_START()
{
	ledc_channel_config_t channel_config;
		channel_config.channel=LEDC_CHANNEL_1;
		channel_config.timer_sel=LEDC_TIMER_0;
		channel_config.duty=128;
		channel_config.gpio_num=27;
		channel_config.hpoint=0;
		channel_config.speed_mode=LEDC_HIGH_SPEED_MODE;
		ledc_channel_config(&channel_config);
}
```

Questo file configura tutte le PWM utilizzate, sia per quanto riguarda il controllo del LED RGB sia per quanto riguarda il pilotaggio di dispositivi esterni. La funzione **init_PWM** serve ad inizializzare il timer per il LED control e tutti i canali PWM utilizzati, si è scelto di utilizzare una frequenza di 4kHz e una risoluzione di duty-cycle di 8bit, entrambi sono valori intermedi che vanno più che bene normali applicazioni. Le funzioni **PWM_STOP** e **PWM_START** sono invece utilizzare per far partire o fermare i vari canali.

MAIN.c

Questo è il file principale dove praticamente tutto il programma viene svolto.

```
#include "freertos/FreeRTOS.h"
#include "esp_wifi.h"
#include "esp_system.h"
#include "esp_event.h"
#include "esp_event_loop.h"
#include "nvs_flash.h"
#include "driver/gpio.h"
#include "string.h"
#include <esp_http_server.h>
#include "driver/timer.h"
#include "adc1.h"
#include "adc_timers.h"
#include "counter.h"
#include "pwm.h"
...
```

La prima parte è composta da tutte le include necessarie al corretto funzionamento del programma, questo comprende le include contenenti le funzioni descritte precedentemente e le include riguardanti le varie periferiche del WROVER-B, oltre a funzioni di gestione di eventi e task.

```
#define ADC1_CHANNEL1 4   //pin 32
#define DIGITAL_PIN_1 GPIO_NUM_33
#define DIGITAL_INPUT_1 GPIO_NUM_34
#define N_DIGITAL_OUTPUT 1
#define N_DIGITAL_INPUT 1
#define LED_STATO GPIO_NUM_18
#define LED_WIFI GPIO_NUM_22
#define LED_ERRORI GPIO_NUM_21
#define ERRORI_SIZE 1
#define BUFFER_SIZE_HTTP 500

#define SSID "ESP32"
#define PASSWORD "PasswordDiProva"  //RISPETTARE LUNGHEZZA MINIMA WPA2

#define ADC_AVG_DEFAULT 0
#define ADC_MAX_DEFAULT 0
#define ADC_MIN_DEFAULT 50
#define ADC_COUNTER_DEFAULT 0
#define ADC_COUNTER_TH_DEFAULT 10
#define ADC_SUM_DEFAULT 0
#define ADC_INDEX_DEFAULT 0
#define ADC_RAW_DEFAULT 0
#define ADC_TH_MAX_DEFAULT 40
#define ADC_TH_MIN_DEFAULT 10
#define ADC_CHANNELS 1

#define PIN_RED 23
#define PIN_GREEN 25
#define PIN_BLUE 26
```

La seconda parte contiene invece tutte le “define” utilizzate per consentire di avere una certa flessibilità nel programma, questo significa che se un giorno qualcuno dovesse cambiare dei valori o dei GPI,O gli basta semplicemente cambiare valore all’ interno delle “define” invece di dover cambiare il valore ad ogni utilizzo del dato. Vengono qui definiti anche SSID e PASSWORD del Wi-Fi generato dalla scheda, e i valori di default dell’ADC.

```
onst float ZERO_GRADI=3.5563;
const float CINQUE_GRADI=2.7119;
const float DIECI_GRADI=2.086;
const float QUINDICI_GRADI=1.6204;
const float VENTI_GRADI=1.2683;
const float VENTICINQUE_GRADI=1.000;
const float TRENTA_GRADI=0.7942;
const float TRENTACINQUE_GRADI=0.6327;
const float QUARANTA_GRADI=0.5074;
const float QUARANTACINQUE_GRADI=0.41026;
const float CINQUANTA_GRADI=0.3336;
const float CINQUANTACINQUE_GRADI=0.27243 ;

static const char *TAG = "ESP32 LOG SERVICE";
bool LED_STATO_level=0;

int PWM_counter=0;
int PWM_counter_th=1000;
int16_t PWM_value=0;

struct canale
{
	int counter;
	int counter_th;
	int buffer_cha[100];
	int ch_avg;
	int ch_max;
	int ch_min;
	int ch_sum;
	int ch_index;
	int ch_raw;
	int ch_th_max;
	int ch_th_min;
	uint32_t ch_mv;
};
struct canale channel1;

...
```

Si prosegue poi con la definizione di gran parte delle variabili utilizzate all’ interno del programma. Molte sono utilizzate nella logica, altre contengono invece dei valori utilizzati per compiere calcoli.

Inseriamo ora la funzione **main** del programma, che nella stesura del firmware appare per ultima, ma per facilità di lettura inserisco per prima:

```
void app_main(void)
{
	nvs_flash_init();
	gpio_init();		//inizializzazione gpio
	init_PWM();		//inizializzazione acquisizione e uscite PWM
	init_wifi();		//inizializzazione wifi
	init_server();		// inizializzazione server http
	struct_init();		// inizializzazione strutture adc
	adc_calib(3,3);		//default adc1 cha 4 ,pin 32, 12 bit e 11dB di attenuazione
	sd_init();		//inizializzazione sd
	adc_timers();		// inizializzazione timers
	adc_timers_intr();
	timer_start(0,1);	// inizio conteggio timer adc
	DC_timers();		// inizio conteggio timer duty cycle
	counter_init();		// inizializzazione pulse counter
	counter_clear();	// azzeramento pulse counter
	PWM_STOP();		// stop PWM, serve a provarne il funzionamento all accensione
	spiff_init();		// inizializzazione partizione pagina HTML


	xTaskCreate(&blink,"LED STATO",4000,NULL,5,NULL);   // creazione task blink
	xTaskCreate(&errors_check,"ERRORI",4000,NULL,5,NULL);//creazione task errori
	xTaskCreate(&temp_calc,"Temperatura",4000,NULL,5,NULL);
	//creazione task calcolo temperatura
	xTaskCreate(&log_write,"LOG",4000,NULL,5,NULL);     //creazione task log su SD

}
```

NVS_FLASH_INIT()

Per prima cosa si inizializza la NVS (non volatile storage) necessaria per il corretto funzionamento della scheda, in pratica si crea una partizione che consente la memorizzazione di informazioni all’ interno della memoria flash della scheda, questa memoria essendo non volatile consente di mantenere memorizzate le informazioni anche in assenza di alimentazione o in caso di reset della scheda. All’ interno di questa partizione i dati vengono salvati in coppia chiave/valore dove quindi ad ogni dato viene assegnato un nome. È inoltre possibile memorizzare diversi tipi di dati. La partizione nel mio caso non è direttamente utilizzata, bensì il Wi-Fi esegue delle operazioni interne che ne richiedono l’utilizzo.

GPIO_INIT()

Si vanno poi ad inizializzare le GPIO [1], procedura richiesta per il corretto funzionamento delle porte nella scheda e la corretta implementazione degli interrupt necessari in alcune parti del programma. Si definiscono OUTPUT per i vari led di stato e per l’uscita

programmabile tramite WEB, gli INPUT per la PWM e per il singolo ingresso digitale definito dalle specifiche. Gli interrupt nell' ESP32 funzionano su priorità, definita in base a livelli crescenti, quasi sempre nel corso del programma si è utilizzato un livello intermedio, in quanto non c'è una particolare fretta di esecuzione in nessuno degli interrupt presenti. Nel caso delle GPIO si utilizza un livello 3 da 1 a 5.L' interrupt è utilizzato per il calcolo del duty-cycle del segnale PWM in ingresso.

[1]

