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

Elenco delle figure

1 Introduzione

Nel contesto del corso "Sistemi Operativi Dedicati", il presente progetto si propone di realizzare un sistema completo per il monitoraggio, la visualizzazione e il controllo dei parametri ambientali. Utilizzando una combinazione di hardware e software, il sistema è progettato per acquisire, elaborare e visualizzare dati ottenuti tramite molteplici sensori.

1.1 Funzionamento Generale

Il sistema è composto da diversi componenti, tra cui una scheda ESP32, una Raspberry Pi 4, un sensore di temperatura e pressione BMP280, una ventola, un display OLED e un 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. I dati acquisiti vengono inviati tramite comunicazione seriale alla Raspberry Pi che agisce come server IoT. Attraverso l'utilizzo di un'interfaccia web semplice e intuitiva, la Raspberry Pi mette a disposizione degli utenti i dati raccolti e archiviati, consentendo loro di visualizzare le tendenze nel tempo e di eseguire analisi approfondite. Inoltre, il sistema offre la possibilità di richiedere dati relativi a intervalli di tempo specifici tramite dei pulsanti digitali implementati sul bot Telegram "SensorDataRPI_bot", che risponde fornendo una visualizzazione grafica dei parametri richiesti, accompagnata dai relativi valori statistici.

Il sistema monitora costantemente i parametri ambientali tramite il sensore BMP280; quando la temperatura supera una soglia predefinita, la ventola viene attivata per raffreddare il sistema e mantenere la temperatura entro il range desiderato. La velocità della ventola è proporzionale allo scostamento della temperatura attuale rispetto a quella desiderata. Le informazioni sulla temperatura, la pressione e la velocità della ventola vengono visualizzate sul display OLED e inviate alla Raspberry Pi insieme al timestamp corrispondente. La Raspberry Pi memorizza i dati all'interno di un database e genera notifiche di alert qualora la temperatura superi una determinata soglia critica. Gli alert vengono inviati tramite il bot Telegram a tutti gli utenti registrati, quando la temperatura non rientra in un range accettabile. Inoltre, come già detto, la Raspberry Pi mette a disposizione i dati memorizzati nel database attraverso una semplice pagina web.

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 sulla semplicità d'uso, l'affidabilità e la reattività alle situazioni critiche.

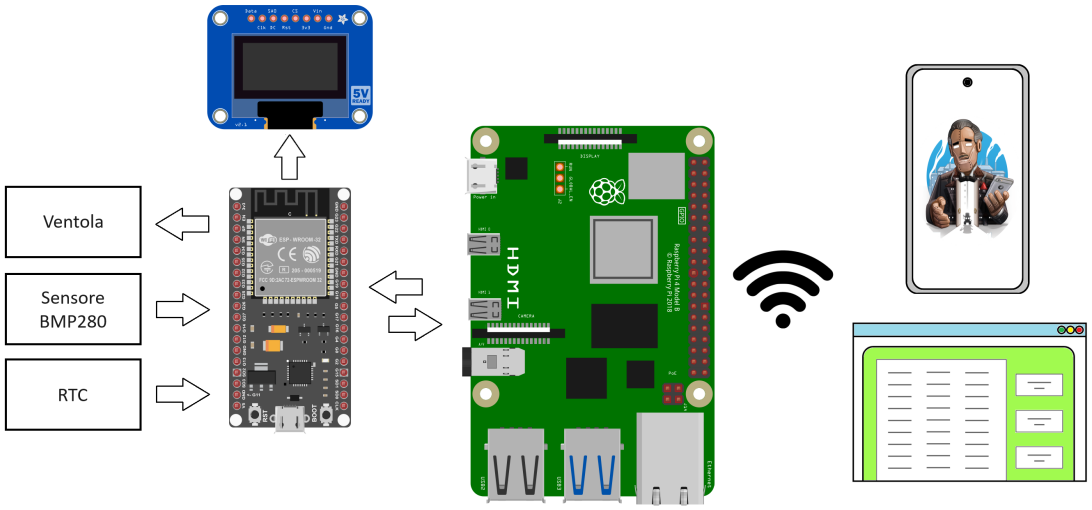

Figura 1: Schema a blocchi

2 Strumenti utilizzati

2.1 Componenti Hardware

Nell'implementazione del progetto sono stati impiegati diversi componenti hardware; si suddividono dapprima i protagonisti in due macro categorie: le schede e i sensori. Lo schema globale del cablaggio, per ovvie ragioni, ha fatto uso di una breadboard per il corretto collegamento dei jumper ai pin interessati.

