

Università Politecnica delle Marche
Dipartimento di Ingegneria dell'Informazione

Facoltà di Ingegneria Informatica e dell'Automazione



**Implementazione di un sistema di acquisizione dati per il
monitoraggio ambientale**

Professori:

Prof. Marcozzi Daniele
Prof. Dragoni Aldo Franco

Studenti:

Ciuffreda Silvia
Liberatore Luca
Serafini Andrea

ANNO ACCADEMICO 2023/2024

Indice

1 Stato dell'arte	3
1.1 Funzionamento Generale	3
2 Strumenti utilizzati	4
2.1 Componenti Hardware	4
2.1.1 Le Board	4
2.1.2 Sensori	4
2.2 Componenti Software	7
2.3 Linguaggi di Programmazione	7
3 Configurazione dispositivi	8
3.1 Esp32	8
3.1.1 Arduino IDE	8
3.1.2 Configurazione dell'Arduino IDE per ESP32	8
3.1.3 Installazione delle librerie in Arduino IDE	9
3.2 Raspberry Pi 4	10
3.2.1 Installazione di L.A.M.P. e librerie necessarie	10
3.3 Ambiente di sviluppo	13
3.4 Framework Flask	14
4 Sistema di acquisizione dati	15
4.1 Descrizione generale sul sistema hardware implementato	15
4.2 Descrizione del codice implementato	17
4.2.1 Inizializzazione e blocco <code>setup()</code>	17
4.2.2 Task per l'acquisizione dei dati	19
4.2.3 Task per l'invio dei dati	20
4.2.4 Task per la sincronizzazione del modulo RTC	20
5 Sistema di elaborazione e divulgazione dei dati acquisiti	21
5.1 Panoramica generale	21
5.2 Sincronizzazione del modulo RTC	21
5.3 Creazione del database	22
5.4 Visualizzazione della webapp	22
5.5 Implementazione del chatbot	23
5.5.1 Esempio generazione grafico - disponibilità di dati	24
5.5.2 Esempio generazione grafico - mancanza di dati	25
5.6 Esecuzione dei daemon	25
6 Conclusioni	29
7 Appendice	30
7.1 <code>requirements.txt</code>	30
7.2 <code>esp32freeRTOS.ino</code>	30
7.3 <code>sync_timer.py</code>	34
7.4 <code>database.py</code>	35
7.5 <code>app.py</code>	36
7.6 <code>index.html</code>	37
7.7 <code>style.css</code>	38
7.8 <code>chatbot.py</code>	39

Elenco delle figure

1	Schema a blocchi	3
2	ESP32 Piedinatura	4
3	RTC	5
4	BMP280	5
5	Display OLED	6
6	Fan	7
7	Configurazioni URL aggiuntivo Arduino IDE	8
8	Installazione board ESP32	9
9	Installazione della libreria relativa al modulo <i>RTC</i>	10
10	Schermata iniziale di <i>Raspberry Pi Imager</i>	10
11	Configurazione delle impostazioni personalizzate in <i>Imager</i>	11
12	Descrizione dei campi che costituiscono il database creato con MariaDB	13
13	Schema del cablaggio	15
14	Schema di collegamento del pin dedicato al segnale del tachimetro.	16
15	Schema riassuntivo dell'architettura hardware-software.	17
16	Visualizzazione grafica della webapp	23
17	Esempio generazione grafico e valori statistici - dati disponibili	25
18	Esempio generazione grafico e valori statistici - mancanza di dati	25
19	Logica di attivazione dei servizi.	26

1 Stato dell'arte

Nel contesto del corso di *Sistemi Operativi Dedicati*, il presente progetto mira a sviluppare un sistema completo per il monitoraggio, la visualizzazione e il controllo dei parametri ambientali. Il sistema utilizza una combinazione di hardware e software per acquisire, elaborare e visualizzare i dati raccolti da vari sensori.

1.1 Funzionamento Generale

Il sistema è composto da diversi componenti:

- scheda ESP32,
- Raspberry Pi 4,
- sensore di temperatura e pressione BMP280,
- ventola,
- display OLED,
- modulo RTC

La scheda ESP32 è responsabile dell'acquisizione dei dati dal sensore BMP280, del controllo della ventola tramite segnale PWM, della sincronizzazione del modulo RTC tramite un comando inviato periodicamente dalla RPi e della visualizzazione delle informazioni sul display OLED.

La Raspberry Pi funge da server IoT, riceve i dati acquisiti dall'ESP tramite comunicazione seriale e mette a disposizione degli utenti i dati raccolti e archiviati attraverso un'interfaccia web, consentendo loro di visualizzare le tendenze nel tempo e di eseguire analisi approfondite. Inoltre, gli utenti possono richiedere dati specifici tramite comandi al bot Telegram "SensorDataRPI_bot", che risponde fornendo una visualizzazione grafica dei parametri richiesti e i relativi valori statistici.

In linea generale, il sistema monitora costantemente i parametri ambientali tramite il sensore BMP280, attivando la ventola per raffreddare il sistema quando la temperatura supera una soglia predefinita, con velocità proporzionale allo scostamento della temperatura attuale rispetto a quella desiderata.

Le informazioni acquisite sulla temperatura, la pressione e la velocità della ventola vengono visualizzate sul display OLED e inviate alla Raspberry Pi con il relativo timestamp. La Raspberry Pi memorizza tali dati all'interno di un database e se la temperatura supera una soglia critica genera notifiche di alert che vengono inviate tramite il bot Telegram "SensorDataRPI_bot" a tutti gli utenti registrati.

In sintesi, il progetto si propone di offrire un sistema completo e integrato per il monitoraggio e il controllo dei parametri ambientali, con un focus particolare alla semplicità d'uso, all'affidabilità e alla reattività alle situazioni critiche.

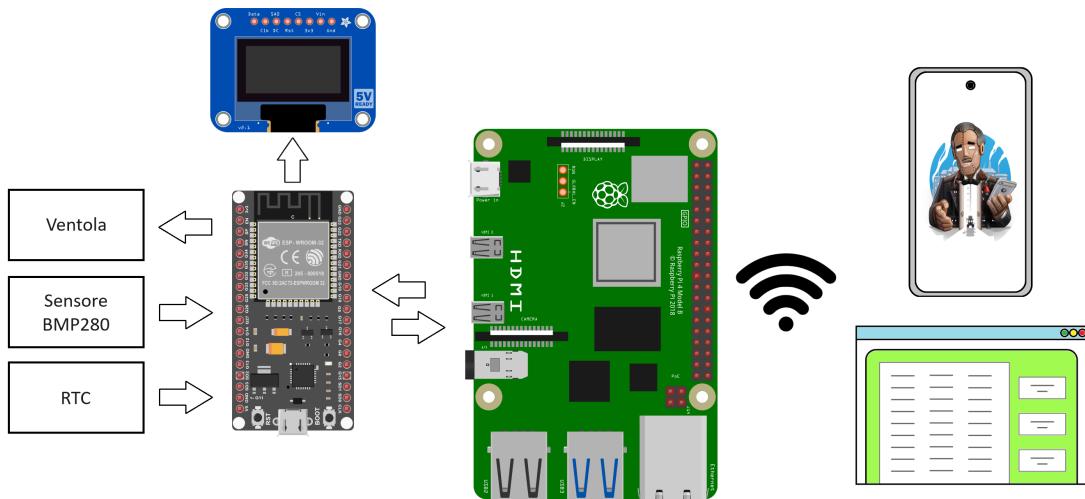


Figura 1: Schema a blocchi

2 Strumenti utilizzati

2.1 Componenti Hardware

Nell'implementazione del progetto sono stati impiegati diversi componenti hardware, suddivisi in due macro categorie: le schede e i sensori. Per il corretto collegamento dei jumper ai pin interessati, è stata utilizzata una breadboard come base per lo schema globale del cablaggio.

2.1.1 Le Board

Come anticipato nel capitolo precedente le due schede utilizzate per la parte real-time e per la funzionalità server IoT sono, rispettivamente, l'ESP32 e la Raspberry Pi:

- **ESP32:** microcontrollore utilizzato per acquisire dati dai sensori e gestire le interazioni con l'ambiente circostante. È alimentato a 3.3V e dispone di vari pin con funzionalità specifiche, tra cui:

- **memoria SPI:** alcuni pin sono riservati per la memoria SPI;
- **canali I2C:** dispone di due canali I2C, anche se qualsiasi pin è configurabile come SDA o SCL;
- **modalità deep sleep:** supporta la modalità deep sleep per il risparmio energetico;
- **PWM:** tutti i pin utilizzabili come uscite possono essere configurati come pin PWM, ad eccezione dei GPIO da 34 a 39.

Inoltre, è consigliabile evitare l'uso dei seguenti pin: GPIO 1, GPIO 3, GPIO 5, GPIO 6 fino a GPIO 11 (collegati alla memoria flash SPI integrata), GPIO 14 e GPIO 15.

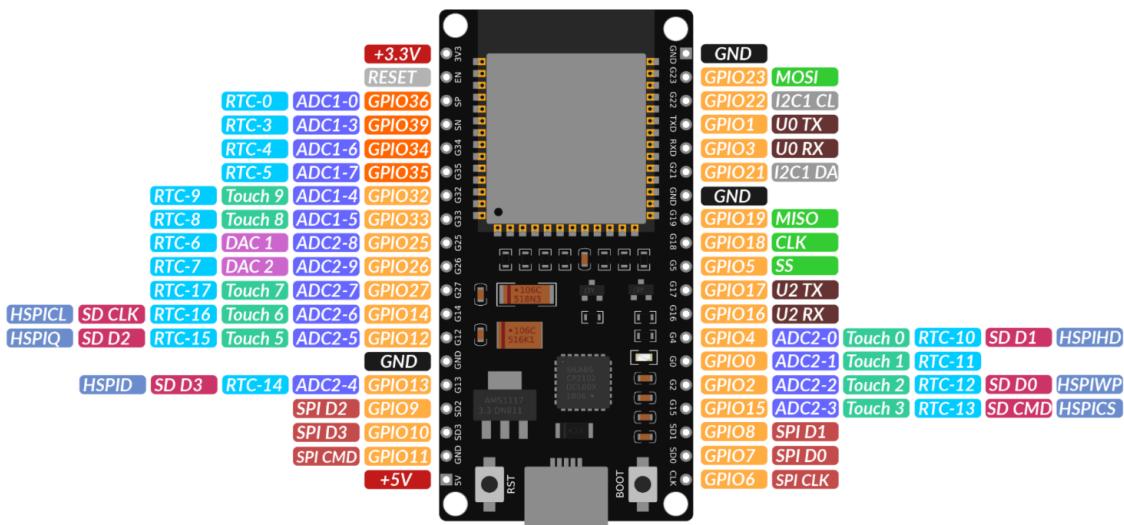


Figura 2: ESP32 Piedinatura

- **Raspberry Pi 4B (RPi):** scheda di sviluppo utilizzata come server IoT per la gestione dei dati e delle comunicazioni con i dispositivi connessi. Questa scheda computer ad alte prestazioni è progettata per una vasta gamma di progetti informatici e, nel caso preso in esame, è dotata di 4GB; questo le conferisce un notevole potenziale di elaborazione e multitasking per soddisfare le esigenze di applicazioni complesse.

2.1.2 Sensori

Prima di procedere con l'integrazione dei singoli sensori nel sistema, sono stati condotti dei test per verificarne il corretto funzionamento. Questi test sono stati eseguiti utilizzando gli sketch di

2.1 Componenti Hardware

esempio forniti dalle librerie ufficiali di ciascun sensore. Tale approccio ha permesso di assicurare che ciascun componente hardware rispondesse correttamente e fosse pronto per essere integrato nel sistema.

Inoltre, gli indirizzi I2C dei singoli dispositivi sono stati lasciati invariati: 0x77 per il BMP280, 0x3D per lo schermo OLED e 0x68 per il modulo RTC.

Nel dettaglio si elencano i singoli sensori e le loro principali caratteristiche:

- **RTC (PCF8523):** Sensore utilizzato per generare il riferimento temporale a cui verranno associati i dati, sincronizzabile attraverso comandi inviati dalla RPi.

Il modulo RTC è dotato di cinque pin principali:

- **Vcc:** pin che fornisce l'alimentazione al chip. Il chip può essere alimentato con 3-5VDC senza alcun regolatore integrato;
- **GND:** pin di terra;
- **SCL:** pin di clock I2C;
- **SDA:** pin di dati I2C;
- **SQW:** pin utilizzato per l'uscita di un'onda quadra se è abilitata la funzionalità di contatore.

E' consigliato utilizzare la libreria **RTCLib** scaricabile dall'Arduino IDE per gestire il modulo RTC. Inoltre, Adafruit suggerisce di inserire una batteria a bottone, anche se scarica, nel RTC per mantenere l'orologio in funzione anche quando l'alimentazione esterna non è disponibile.

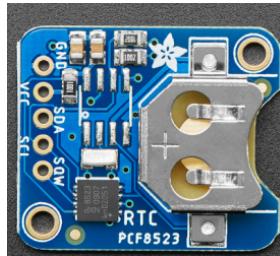


Figura 3: RTC

- **BMP280:** sensore di temperatura e pressione utilizzato per monitorare i parametri ambientali. E' stato testato con l'ESP32 utilizzando la libreria ufficiale **Adafruit BMP280 Library**. Il modulo BMP280 è collegato tramite un connettore STEAMM QT al display OLED da un lato, e all'ESP tramite i pin (Vin, GND, SCK e SDI) dall'altro, seguendo lo schema logico utilizzato per l'RTC. Il sensore BMP280 è in grado di misurare la pressione atmosferica nell'intervallo da 300 a 1100 hPa; ha un basso consumo di corrente, circa 2.7 μ A a una frequenza di campionamento di 1 Hz, e può operare in un intervallo di temperatura che va da -40 a +85 °C. E' possibile modificare la risoluzione delle misurazioni sia di temperatura che di pressione agendo su specifici bit associati ai registri del sensore.

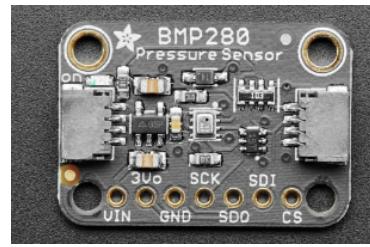


Figura 4: BMP280

2.1 Componenti Hardware

- **SSD1306:** display OLED utilizzato per visualizzare informazioni sull'ambiente circostante, quali pressione, temperatura e RPM della ventola. I collegamenti tra il sensore BMP280 e l'OLED sono stati effettuati tramite un cavo STEMMA QT, che segue il seguente schema di colori:

- Nero - GND
- Rosso - Vcc
- Blu - SDA
- Giallo - CLK

Lo schermo grafico OLED è monocromatico, ha dimensione di 0.96 pollici ed una risoluzione 128x64 pixel. Per utilizzare il modulo è necessario installare due librerie: **Adafruit SSD1306**, che gestisce la comunicazione a basso livello con l'hardware, e **Adafruit GFX**, che si basa sulla libreria **Adafruit SSD1306** per aggiungere funzioni grafiche come linee, cerchi e testo.

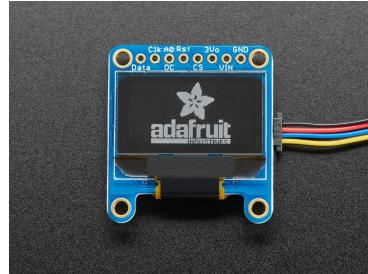


Figura 5: Display OLED

- **Ventola Noctua NF-A4x10 5v PWM 40x10mm:** dispositivo utilizzato per raffreddare il sistema in modo proporzionale all'aumento della temperatura rilevata dal sensore BMP280, tramite controllo PWM. La ventola è dotata di 4 pin collegati al filo/connettore:

- giallo (5V): pin di alimentazione, accetta 5V;
- nero (GND): pin di terra;
- verde (Tachimetro): fornisce un segnale tachimetrico per monitorare la velocità della ventola in tempo reale;
- Blu (PWM): controlla la velocità della ventola tramite segnale PWM gestito dall'ESP32.