```
void gpio_init()
{
	gpio_config_t gp_config;
	gp_config.intr_type=0;
	gp_config.mode=GPIO_MODE_OUTPUT;
	gp_config.pull_down_en=GPIO_PULLDOWN_ENABLE;
	gp_config.pull_up_en=0;
	gp_config.pin_bit_mask=1ULL<<DIGITAL_PIN_1|1ULL<<LED_STATO|
	1ULL<<LED_WIFI|1ULL<<LED_ERRORI;

	if(gpio_config(&gp_config)==ESP_ERR_INVALID_ARG)
...
```

INIT_PWM()

Successivamente si configura la PWM con la sua funzione di inizializzazione dove si creano le configurazioni per tutte le uscite del LED RGB (tre canali) da pilotare e per la ventola.

INIT_WIFI()

A seguire abbiamo l'inizializzazione del Wi-Fi [2], per prima cosa si crea uno stack TCP-IP necessario per la corretta assegnazione degli indirizzi all' interno della rete creata dall' ESP32, si va poi a inserire la configurazione di default per l'hardware. Successivamente si configurano i parametri di autenticazione come SSID, password e modalità di autenticazione, e si imposta il Wi-Fi in modalità access point, in quanto sono le altre apparecchiature che si connettono alla scheda ed infine si avvia il Wi-Fi. È necessario anche configurare una routine di gestione di eventi specifica per il Wi-Fi necessaria per gestire le handle di connessione/disconnessione.

[2]

```
void init_wifi()
{

	tcpip_adapter_init();

	esp_event_loop_create_default();

	wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();

	esp_wifi_init(&cfg)

	wifi_config_t ap_config = {
			.ap = {
			.ssid = SSID,
			.ssid_len = strlen(SSID),
			.max_connection =2,
			.password = PASSWORD,
			.authmode = WIFI_AUTH_WPA_WPA2_PSK

	...
```

INIT_SERVER()

Il passo successivo e l'inizializzazione del server http necessario per l'interfacciamento con la pagina web. L' ESP32 permette di eseguire un vero proprio server http al suo interno, ovviamente nei limiti delle capacità dell'hardware del microcontrollore. Per prima cosa è necessario configurare il server, per semplicità è stata utilizzata la configurazione di default con qualche piccola modifica al numero delle URI gestibili e al numero di massime connessione accettate in ingresso. La prima modifica è necessaria vista la grande quantità di URI necessarie, la seconda invece per la potenza della scheda che non è "troppo" elevata. [3]

Si vanno poi a definire tutte le varie URI utilizzate con i relativi metodi:

- api_analog_get_status
- api_dl_get
- api_analog_get
- api_analog_post
- api_rgb_post
- api_analog_tc_post
- api_errori_get

- api_PWM_get
- api_set_pin_level_post
- api_get_pin_level_get
- api_main_http
- api_error_reset
- api_set_limit_post

In totale sono tredici URI tutte utilizzate nella comunicazione con la pagina WEB, ma nulla vieta visto la loro costruzione, di utilizzarle ad esempio con un'altra scheda. Se ad esempio colleghiamo un'altra scheda ESP32 tramite Wi-Fi possiamo comunicare semplicemente tramite richieste http visto che in tutte le URI si utilizzano strutture JSON per lo scambio di dati.

[3]

```
void init_server()
{
	httpd_handle_t server = NULL;
	httpd_config_t config;

	config.task_priority = tskIDLE_PRIORITY+5;
	config.stack_size         = 10000;
	config.core_id            = tskNO_AFFINITY;
	config.server_port        = 80;
	config.ctrl_port          = 32768;
	config.max_open_sockets   = 7;
	config.max_uri_handlers   = 15;
	config.max_resp_headers   = 14;
	config.backlog_conn       = 7;
	config.lru_purge_enable   = false;
	config.recv_wait_timeout  = 5;
	config.send_wait_timeout  = 5;
	config.global_user_ctx = NULL;
	config.global_user_ctx_free_fn = NULL;
	config.global_transport_ctx = NULL;
	config.global_transport_ctx_free_fn = NULL;
	config.open_fn = NULL;
	config.close_fn = NULL;
	config.uri_match_fn = NULL;

	httpd_uri_t set_limit_post = {
			.uri      = "/api/analog/limit",
			.method   = HTTP_POST,
			.handler  = analog_limit_post_handler,
			.user_ctx = NULL
	};
...
```

Una volta assegnate le URI bisogna anche affiancargli una handler in modo da eseguire determinate azioni quando avviene una richiesta.

```
esp_err_t set_pin_level_post_handler(httpd_req_t *req)
{
	bool setarray[6]={0,0,0,0,0,0};
	cJSON* pins=NULL;
	//creo due oggetti json null
	cJSON* pin=NULL;
	char http_string[500];
	//buffer in ingresso

	esp_err_t receive=httpd_req_recv(req,http_string,500);
....
```

STRUCT_INIT()

Dopo il server, vado a definire una struttura utilizzata nella configurazione del mio ADC, in pratica questa struttura conterrà tutti i valori relativi ad un singolo canale ADC, in questo caso conterrà tutti i dati relativi all' ingresso del nostro sensore di temperatura, come il valore medio, massimo e minimo oppure come la soglia entro quale far scattare l'errore, il tutto viene inizializzato con i valori di default prestabiliti.

```
void struct_init()
{
	channel1.ch_avg=ADC_AVG_DEFAULT;
	channel1.ch_index=ADC_INDEX_DEFAULT;
	channel1.ch_min=ADC_MIN_DEFAULT;
	channel1.ch_raw=ADC_RAW_DEFAULT;
	channel1.ch_sum=ADC_SUM_DEFAULT;
	channel1.ch_th_max=ADC_TH_MAX_DEFAULT;
	channel1.ch_max=ADC_MAX_DEFAULT;
	channel1.counter=ADC_COUNTER_DEFAULT;
	channel1.counter_th=ADC_COUNTER_TH_DEFAULT;
	channel1.ch_th_min=ADC_TH_MIN_DEFAULT;

}
```

Successivamente vado a calibrare l'ADC indicando l'attenuazione e i bit di risoluzione da utilizzare, di default si utilizzano 12bit di risoluzione e 11dB di attenuazione in modo da avere la maggior risoluzione possibile ed un valore di fondo scala abbastanza ampio.

Il prossimo passo è la configurazione della scheda micro-SD, dove vengono impostati i vari GPIO per la comunicazione e la frequenza che pregiudica la velocità di scrittura. Si è scelto di utilizzare 500kHz visto che la scheda è stata implementata su mille fori e non su PCB, questo significa che si sono utilizzate saldature e fili per connettere lo slot della SD all' ESP32, i quali presentano una molta più alta impedenza alle altre frequenze rispetto alle tracce utilizzate su circuito stampato. I 500kHz assicurano una velocità più che sufficiente per la piccola mole di dati che bisogna scrivere e una buona affidabilità. Una volta configurato lo slot utilizzato è necessario montare la partizione della scheda per poter poi andare a scrivere in essa, viene utilizzata una partizione FAT compatibile con quasi tutti i dispositivi con una dimensione di allocazione di 16KB, utilizzando poi le normali funzioni di gestione file del linguaggio C si va a creare una nuova cartella nella partizione, a meno che non ne esista già una con lo stesso nome e si va poi a creare un nuovo file in essa, che sarà appunto il nostro file di LOG.