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. L'ESP32 è un componente alimentato a 3.3V, in cui alcuni pin sono riservati per la memoria SPI e altri hanno funzionalità specifiche come i due canali I2C, anche se qualsiasi pin può essere impostato come SDA o SCL. È in grado di entrare in modalità deep sleep per il risparmio energetico. Inoltre, tutti i pin che possono agire come uscite possono essere utilizzati come pin PWM, ad eccezione dei GPIO da 34 a 39. 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.

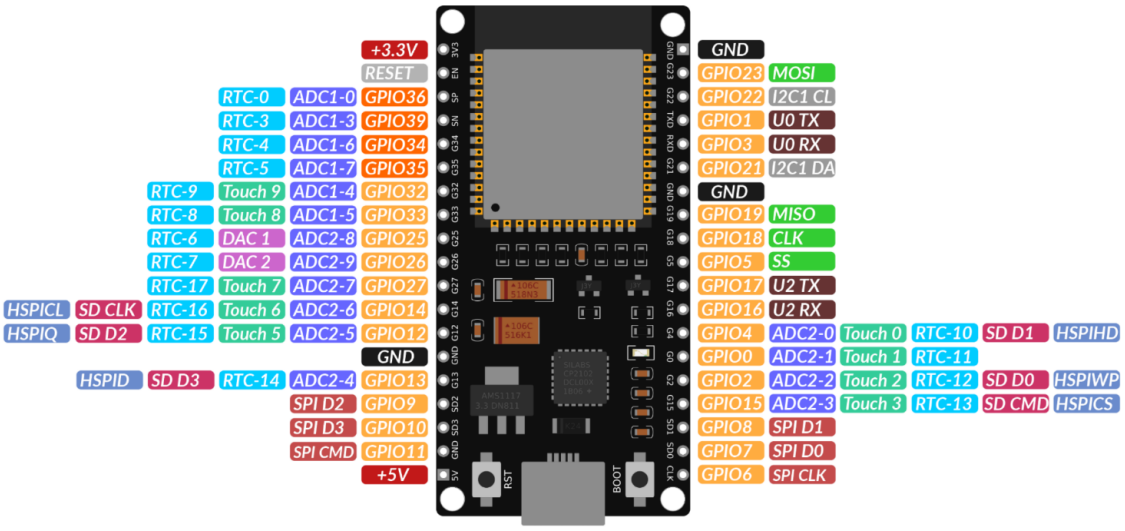

Figura 2: ESP32 Piedinatura

- **Raspberry Pi 4B** (RPi): Scheda di sviluppo utilizzata come server IoT per la gestione dei dati e delle comunicazioni con i dispositivi connessi. La Raspberry Pi 4 è una scheda computer ad alte prestazioni progettata per una vasta gamma di progetti informatici. Dotata, nel nostro caso, di 4GB di RAM, offre un notevole potenziale di elaborazione e multitasking per soddisfare le esigenze di applicazioni complesse. Per iniziare, abbiamo installato il sistema operativo Raspbian, una distribuzione Linux ottimizzata per le Raspberry Pi. Successivamente, ci siamo collegati alla Raspberry Pi 4 tramite SSH per accedere e controllare la scheda da un altro computer sulla stessa rete. Inoltre, abbiamo anche abilitato il servizio VNC (Virtual Network Computing) utilizzando l'utility 'raspi-config', consentendoci di interagire con l'interfaccia grafica della Raspberry Pi tramite un client VNC da remoto.

2.1.2 Sensori

Prima di procedere con l'integrazione dei singoli sensori nel sistema, abbiamo condotto dei test per verificare il loro corretto funzionamento. Questi test sono stati eseguiti utilizzando gli sketch di esempio forniti dalle librerie ufficiali di ciascun sensore. Tale approccio ci ha permesso di assicurarci che ciascun componente hardware rispondesse correttamente e fosse pronto per essere integrato nel sistema. Gli indirizzi I2C dei singoli dispositivi sono stati lasciati invariati e sono rispettivamente: 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: Questo pin 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: Questo pin è utilizzato per l'uscita di un'onda quadra se abilitata la funzionalità di contatore.

  Si consiglia di 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.