Il segnale di tachimetro della ventola Noctua è caratterizzato da una frequenza, espressa in Hertz, che corrisponde alla velocità di rotazione della ventola. Per calcolare la velocità effettiva della ventola in giri al minuto (RPM), la frequenza del segnale in uscita dalla ventola deve essere moltiplicata per 60 e divisa per 2, poiché la ventola emette due impulsi per ogni rivoluzione. La formula per calcolare la velocità della ventola è la seguente:

Il segnale di tachimetro della ventola Noctua è caratterizzato da una frequenza, espressa in Hertz, che corrisponde alla velocità di rotazione della ventola. Per calcolare la velocità effettiva della ventola in giri al minuto (RPM), la frequenza del segnale in uscita dalla ventola deve essere moltiplicata per 60 e divisa per 2, poiché la ventola emette due impulsi per ogni rivoluzione. La formula per calcolare la velocità della ventola è la seguente:

$$\text{fan speed [rpm]} = \text{frequency [Hz]} \times 60 \div 2 \quad (1)$$

Questo tipo di ventola ha un assorbimento di potenza massimo di 0.35W e una corrente massima di 70mA. Poiché tale consumo non è troppo elevato, si è deciso di alimentare il dispositivo direttamente dal pin 5V dell'ESP32. Tuttavia, questa scelta risulta "accettabile" solo per utilizzi limitati come nel caso in esame, in quanto il pin 5V dell'ESP32 è destinato all'alimentazione del microcontrollore stesso quando non viene utilizzato il cavo USB.

2.2 Componenti Software

A tal proposito, si è scelto di alimentare l'ESP32 mediante un cavo USB collegato alla Raspberry Pi; tale cavo USB viene utilizzato anche per la comunicazione seriale tra i due dispositivi. In questo modo, è possibile sfruttare i circuiti della RPi per fornire l'alimentazione necessaria alla ventola.



Figura 6: Fan

2.2 Componenti Software

Per lo sviluppo e l'esecuzione del progetto sono stati utilizzati i seguenti componenti software:

- **Arduino IDE**: ambiente di sviluppo integrato open source utilizzato per la programmazione di microcontrollori in applicazioni progettuali embedded.
- **MariaDB**: sistema di gestione di database relazionale utilizzato per memorizzare e gestire i dati raccolti dai sensori.
- **Flask**: framework web leggero utilizzato per creare l'interfaccia web per la visualizzazione dei dati memorizzati nel database.
- **FreeRTOS**: sistema operativo in tempo reale utilizzato per il coordinamento dei task sull'ESP32.
- **Apache**: server web utilizzato per l'hosting dell'interfaccia web Flask.

2.3 Linguaggi di Programmazione

Nell'implementazione del progetto sono stati impiegati diversi linguaggi di programmazione per lo sviluppo del software:

- **HTML e CSS**: linguaggi di markup e stile utilizzati per la definizione e la formattazione dell'interfaccia web.
- **Python**: linguaggio di programmazione utilizzato per lo sviluppo del backend dell'applicazione.
- **C++**: linguaggio di programmazione utilizzato per lo sviluppo del codice su ESP32 tramite l'IDE di Arduino.
- **SQL**: linguaggio di interrogazione strutturato utilizzato per definire e manipolare i dati all'interno del database MariaDB.

3 Configurazione dispositivi

3.1 Esp32

3.1.1 Arduino IDE

La procedura per l'installazione dell'IDE di Arduino è molto semplice e può essere riassunta nei seguenti passaggi, che si distinguono a seconda del sistema operativo utilizzato:

- **Windows:** Scaricare il file .exe dal sito ufficiale di Arduino e seguire le istruzioni nella finestra di installazione.
- **Linux:** Scaricare il file .tar.xz, estrarrelo, quindi eseguire i seguenti script nel terminale:

```
1 sh arduino-linux-setup.sh user_name  
2 sh install.sh
```

Sostituire `user_name` con il nome del superutente del sistema Linux. Dopo l'installazione, l'Arduino IDE sarà disponibile nella sezione "Tutte le App".

- **Mac:** Scaricare il file .zip, estrarrelo e trascinare l'icona dell'Arduino IDE nella cartella "Applicazioni".

Una volta installato, aprire l'Arduino IDE.

3.1.2 Configurazione dell'Arduino IDE per ESP32

Al fine di configurare l'Arduino IDE per utilizzarlo con una board di tipo ESP32, si devono seguire alcuni passaggi. Innanzitutto, bisogna installare il driver per la comunicazione USB-seriale. Se il driver USB-seriale non viene installato automaticamente al collegamento del microcontrollore, è necessario scaricarlo dal sito di Silicon Labs: Silicon Labs CP2102 USB to UART Bridge VCP Drivers

Aggiungere URL del gestore di schede Ora bisogna abilitare il supporto per la piattaforma EPS32 nell'Arduino IDE. Quindi, bisogna aprire l'Arduino IDE e andare su **File > Preferenze**, nella sezione "URL aggiuntivi per il Gestore delle Schede", e incollare il seguente URL:

https://dl.espressif.com/dl/package_esp32_index.json

Se ci sono già altri URL, aggiungere una virgola alla fine e poi incollare questo nuovo URL.

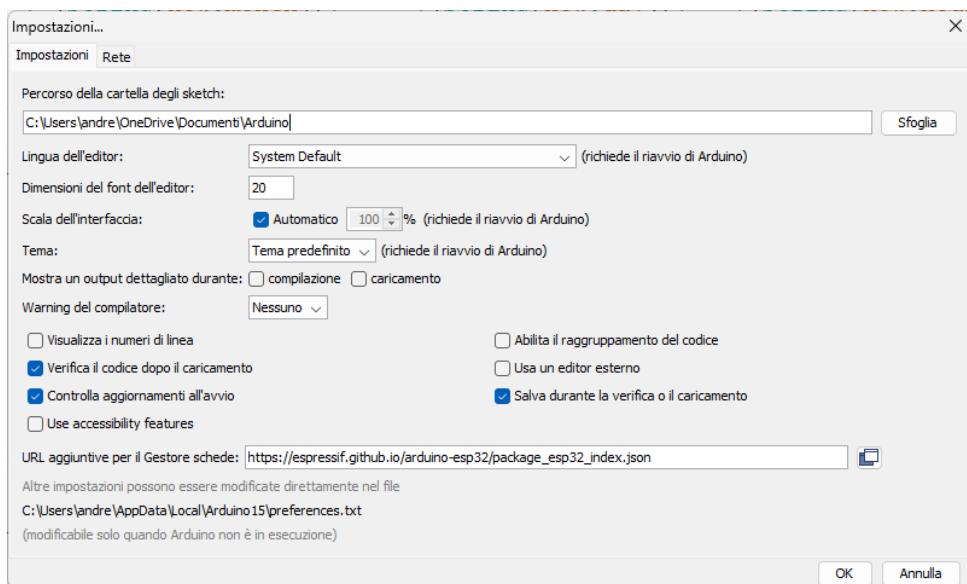


Figura 7: Configurazioni URL aggiuntivo Arduino IDE

3.1 Esp32

Installazione del tipo di scheda ESP32 Per installare il tipo di scheda ESP32, andare su **Strumenti > Scheda > Gestore delle Schede**. Nella casella di ricerca, digitare `esp32` e installare la versione 2.0.11 della scheda prodotta da Espressif Systems.

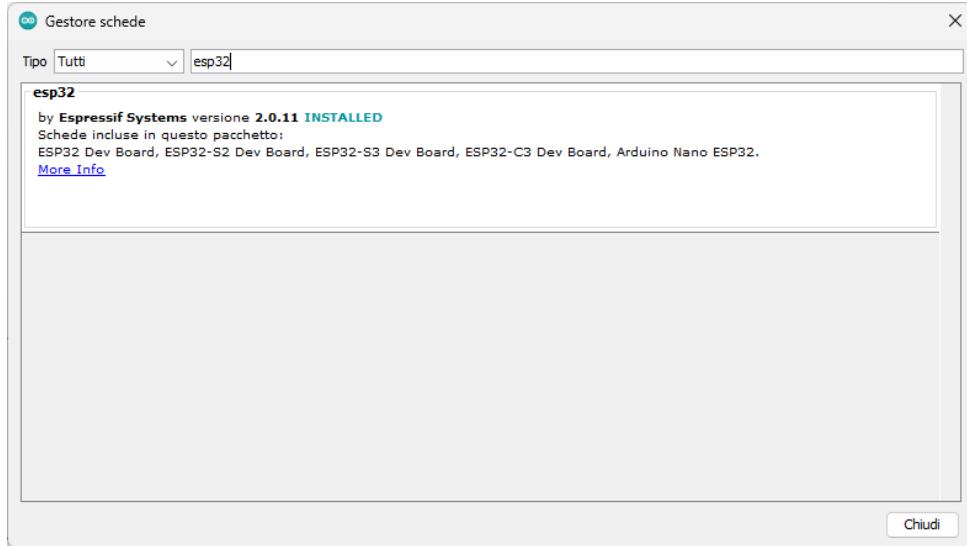


Figura 8: Installazione board ESP32

Selezione della scheda ESP32 Andare su **Strumenti > Scheda** e selezionare **ESP32 Dev Module**.

Selezione della porta Collegare la scheda ESP32 al computer tramite un cavo USB. Andare su **Strumenti > Porta** e selezionare la porta corrispondente alla scheda (es. `COM3` su Windows o `/dev/ttyUSB0` su Linux).

Caricamento del codice su ESP32 Scrivere o aprire uno sketch per ESP32 e cliccare su **Carica**. Se il caricamento non funziona al primo tentativo, può essere utile premere il pulsante **Boot** sul modulo ESP32 durante il caricamento.

3.1.3 Installazione delle librerie in Arduino IDE

Per utilizzare correttamente il progetto con l'ESP32, è stato necessario scaricare e installare alcune librerie specifiche nell'ambiente di sviluppo Arduino IDE (v1.8.19). Di seguito sono riportati i passaggi seguiti per ottenere e integrare le librerie necessarie nel progetto:

- Apertura di Arduino IDE:** Viene avviato l'ambiente di sviluppo Arduino IDE sul computer.
- Apertura del Gestore Librerie:** Dall'interfaccia di Arduino IDE, viene aperto il **Gestore Librerie** selezionandolo dal menu **Sketch > Includi Libreria > Gestione Librerie**.
- Ricerca delle Librerie:** In **Gestore Librerie**, viene utilizzata la funzione di ricerca per individuare le librerie necessarie al progetto.
- Installazione delle Librerie:** Dopo aver trovato le librerie desiderate, viene cliccato il pulsante **Installa** accanto a ciascuna libreria per avviare il processo di installazione. Arduino IDE scarica e installa automaticamente le librerie sul sistema.

Nella Fig9 si fa un esempio dell'installazione della libreria `RTClib.h`.

3.2 Raspberry Pi 4

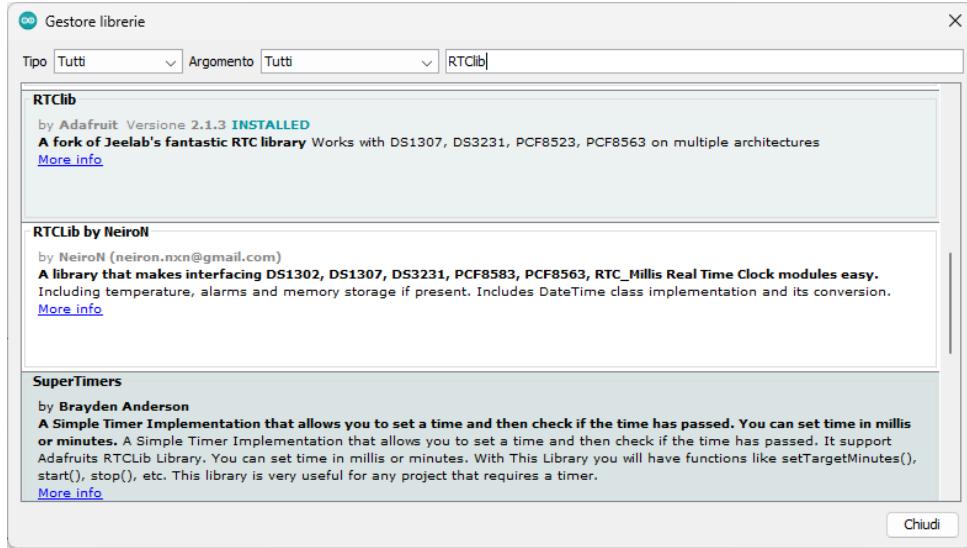


Figura 9: Installazione della libreria relativa al modulo *RTC*

Nel progetto sono state utilizzate diverse librerie per il corretto funzionamento del dispositivo Arduino. Di seguito sono elencate le librerie utilizzate con le rispettive versioni:

RTClib	- v2.1.3
Adafruit BMP280 Library	- v2.6.8
Adafruit GFX Library	- v1.11.9
Adafruit SSD1306	- v2.5.9

3.2 Raspberry Pi 4

3.2.1 Installazione di L.A.M.P. e librerie necessarie

Installazione del Sistema Operativo sulla Raspberry Pi Per installare il sistema operativo sulla Raspberry Pi, è stata impiegata l'applicazione *Raspberry Pi Imager*, resa disponibile sul sito ufficiale di Raspberry Pi. Una volta terminata l'installazione, la schermata iniziale del tool è mostrata in Fig.10

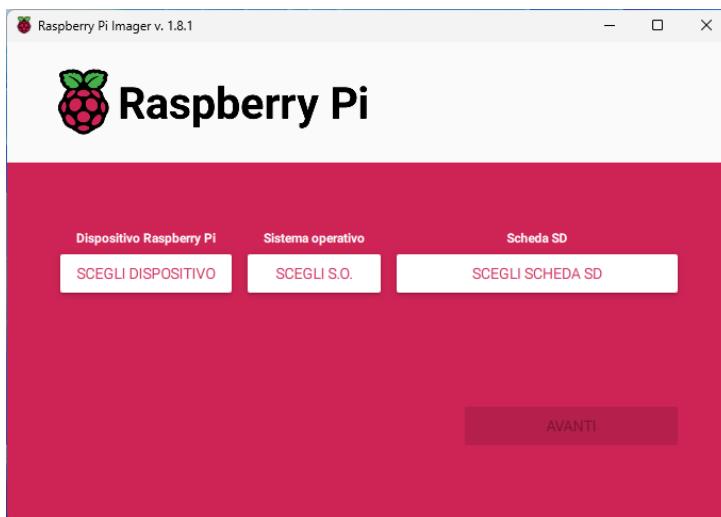


Figura 10: Schermata iniziale di *Raspberry Pi Imager*

Questo strumento offre una serie di funzionalità utili:

3.2 Raspberry Pi 4

- *Selezione del dispositivo Raspberry Pi*: consente di scegliere il modello specifico di Raspberry Pi su cui installare il sistema operativo;
- *Selezione del sistema operativo*: permette di scegliere tra diversi sistemi disponibili. La scelta è ricaduta su *Raspberry Pi OS Lite (64-bit)* per la sua leggerezza e efficienza, adatte alle esigenze del progetto.
- *Selezione della scheda SD*: il software rileva automaticamente la scheda SD inserita, semplificando il processo di memorizzazione del sistema operativo.

Prima di avviare il processo di installazione, è stata aperta una nuova finestra di dialogo (Fig.11) per la configurazione personalizzata delle seguenti impostazioni:

- *Nome Host*: viene richiesto di impostare un nome host per identificare facilmente il dispositivo nella rete;
- *SSH*: è possibile abilitare questa opzione per consentire connessioni sicure e crittografate al dispositivo;
- *Wi-Fi*: è richiesto di configurare il Wi-Fi inserendo l'SSID e la password della rete wireless per garantire l'accesso a Internet.