```
void sd_init()
{
	sdmmc_host_t host = SDMMC_HOST_DEFAULT();
	sdmmc_slot_config_t slot_config = SDMMC_SLOT_CONFIG_DEFAULT();
	host.max_freq_khz=500;

	gpio_set_pull_mode(15, GPIO_PULLUP_ONLY);
	gpio_set_pull_mode(2, GPIO_PULLUP_ONLY);
	gpio_set_pull_mode(4, GPIO_PULLUP_ONLY);
	gpio_set_pull_mode(12, GPIO_PULLUP_ONLY
	gpio_set_pull_mode(13, GPIO_PULLUP_ONLY);

	esp_vfs_fat_sdmmc_mount_config_t mount_config = {
			.format_if_mount_failed = false,
			.max_files = 5,
			.allocation_unit_size = 16 * 1024
	};
```

Si vanno poi a configurare i timer per l'ADC, come precedentemente detto si vuole andare a campionare la temperatura ogni 1mS, significa che si andrà ad impostare il timer in modo da avere un interrupt ogni 5000 tick, derivati dal clock di 80MHz e il prescaler di 16.

```
void timers_isr(void* arg)
{

	channel1.counter++;
	PWM_counter++;



	if(PWM_counter >= PWM_counter_th)
	{

		if(pcnt_get_counter_value(PCNT_UNIT_0,&PWM_value)==ESP_ERR_INVALID_ARG)
		{
			ESP_LOGE(TAG,"PCNT GET COUNTER VALUE parameter error");
		}
		counter_clear();
		PWM_counter=0;
	}


	if(channel1.counter>=channel1.counter_th)
	{

		channel1.ch_raw=multisample(1,ADC1_CHANNEL1);
		channel1.ch_mv= get_mv(&calib,channel1.ch_raw);
		channel1.ch_sum=channel1.ch_sum-
		channel1.buffer_cha[channel1.ch_index]+channel1.ch_mv
		channel1.buffer_cha[channel1.ch_index]=channel1.ch_mv;
		channel1.ch_index++;
		if(channel1.ch_index==32)
		{

			channel1.ch_index=0;
		}
		channel1.counter=0;

	}

	TIMERG0.int_clr_timers.t1 = 1;
	TIMERG0.hw_timer[1].config.alarm_en = TIMER_ALARM_EN;
}
```

Nella mia routine di interrupt vado a campionare i vari canali dell’ADC, in questo caso soltanto quello relativo alla NTC collegata in ingresso. Visto che si cerca una certa flessibilità si è deciso di impostare una soglia di campionamento, ovvero la possibilità di cambiare ogni quanto effettuare la lettura del valore in ingresso. Ad esempio, il valore di default è 1mS, questo significa che andrò a leggere il valore una volta ogni millisecondo, se però ho a che fare con grandezze che variano lentamente nel tempo come la temperatura, posso decidere di aumentare questo tempo fino a cinque secondi.

Utilizzo, inoltre, un buffer di dimensione 32 per ottenere un valore medio più preciso, nel quale vado continuamente a sostituire valori man mano che campiono.

Il prossimo passo è lo start dei vari timer utilizzati dall' ADC in modo da poter cominciare a campionare, e lo start dei timer per il conteggio del duty-cycle. Inoltre, si inizializza anche il pulse counter per il calcolo della frequenza e del duty-cycle.

L' ultima inizializzazione è quella della partizione spiff:

```
void spiff_init(){

	esp_vfs_spiffs_conf_t conf = {
			.base_path = "/spiffs",
			.partition_label = NULL,
			.max_files = 5,
			.format_if_mount_failed = false
	};

	esp_err_t ret = esp_vfs_spiffs_register(&conf);
	if (ret != ESP_OK) {
		if (ret == ESP_FAIL) {
			ESP_LOGE(TAG, "Failed to mount or format filesystem");
		} else if (ret == ESP_ERR_NOT_FOUND) {
			ESP_LOGE(TAG, "Failed to find SPIFFS partition");
		} else {
			ESP_LOGE(TAG, "Failed to initialize SPIFFS (%s)",
			esp_err_to_name(ret));
		}
		return;
	}
	size_t total = 0, used = 0;
	ret = esp_spiffs_info(NULL, &total, &used);
	if (ret != ESP_OK) {
		ESP_LOGE(TAG, "Failed to get SPIFFS partition information (%s)",
		esp_err_to_name(ret));
	} else {
		ESP_LOGI(TAG, "Partition size: total: %d, used: %d", total, used);
	}
}
```

Questa partizione viene utilizzata per contenere la pagina WEB che deve essere visualizzata quando ci si collega all' indirizzo IP della scheda. Per prima cosa inizializzo la partizione spiffs con un massimo di cinque file, registro poi il file system virtuale (vfs) ed

infine controllo se tutto è andato a buon fine. In questa partizione al momento del flash del firmware verranno caricati pagina WEB e logo.

Infine, si creano quattro task che gireranno nel programma

```
xTaskCreate(&blink,"LED STATO",4000,NULL,5,NULL);
xTaskCreate(&errors_check,"ERRORI",4000,NULL,5,NULL);
xTaskCreate(&temp_calc,"Temperatura",4000,NULL,5,NULL);
xTaskCreate(&log_write,"LOG",4000,NULL,5,NULL);
```

L' ESP32 è basato su una CPU dual core, ma normalmente le operazioni vengono eseguite solamente sul Core 0, se si vuole utilizzare la funzionalità multitasking della scheda, è necessario affidarsi al sistema operativo FreeRTOS (Free Real Time Operating System), il cui compito è quello di schedulare le varie task secondo una priorità preimpostata. Questo ovviamente può portare ad una temporizzazione delle istruzioni non esattamente definita, ed è quindi necessario capire se le specifiche consentono o meno l'utilizzo del multitasking.

Le task in totale sono quattro:

1. LED di stato
2. Gestione errori
3. Calcolo della temperatura
4. Scrittura del LOG

Una volta che le task vanno in esecuzione, il sistema operativo le assegna a uno dei core che ha libero in quel momento, e si comporteranno a tutti gli effetti come dei LOOP.

LED STATO

Questa è la prima task:

```
void blink()
{
	while(1)
	{
		gpio_set_level(LED_STATO,LED_STATO_level);
		LED_STATO_level= !LED_STATO_level;

		if(WIFI_LED>0)
		{
			gpio_set_level(LED_WIFI,1);
		}
		else
		{
			gpio_set_level(LED_WIFI,0);
		}
		vTaskDelay(1000/ portTICK_PERIOD_MS);
	}