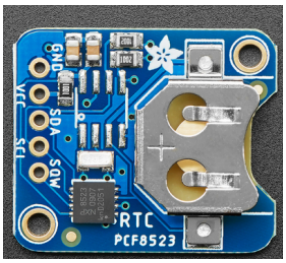

Figura 3: RTC

- **BMP280**: Sensore di temperatura e pressione utilizzato per monitorare i parametri ambientali. Il sensore BMP280 è stato testato con l'ESP32 utilizzando la propria libreria ufficiale: "Adafruit BMP280 Library". Il modulo BMP280 è collegato tramite un connettore STEAMM QT al display OLED da un lato, mentre dall'altro è collegato all'ESP32 tramite i propri pin (Vin, GND, SCK e SDI) seguendo lo schema logico già visto in precedenza per l'RTC. Facendo riferimento alle specifiche tecniche, il sensore è in grado di misurare la pressione atmosferica nell'intervallo da 300 a 1100 hPa; in aggiunta, il sensore ha anche un basso consumo di corrente, di circa 2.7 µA 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, inoltre, agendo su specifici bit associati a registri del sensore, andare a modificare la risoluziore delle misurazioni sia di temperatura che di pressione.

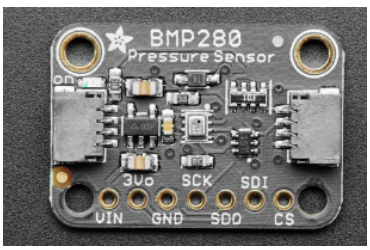

Figura 4: BMP280

- **SSD1306**: Display OLED utilizzato per visualizzare informazioni sull'ambiente circostante: Pressione, Temperatura e RPM della ventola. I collegamenti tra il sensore BMP280 e l'OLED sono stati effettuati, come già detto, 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 con una dimensione di 0.96 pollici ed ha una risoluzione 128x64 pixel. Al fine di 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 su questa per aggiungere funzioni grafiche come linee, cerchi e testo.

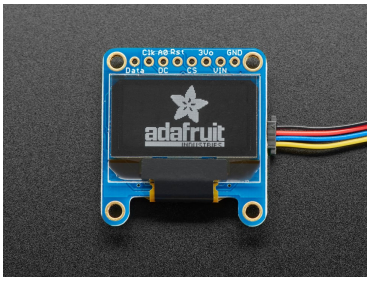

Figura 5: Display OLED

- **Ventola** (Noctua NF-A4x10 5v PWM 40x10mm): Dispositivo utilizzato per azionarsi in modo proporzionale, tramite controllo PWM, all'aumentare della temperatura rilevata dal BMP280. Dotata di 4 pin, questa ventola accetta un'alimentazione di 5V tramite il filo giallo, mentre il cavo nero è utilizzato per il collegamento a terra. Il connettore verde fornisce un segnale di tachimetro, che può essere utilizzato per monitorare la velocità della ventola in tempo reale. Il connettore blu è dedicato al controllo della velocità tramite segnale PWM da parte dell'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. Si utilizza dunque la seguente formula:

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

Questo tipo di ventola è stata progettata per assorbire una corrente massima di 5mA, quindi è stato necessario il calcolo della resistenza per adattare le tensioni e correnti di uscita della ventola a quelle di ingresso del ESP32.

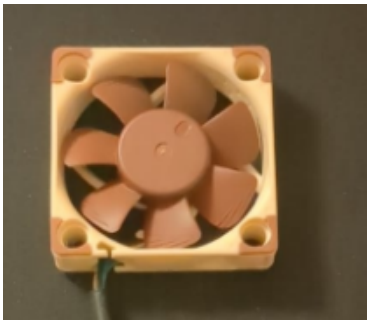

Figura 6: Fan

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 Sistema di acquisizione dati

3.1 Descrizione generale sul sistema hardware implementato

Dal punto di vista elettronico, è possibile avere un'idea di come il sistema è stato implementato guardando figura 7.