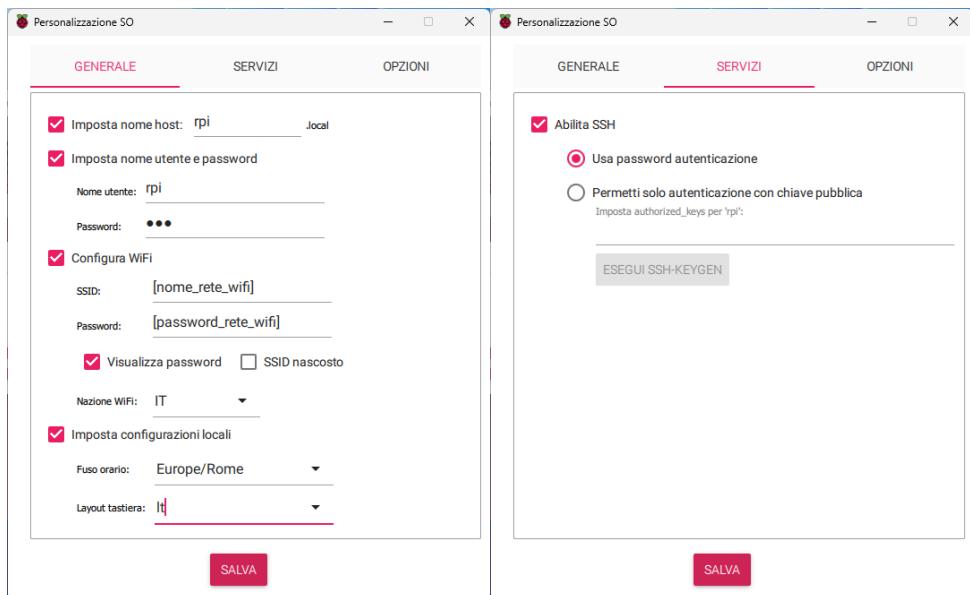


Figura 11: Configurazione delle impostazioni personalizzate in *Imager*

Dopo aver configurato tutti i parametri, è stata avviata la scrittura dell'immagine del sistema operativo sulla microSD. Al termine del processo, è stata inserita la microSD nella Raspberry Pi. All'avvio della Raspberry Pi, il sistema operativo viene caricato e le configurazioni personalizzate vengono applicate. Il servizio SSH è attivo e ci si può connettere alla Raspberry Pi tramite il terminale di un altro dispositivo connesso alla stessa rete Wi-Fi utilizzando il comando:

```
1 ssh [username]@[IP_address]
```

Per ulteriori personalizzazioni e ottimizzazioni del sistema operativo, è possibile utilizzare il comando `sudo raspi-config`, il quale consente di accedere a una serie di opzioni di configurazione avanzate, consentendo di adattare ulteriormente la Raspberry Pi alle proprie esigenze specifiche. In particolare, è stato abilitato il servizio VNC (Virtual Network Computing) consentendo di interagire con l'interfaccia grafica della Raspberry Pi tramite un client VNC da remoto.

Prima di procedere con l'installazione dei pacchetti necessari è buona pratica aggiornare l'elenco dei pacchetti, risolvere le dipendenze e migliorare la sicurezza del sistema, tramite il comando:

3.2 Raspberry Pi 4

```
1 sudo apt update && sudo apt upgrade
```

Installazione di Apache2 E' possibile scaricare Apache2 eseguendo il seguente comando

```
1 sudo apt install apache2==2.4.57-2
```

Per la verifica dello stato e il riavvio del servizio si utilizzino i seguenti comandi

```
1 sudo systemctl status apache2
2 sudo systemctl restart apache2
```

Installazione di MariaDB Per l'installazione di MariaDB sul sistema sono stati seguiti i seguenti passaggi:

1. **Installazione di MariaDB Server:** è stato installato il server MariaDB utilizzando il comando:

```
1 sudo apt install mariadb-server==1.10.11.4-1
```

2. **Verifica della versione di MariaDB:** per verificare la versione di MariaDB installata, eseguire il seguente comando:

```
1 mariadb -V
```

Inoltre, è stato riavviato il servizio MariaDB al fine di verificare che fosse tutto installato correttamente:

```
1 sudo systemctl restart mariadb.service
```

3. **Configurazione della sicurezza:** Dopo aver installato MariaDB Server, è consigliabile eseguire il seguente comando per proteggere l'installazione:

```
1 sudo mysql_secure_installation
```

Una volta eseguito il comando, verrà visualizzata una finestra di dialogo da terminale, chiedendo di inserire la password corrente per l'utente root e di impostare una nuova password. Al termine del processo di configurazione, verrà visualizzato il messaggio "*Grazie per aver utilizzato MariaDB!*".

4. **Accesso a MariaDB:** Infine, è stato effettuato l'accesso a MariaDB utilizzando il comando

```
1 sudo mysql --user=root --password
```

Questi passaggi hanno permesso di configurare correttamente MariaDB sul sistema e di iniziare a utilizzarlo per le applicazioni.

Creazione del database Una volta connessi a MariaDB, è stato creato il database e la tabella contenente le misurazioni dei sensori installati sull'EPS32. Si utilizzano i seguenti comandi:

```
1 MariaDB [mysql]> create database SensorData;
2 MariaDB [mysql]> use SensorData;
3 MariaDB [SensorData]> CREATE TABLE misure(
4 -> id INT NOT NULL AUTO_INCREMENT,
5 -> Data DATETIME,
6 -> Temperatura FLOAT,
7 -> Pressione FLOAT,
8 -> RPM INT,
9 -> PRIMARY KEY (id)
10-> );
```

Questi comandi hanno permesso di creare il database *SensorData* e di definire la tabella denominata *misure*, la quale contiene i seguenti campi:

3.3 Ambiente di sviluppo

- **id**: un identificatore univoco per ogni record, generato automaticamente.
- **Data**: la data e l'ora della misura.
- **Temperatura**: il valore della temperatura registrato.
- **Pressione**: il valore della pressione registrato.
- **RPM**: il numero di giri al minuto registrato.

In Fig.12 è mostrata la tabella creata visibile con il comando

```
1 MariaDB [SensorData]> describe misure;
```

Field	Type	Null	Key	Default	Extra
id	bigint(20) unsigned	NO	PRI	NULL	auto_increment
Data	datetime	YES		NULL	
Temperatura	float(5,2)	YES		NULL	
Pressione	float(5,2)	YES		NULL	
RPM	int(11)	YES		NULL	

Figura 12: Descrizione dei campi che costituiscono il database creato con MariaDB

PHP Per installare PHPMyAdmin su un Raspberry Pi, è sufficiente eseguire i seguenti comandi nel terminale:

```
1 sudo apt install php==2:8.2+93
2 sudo apt install phpmyadmin==4:5.2.1+dfsg-1
```

Durante l'installazione di PHPMyAdmin verrà visualizzata una finestra di dialogo in cui verrà chiesto di confermare l'utilizzo di *apache2* e *dbconfig-common* per la configurazione del database. Successivamente, verrà richiesta la configurazione di una password per l'accesso a MySQL tramite PHPMyAdmin.

Qualora si riscontrassero errori durante l'accesso a PHPMyAdmin, potrebbe essere necessario creare un nuovo utente per accedere. Questi comandi creeranno un nuovo utente con nome (*admin*) e password (*your_password*). Le credenziali non devono necessariamente coincidere con quanto riportato, ma possono essere liberamente selezionate dall'utente. Qui vengono fornite delle credenziali esemplificative, offrendo all'utente la flessibilità di scegliere le proprie informazioni di accesso.

```
1 sudo mysql
2 > create user admin@localhost identified by 'your_password';
3 > grant all privileges on *.* to admin@localhost;
4 > FLUSH PRIVILEGES;
5 > exit;
```

3.3 Ambiente di sviluppo

L'ambiente di sviluppo virtuale è un ambiente che include tutte le risorse necessarie per scrivere, testare e eseguire applicazioni. L'utilizzo di questo è stato dettato dalle sue peculiarità, le quali permettono la riproducibilità di un qualsiasi progetto isolandone le dipendenze dagli altri. Per creare ed installare le prime librerie all'interno dell'ambiente sono state eseguite le seguenti istruzioni direttamente nel terminale della Raspberry Pi:

1. **Creazione dell'ambiente virtuale.** Questo comando deve essere eseguito solo la prima volta; Nel nostro caso è stato creato un ambiente virtuale chiamato *progett_sod* al cui interno viene installata l'ultima versione del linguaggio Python (v. 3.11.2) e del gestore dei pacchetti *pip* (v. 23.0.1)

```
1 python3 -m venv progett_sod python=3.11.2
```

3.4 Framework Flask

2. **Attivazione dell'ambiente virtuale.** Questo comando va eseguito ogni volta che si vuole usufruire dei pacchetti installati all'interno di un determinato ambiente virtuale.

```
1 source progett_sod/bin/activate
```

3. **Installazione delle librerie utilie.** Successivamente alla prima installazione sono state installate tutte le librerie necessarie alla comunicazione su seriale, alla creazione e salvataggio dei dati su database e al funzionamento del chatbot. All'interno del sottostante listato vengono riportati i principali pacchetti utilizzati ai quali, per ciascuno di essi, sono state installate dal gestore pip le loro dipendenze. In Appendice 7.1 viene riportata la lista completa di tutti i pacchetti.

```
1 pip install pyserial == 3.5
2 pip install ntplib == 0.4.0
3 sudo apt-get install libmariadb-dev == 1:10.11.4-1 deb12u1
4 pip install mariadb == 1.1.9
5 pip install numpy == 1.26.3
6 pip install matplotlib == 3.8.1
7 pip install Flask == 3.0.1
8 pip install python-telegram-bot == 20.7
```

3.4 Framework Flask

Flask è un micro framework web per Python, sviluppato per essere semplice e flessibile. Questo framework permette di costruire rapidamente applicazioni web, fornendo solo gli elementi essenziali necessari per iniziare, mentre lascia ampia libertà per l'organizzazione del codice e la scelta delle estensioni. Flask si basa su Jinja2 come motore di template. Per mantenere il codice ben organizzato, è importante seguire una struttura delle cartelle chiara e logica. La struttura di progetto seguita per l'applicazione sviluppata è la seguente:

```
1 Flask/
2 |
3 |-- app.py
4 |
5 |-- static/
6 |   |__ style.css
7 |
8 |-- templates/
9   |__ index.html
```

La procedura di installazione di questo strumento è stato trattato all'interno del paragrafo precedente.

4 Sistema di acquisizione dati

4.1 Descrizione generale sul sistema hardware implementato

Dal punto di vista elettronico, è possibile comprendere l'implementazione del sistema osservando la Fig. 13.

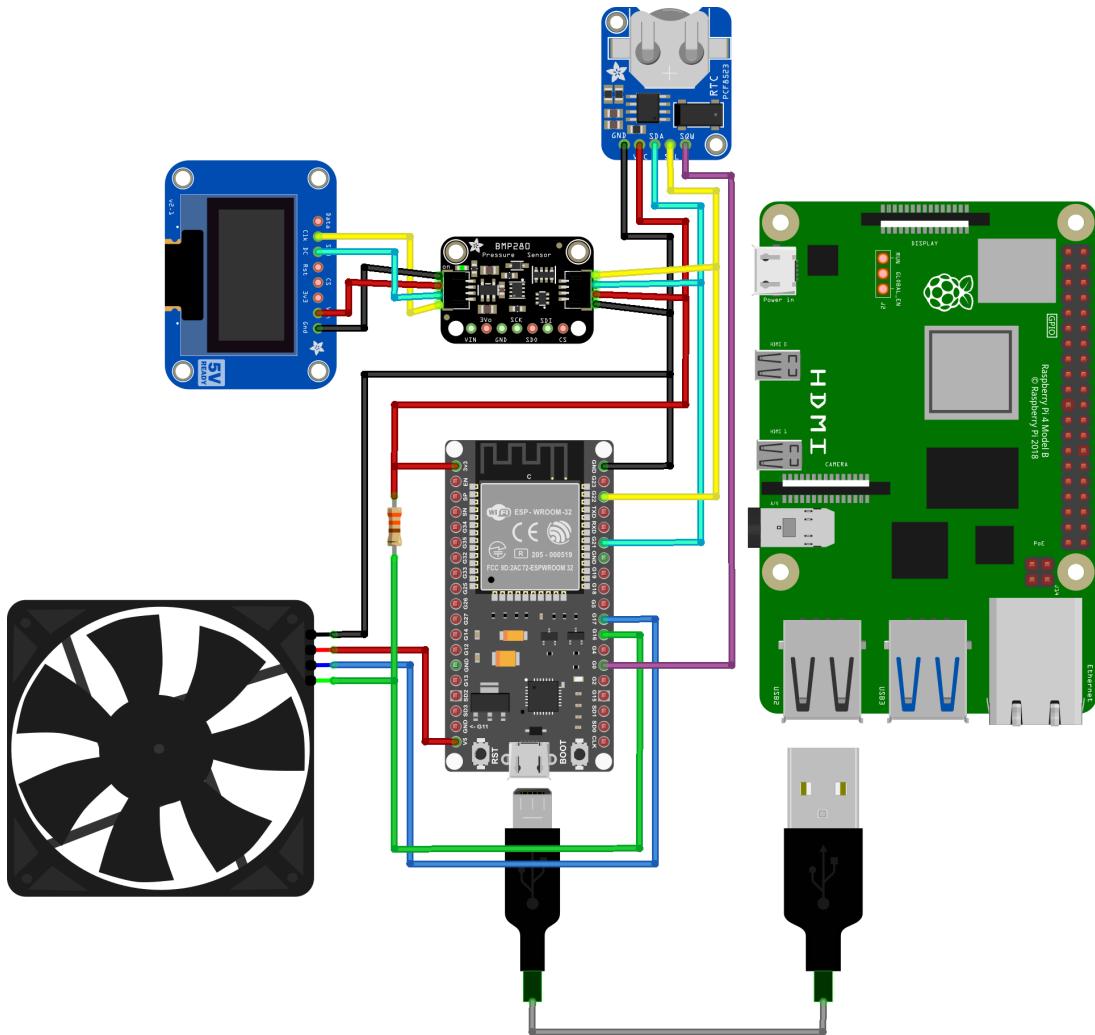


Figura 13: Schema del cablaggio.

Il microcontrollore ESP32 e la Raspberry Pi (RPi) sono stati interconnessi tramite il protocollo seriale. Ciascuno di questi dispositivi avrà il proprio ruolo specifico: l'ESP è responsabile dell'acquisizione dei dati, mentre l'RPi si occupa della loro ricezione e successiva elaborazione.

Come mostrato nella Fig 13, i seguenti dispositivi sono collegati all'ESP:

- sensore BMP280: misura la temperatura e la pressione atmosferica dell'ambiente circostante;
- modulo RTC: mantiene l'orario corretto all'interno del sistema;
- display OLED: visualizza i dati acquisiti;
- ventola controllata da PWM: utilizzata per la dispersione del calore.

Il sensore BMP280, il modulo RTC e il display OLED utilizzano il protocollo di comunicazione bifilare I²C per interagire con l'ESP32. In particolare, l'ESP, configurato come dispositivo "Master",

4.1 Descrizione generale sul sistema hardware implementato

- emette il segnale di clock sul pin G22 (collegamenti di colore gialli) e
- preconfigura la linea per la trasmissione dei dati sul pin G21 (collegamenti di colore ciano).

La ventola, invece, è controllata tramite PWM e dispone di 4 pin:

- due pin per l'alimentazione a 5V (connessi ai fili rossi e neri),
- un pin per il controllo mediante segnale PWM (connesso al pin G17 tramite il cavo blu)
- e un pin per rilevare il numero di rotazioni al minuto (connesso sia direttamente al pin G16 e sia, tramite una resistenza di pull-up, al pin di alimentazione 3V3 mediante la linea verde).