}
```

Il suo compito è far lampeggiare un LED per indicare il che il microcontrollore non è bloccato, ed inoltre accende il LED Wi-Fi se qualche dispositivo è connesso.

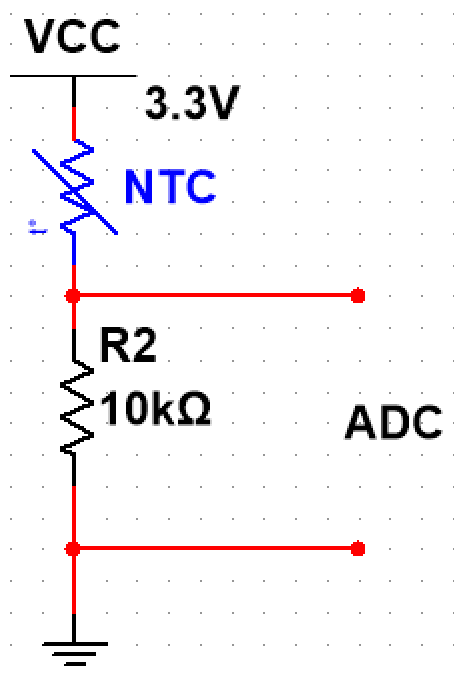
GESTIONE ERRORI

```
void errors_check()
{
	while(1)
	{
		int error_counter=0;
		for(int i=0;i<ERRORI_SIZE;i++)
		{
			if(errori[i]==1)
			{
				error_counter++;
			}

		}

		if(error_counter>=1)
		{
			gpio_set_level(LED_ERRORI,1);
		}
		else
		{
			gpio_set_level(LED_ERRORI,0);
		}
		vTaskDelay(500/ portTICK_PERIOD_MS);
	}
}
```

Questa è la task relativa alla gestione degli errori, per prima cosa faccio un controllo nel array contenente i bit di errore, se almeno uno di quei bit è 1, accendo il LED di segnalazione, altrimenti o lo tengo spento o lo spengo. Gli errori scattano se vado sopra il limite massimo o sotto il limite minimo relativo ad un dato canale ADC, nel nostro caso si intende un limite di temperatura avendo a che fare con un solo canale ed entrambi i limiti sono impostabili da pagina WEB.

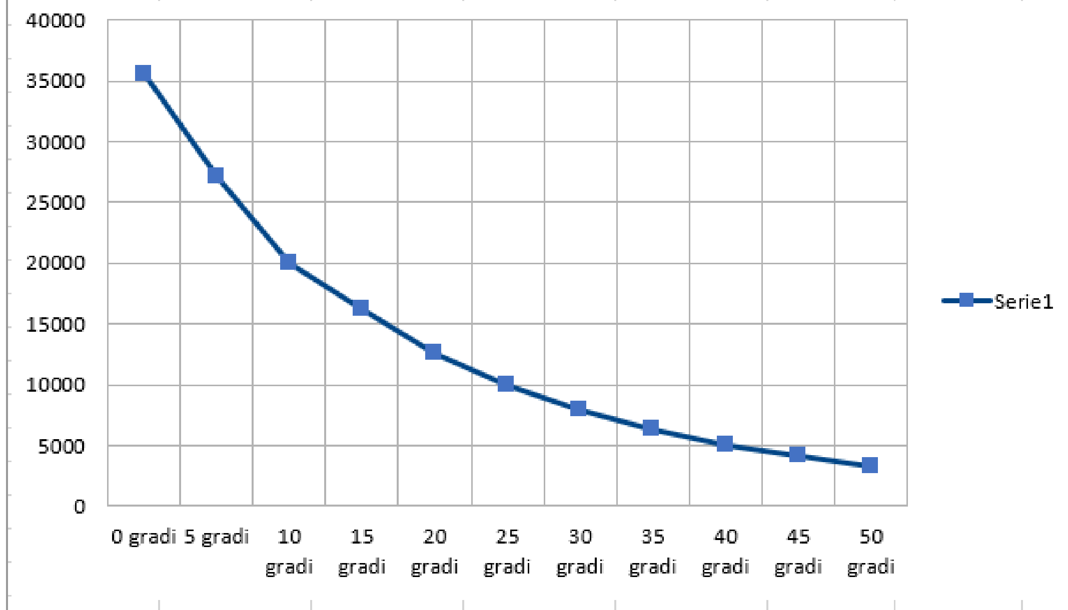
CALCOLO DELLA TEMPERATURA

Questa è la task dedicata alla conversione della lettura dell'ADC in temperatura vera e propria. All' ingresso del canale del convertitore abbiamo questo schema:



In pratica la NTC e la resistenza da 10K sono connesse in serie in modo da creare un partitore di tensione, visto che la NTC utilizzata a 25 gradi presenta una resistenza da 10K avremo precisamente una tensione di 1.65V in ingresso al nostro ADC a quella temperatura. Il problema però, è che la NTC non si comporta in maniera lineare al variare della temperatura quindi affidandomi al datasheet che mi forniva i vari valori di resistenza ad una data temperatura, sono andato a calcolare i volt che dovrei avere nel mio ingresso, ottenendo il seguente grafico:

10000	RT/R25	resistenza	volt
0 gradi	3.55	35500	2.57472527
5 gradi	2.7119	27119	2.41096743
10 gradi	2	20000	2.2
15 gradi	1.62	16200	2.04045802
20 gradi	1.26	12600	1.83982301
25 gradi	1	10000	1.65
30 gradi	0.79	7900	1.45642458
35 gradi	0.63	6300	1.27546012
40 gradi	0.5	5000	1.1
45 gradi	0.41	4100	0.95957447
50 gradi	0.33	3300	0.81879699



Il datasheet forniva il rapporto RT/R25 che significa il valore che ha la NTC alla data temperatura rispetto a quello che ha a 25 gradi. Ad esempio, a 10 gradi si ha un RT/R25=2 il che significa che la resistenza sarà il doppio rispetto a quella a 25 gradi, è anche possibile vedere che al calare della temperatura la resistenza aumenta, quindi si avrà una caduta di tensione maggiore nella NTC e un valore minore in ingresso dell'ADC. Nel caso invertissimo la resistenza con la NTC avremmo l'effetto contrario, ovvero all' aumentare della temperatura avremmo un abbassamento della tensione in ingresso. Si è optato per l'altra opzione per mantenere la conversione più logica. Per conoscere quindi la temperatura effettiva, è necessari effettuare una linearizzazione della curva.

Per farlo basta pensare che R25=10KΩ, che è esattamente il valore dell'altra resistenza, possiamo quindi andare a scrivere:

$$\frac{3.3V * R25}{R25 + RT} = \frac{3.3V}{1 + \frac{RT}{R25}} = V$$

Otteniamo quindi:

$$\frac{RT}{R25} = \frac{3.3}{V} - 1$$

A questo punto visto che il datasheet ci fornisce i valori di RT/R25 possiamo sempre sapere in quale parte della curva ci troviamo.