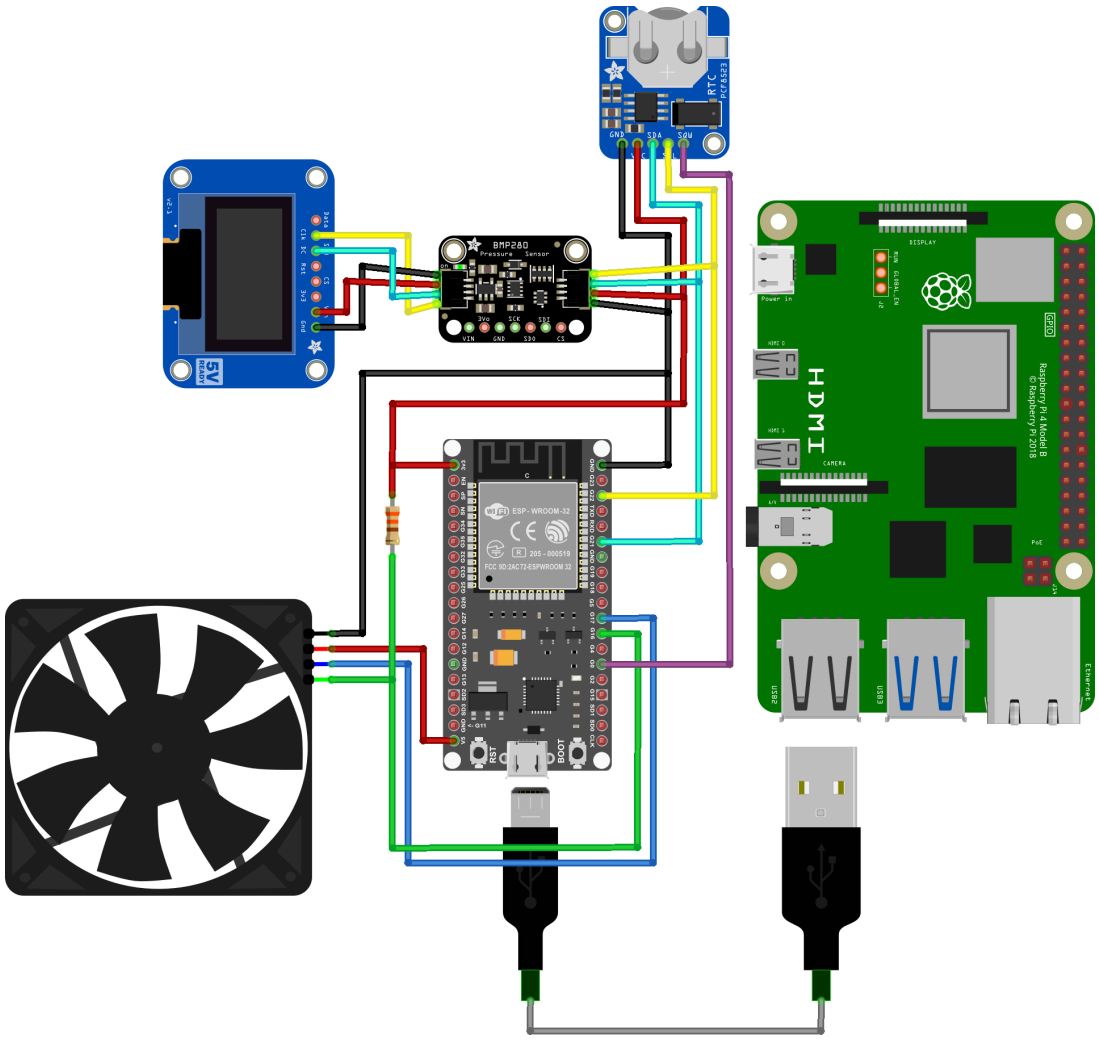

Figura 7: Schema del cablaggio.

Innanzitutto, il microcontrollore ESP32 e la Raspberry sono stati interconnessi utilizzando il protocollo seriale. Ciascuno di questi dispositivi avrà il proprio ruolo ben definito: il microcontrollore sarà responsabile dell'acquisizione dei dati, mentre la Raspberry si occuperà della loro ricezione e della loro successiva elaborazione.
Dalla figura, possiamo distinguere i seguenti dispositivi collegati al microcontrollore:

- un sensore BMP280, misura la temperatura e la pressione atmosferica dell'ambiente circostante;
- un modulo RTC, utilizzato per mantenere l'orario corretto all'interno del sistema;
- un display OLED, su cui è possibile visualizzare i dati acquisiti;
- una ventola controllata da PWM, utilizzata per la dispersione del calore.

I primi tre dispositivi si avvalgono del protocollo di comunicazione bifilare I^2C per interagire con l'ESP32. Quest'ultimo, configurato come dispositivo "Master", 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). Il quarto dispositivo illustrato nella figura è una ventola controllata tramite PWM. Essa è dotata di 4 pin: due per l'alimentazione a 5V (connessi ai cavi rossi e neri), uno per il controllo mediante segnale PWM (connesso al `pin G17` attraverso il cavo blu) e uno per rilevare il numero di rotazioni al minuto (connesso al `pin G16` tramite una resistenza da 330 Ω e cavo verde).
In generale, la logica di funzionamento implementata per il sotto sistema fin qui illustrato si divide in 5 operazioni:

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 su seriale da RPi.