Tutti i dispositivi sono alimentati mediante il pin 3v3 di ESP32¹

Inoltre, per quanto riguarda la ventola, è stato necessario dimensionare la resistenza di pull-up poichè il pin dedicato al tachimetro presenta un transistor in configurazione open collector (vedi figura 14). Dal datasheet si evince che la corrente massima che può scorrere sul collettore di tale transistor è pari a 5mA. Con una tensione di alimentazione erogata di 3.3V fornita dall'ESP, il valore della resistenza calcolata è di 660Ω . In mancanza di un resistore di questo valore, è preferibile utilizzare una resistenza maggiore, ma a causa delle limitazioni di risorse hardware disponibili, è stato utilizzato un resistore da 330Ω , il quale permette il passaggio di una corrente doppia rispetto alla massima raccomandata dal produttore).

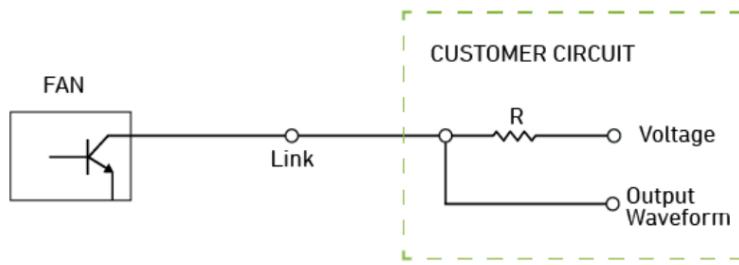


Figura 14: Schema di collegamento del pin dedicato al segnale del tachimetro.

In generale, la logica di funzionamento implementata per il sotto-sistema illustrato finora si suddivide in 5 operazioni principali:

1. Acquisizione dei dati dal sensore BMP280 e della marca temporale dall'orologio RTC
2. Modulazione della velocità della ventola;
3. Impacchettamento dell'informazione in una stringa;
4. Invio della stringa alla RPi;
5. Visualizzazione su display OLED dei valori di temperatura, pressione e velocità ventola;
6. Controllo di comandi di sincronizzazione ricevuti tramite seriale dalla RPi.

Queste operazioni sono state raggruppate in 3 attività (denominate **task**), che vengono eseguite "contemporaneamente" grazie alla libreria FreeRTOS, la quale le schedula in tempo reale.

La Fig. 15 mostra uno schema approssimativo di come hardware e software interagiscono tra loro. Nelle sezioni successive, ciascun task verrà descritto in dettaglio, insieme alla loro integrazione con FreeRTOS.

Inoltre, il codice di riferimento è consultabile in appendice 7.2.

¹Questo collegamento non è convenzionale in quanto, per i motivi spiegati in 2.1.2, tale pin è progettato per alimentare il microcontrollore da una fonte esterna e non per fornire energia in uscita.

4.2 Descrizione del codice implementato

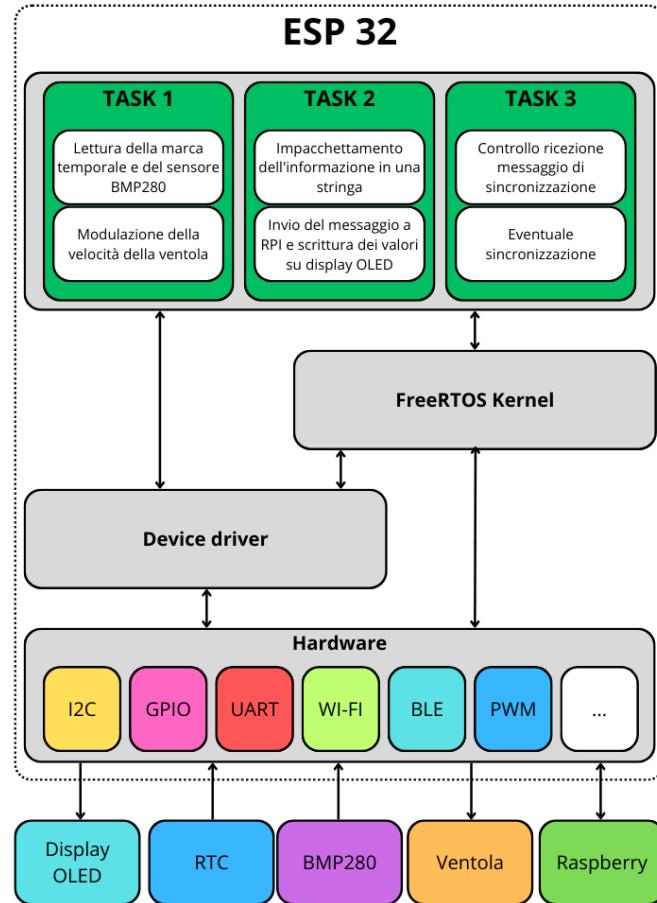


Figura 15: Schema riassuntivo dell'architettura hardware-software.

4.2 Descrizione del codice implementato

Il codice caricato nel microcontrollore (vedi appendice 7.2) è strutturato per gestire le attività del sistema in un ambiente embedded multitasking in tempo reale, impiegando il kernel FreeRTOS. Dopo l'inizializzazione e la configurazione dei dispositivi hardware necessari, l'associazione di funzioni per la gestione degli interrupt e la dichiarazione delle variabili globali, vengono creati diversi task tramite la funzione `xTaskCreate()`. Ciascun task ha una priorità e un compito specifico all'interno del sistema, come la sincronizzazione dell'orologio RTC, la lettura dei sensori e l'invio dei dati alle periferiche.

Il kernel FreeRTOS si occupa della schedulazione dei task, decidendo quale debba essere eseguito in base alla priorità assegnata. La comunicazione e la cooperazione dei task avvengono attraverso l'uso di una coda (`QueueHandle`) e variabili di stato, che consentono lo scambio di messaggi in modo sicuro e sincronizzato, evitando conflitti e garantendo un funzionamento coerente del sistema. Questo meccanismo permette ai task di condividere dati e informazioni rilevanti per il funzionamento del sistema.

Infine, gli interrupt sono gestiti in modo sicuro all'interno dell'ambiente multitasking di FreeRTOS e vengono utilizzati per contare gli impulsi del tachimetro, assicurando una gestione efficiente degli eventi esterni.

4.2.1 Inizializzazione e blocco `setup()`

Analizzando le righe che vanno da 1 a 149 del codice di riferimento 7.2 si può scomporre la logica implementata in due parti: la prima dedicata alla preparazione e alle definizioni iniziali e la seconda destinata all'inizializzazione dei dispositivi hardware e dei task. Per quanto concerne la prima sezione (da 1 a 66) si evidenziano:

- **Inclusione delle librerie.** Il codice inizia includendo le librerie necessarie per il funzionamento dei dispositivi utilizzati, come `RTCLib.h` per il modulo RTC, `Adafruit_SSD1306.h` e `Adafruit_GFX.h`

4.2 Descrizione del codice implementato

per il display OLED, `Adafruit_BMP280.h` per il sensore BMP280 e `Wire.h` per la comunicazione I²C.

- **Definizione delle costanti.** Vengono definite alcune costanti che verranno utilizzate nel codice per identificare pin GPIO specifici per il segnale PWM, per il tachimetro, per la soglia di temperatura e per l'interrupt del countdown del RTC.
- **Dichiarazione delle variabili globali.** Vengono dichiarate variabili globali utilizzate all'interno del programma. Queste includono variabili per gestire gli interrupt, tenere traccia della velocità della ventola, memorizzare la data della misurazione precedente e una struttura per i dati dei messaggi da inviare in coda.
- **Definizione delle funzioni di interrupt.** Vengono definite due funzioni di interrupt: `countPulses()` per contare gli impulsi del tachimetro e `countdownOver()` per gestire gli impulsi generati dal RTC. Queste funzioni sono dichiarate come `void()` poiché possono essere chiamate in modo asincrono durante l'esecuzione del programma principale e si appoggiano a variabili globali volatili.
- **Definizione della funzione dateCheck().** Questa funzione viene utilizzata per confrontare la data prima dell'invio di ogni messaggio su seriale al fine di evitare l'invio di messaggi identici. Restituisce `True` se la nuova data è diversa dalla data precedente, altrimenti restituisce `False`.
- **Definizione dei task.** Vengono definiti i task che verranno eseguiti dal sistema. Questi task includono `TaskSerial()`, `Task_Sensors()` e `Task_Sync()`.
- **Definizione della coda e delle variabili oggetto di ciascun dispositivo.** Viene definito l'oggetto coda riservandole uno spazio in memoria. Successivamente vengono definite le variabili oggetto necessarie per la successiva inizializzazione dei dispositivi.

La seconda sezione (da 68 a 149) è riservata al set up iniziale di tutto il sistema hardware. A tal proposito si evidenziano:

- **Inizializzazione dei dispositivi hardware.** Prima dell'effettivo utilizzo, è stato necessario procedere all'inizializzazione dei dispositivi utilizzati:
 - *modulo RTC*: il datasheet del dispositivo PCF8523 dichiara che l'oscillatore interno è soggetto a delle perdite di precisione che possono essere risolte calibrando il dispositivo successivamente al suo avvio. Tale operazione dovrebbe essere effettuata impostando dei parametri di drift segnalati sulla scheda tecnica ma, date le specifiche di implementazione, si è scelto di affettuare questa procedura in un altro modo: sfruttando l'implementazione del task `Task_Sync()`. Inoltre, è stato abilitato e configurato un timer di conteggio decrescente con una frequenza di 64Hz e un valore di partenza di 64. Con queste impostazioni, il timer raggiunge lo zero dopo circa 1s generando un impulso di basso livello di otto cicli di clock.
 - *sensore BMP280*: nell'inizializzazione è possibile definire i parametri per la configurazione del sensore. Essi possono includere la scelta della modalità operativa di misurazione (ad esempio, modalità normale, modalità forzata), la risoluzione della misurazione (sia per la temperatura che per la pressione), il modo in cui deve essere filtrato il segnale misurato e il tempo di standby. Tale configurazione dipende dalle esigenze specifiche dell'applicazione e possono essere regolate utilizzando le funzioni fornite dalla libreria. Ad ogni modo, nel nostro caso sono state lasciate le impostazioni di default.
 - *Display OLED*: è stato opportuno impostare per tale dispositivo la modalità di alimentazione tramite circuito di carica del condensatore integrato, mediante il parametro `SSD1306_SWITCHCAPVCC`, e l'indirizzo per la comunicazione I²C.
- **Creazione dei task.** Utilizzando la funzione `xTaskCreate()`, vengono creati i task e vengono specificati i loro parametri come nome, dimensione dello stack e priorità.
- **Inizializzazione del PWM e degli interrupt.** Viene inizializzato il modulo PWM per controllare la velocità della ventola e vengono configurati gli interrupt per gestire gli impulsi del tachimetro e del countdown del modulo RTC.

4.2 Descrizione del codice implementato

4.2.2 Task per l'acquisizione dei dati

Il task dedicato all'acquisizione dei dati è stato denominato `Task_Sensors()`. Analizzando la porzione di codice compresa tra riga 188 e 216 del codice è possibile intuire come la logica implementata si suddivida nei seguenti tre step:

1. Lettura del modulo RTC e del sensore BMP280.

Dopo l'esecuzione della funzione `rtc.now()`, necessaria per l'ottenimento della data e l'orario, viene interrogato il sensore tramite le funzioni `bmp.readTemperature()` e `bmp.readPressure()`. Questi comandi permettono di leggere la temperatura (in gradi Celsius) e la pressione (in Pascal) che misura il dispositivo negli istanti in cui vengono eseguite. Per una migliore interpretazione della pressione rilevata si è scelto di effettuare una conversione dell'unità di misura da Pascal in atm moltiplicando il valore acquisito per $9.8692 \cdot 10^{-6}$

2. Modulazione del segnale PWM.

Da specifiche è stato richiesto che, successivamente al superamento di una temperatura limite (impostata a 28°C), la ventola si avviasse mediante segnale PWM e con una velocità proporzionale alla differenza tra il valore attuale misurato e la soglia predisposta. Tale richiesta è stata implementata combinando le funzioni:

- `map()`, mappa il primo parametro in input dall'intervallo delle temperature [28°C , 32°C] in quello numerico [0, 255], necessario per la generazione del valore da dare in input alla funzione `ledcWrite()`;
- `constrain()`, vincola il primo parametro in input a rimanere all'interno dell'intervallo [0, 255] risolvendo i problemi di mappatura quando il valore `message.temperatura` è molto più grande del limite superiore dell'intervallo delle temperature dichiarato in `map()`;
- `ledcWrite()`, permette di generare il segnale PWM sul pin 17 modulando il duty cycle nel range [0%, 100%] mediante i livelli [0, 255].

3. Calcolo della velocità della ventola.

Per acquisire la velocità della ventola è stato opportuno in prima battuta implementare due importanti comandi:

- `countPulses()` (definita tra riga 32 e 35), utilizzata per rilevare gli impulsi emessi dal contagiri sul pin designato ogni volta che la ventola esegue 1/2 di giro;
- `countdownOver()` (definita tra riga 37 e 41), utilizzata per rilevare gli impulsi emessi dal modulo RTC tramite il pin SWQ ogni volta che trascorre 1 secondo.

Grazie alla definizione di queste due funzioni, alla decorrenza di ogni secondo viene effettuata la conversione degli impulsi rilevati da `countPulses()` in rivoluzioni per minuto (rpm), in accordo con la seguente formula

$$rpm = \frac{n_{impulsi}}{2} \cdot 60 \quad (2)$$

4. Invio della variabile struttura `message` nella coda.

Al termine del passaggio precedente la variabile struttura denominata `message` contiene tutti i valori che verranno utilizzati per preparare la stringa di dati da trasmettere. Di seguito viene riportato un esempio della variabile definita tra le righe 23 e 29 arricchita:

```
1 message
2   |_ .temperatura = 19.8
3   |_ .pressione = 1.01
4   |_ .rpm = 1200
5   |_ .time_stamp = 2024-03-12 12:12:12
```

Prima dell'invio di questa variabile al task `TaskSerial`, tramite la funzione `xQueueSend()`, viene eseguita un controllo mediante la funzione `dateCheck()` in riga 211 verificando se l'orario del messaggio non coincide con quello precedentemente trasmesso. Tale azione permette di ridurre drasticamente l'invio di messaggi evitando così intasamento dei canali di trasmissione, ritardi nella trasmissione dei dati e ridondanza di informazione nel database.

4.2 Descrizione del codice implementato

Al fine di garantire una corretta temporizzazione del task è stato necessario l'utilizzo della funzione `vTaskDelayUntil()` la quale tiene conto del tempo di esecuzione del codice.

4.2.3 Task per l'invio dei dati

Per la trasmissione dei dati al display OLED e alla Raspberry è stata implementata la funzione chiamata `TaskSerial()`, compresa tra le righe 158 e 186. Tale task sfrutta due protocolli di comunicazione diversi al fine di visualizzare i dati sul display e, contemporaneamente, inviarli alla RPi secondo i seguenti step:

1. tramite comunicazione seriale (riga 165), una stringa contenente tutti gli attributi della variabile `message` strutturata nel seguente modo

M!#YYYY-MM-DD hh:mm:ss#temperatura#pressione#rpm (3)

2. tramite protocollo I²C (da 168 a 181), i valori da visualizzare su display OLED.

4.2.4 Task per la sincronizzazione del modulo RTC

Da specifiche di progetto è necessario l'utilizzo di un modulo RTC che, non essendo equipaggiato da scheda Wi-Fi per l'accesso a Server NTP, può perdere precisione durante il funzionamento. Per ovviare a questa problematica è stata implementata la funzione `Task_Sync()` (visualizzabile da riga 219 a 253) che consente al microcontrollore di ricevere comandi di sincronizzazione da Raspberry tramite la porta seriale e di aggiornare l'orologio RTC di conseguenza, utilizzando i dati forniti dalla stessa. La struttura di un messaggio di sincronizzazione tipico è la seguente

#sync#anno#mese#giorno#ora#minuti#secondi# (4)

e sulla base della conoscenza di questa è stato implementato il processo di ricezione, decodifica e calibrazione nei seguenti step:

- **Lettura del messaggio seriale.**

La funzione controlla periodicamente se ci sono dati disponibili sulla porta seriale. Quando dei dati sono disponibili, legge l'intero messaggio seriale utilizzando la funzione `Serial.readString()`.