```
void temp_calc()
{
	while(1){

		for(int i=0;i<sizeof(temp_array);i++)
		{
			channel1.ch_avg=channel1.ch_avg+temp_array[i];
		}
		channel1.ch_avg=channel1.ch_avg/20;
		temp_index++;
		int errore_sel=0;
		float Vmax=0.0;
		float Vmin=0.0;
		float N_Gradi=0.0;
		float Res_ratio = (3300.00/(channel1.ch_mv-20.00))-1.00;
		if(Res_ratio<=ZERO_GRADI && Res_ratio>CINQUE_GRADI)
		{   Vmax =  ZERO_GRADI;
		Vmin =  CINQUE_GRADI;
		}
		if(Res_ratio<=CINQUE_GRADI && Res_ratio>DIECI_GRADI)
		{   Vmax =  CINQUE_GRADI;
		Vmin =  DIECI_GRADI;
		N_Gradi=1.0;
		}
		if(Res_ratio<=DIECI_GRADI && Res_ratio>QUINDICI_GRADI)
		{   Vmax =  DIECI_GRADI;
		Vmin =  QUINDICI_GRADI;
		N_Gradi=2.0;
		}
		if(Res_ratio<=QUINDICI_GRADI && Res_ratio>VENTI_GRADI)
		{   Vmax =  QUINDICI_GRADI;
		Vmin =  VENTI_GRADI;
		N_Gradi=3.0;
		}
...
```

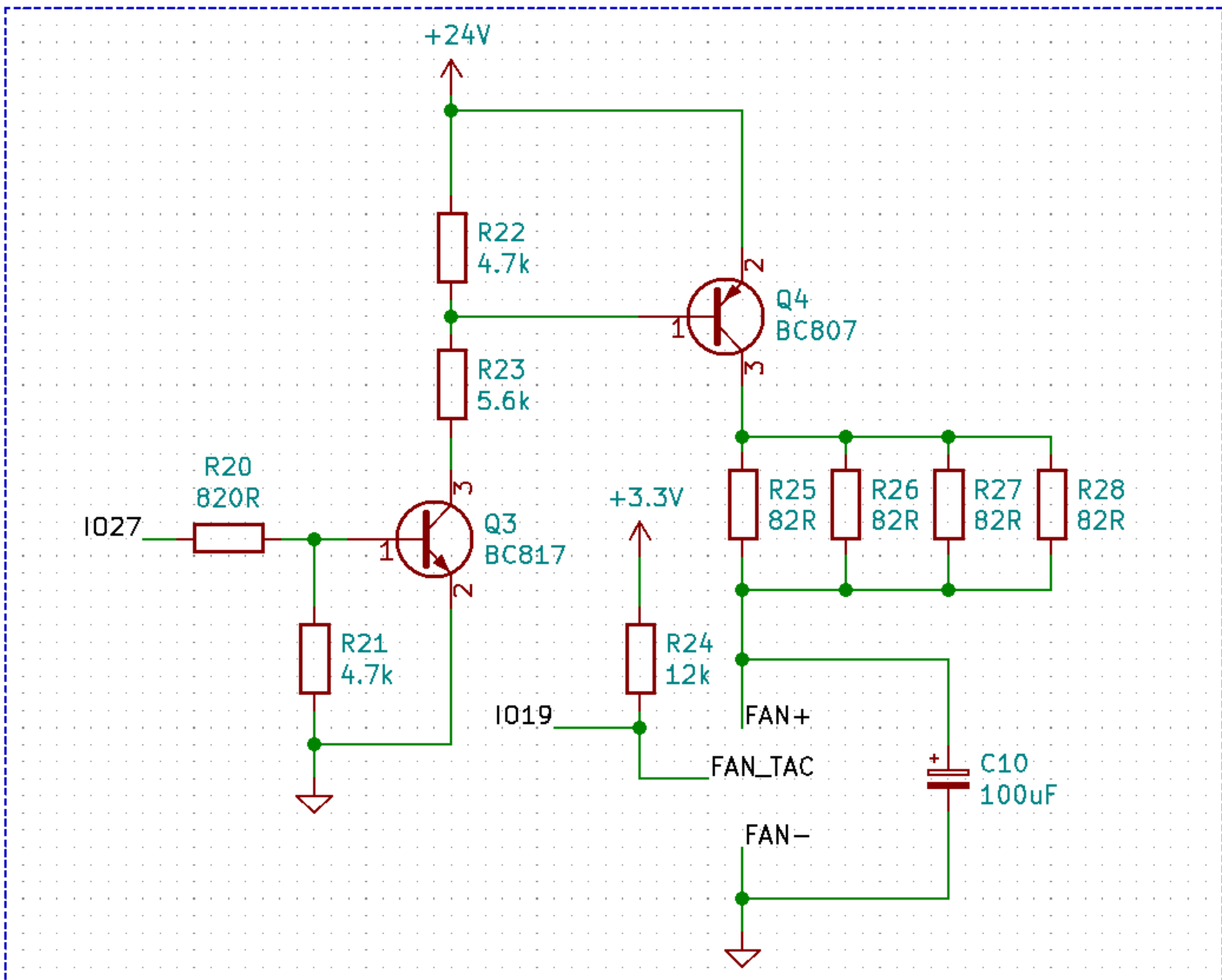
Si calcola poi la tensione massima e minima che si ha nei vari tratti della curva e si calcola quindi la temperatura con un semplice rapporto.

Nella seconda part della task si va invece a pilotare la ventola in base alla temperatura rilevata:

```
if(temp>channel1.ch_max)
		{
			channel1.ch_max=temp;
		}

		if(temp<channel1.ch_min)
		{
			channel1.ch_min=temp;
		}

		if(channel1.ch_avg<28)
		{
			PWM_STOP();

		}
		else if(channel1.ch_avg<30)
		{
			PWM_START();
	ledc_set_fade_with_time(LEDC_HIGH_SPEED_MODE,LEDC_CHANNEL_1,128,1000);

	ledc_fade_start(LEDC_HIGH_SPEED_MODE,LEDC_CHANNEL_1,LEDC_FADE_NO_WAIT);
		}

		else if(channel1.ch_avg<35)
		{


	ledc_set_fade_with_time(LEDC_HIGH_SPEED_MODE,LEDC_CHANNEL_1,200,500);

	ledc_fade_start(LEDC_HIGH_SPEED_MODE,LEDC_CHANNEL_1,LEDC_FADE_NO_WAIT);
		}

		else
		{


	ledc_set_fade_with_time(LEDC_HIGH_SPEED_MODE,LEDC_CHANNEL_1,256,500);

	ledc_fade_start(LEDC_HIGH_SPEED_MODE,LEDC_CHANNEL_1,LEDC_FADE_NO_WAIT);
		}
		if(errore_sel==1){