Queste sono state aggregate tra loro all'interno di 3 attività (chiamati anche `task`), eseguibili "contemporaneamente" grazie alla libreria FreeRTOS, che le schedula in tempo reale. Nella figura 8 viene riportato uno schema approssimativo di come hardware e software siano messi in comunicazione tra loro. Nei successivi sotto-paragrafi ciascun task verrà commentato nel dettaglio e si accennerà all'integrazione con FreeRTOS, inoltre, è possibile consultare il codice di riferimento in appendice 6.1.

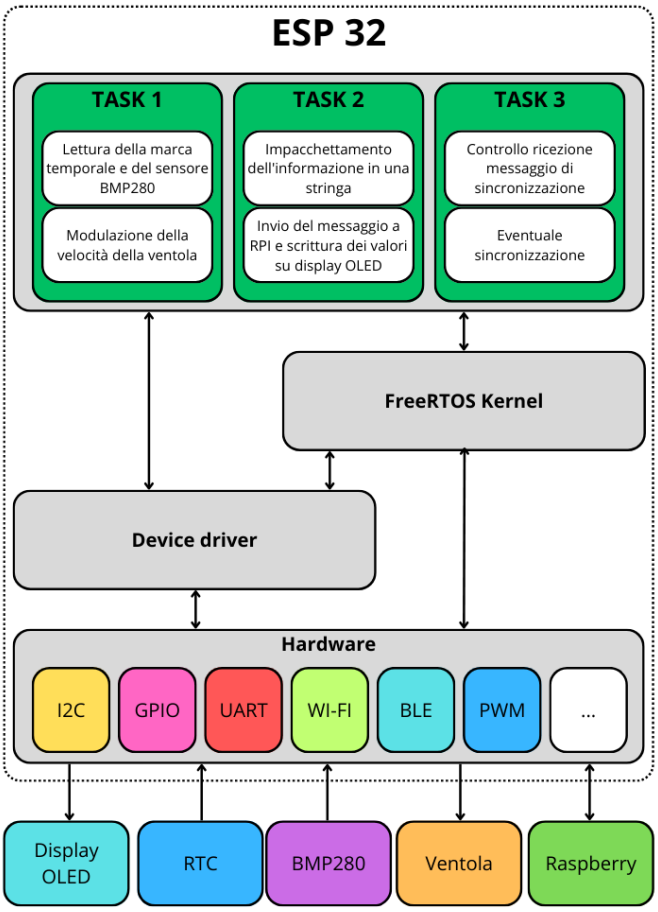

Figura 8: Schema riassuntivo dell'architettura hardware-software.

3.2 Descrizione del codice implementato.

Il codice caricato nel microcontrollore (vedi appendice 6.1) 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.

3.2.1 Inizializzazione e blocco `setup()`

Analizzando le righe che vanno da 1 a 148 del codice di riferimento 6.1 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` per il display OLED, `Adafruit_BMP280.h` per il sensore BMP280 e `Wire.h` per la comunicazione I^2C.
- **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 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 volatile poiché possono essere chiamate in modo asincrono durante l'esecuzione del programma principale.
- **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 148) 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 I^2C.

- **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.

3.2.2 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 213 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 effetturare 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 21°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 [21°C, 28°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()` (righe da 32 a 35), utilizzata per rilevare gli impulsi emessi dal contagiri sul pin designato ogni volta che la ventola esegue 1/2 di giro;
- `countdownOver()` (righe da 37 a 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