- **Decodifica del messaggio.**

Dopo aver letto il messaggio seriale, il codice estrae i dati utili da esso. Utilizza il carattere "#" come delimitatore per separare la stringa. Questo è implementato con un ciclo `for` che cerca la posizione di ciascun delimitatore "#" all'interno del messaggio e lo divide.

- **Verifica del comando di sincronizzazione.**

Una volta che i dati sono stati estratti dal messaggio, il codice verifica se il primo elemento estratto è "sync". Questo indica che il comando richiede la sincronizzazione del modulo.

- **Calibrazione del modulo RTC.**

Se il primo elemento dell'array `msg_rx` è riconosciuto come "sync" e tutti gli altri elementi di questo sono all'interno di range accettabili allora il codice procede con la funzione `rtc.adjust()` per impostare la data e l'ora dell'orologio esterno con i valori forniti dal messaggio ricevuto. Dopo la calibrazione si effettua un riavvio con la funzione `rtc.start()`.

5 Sistema di elaborazione e divulgazione dei dati acquisiti

5.1 Panoramica generale

La Raspberry Pi 4 svolge un ruolo fondamentale all'interno del sistema di monitoraggio ambientale. Essa, infatti, funge da server IoT, coordinando diverse operazioni cruciali come la memorizzazione dei dati nel database, la gestione degli alert tramite un chatbot Telegram e la visualizzazione dei dati su una pagina web dedicata. Nel dettaglio, nel corso di questa sezione, verranno descritte le seguenti attività:

- **Sincronizzazione del modulo RTC**

Si spiegherà come la Raspberry Pi assicuri una sincronizzazione accurata del modulo di tempo reale RTC collegato all'ESP32, garantendo una gestione precisa e costante del tempo nell'intero sistema.

- **Creazione del database**

Si illustrerà il processo di memorizzazione dei dati ricevuti dall'ESP32 all'interno di un database MariaDB, includendo i dettagli relativi alla gestione dell'orario e alla memorizzazione dei dati stessi.

- **Visualizzazione della webapp**

Si presenterà l'applicazione web sviluppata con Flask per consentire agli utenti di visualizzare i dati storici in modo intuitivo e user-friendly.

- **Implementazione del chatbot**

Si descriverà come è stato implementato un chatbot Telegram per inviare alert agli utenti registrati quando la temperatura supera una soglia critica prefissata a 33°C e per consentire agli utenti di richiedere grafici relativi ai dati storici.

- **Esecuzione dei daemon**

Si forniranno istruzioni su come sono stati creati e avviati i daemon per garantire che l'intero sistema funzioni autonomamente, senza richiedere l'intervento diretto degli utenti.

Queste attività rappresentano le componenti chiave del sistema di monitoraggio ambientale gestito dalla Raspberry Pi, consentendo un'operatività fluida e automatizzata del sistema nel suo complesso.

5.2 Sincronizzazione del modulo RTC

Per completare la fase relativa alla corretta sincronizzazione tra l'ESP32 e la Raspberry, è stato implementato uno script consultabile in appendice 7.3. Tale script si occupa di mantenere sincronizzato il modulo di tempo reale RTC collegato all'ESP32 con un server NTP (Network Time Protocol) attraverso una connessione seriale con la Raspberry, al fine di garantire una gestione precisa e costante del tempo nell'intero sistema.

Come da specifiche di progetto, la sincronizzazione del modulo RTC deve avvenire mediante un comando proveniente dalla Raspberry che, attraverso il protocollo NTP, avrà accesso all'orario corretto. Inoltre, la sincronizzazione dovrà avvenire all'accensione del sistema e, successivamente, ad intervalli regolari.

Per fare ciò, inizialmente, l'utilizzo della funzione `get_ntp_time()` permette di ottenere l'orario iniziale dal server NTP; questo consente di sincronizzare il modulo RTC all'accensione del sistema e a successivi intervalli regolari, fissati ad un minuto nell'esempio, inviando un messaggio contenente l'orario specifico attraverso la porta seriale all'ESP32.

In prima battuta, lo script era stato implementato come un loop principale, il quale ciclicamente otteneva l'orario corrente dal server NTP, calcolava la differenza temporale rispetto all'orario iniziale e, qualora tale differenza fosse superiore o uguale a 60 secondi, preparava un messaggio di sincronizzazione con l'orario e lo inviava all'ESP32 tramite comunicazione seriale. In ultima analisi, ai fini di ottimizzare il codice, è stato semplificato lo script privandolo del loop interno e richiamandolo periodicamente da un timer predisposto da `systemd`, il quale si occupa di compensare l'eliminazione dell'iniziale loop insito nello script. La relazione tra lo script in questione, i demoni ed il timer verrà approfondita nel paragrafo 5.6.

Un ultimo approfondimento su questa sezione riguarda la formattazione del messaggio di sincronizzazione che viene inviato dalla RPi al ESP32. In particolare, il messaggio è strutturato nel seguente modo:

5.3 Creazione del database

```
#sync#anno#mese#giorno#ora#minuti#secondi#
```

 (5)

poiché la funzione `adjust()` presente nel codice lato ESP32 utilizzata per la calibrazione del modulo RTC richiede parametri di input separati per data e ora in forma numerica intera. Un esempio è rappresentato da:

```
adjust(YYYY,MM,DD,hh,mm,ss) → adjust(2024,02,19,13,14,59)
```

 (6)

5.3 Creazione del database

Una volta completata la gestione della comunicazione seriale tra ESP32 e Raspberry per garantire una sincronizzazione accurata, ci siamo dedicati all'elaborazione dei dati ricevuti dall'ESP e al loro archivio all'interno di un database MariaDB. Lo script di riferimento è consultabile in appendice 7.4. Entrando nel merito della memorizzazione dei dati all'interno del database, si entra in un loop principale che esegue ciclicamente i seguenti passaggi:

- **Lettura e decodifica dei dati dalla porta seriale.**

Il metodo `readline()` legge una linea inviata dall'ESP attraverso la comunicazione seriale. Dopo-dichè, questa linea viene decodificata utilizzando l'UTF-8 e vengono ignorati i caratteri non validi. Inoltre, eventuali spazi bianchi in eccesso vengono rimossi tramite l'uso del metodo `strip()`. Il risultato di tale processo viene memorizzato nella variabile chiamata `line`.

- **Divisione della riga in base al carattere #**

La riga viene quindi divisa utilizzando il carattere `#`, il che consente l'estrazione dei singoli valori dal messaggio inviato dall'ESP32.

- **Controllo del messaggio ricevuto**

Una volta convertito il messaggio nel formato stabilito, ne viene verificata sommariamente l'idoneità per il salvataggio sul DB. Questo controllo avviene esaminando che la lunghezza del vettore `line` sia diversa da 1 (indicando che il messaggio contiene più di un valore) e che il primo elemento della lista sia `"M!"`. Queste osservazioni denotano che si tratta di un messaggio presumibilmente valido, cioè che contiene lo stesso numero di campi necessari per richiesti per popolare una riga del database, ma che non ne verifica effettivamente l'integrità come farebbero algoritmi di hashing e di checksum.

- **Assegnazione dei valori alle variabili temporanee**

Se il messaggio è valido, i valori estratti vengono assegnati alle variabili temporanee `new_Data`, `new_Temp`, `new_Pres` e `new_RPM` che rappresentano rispettivamente la data, la temperatura, la pressione e la RPM ricevute dall'ESP32.

- **Controllo della temperatura e invio di alert**

In parallelo viene verificato se la temperatura ricevuta supera una soglia critica prefissata di 33°C. In caso affermativo, viene incrementato un contatore ed eseguita la funzione asincrona `chatbot.funz_alert()` per inviare un alert tramite chatbot, definita all'interno dello script dedicato all'implementazione del chatbot stesso. Se la temperatura scende sotto la soglia di 33°C, il contatore viene azzerato.

- **Inserimento dei dati nel database**

Infine, viene chiamata la funzione `add_measure()` per aggiungere i dati ricevuti al database utilizzando il cursore della connessione al database stesso. Questa funzione prende come argomenti la data, la temperatura, la pressione e gli RPM letti e li inserisce nella tabella del database.

5.4 Visualizzazione della webapp

Terminata la fase di memorizzazione delle acquisizioni provenienti dalla sensoristica connessa ad ESP32, il passo successivo è stato rendere fruibili i dati archiviati all'interno del database attraverso una semplice pagina web per visualizzarli. A tale scopo, è stata sviluppata un'applicazione web utilizzando il framework Flask e lo script di riferimento è consultabile in appendice 7.5.

5.5 Implementazione del chatbot

In particolare, dopo il setup dell'applicazione Flask e la connessione al database, lo script si divide in 3 sezioni:

- **Esecuzioni delle query SQL**

Vengono eseguite due query SQL: la prima per selezionare tutte le righe presenti nel database, e la seconda per selezionare tutte le righe relative agli ultimi 5 minuti, al fine di calcolare valori statistici.

- **Calcolo dei valori statistici**

I dati relativi alla temperatura, alla pressione e gli RPM vengono estratti e utilizzati per calcolare la media di ciascuna grandezza.

- **Trasferimento dei dati alla pagina HTML**

Infine, i dati estratti dal database e i valori medi calcolati vengono passati alla funzione `render_template()` di Flask. Questa funzione carica un file HTML di template chiamato `index.html` e sostituisce le variabili presenti nel template con i valori passati come argomenti.

L'interfaccia grafica, visualizzabile dall'utente, è consultabile in appendice 7.6, con il relativo linguaggio che gestisce il design in appendice 7.7. La schermata è divisa in due parti: una tabella per visualizzare i dati storici e delle card che mostrano la media dei valori nell'ultima ora. La visualizzazione della schermata è mostrata in Figura 16.



Figura 16: Visualizzazione grafica della webapp

5.5 Implementazione del chatbot

Dopo aver completato la gestione dei dati acquisiti dall'ESP e averli resi fruibili in una semplice pagina web, il passo successivo è stato implementare un chatbot Telegram. Questo chatbot ha il compito di inviare un alert a tutti gli utenti registrati nel caso in cui la temperatura ambientale superi una soglia critica prefissata. Gli utenti possono, inoltre, richiedere al bot di generare e inviare un grafico con i dati relativi a un intervallo di tempo passato: 1 minuto, 5 minuti, 30 minuti, 60 minuti. Lo script di riferimento è consultabile in appendice 7.8, le cui principali funzioni sono documentate di seguito:

- **Gestione per l'invio dell'alert**

La funzione `funz_alert()` viene richiamata all'interno dello script `database.py` in risposta al superamento della soglia di temperatura: quando la temperatura registrata supera i 33°C, essa invia un messaggio di alert a tutti gli utenti registrati.

- **Inizializzazione e terminazione del bot**

La funzione `start()` inizializza il bot e salva l'ID dell'utente in un semplice file di testo `bot_users.txt`, dove sono salvati tutti gli utenti registrati al bot.

La funzione `stop()`, invece, termina il bot e rimuove l'ID dell'utente dal file di testo.

- **Gestione per la richiesta del grafico da parte dell'utente**

La funzione `interval_options()` gestisce la richiesta di generazione del grafico da parte degli utenti. A partire dalla richiesta dell'utente, invia una serie di bottoni tramite i quali gli utenti possono selezionare l'intervallo temporale desiderato.

- **Generazione del grafico**

La funzione `funz_grafico()` genera il grafico dei dati relativi all'intervallo temporale selezionato dagli utenti.

Inizialmente esegue una query al database per recuperare i dati dell'intervallo selezionato.

Successivamente, calcola i valori statistici (media, minimo e massimo) dei valori di temperatura, pressione e velocità della ventola.

Inoltre, gestisce il controllo sulla quantità di dati disponibili nel database per l'intervallo richiesto. In dettaglio:

- La condizione `temp_len >= 59 * int(minutes)` controlla se il numero di misurazioni di temperatura disponibili è maggiore o uguale a 59 volte il numero di minuti richiesto. Questo valore, 59, deriva dal fatto che viene eseguita una misurazione ogni secondo e che non si è certi di avere esattamente 60 campioni.
- Se la condizione è vera, significa che ci sono abbastanza dati disponibili nel database per visualizzare l'intervallo richiesto. In tal caso, viene assegnata alla variabile `caption_msg_bot` una stringa che contiene i valori statistici della temperatura, della pressione e della velocità della ventola.
- Se la condizione è falsa, significa che non ci sono abbastanza dati disponibili nel database per l'intervallo richiesto. In questo caso, viene assegnata alla variabile `caption_msg_bot` una stringa che include i valori statistici della temperatura, della pressione e della velocità della ventola, seguiti da un messaggio di avviso che spiega le possibili cause del problema. Queste cause includono la mancanza di dati nel database per visualizzare l'intervallo richiesto o un malfunzionamento del sistema di acquisizione che ha comportato l'interpolazione dei dati mancanti.

Infine genera il grafico, lo salva su disco e restituisce un messaggio contenente i valori statistici.

- **Gestione dei bottoni**

La funzione `button()` gestisce la selezione di un intervallo di tempo tramite i bottoni da parte degli utenti. In particolare, a partire dalla selezione del bottone da parte dell'utente, invia un messaggio di conferma con l'intervallo selezionato. Successivamente chiama la funzione `funz_grafico()` per generare il grafico. Infine, invia il grafico e i valori statistici agli utenti.

- **Gestione dei comandi**

La funzione `gestione_comandi()` crea l'applicazione del bot e associa i comandi inviati dagli utenti alle rispettive funzioni. Infine, avvia il polling per attendere i comandi degli utenti.

Per completezza, vengono presentati due esempi di screenshot che mostrano il chatbot in azione su Telegram, fornendo una dimostrazione pratica delle funzioni appena discusse.

5.5.1 Esempio generazione grafico - disponibilità di dati

Un primo esempio 17 riguarda la generazione del grafico richiesto dall'utente con la disponibilità dei dati nel database per visualizzare l'intervallo richiesto. Come si può notare dal grafico, viene evidenziato un picco sul valore della temperatura, la ventola viene attivata e vengono infatti mostrati relativi aumenti della velocità ogni minuto della ventola stessa.

5.6 Esecuzione dei daemon

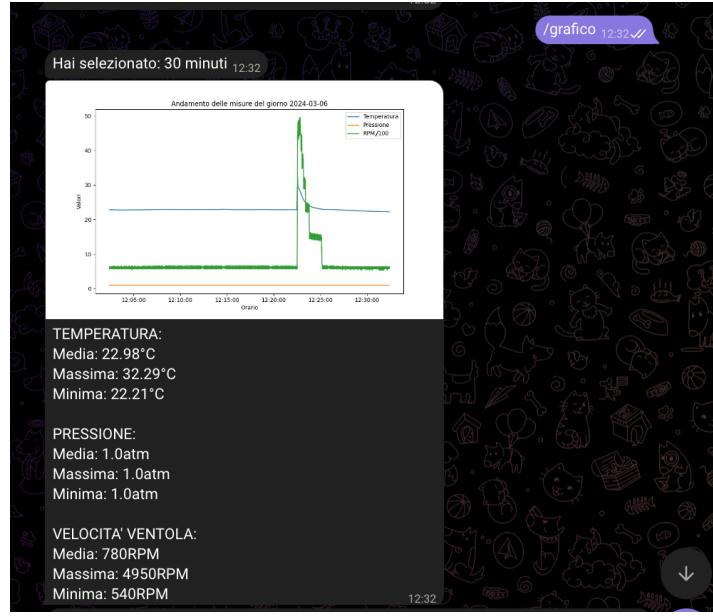


Figura 17: Esempio generazione grafico e valori statistici - dati disponibili

5.5.2 Esempio generazione grafico - mancanza di dati

Un secondo esempio 18 riguarda la generazione del grafico con una scarsa disponibilità dei dati nel database per visualizzare l'intervallo richiesto. Come si può notare, il grafico viene generato lo stesso ma viene seguito da un messaggio di avviso che spiega le possibili cause del problema.