	if(channel1.ch_avg>=channel1.ch_th_max||channel1.ch_avg<=channel1.ch_th_min)
			{
				errori[0]=1;
			}
		}
```

Utilizzando il LED CONTROL andiamo a variare in modo lineare la velocità della ventola che comunque parte solo dopo aver raggiunto una certa temperatura di soglia di 28 gradi, inoltre se si nota che la temperatura media ha superato quella di soglia si mette il bit di errore relativo a 1.

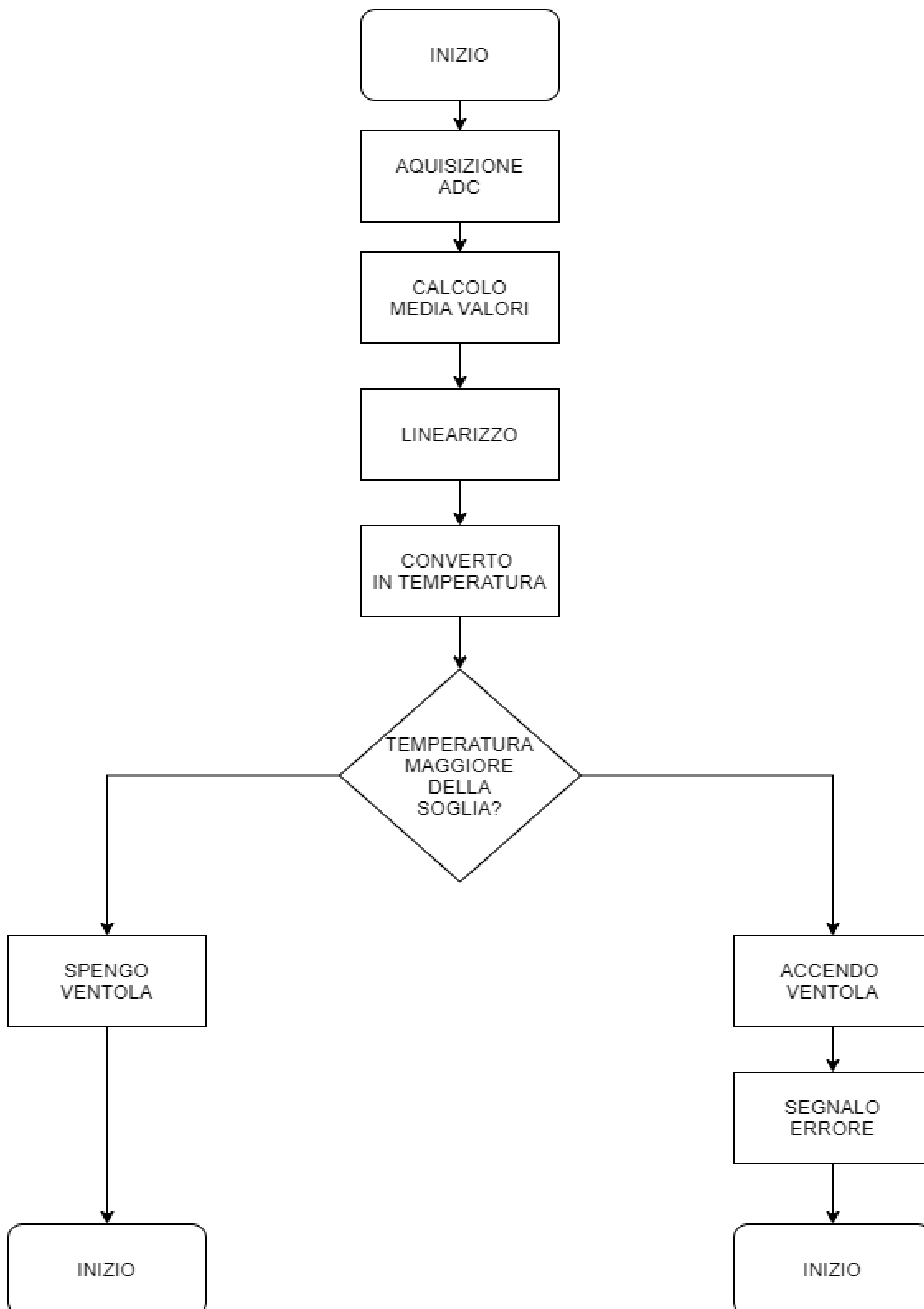
La ventola è collegata nella seguente maniera:



La GPIO 27 è quella responsabile del PWM della ventola mentre la 19 è quella responsabile dell’ingresso PWM per il calcolo della velocità della ventola, il condensatore in parallelo alla ventola è stato inserito per facilitare il pilotaggio PWM di ventole che non funzionano necessariamente in Pulse Width Modulation.

Nella seguente flow chart si riassume la task, per prima cosa parto dall’ acquisizione del valore dell’ ADC, seguita dal calcolo della media e dalla linearizzazione della curva di resistenza, una volta che si sa in quale parte della curva in cui ricade il valore si effettua la

conversione in temperatura e successivamente si effettua il controllo sulla soglia, se la temperatura è maggiore allora si accende la ventola e si segnala l' errore , se è minore si spegne o si lascia spenta.



LOG WRITE

Il compito di questa task è quello di scrivere il file di LOG al cui interno vengono salvati tutti i valori riguardanti pin digitali e analogici sia di INPUT che di OUTPUT e anche i valori degli errori.

```
void log_write()
{
	while(1){
		FILE *f =fopen(nome_file,"a");
		if(f!=NULL)
		{
	fprintf(f,"%d;%d;%d;%2.0f;%d;%d;%d;%f;%d;\n",channel1.ch_max,channel1.ch_min,chan
	nel1.ch_avg,temp,digital_pin_level[1],gpio_get_level(DIGITAL_INPUT_1),PWM_value,D
	C,errori[1]);
			stat(nome_file,&st
			int file_size=st.st_size;
			fclose(f);

			if (file_size>=4000000)
			{
				new_file();
			}
		}
		else
		{
			ESP_LOGE(TAG,"File open error");
			fclose(f);
		}
		for(int i=0;i<50;i++)
		{
			loghtml[i]=0;
		}

	sprintf(loghtml,"%d;%d;%d;%2.0f;%d;%d;%d;%f;%d;\n",channel1.ch_max,channel1.ch_mi
n,channel1.ch_avg,temp,digital_pin_level[1],gpio_get_level(DIGITAL_INPUT_1),PWM_value,D
C,errori[1]);
		vTaskDelay(1000/ portTICK_PERIOD_MS);
	}