$$rpm = \frac{n_{impulsi}}{2} \cdot 60 \tag{1}$$

Al termine dell'ultima porzione di codice 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
```

3.2.3 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 157 e 186. La prima operazione svolta consiste nell'ispezionare la variabile `message`: la funzione `dateCheck()` in riga 164 verifica se l'orario del messaggio in attesa dell'invio coincide con quello precedentemente trasmesso, scartandolo di conseguenza. 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. Una volta passato questo primo test si procede trasferendo:

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

   `M!#YYYY-MM-DD  hh:mm:ss#temperatura#pressione#rpm` (2)

2. tramite protocollo I^2C (da 167 a 180), i valori da visualizzare su display OLED.

3.2.4 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 wifi 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 215 a 247) 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#` (3)

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 comando è riconosciuto come `"sync"`, 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()`. Il comando `delay()` garantisce il corretto riavvio del modulo.

4 Sistema di elaborazione e divulgazione dei dati acquisiti

4.1 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 30°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.

4.2 Sincronizzazione del modulo RTC

Per completare la fase relativa alla corretta sincronizzazione tra l'ESP32 e la Raspberry, è stato implementato uno script altamente efficiente consultabile in appendice 6.2. 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_timer()` permette di ottenere l'orario iniziale dal server NTP. Questo consente di sincronizzare il modulo RTC all'accensione del sistema e quando passa il tempo dopo la quale l'RTC si deve sincronizzare (1 minuto nel nostro caso), inviando un messaggio contenente l'orario specifico attraverso la porta seriale all'ESP32.

Il fulcro dello script è rappresentato dal loop principale, il quale ciclicamente ottiene l'orario corrente dal server NTP, calcola la differenza temporale rispetto all'orario iniziale e, qualora tale differenza superi o sia uguale a 60 secondi, prepara un messaggio di sincronizzazione che include l'orario attuale e lo invia all'ESP32 tramite la comunicazione seriale. E' interessante notare che il formato del messaggio è strutturato nel seguente modo:

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

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)` (5)

4.3 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 6.3.

Prima di esaminare nel dettaglio il processo di memorizzazione dei dati nel database, è importante fare un appunto sulle righe 28 a 35: all'avvio del database e, di conseguenza, dell'intero sistema associato, viene eseguita una prima inizializzazione del modulo RTC. Successivamente, tramite lo script `sync_timer.py`, la calibrazione dell'orario viene eseguita ogni minuto di acquisizioni.

Questo approccio garantisce una gestione accurata del tempo nel sistema, assicurando che l'orario sia sempre allineato e calibrato in modo appropriato per le operazioni di raccolta dati e archiviazione nel database.

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. Dopodichè, 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 dell'integrità del messaggio**
  Una volta convertito il messaggio in un formato idoneo al corretto utilizzo, viene verificata l'integrità del messaggio ricevuto. Questo controllo avviene esaminando che la lunghezza della lista `line` sia diversa da `1` (indicando che il messaggio contiene più di un valore) e che il primo elemento della lista sia `"M!"`, il che denota che si tratta di un messaggio valido (o comunque di una stringa che non è stata corrotta durante il procedimento di scruttura/lettura su seriale).

- **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 30°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 30°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 le RPM lette e le inserisce nella tabella del database.

4.4 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 6.4.

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 60 minuti, al fine di calcolare valori statistici.
- **Calcolo dei valori statistici**
  I dati relativi alla temperatura, alla pressione e alle 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 6.5, con il relativo linguaggio che gestisce il design in appendice 6.6. 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 9.

Figura 9: Visualizzazione grafica della webapp

4.5 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 6.7, 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 30°, 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 minuto 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.

4.5.1 Esempio generazione grafico - disponibilità di dati

Un primo esempio 10 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.

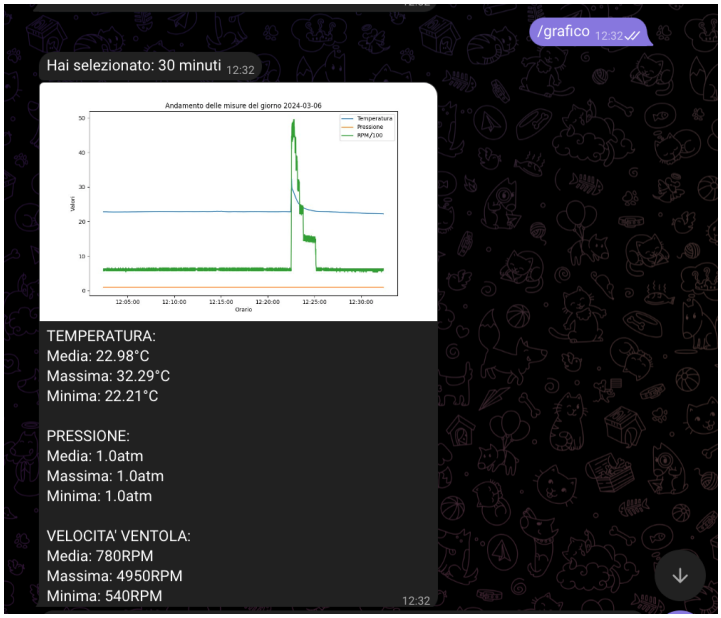

Figura 10: Esempio generazione grafico e valori statistici - dati disponibili

4.5.2 Esempio generazione grafico - mancanza di dati

Un secondo esempio 11 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.

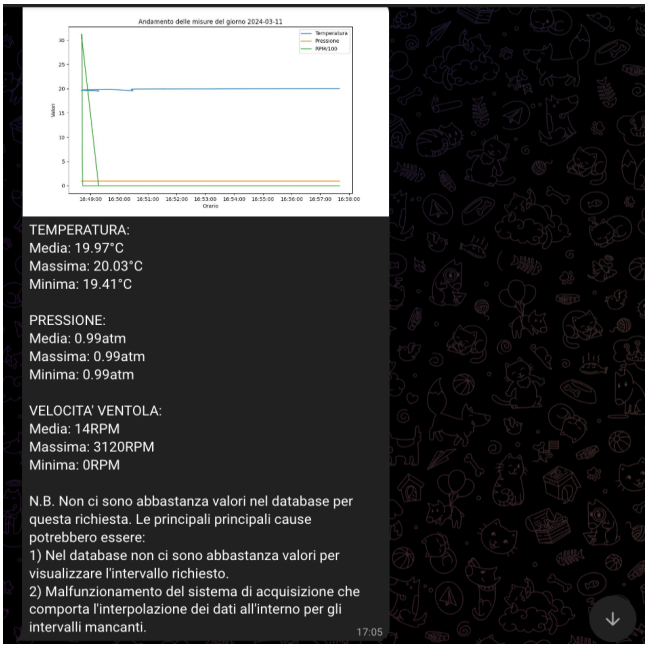

Figura 11: Esempio generazione grafico e valori statistici - mancanza di dati

4.6 Esecuzione dei daemon

Terminata l'implementazione del sistema di monitoraggio ambientale, l'ultima fase è stata occupata dalla creazione dei servizi di routine al fine di rendere l'intero sistema autonomo e, quindi, non dipendente dall'avvio della Raspberry di un operatore umano. Si sta parlando dei cosiddetti "daemon", processi 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 monitaggio ambientale (`sync_timer.py, database.py, app.py, chatbot.py`). Per implementare i servizi di routine sono stati seguiti passaggi stardard che per semplicità si mostraranno i passaggi seguiti per al creazione del daemon associato al servizio `chatbot.py`, in quanto per gli altri si è proceduto analogamente. I passaggi seguiti sono mostrati di seguito:

- **Scrittura dello script del daemon**
  Prima di tutto, è stato realizzato il file per la creazione del daemone `service_bot.sh`; si tratta sostanzialmente di un file di appoggio per lanciare `chatbot.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  sleep 10
3  source /home/RPi/progett:sod/bin/activate
4  sleep 5
5  python3 /home/RPi/chatbot.py
```