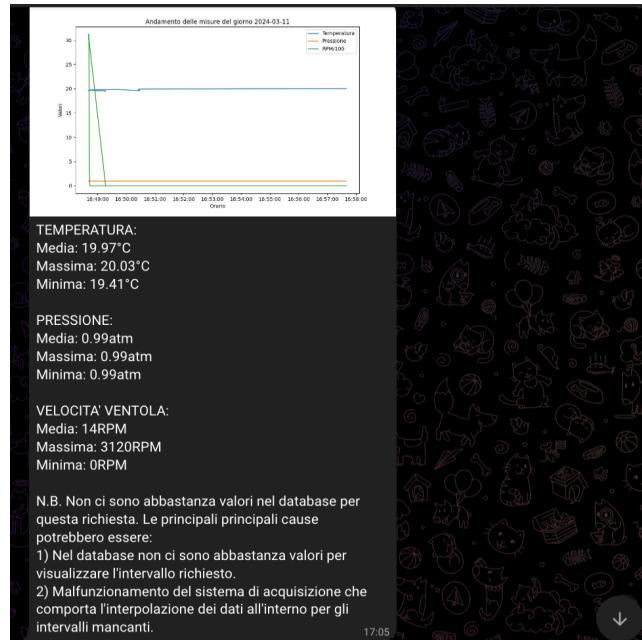


Figura 18: Esempio generazione grafico e valori statistici - mancanza di dati

5.6 Esecuzione dei daemon

L'ultima fase è stata occupata dalla creazione dei servizi di routine al fine di rendere l'intero sistema autonomo e, quindi, non dipendente da un operatore umano. Si sta parlando dei cosiddetti *daemon*, pro-

5.6 Esecuzione dei daemon

cessi di background che eseguono varie funzioni o servizi senza l'interazione diretta degli utenti, essenziali per il funzionamento di molti sistemi informatici.

Nel caso specifico di progetto, sono stati creati da riga di comando 4 daemon, uno per ciascun file implementato per il funzionamento del sistema di monitoraggio ambientale (`sync_timer.py`, `database.py`, `app.py`, `chatbot.py`). Innanzitutto, ad ogni script Python è associato un file bash `.sh`; si tratta sostanzialmente di un file di appoggio per lanciare il codice per l'applicazione che si vuole eseguire, appunto, come daemon. Inoltre, è necessario creare il file di configurazione del demone `.service` che gestisce l'esecuzione del file di appoggio. È stata, successivamente, necessaria l'implementazione di un'ulteriore file `.timer` che funge da trigger.

Prendendo in considerazione la Figura 19, può essere descritta la logica di attivazione dei vari servizi disponibili:

1. Il Timer (in rosso) permette di eseguire periodicamente lo script `sync_timer.py` (in giallo), tramite il servizio `sync_service.service` (in blu) associato al relativo file di appoggio (in verde).
2. Le ultime tre righe del grafico evidenziano una dipendenza rispetto al timer, e quindi al relativo servizio di sincronizzazione, da parte dei servizi associati al database, alla web app e al chatbot. In particolare, il servizio associato al database, `db_service.service`, è reso dipendente dalla prima esecuzione del timer di sincronizzazione; mentre, la web app ed il chatbot dipendono e sono temporalmente conseguenti alla corretta esecuzione del servizio associato al db.

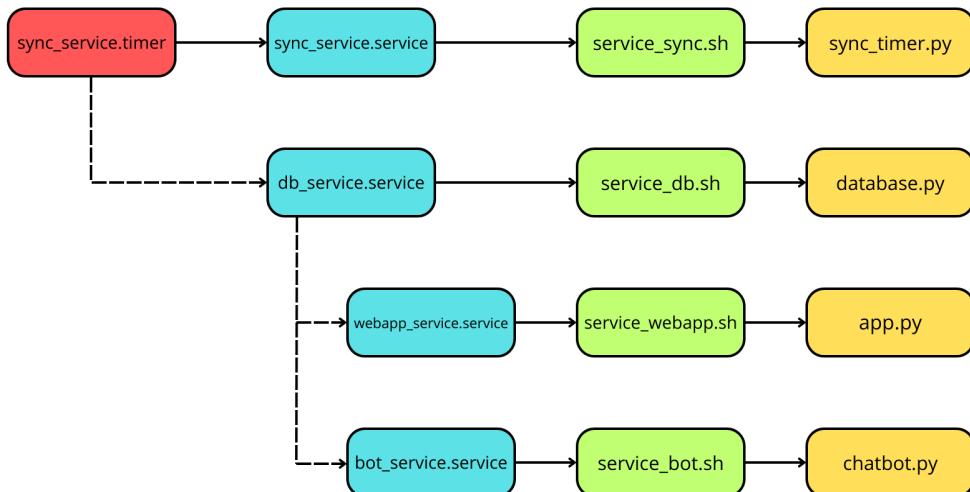


Figura 19: Logica di attivazione dei servizi.

L'implementazione appena descritta garantisce una corretta sincronizzazione temporale del modulo RTC, tramite protocollo NTP, che da il via al popolamento del database. Una volta attivo il servizio associato al database, vengono correttamente avviati i servizi relativi all'applicazione web e al chatbot, i quali possono usufruire correttamente delle informazioni contenute nella base di dati.

Di seguito vengono mostrate le implementazioni relative al servizio ed al timer di sincronizzazione, che richiedono un'analisi a parte. Infine, si andrà ad esporre la configurazione del servizio relativo al database, come caso generale che, previe alcune modifiche relative alle dipendenze, può essere replicata per web app e chatbot.

- **Scrittura del file bash - `service_sync.sh`**

Come anticipato, è stato realizzato il file per la creazione del daemon `service_sync.sh`; si tratta

5.6 Esecuzione dei daemon

sostanzialmente di un file di appoggio per lanciare `sync_timer.py` al cui interno contiene il codice per l'applicazione che si vuole eseguire, appunto, come daemon. Di seguito è mostrato il codice:

```
1 #!/bin/bash
2 source /home/rpi/progett_sod/bin/activate
3 python3 /home/rpi/sync_timer.py
```

Il comando `source /home/rpi/progett_sod/bin/activate` attiva un ambiente virtuale di Python, creato per isolare le dipendenze di un progetto Python e garantire che l'applicazione Python abbia accesso solo a quelle specifiche dipendenze.

Inoltre, è necessario assicurarsi che tale script risulti eseguibile utilizzando il comando:

```
1 sudo chmod +x service_sync.sh
```

- **Creazione del servizio di configurazione del daemon - `sync_service.service`**

Successivamente si va a creare il servizio vero e proprio. È stato, dunque, necessario creare il file `sync_service.service`, all'interno del percorso `/lib/systemd/system/`, che gestisce l'esecuzione del file `service_sync.sh`. Il codice è mostrato di seguito:

```
1 [Unit]
2 Description=Servizio di monitoraggio ambientale - sincronizzazione RTC
3 After=network.target
4
5 [Service]
6 Type=oneshot
7 User=rpi
8 Group=rpi
9 LimitNOFILE=65536
10 ExecStart=/home/rpi/service_sync.sh
11 KillMode=control-group
12 Restart=on-failure
13
14 [Install]
15 WantedBy=multi-user.target
16 Alias=sync_service.service
```

Questo file è diviso in tre sezioni principali:

- **[Unit]**

Definisce la descrizione del servizio e il momento in cui deve essere avviato.

- **[Service]**

Contiene le impostazioni per il servizio stesso, come il tipo di servizio, l'utente e il gruppo che eseguiranno il servizio, il comando di avvio e le opzioni di controllo del servizio. In particolare, il campo `ExecStart` definisce il comando o lo script che lancia il servizio. Inoltre, il campo `Restart` specifica come gestire il riavvio del servizio in caso di fallimento.

- **[Install]**

Specifica quando il servizio deve essere avviato e fornisce un alias per il servizio, che può essere utilizzato per riferirsi al servizio in altri script o file di configurazione. In particolare, il campo `WantedBy` specifica il momento in cui il servizio deve essere avviato, mentre il campo `Alias` fornisce un nome alternativo per il servizio.

- **Configurazione del timer - `sync_service.timer`**

Il timer si occupa di eseguire, ad intervalli più o meno regolari, il servizio relativo la sincronizzazione illustrato nel punto precedente: `sync_service.service`. Per fare ciò si è creato un ulteriore file nell'omonima cartella del servizio da temporizzare, con lo stesso nome del servizio in esame ma con estensione `.timer`. In dettaglio, il timer si occupa di attuare la prima sincronizzazione 10 secondi dopo l'avvio del sistema (`OnBootSec`) e successivamente ogni 60 secondi (`OnUnitActiveSec`).

```
1 [Unit]
2 Description=Avvia il servizio sync_service.service ogni minuto
3
4 [Service]
```

5.6 Esecuzione dei daemon

```
5 OnBootSec=10s
6 OnUnitActiveSec=1min
7 Persistent=false
8
9 [Install]
10 WantedBy=timers.target
```

Una volta salvati tutti i file, è necessario abilitare il timer in modo tale da automatizzare la propria esecuzione all'accensione del sistema; questo è reso possibile dal comando di sistema:

```
1 systemctl enable sync_service.timer
```

Successivamente, è possibile verificare lo stato del servizio di sincronizzazione e del timer rispettivamente tramite i comandi:

```
1 systemctl status sync_service.timer
2 systemctl status sync_service.service
```

Se i precedenti passaggi sono stati effettuati correttamente allora lo stato del timer sarà "enabled", mentre il servizio di sincronizzazione sarà "disabled" ma, dato che quest'ultimo viene attivato dal primo, si noterà alla voce "TriggeredBy" un pallino verde seguito dal nome del file che lo esegue, in questo caso `sync_service.timer`.

Come anticipato nella trattazione, ora si descrive la configurazione del servizio database come caso generale che, ad eccezione di piccole variazioni nelle dipendenze, può essere replicata negli altri due servizi.

- **Scrittura del file bash - service_db.sh**

Analogalmente al caso precedente è stato creato il file di appoggio bash per lanciare lo script Python relativo.

```
1 #!/bin/bash
2 source /home/rpi/progett_sod/bin/activate
3 python3 /home/rpi/database.py
```

- **Creazione del servizio di configurazione del daemon - db_service.service**

Successivamente è creato il servizio associato al database, il quale è stato reso dipendente dal servizio del timer di sincronizzazione tramite i parametri `After` e `Requires`.

```
1 [Unit]
2 Description=Servizio di monitoraggio ambientale - database
3 After=network.target sync_service.timer
4 Requires=sync_service.timer
5
6 [Service]
7 Type=simple
8 User=rpi
9 Group=rpi
10 LimitNOFILE=65536
11 ExecStart=/home/rpi/service_db.sh
12 KillMode=control-group
13 Restart=on-failure
14
15 [Install]
16 WantedBy=multi-user.target
17 Alias=db_service.service
```

- **Abilitazione e avvio del daemon**

Dopo aver configurato il file del daemon, è necessario abilitarlo utilizzando il comando:

```
1 systemctl enable db_service.service
```

Per concludere, in seguito alla configurazione di tutti i daemon, è necessario un reboot della RPi al fine di avviare in modo ordinato e automatizzato tutte le operazioni di raccolta, visualizzazione ed elaborazione dati previste dalle specifiche di progetto.

6 Conclusioni

In conclusione, il progetto ha permesso di realizzare con successo un sistema IoT basato su ESP32 e Raspberry Pi, con l'obiettivo di monitorare i parametri ambientali e controllare una ventola per mantenere la temperatura all'interno di un range desiderato. Durante lo sviluppo del progetto, sono state affrontate sfide tecniche legate alla programmazione, all'integrazione dei componenti hardware, alla gestione e sincronizzazione dei dati.

Grazie all'utilizzo di sensori come il BMP280 e un display OLED, siamo stati in grado di acquisire e visualizzare con precisione i dati ambientali, mentre la ventola Noctua NF-A4x10 ci ha fornito un controllo affidabile della temperatura all'interno del sistema. L'implementazione di un server IoT su Raspberry Pi ci ha permesso di archiviare e analizzare i dati raccolti tramite una semplice interfaccia web, mentre l'integrazione con un bot Telegram ha consentito agli utenti di ricevere alert e visualizzare i dati graficamente. La sincronizzazione tra le due schede, mediante il modulo RTC, ha permesso di coordinare le operazioni tra i vari componenti e di associare il relativo timestamp alle misure raccolte dall'edge device. Va notato che il modulo RTC può essere evitato, poiché l'ESP32 supporta una scheda Wi-Fi che permette l'interrogazione diretta del server NTP per ottenere orario e data. Questa funzionalità rende il sistema più semplice e meno dipendente dai componenti hardware aggiuntivi, riducendo così i costi e la complessità. Tuttavia, è importante gestire correttamente le riconnessioni Wi-Fi e i possibili ritardi nella sincronizzazione NTP per garantire un'accuratezza temporale adeguata alle esigenze del progetto. Inoltre, per progetti più complessi e strutturati, è opportuno utilizzare un metodo di controllo di integrità dei messaggi di tipo CRC o simili (MD5, SHA). L'uso di checksum (o di digest) è essenziale per garantire l'integrità dei dati trasmessi, soprattutto in ambienti rumorosi o quando si utilizzano reti wireless. Implementare un controllo di integrità dei messaggi aiuta a rilevare e correggere eventuali errori di trasmissione, migliorando la robustezza e l'affidabilità del sistema.

In futuro, il sistema potrebbe essere migliorato ulteriormente con l'aggiunta di nuove caratteristiche come: il controllo remoto della ventola, ulteriori funzionalità da parte del chatbot, implementazione di meccanismi di sicurezza informatica e l'espansione della capacità di archiviazione e analisi dei dati, mediante tecniche di data science, da parte della RPi. Nel complesso, il progetto ha fornito una solida base per esplorare ulteriormente le applicazioni degli ambienti IoT e ha dimostrato il potenziale delle tecnologie utilizzate e la capacità di interfacciarsi tra loro.