}
```

Per prima cosa si apre il file, poi si scrivono i valori al suo interno ed infine si chiude il file. Se la dimensione del file supera i 4MB allora se ne genera uno nuovo, questo per prevenire errori in apertura e soprattutto per evitare lunghi tempi di attesa.

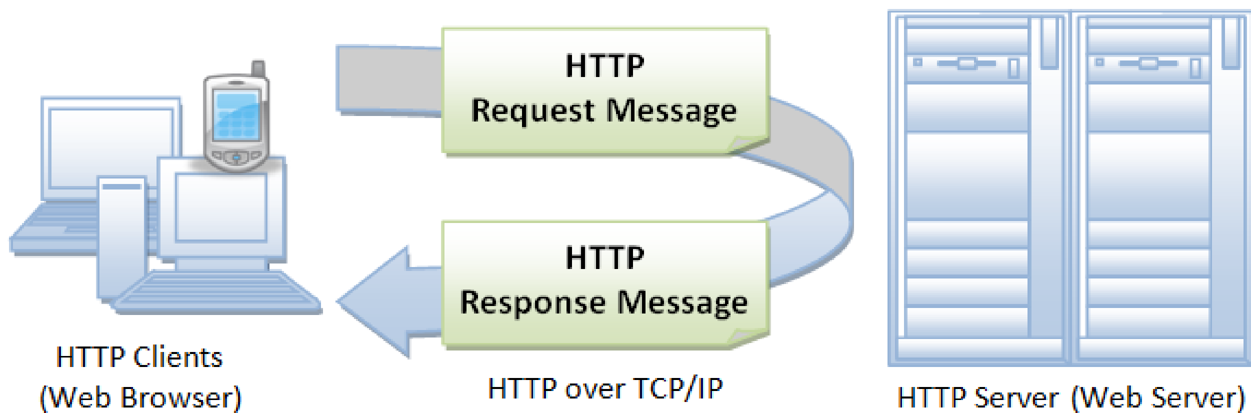
Il LOG viene salvato in formato CSV in modo da poter essere aperto con Excel, ed inoltre viene anche generata una riga di LOG che sarà poi inviata alla pagina web quando richiesto.

API

Sono state generate diverse funzioni per la trasmissione di dati da e verso la scheda, il fine principale è quello di comunicare con la pagina WEB ma è anche possibile utilizzarle per comunicare direttamente con altre periferiche. In totale sono presenti tredici funzioni funzione che costituiscono gran parte del corpo del firmware, vista la loro lunghezza non sarà riportato codice ma se ne darà soltanto una spiegazione.

Visto che la scheda utilizza un server http, tutte le funzioni devono rispettare i protocolli sia per quanto riguarda l'invio che per quanto riguarda la ricezione di informazioni.

In pratica all' interno della nostra rete si ha un server, che è la nostra scheda ESP32 e dei client, che possono essere sia PC che smartphone che altre schede. Il funzionamento è abbastanza semplice [1]:



[1]

In pratica il client invia richieste al server, che possono essere di due tipi:

- GET
- POST

Il tipo di richiesta viene chiamato metodo, in realtà ne esistono anche altri come HEAD,PUT o CONNECT ma non sono stati utilizzati nel progetto.

La richiesta GET serve a recuperare informazioni dal server utilizzando la URI prestabilita, questo metodo dovrebbe essere utilizzato solo per recuperare dati dal server e non dovrebbe mai avere altri effetti su di essi.

La richiesta POST è invece solitamente utilizzata per l'invio di dati verso il server, che possono essere upload o informazioni di qualsiasi tipo. Una volta che il server riceve una data richiesta, risponde di conseguenza mandando come risposta dei dati oppure impostando qualcosa all' interno della scheda come delle uscite digitali o dei parametri riguardanti l'ADC.

API_ANALOG_GET_STATUS

L' obbiettivo di questa prima funzione è l'invio di tutte le informazioni riguardanti l'ADC, che vengono tutte inserite in una struttura JSON la quale viene convertita in stringa ed inviata come risposta alla richiesta. Contiene dati come l'attenuazione, i bit di risoluzione e i limiti massimi e minimi entro quale far scattare l'errore relativi ad un canale dell'ADC.

API_DL_GET

Questa funzione è invece utilizzata per scaricare l'ultimo file di log salvato nell' SD, l'idea iniziale era quella di creare un server FTP che potesse affiancare quello http per poter inviare o ricevere file direttamente dalla micro-SD senza bisogno di rimuoverla dalla scheda. Purtroppo, l'ESP32 al momento dello sviluppo del progetto non supportava questa feature, si è quindi creata questa funzione per permettere comunque di scaricare almeno l'ultimo file salvato.

API_ANALOG_GET

Questa parte di codice ha il compito di rispondere alla richiesta riguardante i valori acquisiti dall' ADC, si invia quindi come risposta una struttura JSON contenente il nome del canale richiesto il valore massimo misurato, il minimo, la media e il valore attuale.

API_ANALOG_POST

Questa funzione è la risposta alla richiesta di configurazione dell'ADC inviata con metodo POST, si riceve quindi una struttura JSON dalla quale vengono estrapolati i valori di bit e attenuazioni relativi ad un dato canale, si riconfigura quindi l'ADC con i parametri ricevuti e si invia la risposta di OK se l'operazione è andata a buon fine oppure si invia un codice di errore.

API_RGB_POST

Questa handler serve a gestire il LED RGB presente nella scheda, si riceve una struttura JSON contenete delle coppie nome-valore relative ai canali RED, GREEN e BLUE i valori vanno da 0 a 256 e corrispondono a l'intensità minima e massima del rispettivo canale. Si effettua puoi un fading nei canali modificati, che consiste in una modifica lineare del dato canale che produce una variazione nel colore.

API_ANALOG_TC_POST

Questa funzione serve a modificare il tempo ogni quale si va a campionare il segnale su un dato canale dell'ADC, si riceve in ingresso un JSON contenente una coppia nome-valore contenente il nome del canale da impostare e il nuovo tempo che viene quindi impostato. Nel caso la struttura JSON non sia corretta o se qualcosa non va a buon fine si invia un errore al client, altrimenti si manda un OK.

API_ERRORI_GET

In questa parte di crea una struttura JSON da inviare al client contenente coppie chiave-valore corrispondenti al nome del canale e il suo effettivo stato di errore.

API_PWM_GET

Qui si calcolano la frequenza e il duty-cycle del segnale PWM in ingresso, nel nostro caso per calcolare la velocità della ventola e si crea poi una struttura JSON contenente i valori della frequenza e del duty-cycle e la si invia come risposta al client.

API_SET_PIN_LEVEL_POST

In questa funzione si riceve dal client una struttura JSON contenente il nome e il valore di un dato OUTPUT digitale e si procede poi con l'update dei valori, nella scheda è possibile provare questa funzione tramite un LED posto nell' uscita digitale.

API_GET_PIN_LEVEL_GET

Questa funzione è necessaria per conoscere i valori degli output digitali, si invia al client una struttura contenente la coppia chiave valore delle uscite digitali. Se ad esempio apriamo la nostra pagina WEB dopo qualcun altro, essa deve essere in grado di aggiornarsi con gli attuali valori di tutti gli OUTPUT della scheda quindi si invia questa richiesta e si ricevono come risposta i valori.

API_MAIN_HTTP

Questa è la prima funzione che viene chiamata e serve a caricare la pagina WEB nel dispositivo al momento dell'apertura della URL principale, la pagina è salvata in una locazione di memoria della scheda.

API_ERROR_RESET

Questa handler serve a resettare gli status di errore dei vari canali ADC, nella pagina web è presente un pulsante che alla pressione invia una richiesta la cui scheda risponde resettando l'array contenente gli errori.

API_SET_LIMIT_POST

Quest' ultima funzione serve a modificare i limiti entro quale far scattare l'errore nella scheda, è possibile impostare un limite superiore ed un limite inferiore

PAGINA WEB

ESP32 DATA PAGE

DIGITAL

Name	Value	EN
DIGITAL OUTPUT 1	HIGH	☐

Name	Value
DIGITAL INPUT1	HIGH

PWM

PWM	FREQUENCY	DC
RPM FAN	0	0

ANALOG

Name	MAX C	MIN C	AVG C	ACT C
TEMPERATURA	0	0	0	0

ADC SET CH: 1 Bits 9 Att: 0 SET

ADC TC CH: 1 TC: SET

ERRORS

Name	State
TEMPERATURA	LOW

RESET

ADC STATUS

Attenuation:0=0dB, 1=2.5dB, 2=6dB, 3=11dB

Name	Bits	Attenuation	Limit Max	Limit Min
Analog1	LOW	LOW	LOW	LOW

ADC LIMIT CH: 1 Limit Max: Limit Min: SET

LED COLOR

LOG

MAX C	MIN C	AVG C	ACT C	DIGI OUT1	DIGI IN1	PWM F	PWM DC	ERR	DATE

Come già visto la pagina WEB si presenta in questo modo, è stata creata utilizzando il linguaggio HTML/CSS e Javascript. I primi due sono utilizzati per creare la struttura e la grafica della pagina, mentre il Javascript consente di comunicare con la scheda utilizzando le funzioni della API.

La parte DIGITAL serve a gestire gli INPUT e OUTPUT digitali, la prima tabella mostra lo stato dell'unica uscita presente nel prototipo e consente attraverso la spunta di modificare il suo stato ATTIVO/DISATTIVO. La seconda tabella invece mostra semplicemente lo stato dell'unico ingresso digitale presente nella scheda.

La parte PWM mostra in questo caso la frequenza del segnale presente in ingresso che corrisponde alla velocità di rotazione della ventola e il duty-cycle che in questo caso non è utilizzato per nessun calcolo.

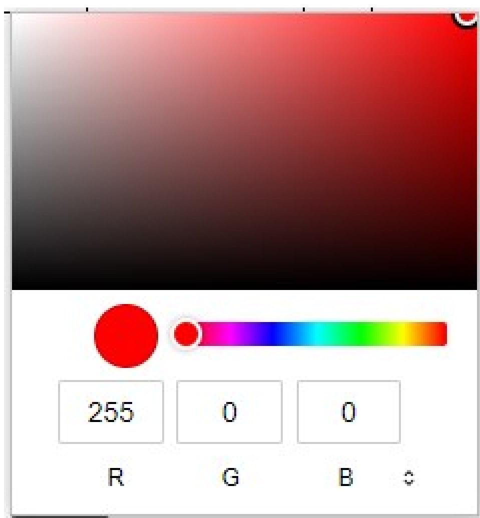
La sezione ANALOG mostra i valori relativi alla temperatura della NTC e consente di modificare i valori di attenuazione e risoluzione di un canale ADC e il tempo di campionamento.

In ADC STATUS si verificano i parametri di configurazione relativi all' ADC e inoltre, è possibile impostare i vari i limiti relativi ai canali presenti. In questo modo è possibile modificare la soglia oltre la quale si verifica l'errore.

ERRORS invece mostra lo status relativo agli errori, presentando il nome e il valore legato ad esso. Ad esempio, se la temperatura media registrata dalla NTC dovesse andare sopra il limite massimo il valore cambia da LOW ad HIGH e la casella diventa rossa. Con il pulsante RESET è possibile azzerare lo status.

La parte LOG mostra tutti i valori relativi alla scheda secondo per secondo, inoltre è anche presente la data e l'ora di ogni riga, cosa non presente nel LOG implementato nella SD vista la mancanza di un RTC.

L' ultima parte è il LED COLOR che ha il compito di modificare il colore del LED RGB connesso alla scheda. Alla pressione del bottone compare una palette dalla cui è possibile impostare il colore direttamente o inserire il suo valore in RGB:



La pagina ha il seguente funzionamento:

Per prima cosa si costruisce la parte grafica, contenente le varie tabelle, bottoni ed etichette, si va poi a creare le variabili utilizzate nel Javascript e le si inizializza con i parametri di default.

Successivamente si va a comunicare con la scheda, questa cosa avviene automaticamente ogni mezzo secondo per le funzioni che implementano il metodo GET oppure ogni volta che si preme il bottone per il metodo POST e si aggiornano poi i parametri di conseguenza.

Per quanto riguarda il codice, si può dire che la pagina è divisa in due parti. Nella prima si definisce tutta la parte grafica creando tabelle, bottoni ed etichette:

```
1   <!DOCTYPE html>
2   <html>
3   <body>
4     <style>
5     table, th, td {
6       border: 1px solid black;
7       padding:5px;
8     }
9
10    #digital_input
11    {
12      position: relative;
13      top:-60px;
14      left:250px;
15    }
16
17    #pwm_tab
18    {
19      position: relative;
20      top:-186px;
21      left:448px;
22      margin-bottom: -185px;
23    }
24    #pwm_header
25    {
26      position: relative;
27      top:-180px;
28      left:525px;
29    }
30
31    #adc_set_form
32    {
33      position: relative;
34      top:5px;
35      left:5px;
36    }
```

Si creano quindi tutte le parti che compongono la pagina utilizzando codice HTML e CSS.

Nella seconda parte si implementa il codice in Javascript che consente la comunicazione con la scheda, in particolare viene utilizzata la funzione fetch che consente di inviare richieste http alla data URI con il metodo specificato.

```
546      function pwm_fetch()
547      {
548        fetch('http://192.168.4.1/api/pwm')
549        .then(pwm => pwm.json())
550        .then(data => {
551          var frequency_value=data.frequency;
552          var rpm=frequency_value*30;
553          var duty_cycle=data.duty_cycle;
554          var pwm_tab_temp=document.getElementById("pwm_tab");
555          pwm_tab_temp.rows[1].cells[1].innerHTML=rpm;
556          pwm_tab_temp.rows[1].cells[2].innerHTML=duty_cycle.toFixed(2);
557
558        })
559      }
```

Ovviamente sia che si inviino i dati sia che si ricevono, si utilizzano struttura JSON per consentire una più facile comunicazione.

CONCLUSIONI

MIGLIORAMENTI POSSIBILI

Sicuramente è possibile effettuare diversi miglioramenti allo sviluppo della scheda, primo fra tutti il miglioramento dell’interfaccia della pagina WEB che risulta molto spartana e poco chiara. Con un po' più di tempo ed una conoscenza leggermente più approfondita del linguaggio HTML e CSS si potrebbe ottenere decisamente una pagina migliore e si potrebbe inserire grafici per la visualizzazione nel tempo di vari valori. Ed è poi sicuramente possibile rendere la scheda molto più modulare inserendo la possibilità, ad esempio, di selezionare diversi tipi di sensori dalla pagina WEB in modo da modificare in tempo reale e senza dover mettere mano al firmware il tipo di dato acquisito.

È possibile poi migliorare il multitasking implementando delle routine che migliorano l'utilizzo della RAM, fortunatamente il grande quantitativo di memoria presente ha reso molto semplice l'allocazione.

STUDIO SU SCHEDE SIMILI

Nella parte finale del tirocinio ho effettuato uno studio riguardante possibili schede sostitutive all' ESP32 le specifiche importanti erano il Wi-Fi integrato e la capacità di gestire server http, oltre che avere una potenza almeno discreta ed un prezzo simile.

WGML160P

Sviluppato dalla Silicon Labs e basato su un ARM Cortex M4 a 72MHz, presenta 512KB di RAM e 2MB di ROM flash. È dotato di 31GPIO e un ADC a 12bit, inoltre è dotato di un'interfaccia USB 2.0 full speed, è purtroppo sprovvisto di Bluetooth. Prezzo di circa 7.50€.

CC3220MODx

Questi sono una famiglia di microcontrollori sviluppati dalla TI, basati su Cortex M4 a 80MHz e con 256KB di RAM. Presentano una ROM Flash interna da 1MB e 27GPIO. Non hanno Bluetooth e l'ADC è a solo quattro canali con risoluzione a 21Bit. Il prezzo si aggira sugli 8€.

ATMEL ATWINC1500B

Prodotto dalla ATMEL è dotato di una CPU Cortus APS32 a 32bit, con clock da 32MHz. Ha una RAM da 224KB e una ROM da 128KB, presenta però soltanto 9GPIO e non è dotato né di ADC né di DAC. Ha entrambe le connettività Wi-Fi e Bluetooth con prezzo intorno agli 8€.

ATSAMW25

Prodotto dalla Microchip e provvisto di Cortex M0 a 48MHz presenta delle spicifiche piu basse rispetto agli altri, con 32KBdi RAM e 256KB di ROM, le GPIO sono 21 in totale ed è

provvisto sia di ADC che di DAC entrambi a 12bit di risoluzione. Non ha Bluetooth e il prezzo si aggira sui 9€.

Questi sono i candidati che ho trovato per sostituire l'ESP32, sono praticamente tutti basati su architettura ARM che è sempre più diffusa anche nel mondo dei microcontrollori e sono tutti in grado almeno teoricamente di sostituire l'ESP32 con le specifiche date nella realizzazione del progetto.

CONCLUSIONI GENERALI

La scheda sviluppata nelle 150 ore di tirocinio ha decisamente superato le mie aspettative, si sono aggiunte diverse funzionalità non previste all' inizio e si è anche realizzato un prototipo funzionante.

Ringrazio l'impresa Loccioni per avermi concesso la possibilità di effettuare il tirocinio da loro e per l'aiuto fornitomi durante lo sviluppo del progetto, e i professori Turchetti Claudio e Falaschetti Laura per aver accettato di farmi da relatori e per l'aiuto fornitomi durante la stesura della tesi.