  Il comando `sleep` ha il compito di garantire l'avvio dei servizi caratterizzanti.

  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 sia eseguibile utilizzando il comando `chmod +x service_bot.sh`; questo passaggio permette al kernel del sistema operativo di eseguire lo script all'avvio del sistema.

- **Creazione del file di configurazione del daemon**
  Inoltre, è necessario creare un ulteriore file `bot_service.service` che gestisce l'esecuzione del file `service_bot.sh` il cui codice è mostrato di seguito:

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

  Il codice rappresenta un file di configurazione per un servizio di sistema su Linux, gestito da systemd. 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.

- **Abilitazione e avvio del daemon**
  Dopo aver configurato il file del daemon, è necessario abilitarlo utilizzando il comando `systemctl enable bot_service.service`. Quindi, si avvia il daemon con il comando `systemctl start bot_service.service`.

5 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.

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.

6 Appendice

6.1 esp32freeRTOS.ino

```
#include "RTClib.h"
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>
#include <Adafruit_BMP280.h>
#include <Wire.h>

#define CANALE 0
#define RISOLUZIONE_PWM 8
#define FREQUENZA 5000
#define PIN_PWM 17

#define PIN_TACHO 16
#define SOGLIA 21
#define PIN_COUNTDOWNINT 0

//Inizializzazione variabili globali utili per l'RPM
volatile bool countdownInterruptTriggered = false;
volatile int numCountdownInterrupts = 0;
volatile unsigned long counter = 0;
int fanSpeed=0;
String oldDate;

//struttura contenente i dati acquisiti da utilizzare per lo scambio di messaggi in coda
typedef struct{
  float temperatura;
  float pressione;
  long rpm;
  String time_stamp;
} message_t;


//Funzione per contare gli impulsi per la lettura della velocit  della ventola (tachimetro)
void countPulses() {
  counter ++;
}

//Funzione per contare gli impulsi da parte di RTC che emette in 1 secondo
void countdownOver () {
  countdownInterruptTriggered = true;
  numCountdownInterrupts++;
}

//Funzione per confrontare la data prima dell'invio di ogni messaggio su seriale (per evitare l'invio di messaggi identici)
bool dateCheck(String newDate){
  if (oldDate == newDate){
    return 0;
    }
  else if (oldDate != newDate){
    oldDate = newDate;
    return 1;
      }
}

//Definizione dei task
void TaskSerial(void *pvParameters);
void Task_Sensors(void *pvParameters);
void Task_Sync(void *pvParameters);

//Definizione della coda
QueueHandle_t QueueHandle;
const int QueueElementSize = 10;

//Definizioni delle variabili oggetto dei tre dispositivi
Adafruit_BMP280 bmp;
```