7 Appendice

7.1 requirements.txt

```
1 aiohttp==3.9.1
2 aiosignal==1.3.1
3 anyio==4.2.0
4 attrs==23.2.0
5 blinker==1.7.0
6 certifi==2023.11.17
7 click==8.1.7
8 contourpy==1.2.0
9 cycler==0.12.1
10 Flask==3.0.1
11 fonttools==4.47.2
12 frozenlist==1.4.1
13 h11==0.14.0
14 httpcore==1.0.2
15 httpx==0.25.2
16 idna==3.6
17 itsdangerous==2.1.2
18 Jinja2==3.1.3
19 kiwisolver==1.4.5
20 mariadb==1.1.9
21 MarkupSafe==2.1.3
22 matplotlib==3.8.2
23 multidict==6.0.4
24 ntplib==0.4.0
25 numpy==1.26.3
26 packaging==23.2
27 pillow==10.2.0
28 pyparsing==3.1.1
29 pyserial==3.5
30 python-dateutil==2.8.2
31 python-telegram-bot==20.7
32 six==1.16.0
33 sniffio==1.3.0
34 urllib3==2.1.0
35 Werkzeug==3.0.1
36 yarl==1.9.4
```

7.2 esp32freeRTOS.ino

```
1 #include "RTClib.h"
2 #include <Adafruit_GFX.h>
3 #include <Adafruit_SSD1306.h>
4 #include <Adafruit_BMP280.h>
5 #include <Wire.h>
6
7 #define CANALE 0
8 #define RISOLUZIONE_PWM 8
9 #define FREQUENZA 5000
10 #define PIN_PWM 17
11
12 #define PIN_TACHO 16
13 #define SOGLIA 28
14 #define PIN_COUNTDOWNINT 0
15
16 //Inizializzazione variabili globali utili per l'RPM
17 volatile bool countdownInterruptTriggered = false;
18 volatile int numCountdownInterrupts = 0;
19 volatile unsigned long counter = 0;
20 int fanSpeed=0;
21 String oldDate;
22
23 //struttura contenente i dati acquisiti da utilizzare per lo scambio di messaggi in coda
24 typedef struct{
```

7.2 esp32freeRTOS.ino

```
25     float temperatura;
26     float pressione;
27     long rpm;
28     String time_stamp;
29 } message_t;
30
31
32 //Funzione per contare gli impulsi per la lettura della velocità della ventola
33 // (tachimetro)
34 void countPulses() {
35     counter++;
36 }
37
38 //Funzione per contare gli impulsi da parte di RTC che emette in 1 secondo
39 void countdownOver () {
40     countdownInterruptTriggered = true;
41     numCountdownInterrupts++;
42 }
43
44 //Funzione per confrontare la data prima dell'invio di ogni messaggio su seriale (per
45 // evitare l'invio di messaggi identici)
46 bool dateCheck(String newDate){
47     if (oldDate == newDate){
48         return 0;
49     }
50     else if (oldDate != newDate){
51         oldDate = newDate;
52         return 1;
53     }
54 }
55
56 //Definizione dei task
57 void TaskSerial(void *pvParameters);
58 void Task_Sensors(void *pvParameters);
59 void Task_Sync(void *pvParameters);
60
61 //Definizione della coda
62 QueueHandle_t QueueHandle;
63 const int QueueElementSize = 10;
64
65 //Definizioni delle variabili oggetto dei tre dispositivi
66 Adafruit_BMP280 bmp;
67 Adafruit_SSD1306 display(128, 64, &Wire, -1);
68 RTC_PCF8523 rtc;
69
70 void setup() {
71     Serial.begin(115200);
72     while(!Serial){delay(10);}
73     delay(5000); //tempo necessario per la corretta inizializzazione della seriale da
74     // parte di ESP32 ed RPI
75
76     //inizializzazione dei tre dispositivi:
77
78     //BMP280
79     unsigned status;
80     status = bmp.begin();
81     if (!status) {
82         Serial.println("Errore BMP");
83     }
84     bmp.setSampling(Adafruit_BMP280::MODE_NORMAL,           // Modalità operativa
85                     Adafruit_BMP280::SAMPLING_X2,           // Campionamento della temperatura
86                     Adafruit_BMP280::SAMPLING_X16,          // Campionamento della pressione
87                     Adafruit_BMP280::FILTER_X16,            // Filtraggio dei segnali
88                     Adafruit_BMP280::STANDBY_MS_500); // Tempo di stand-by
89
90     //OLED
91     if (!display.begin(SSD1306_SWITCHCAPVCC, 0x3D)) {
92         Serial.println("Errore OLED");
93     }
94 }
```

7.2 esp32freeRTOS.ino

```
92     display.clearDisplay();
93     display.display();
94
95     //RTC
96     if (! rtc.begin()) {
97         Serial.println("Couldn't find RTC");
98         while (1) delay(10);
99     }
100    if (! rtc.initialized() || rtc.lostPower()) {
101        rtc.enableCountdownTimer(PCF8523_Frequency64Hz, 64, PCF8523_LowPulse8x64Hz);
102    }
103    rtc.start();
104
105    //Creazione della coda
106    QueueHandle = xQueueCreate(QueueElementSize, sizeof(message_t));
107    if(QueueHandle == NULL){
108        while(1) delay(1000);
109    }
110
111    //Creazione dei task
112    xTaskCreate(
113        TaskSerial
114        , "TaskSerial"
115        , 2048
116        , NULL
117        , 2
118        , NULL
119    );
120
121    xTaskCreate(
122        Task_Sensors
123        , "Task_Sensors"
124        , 2048
125        , NULL
126        , 1
127        , NULL
128    );
129
130    xTaskCreate(
131        Task_Sync
132        , "TaskSync"
133        , 2048
134        , NULL
135        , 0
136        , NULL
137    );
138
139    //inizializzazione PWM
140    ledcSetup(CANALE, FREQUENZA, RISOLUZIONE_PWM);
141    ledcAttachPin(PIN_PWM, CANALE);
142    ledcWrite(CANALE,0);
143
144    //Creazione degli interrupt associando i pin alle funzioni definite in precedenza
145    pinMode(PIN_COUNTDOWNINT, INPUT_PULLUP);
146    pinMode(PIN_TACHO, INPUT_PULLUP);
147    attachInterrupt(digitalPinToInterruption(PIN_TACHO), countPulses, FALLING); //RPM
148    attachInterrupt(digitalPinToInterruption(PIN_COUNTDOWNINT), countdownOver, FALLING);
149    //RTC
150}
151 void loop(){
152}
153
154 /*-----*/
155 /*----- Tasks -----*/
156 /*-----*/
157
158 //Task dedicato alla trasmissione dei dati tramite protocollo seriale alla RPI e tramite
159 // I2C al display oled
160 void TaskSerial(void *pvParameters){
```

7.2 esp32freeRTOS.ino

```

160     message_t message;
161     for (;;) {
162       if (QueueHandle != NULL) {
163         int ret = xQueueReceive(QueueHandle, &message, portMAX_DELAY);
164         if (ret == pdPASS) {
165           Serial.println("M!#" + message.time_stamp + "#" + message.temperatura +
166                         "#" + message.pressione + "#" + message.rpm);
167
168           display.clearDisplay();
169           display.setTextColor(WHITE);
170           display.setTextSize(1);
171           display.setCursor(0, 0);
172           display.print("Temperatura: ");
173           display.print(message.temperatura);
174           display.println((char)247);
175           display.println("C");
176           display.print("Pressione: ");
177           display.print(message.pressione);
178           display.println(" atm");
179           display.print("RPM:");
180           display.print(message.rpm);
181           display.display();
182         } else if (ret == pdFALSE) {
183       }
184     }
185   }
186 }
187
188 //Task dedicato alla lettura della temperatura e pressione (BMP280)
189 void Task_Sensors(void *pvParameters){
190   message_t message;
191   TickType_t start_time = xTaskGetTickCount();
192   for (;;) {
193     DateTime now = rtc.now();
194     message.time_stamp = (String)now.year() + "-" + (String)now.month() + " - "
195       + (String)now.day() + " " + (String)now.hour() + ":" + (String)now.minute() + ":" +
196       (String)now.second();
197     message.temperatura = bmp.readTemperature();
198     message.pressione = bmp.readPressure() * 9.8692 * pow(10, -6);
199     //map e constrain hanno l'obiettivo di rendere proporzionale la velocit della
200     //ventola all'aumentare
201     //della temperatura limitando il valore associato alla PWM in un range 0-255
202     fanSpeed = map(message.temperatura, SOGLIA, 32, 0, 255);
203     ledcWrite(CANALE, constrain(fanSpeed, 0, 255));
204
205     //conversione degli impulsi in RPM ad ogni secondo
206     if(countdownInterruptTriggered && numCountdownInterrups == 1){
207       message.rpm = counter * 60 / 2;
208       counter = 0;
209       countdownInterruptTriggered = false;
210       numCountdownInterrups = 0;
211     }
212     if(dateCheck(message.time_stamp)){
213       xQueueSend(QueueHandle, &message, portMAX_DELAY);
214     }
215     vTaskDelayUntil(&start_time, pdMS_TO_TICKS(100));
216   }
217
218
219 //Task per la sincronizzazione del modulo RTC tramite seriale al fine di mantenere
220 //l'orario corretto
221 void Task_Sync(void* pvParameters) {
222   message_t message;
223
224   //dichiarazioni di variabili utili per la decodifica del messaggio
225   int delimiter_start, delimiter_end;
226   String string, msg_rx[7];
227   for(;;){
```

7.3 sync_timer.py

```
228     if (Serial.available() > 0) {
229         string = Serial.readString();
230         delimiter_start = string.indexOf("#"); //posizione del primo valore "#" nel
231         messaggio ricevuto
232
233         //Ciclo per lo split della stringa arrivata
234         for(int idx_str = 0; idx_str < 7; idx_str++){
235
236             //posizione del successivo valore "#" rispetto a quello contenuto in
237             delimiter_start
238             delimiter_end = string.indexOf("#", delimiter_start+1);
239             //allocazione del valore contenuto tra due "#" nella i-esima cella della
240             //stringa msg_rx
241             msg_rx[idx_str] = string.substring(delimiter_start+1, delimiter_end);
242             delimiter_start = delimiter_end;
243         }
244
245         //Verifica se la stringa ricevuta "sync" ed effettua l'eventuale
246         sincronizzazione e controllo dei valori
247         if (msg_rx[0]== "sync" && msg_rx[1].toInt() >= 2024 && msg_rx[2].toInt()<=12 &&
248             msg_rx[3].toInt()<=31 && msg_rx[4].toInt()<= 23 && msg_rx[5].toInt()<= 59 &&
249             msg_rx[6].toInt()<= 59 ) {
250             rtc.adjust(DateTime(msg_rx[1].toInt(),msg_rx[2].toInt(),msg_rx[3].toInt(),
251                         msg_rx[4].toInt(),msg_rx[5].toInt(),msg_rx[6].toInt()));
252             rtc.start();
253             vTaskDelay(pdMS_TO_TICKS(100));
254         }
255     }
256 }
```

7.3 sync_timer.py

```
1 import ntplib
2 #from time import sleep
3 import serial
4 from datetime import datetime, timezone
5
6 # Indirizzo di un server NTP (puoi utilizzare un server NTP locale o uno pubblico)
7 ntp_server = 'pool.ntp.org'
8
9 # Funzione per ottenere l'orario dal server NTP
10 def get_ntp_time():
11     client = ntplib.NTPClient()
12     response = client.request(ntp_server)
13     return response.tx_time
14
15 #Inizializzazione porta seriale
16 port = '/dev/ttyUSB0'
17 # Specifica la velocità di trasmissione (baud rate)
18 baud_rate = 115200
19 # Apre la connessione seriale
20 ser = serial.Serial(port, baud_rate)
21 try:
22     # Ottiene l'orario iniziale dal server NTP e synchronizza il modulo RTC all'avvio
23     # del sistema
24     start_time = get_ntp_time() #tipo float, sono il numero di secondi passati
25     # dall'inizio del tempo
26     date_msg2send = str(datetime.fromtimestamp(start_time)).split(".")
27     date_msg2send = date_msg2send[0].replace(" ", "#").replace(":", "#").replace("-", "#")
28     msg = "#sync#" + date_msg2send + "#"
29     #Scrittura su seriale del messaggio di sincronizzazione
30     #sync#anno#mese#giorno#ora#minuti#secondi#
31     ser.write(msg.encode())
32
33 except KeyboardInterrupt:
```

7.4 database.py

```
32     #Chiusura comunicazione
33     ser.close()
34     print("Contatore spento")
```

7.4 database.py

```
1 import serial
2 import mariadb
3 import sys
4 import chatbot
5 import asyncio
6 import ntplib
7 from time import sleep
8 from datetime import datetime, timezone
9
10 # Aggiungo una singola misurazione
11 def add_measure(cur, Data, Temperatura, Pressione, RPM):
12     cur.execute("INSERT INTO misure(Data, Temperatura, Pressione, RPM) VALUES (?, ?, ?, ?)",
13                 (Data, Temperatura, Pressione, RPM))
14
15 # Inizializzazione porta seriale
16 porta_seriale = '/dev/ttyUSB0'
17 # Specifica la velocità di trasmissione (baud rate)
18 velocita_trasmissione = 115200
19 # Apre la connessione seriale
20 ser = serial.Serial(porta_seriale, velocita_trasmissione)
21
22 # Inizializzazione contatore per la gestione dell'alert
23 contatore = 0
24
25 # Crea la connessione database tramite connettore python
26 try:
27     conn = mariadb.connect(
28         host="127.0.0.1",
29         port=3306,
30         user="root",
31         password="root",
32         autocommit=True,
33         database="SensorData")
34
35 except mariadb.Error as e:
36     print(f"Error connecting to the database: {e}")
37     sys.exit(1)
38
39 # Instanza il cursore
40 cur = conn.cursor()
41
42 # Lettura dei dati da seriale e successivo inserimento nel db
43 try:
44     while True:
45         # Lettura del messaggio e split
46         line = ser.readline().decode("utf-8", "ignore").strip()
47         line = line.split("#")
48         # controllo integrità del messaggio ricevuto
49         if(len(line) != 1) and (line[0] == "M!"):
50             print(line)
51             # assegnazione dei valori in variabili temporanee
52             new_Data = line[1]
53             new_Temp = float(line[2])
54             new_Pres = float(line[3])
55             new_RPM = int(line[4])
56
57             # controllo per l'invio dell'alert
58             if new_Temp > 33:
59                 contatore= contatore+1
60                 asyncio.run(chatbot.funz_alert(True, contatore))
61             else:
```

7.5 app.py

```
62         contatore = 0
63
64         #chiamata della funzione per il salvataggio dei dati nel db
65         add_measure(cur,new_Data,new_Temp,new_Pres,new_RPM)
66
67     else:
68         print(line) #stampa a video eventuali altri messaggi
69
70 except KeyboardInterrupt:
71     ser.close()
72     print("Connessione al dispositivo ESP32 chiusa.")
73     conn.close()
74     print("Connessione al Database 'SensorData' chiusa.")
```

7.5 app.py

```
1 from flask import Flask, render_template
2 import mariadb
3 import sys
4 from numpy import mean, float16, float32
5 import datetime
6
7 #setup app
8 app = Flask(__name__)
9
10 # Definizione della rotta principale
11 @app.route('/')
12 def index():
13
14     try:
15         # Connessione al database MariaDB
16         conn = mariadb.connect(
17             host="127.0.0.1",
18             port=3306,
19             user="root",
20             password="root",
21             autocommit=True,
22             database="SensorData"
23         )
24     except mariadb.Error as e:
25         print(f"Errore connessione al database: {e}")
26         sys.exit(1)
27
28     cur = conn.cursor()
29
30     # Esecuzione della query SQL per selezionare tutte le righe dalla tabella
31     # 'misure',
32     cur.execute("SELECT * FROM misure ORDER BY id DESC")
33     data = cur.fetchall()
34
35     #Esecuzione della query SQL per selezionare tutte le righe nell'intervallo di 5
36     # minuti
37     cur.execute("SELECT * FROM misure WHERE Data >= NOW() - INTERVAL 5 MINUTE AND
38     Data < NOW()")
39     data2 = cur.fetchall()
40
41     conn.close()
42
43     #estrazione dei dati e calcolo dei valori statistici
44     temperatura = [idx_data2[2] for idx_data2 in data2]
45     pressione = [idx_data2[3] for idx_data2 in data2]
46     rpm = [idx_data2[4] for idx_data2 in data2]
47
48     temp_mean = mean(temperatura, dtype=float16)
49     pres_mean = mean(pressione, dtype=float16)
50     rpm_mean = mean(rpm, dtype=float32)
51
52     # Trasferimento dei dati alla pagina HTML usando un template Flask
```

7.6 index.html

```
50     return render_template('index.html', data=data, temp_mean=temp_mean,
51                           pres_mean=pres_mean, rpm_mean=rpm_mean)
52 #Punto di ingresso dell'app Flask
53 if __name__ == '__main__':
54     #Avvio app in modalità debug
55     app.run(host='0.0.0.0', port=5000, debug=True)
```

7.6 index.html

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <!-- <meta http-equiv="refresh" content="10"> Ricarica la pagina ogni 300
6          secondi (5 minuti) -->
7      <meta name="viewport" content="width=device-width", initial-scale=1.0>
8      <title>Progetto SOD</title>
9      <link rel="stylesheet" type="text/css" href="{{ url_for('static',
10                                filename='style.css') }}">
11
12  </head>
13  <body>
14      <div class="titolo">
15          <h1>Progetto Sistemi Operativi Dedicati
16              <span>Sistema di acquisizione dati per il monitoraggio ambientale</span>
17          </h1>
18      </div>
19      <div class="container">
20          <div class="left-column">
21              <h2>Database - storico dei dati</h2>
22              <div class="table-container">
23                  <table border="1">
24                      <tr>
25                          <th>Data</th>
26                          <th>Temperatura</th>
27                          <th>Pressione</th>
28                          <th>RPM</th>
29                      </tr>
30                      <!-- Cicla attraverso i dati passati dall'app Flask --&gt;
31                      {% for row in data %}
32                          &lt;tr&gt;
33                              &lt;td&gt;{{ row[1] }}&lt;/td&gt;
34                              &lt;td&gt;{{ row[2] }}&lt;/td&gt;
35                              &lt;td&gt;{{ row[3] }}&lt;/td&gt;
36                              &lt;td&gt;{{ row[4] }}&lt;/td&gt;
37                          &lt;/tr&gt;
38                      {% endfor %}
39                  &lt;/table&gt;
40          &lt;/div&gt;
41      &lt;/div&gt;
42      &lt;div class="right-column"&gt;
43          &lt;h2&gt;Media dei valori dell'ultima ora&lt;/h2&gt;
44          &lt;div class="card"&gt;
45              &lt;h3&gt;TEMPERATURA&lt;/h3&gt;
46              &lt;p&gt;{{temp_mean}} C&lt;/p&gt;
47          &lt;/div&gt;
48          &lt;div class="card"&gt;
49              &lt;h3&gt;PRESSIONE&lt;/h3&gt;
50              &lt;p&gt; {{pres_mean}} atm &lt;/p&gt;
51          &lt;/div&gt;
52          &lt;div class="card"&gt;
53              &lt;h3&gt;VELOCITA' VENTOLA &lt;/h3&gt;
54              &lt;p&gt; {{rpm_mean}} rpm&lt;/p&gt;
55          &lt;/div&gt;
56      &lt;/div&gt;</pre>
```

7.7 style.css

```
57 </div>
58 </body>
59 </html>
```

```
1 /* STILE TITOLO */
2 .titolo h1 {
3     text-align:center; font-size:50px; text-transform:uppercase; color:#222;
4     letter-spacing:1px;
5     font-family: "Playfair Display", serif; font-weight:400;
6 }
7 .titolo h1 span {
8     margin-top: 5px;
9     font-size:15px; color:#444; word-spacing:1px; font-weight:normal;
10    letter-spacing:2px;
11    text-transform: uppercase; font-family: "Raleway", sans-serif; font-weight:500;
12
13    display: grid;
14    grid-template-columns: 1fr max-content 1fr;
15    grid-template-rows: 27px 0;
16    grid-gap: 20px;
17    align-items: center;
18 }
19 .titolo h1 span:after,.nine h1 span:before {
20     content: " ";
21     display: block;
22     margin-left: 20px;
23     margin-right: 20px;
24     border-bottom: 1px solid #c50000;
25     border-top: 1px solid #c50000;
26     height: 5px;
27     background-color:#f8f8f8;
28 }
29 /* TABLE CONTAINER */
30 .table-container {
31     border: 2px solid #dddddd;
32     padding: 10px;
33     width: 90%;
34     margin: 10px;
35     overflow-x: auto;
36     max-height: 500px;
37     position: relative;
38 }
39 table {
40     width: 100%;
41     border-collapse: collapse;
42 }
43 th, td {
44     border: 1px solid #dddddd;
45     padding: 8px;
46     text-align: center;
47     font-family: "Raleway", sans-serif;
48 }
49 th {
50     background-color: #f2f2f2;
51     position: sticky;
52     top: 0;
53 }
54 .container {
55     display: flex;
56 }
57 }
58
59 .left-column {
60     flex: 2;
61     padding: 20px;
```

7.8 chatbot.py

```
62     background-color: #f0f0f0;
63
64 }
65
66 .right-column {
67     flex: 1;
68     padding: 20px;
69     background-color: #e0e0e0;
70     display: flex;
71     flex-direction: column;
72     align-items: center;
73 }
74
75 .card {
76     margin-top: 5px;
77     width: 40%;
78     padding: 10px;
79     margin-bottom: 20px;
80     height: 100px;
81     border: 1px solid #ddd;
82     border-radius: 8px;
83     background-color: #ffffff;
84     font-size: 15px; color: #444; word-spacing: 1px; font-weight: normal;
85     letter-spacing: 2px;
86     text-transform: uppercase; font-family: "Raleway", sans-serif; font-weight: 500;
87     flex-direction: column;
88     align-items: center;
89     box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1);
90 }
91
92 .card h3 {
93     text-align: center;
94     font-size: 15px;
95     color: #444;
96     word-spacing: 1px;
97     font-weight: normal;
98     letter-spacing: 2px;
99     text-transform: uppercase;
100    font-family: "Raleway", sans-serif;
101    font-weight: 500;
102 }
103
104 .left-column h2, .right-column h2 {
105     margin-top: 5px;
106     color: #333;
107     font-size: 24px;
108     margin-bottom: 10px;
109     text-align: center;
110     font-family: "Playfair Display", serif;
111 }
112
113 p {
114     text-align: center;
115     font-size: 20px;
116     font-family: "Raleway", sans-serif;
117 }
```

7.8 chatbot.py

```
1 import telegram
2 import asyncio
3 from telegram import Update, InlineKeyboardButton, InlineKeyboardMarkup
4 from telegram.ext import Application, CommandHandler, ContextTypes,
5                           CallbackQueryHandler
6 import serial
7 import mariadb
8 import sys
9 import os
9 from datetime import datetime, timedelta
```

7.8 chatbot.py

```
10 import matplotlib.pyplot as plt
11 import matplotlib.dates as mdates
12 import logging
13 from numpy import mean, min, max, float16, float32
14
15 #Variabili globali
16 alert = "La temperatura ha superato la soglia critica"
17 users_file_path = "/home/rpi/bot_users.txt"
18 tokenbot_file_path = "/home/rpi/bot_token.txt"
19
20 #Lettura token del chatbot da un file di testo
21 with open(tokenbot_file_path,'r') as token_f:
22     TOKEN = token_f.read()
23
24 #Gestore per l'invio dell'alert
25 async def funz_alert(trigger, contatore):
26     bot = telegram.Bot(token=TOKEN)
27     if trigger == True and contatore == 1:
28         async with bot:
29             with open(users_file_path, "r") as users_f:
30                 chat_users = [line.strip() for line in users_f.readlines()]
31                 for id in chat_users:
32                     await bot.send_message(chat_id=id, text=alert)
33
34 #Inizializzazione del bot e salvataggio dell'id utente nel file di testo
35 async def start(update: Update, context: ContextTypes.DEFAULT_TYPE) -> None:
36     user_id = update.message.from_user.id
37     with open(users_file_path, "r") as users_f:
38         users_list = [line.strip() for line in users_f.readlines()]
39         if str(user_id) not in users_list: #controllo sull'id
40             users_list.append(user_id)
41         with open(users_file_path, "w") as users_file:
42             for list_idx in users_list:
43                 users_file.write(str(list_idx)+"\n")
44             await update.message.reply_text("Benvenuto, questo bot offre il servizio
45                                         di monitoraggio ambientale. Digita /grafico per richiedere i dati
46                                         storici di un intervallo di tempo passato.")
47     else:
48         await update.message.reply_text("L'utente "+f'{user_id}'+ " ha già
49                                         inizializzato il bot.")
50
51 #stop ed eliminazione dell'id utente dal file di testo
52 async def stop(update: Update, context: ContextTypes.DEFAULT_TYPE) -> None:
53     user_id = update.message.from_user.id
54     with open(users_file_path, "r") as users_f:
55         users_list = [line.strip() for line in users_f.readlines()]
56         if str(user_id) in users_list: #controllo sull'id
57             users_list.remove(str(user_id)) #rimozione id
58             with open(users_file_path, "w") as users_file:
59                 for list_idx in users_list:
60                     users_file.write(str(list_idx)+"\n")
61             await update.message.reply_text("Bot terminato. Digita /start per
62                                         avviarlo nuovamente.")
63
64 #Richiesta del grafico da parte del client e risposta del bot tramite buttoni
65 async def interval_options(update: Update, context: ContextTypes.DEFAULT_TYPE) ->
66     None:
67     bot = telegram.Bot(token = TOKEN)
68     user_id = update.message.from_user.id
69     with open(users_file_path, "r") as users_f:
70         users_list = [line.strip() for line in users_f.readlines()]
71         if str(user_id) in users_list: #controllo sull'id
72
73             #buttoni per la scelta dell'intervallo
74             keyboard = [
75                 [ InlineKeyboardButton("1 minuto", callback_data="1"),
76                   InlineKeyboardButton("5 minuti", callback_data="5")],
77                 [ InlineKeyboardButton("30 minuti", callback_data="30"),
78                   InlineKeyboardButton("60 minuti", callback_data="60")],
```

7.8 chatbot.py

```
75         ]
76
77     reply_markup = InlineKeyboardMarkup(keyboard)
78     await update.message.reply_text("Quale intervallo temporale vuoi
79     selezionare?", reply_markup=reply_markup)
80 else:
81     await bot.send_message(chat_id = user_id, text = "Il bot non      stato
82     inizializzato. Inviare /start e poi richiedere il grafico.")
83
84 #Funzione per l'invio del grafico tramite il bot
85 async def funz_grafico(minutes):
86
87     # Crea la connessione al database
88     try:
89         conn = mariadb.connect(
90             host="127.0.0.1",
91             port=3306,
92             user="root",
93             password="root",
94             autocommit=True,
95             database="SensorData"
96         )
97     except mariadb.Error as e:
98         print(f"Error connecting to the database: {e}")
99     return
100
101 # Istanzia il cursore
102 cur = conn.cursor()
103 # Query per ottenere i dati nell'intervallo selezionato
104 try:
105     match minutes:
106         case "1":
107             cur.execute("SELECT * FROM misure WHERE Data >= NOW() - INTERVAL 1
108                         MINUTE AND Data < NOW()")
109         case "5":
110             cur.execute("SELECT * FROM misure WHERE Data >= NOW() - INTERVAL 5
111                         MINUTE AND Data < NOW()")
112         case "30":
113             cur.execute("SELECT * FROM misure WHERE Data >= NOW() - INTERVAL 30
114                         MINUTE AND Data < NOW()")
115         case "60":
116             cur.execute("SELECT * FROM misure WHERE Data >= NOW() - INTERVAL 60
117                         MINUTE AND Data < NOW()")
118
119     rows = cur.fetchall()
120
121     except mariadb.Error as e:
122         print(f"Errore per l'esecuzione della query: {e}")
123     conn.close()
124     return
125
126 # Chiusura della connessione al database
127 conn.close()
128
129 # Estrazione dei dati
130 timestamps = [row[1] for row in rows]
131 temperatura = [row[2] for row in rows]
132 pressione = [row[3] for row in rows]
133 rpm = [row[4] for row in rows]
134 rpm_norm = [idx_rpm/100 for idx_rpm in rpm]
135
136 #Calcolo dei valori statistici
137 temp_mean = mean(temperatura, dtype=float16)
138 temp_min = min(temperatura)
139 temp_max = max(temperatura)
140 temp_len = len(temperatura)
141
142 pres_mean = mean(pressione, dtype=float16)
143 pres_min = min(pressione)
144 pres_max = max(pressione)
```

7.8 chatbot.py

```
139
140     rpm_mean = round(mean(rpm, dtype = float32))
141     rpm_min = min(rpm)
142     rpm_max = max(rpm)
143
144     #istanziazione del messaggio contenente i valori statistici
145     temp_msg = "TEMPERATURA:\n"+ "Media: "+str(temp_mean)+', C \n'+ "Massima:
146         "+str(temp_max)+', C \n'+ "Minima: "+str(temp_min)+', C \n'
147     pres_msg = "PRESSIONE:\n"+ "Media: "+str(pres_mean)+', atm\n'+ "Massima:
148         "+str(pres_max)+', atm\n'+ "Minima: "+str(pres_min)+', atm\n',
149     rpm_msg = "VELOCITA' VENTOLA:\n"+ "Media: "+str(rpm_mean)+', RPM\n'+ "Massima:
150         "+str(rpm_max)+', RPM\n'+ "Minima: "+str(rpm_min)+', RPM\n',
151
152     #controllo sulla quantit di dati a disposizione
153     if (temp_len >= 59*int(minutes)):
154         caption_msg_bot = temp_msg+"\n"+pres_msg+"\n"+rpm_msg
155     else:
156         nb("N.B. Non ci sono abbastanza valori nel database per questa richiesta.
157             Le principali principali cause potrebbero essere: \n"
158             "1) Nel database non ci sono abbastanza valori per visualizzare l'intervallo
159                 richiesto.\n"
160             "2) Malfunzionamento del sistema di acquisizione che comporta
161                 l'interpolazione dei dati all'interno per gli intervalli mancanti.")
162         caption_msg_bot = temp_msg+"\n"+pres_msg+"\n"+rpm_msg+"\n"+nb
163
164     # Generazione del grafico
165     plt.figure(figsize=(10,6))
166     plt.plot(timestamps, temperatura, label='Temperatura')
167     plt.plot(timestamps, pressione, label='Pressione')
168     plt.plot(timestamps, rpm_norm, label='RPM/100')
169     plt.xlabel('Orario')
170     plt.ylabel('Valori')
171     plt.title('Andamento delle misure del giorno
172                 '+str(timestamps[1].strftime("%Y-%m-%d")))
173     plt.legend()
174     myFmt = mdates.DateFormatter('%H:%M:%S')
175     plt.gca().xaxis.set_major_formatter(myFmt)
176
177     #Salvataggio dell'immagine su disco
178     image_path = "/home/rpi/grafico.png"
179     plt.savefig(image_path)
180     plt.close()
181
182     return caption_msg_bot
183
184 #Funzione per la gestione del bottone selezionato dal client
185 async def button(update: Update, context: ContextTypes.DEFAULT_TYPE) -> None:
186
187     query = update.callback_query
188     await query.answer()
189     await query.edit_message_text(text=f"Hai selezionato: {query.data} minuti")
190     msg = await funz_grafico(minutes = query.data) #chiamata della funzione per la
191         generazione del grafico
192     await query.message.reply_photo(photo= open("/home/rpi/grafico.png", 'rb'),
193         caption=msg) #invio immagine del grafico
194     os.remove("/home/rpi/grafico.png") #una volta inviata l'immagine viene rimossa
195
196 def gestione_comandi() -> None:
197     #Creazione dell'applicazione
198     app_bot = Application.builder().token(TOKEN).build()
199
200     #Associazione dei comandi inviati dal client
201     app_bot.add_handler(CommandHandler("start", start))
202     app_bot.add_handler(CommandHandler(["grafico"], interval_options))
203     app_bot.add_handler(CallbackQueryHandler(button))
204     app_bot.add_handler(CommandHandler("stop", stop))
205
206     #Polling che attende il comando dall'utente
207     app_bot.run_polling(allowed_updates=Update.ALL_TYPES)
```

7.8 chatbot.py

```
200 if __name__ == '__main__':
201     gestione_comandi()
```

Riferimenti bibliografici

- [1] ESP32: https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32d_esp32-wroom-32u_datasheet_en.pdf
- [2] PCF8523: <https://cdn-learn.adafruit.com/downloads/pdf/adafruit-pcf8523-real-time-clock.pdf>
- [3] BMP280: <https://cdn-shop.adafruit.com/datasheets/BST-BMP280-DS001-11.pdf>
- [4] SSD1306: <https://www.adafruit.com/product/326>
- [5] Noctua NF-A4x10: https://noctua.at/pub/media/wysiwyg/Noctua_PWM_specifications_white_paper.pdf