6.1 esp32freeRTOS.ino

```
65  Adafruit_SSD1306 display(128, 64, &Wire, -1);
66  RTC_PCF8523 rtc;
67
68  void setup() {
69    Serial.begin(115200);
70    while(!Serial){delay(10);}
71    delay(5000); //tempo necessario per la corretta inizializzazione della seriale da
          parte di ESP32 ed RPI
72
73    //inizializzazione dei tre dispositivi:
74
75    //BMP280
76    unsigned status;
77    status = bmp.begin();
78    if (!status) {
79      Serial.println("Errore BMP");
80    }
81    bmp.setSampling(Adafruit_BMP280::MODE_NORMAL,     // Modalit  operativa
82                    Adafruit_BMP280::SAMPLING_X2,     // Campionamento della temperatura
83                    Adafruit_BMP280::SAMPLING_X16,    // Campionamento della pressione
84                    Adafruit_BMP280::FILTER_X16,      // Filtraggio dei segnali
85                    Adafruit_BMP280::STANDBY_MS_500); // Tempo di stand-by
86
87    //OLED
88    if (!display.begin(SSD1306_SWITCHCAPVCC, 0x3D)) {
89      Serial.println("Errore OLED");
90    }
91    display.clearDisplay();
92    display.display();
93
94    //RTC
95    if (! rtc.begin()) {
96      Serial.println("Couldn't find RTC");
97      while (1) delay(10);
98    }
99    if (! rtc.initialized() || rtc.lostPower()) {
100     rtc.enableCountdownTimer(PCF8523_Frequency64Hz, 64, PCF8523_LowPulse8x64Hz);
101   }
102     rtc.start();
103
104   //Creazione della coda
105   QueueHandle = xQueueCreate(QueueElementSize, sizeof(message_t));
106   if(QueueHandle == NULL){
107     while(1) delay(1000);
108   }
109
110   //Creazione dei task
111   xTaskCreate(
112     TaskSerial
113     ,  "TaskSerial"
114     ,  2048
115     ,  NULL
116     ,  2
117     ,  NULL
118     );
119
120   xTaskCreate(
121     Task_Sensors
122     ,  "Task_Sensors"
123     ,  2048
124     ,  NULL
125     ,  1
126     ,  NULL
127     );
128
129     xTaskCreate(
130     Task_Sync
131     ,  "TaskSync"
132     ,  2048
133     ,  NULL
```

```
134     ,  0
135     ,  NULL
136     );
137
138     //inizializzazione PWM
139     ledcSetup(CANALE, FREQUENZA, RISOLUZIONE_PWM);
140     ledcAttachPin(PIN_PWM, CANALE);
141     ledcWrite(CANALE,0);
142
143     //Creazione degli interrupt associando i pin alle funzioni definite in precedenza
144     pinMode(PIN_COUNTDOWNINT, INPUT_PULLUP);
145     pinMode(PIN_TACHO, INPUT_PULLUP);
146     attachInterrupt(digitalPinToInterrupt(PIN_TACHO), countPulses, FALLING); //RPM
147     attachInterrupt(digitalPinToInterrupt(PIN_COUNTDOWNINT), countdownOver, FALLING);
            //RTC
148 }
149
150 void loop(){
151 }
152
153 /*------------------------------------------------------*/
154 /*---------------------- Tasks ---------------------*/
155 /*------------------------------------------------------*/
156
157 //Task dedicato alla trasmissione dei dati tramite protocollo seriale alla RPI e tramite
        I2C al display oled
158 void TaskSerial(void *pvParameters){
159   message_t message;
160   for (;;){
161     if(QueueHandle != NULL){
162       int ret = xQueueReceive(QueueHandle, &message, portMAX_DELAY);
163       if(ret == pdPASS){
164         if(dateCheck(message.time_stamp)){
165              Serial.println("M!#"+message.time_stamp+"#"+message.temperatura+"#"+message.pressione+"#"+
166
167               display.clearDisplay();
168               display.setTextColor(WHITE);
169               display.setTextSize(1);
170               display.setCursor(0, 0);
171               display.print("Temperatura: ");
172               display.print(message.temperatura);
173               display.println((char)247);
174               display.println("C");
175               display.print("Pressione: ");
176               display.print(message.pressione);
177               display.println(" atm");
178               display.print("RPM:");
179               display.print(message.rpm);
180               display.display();
181       }
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   for (;;){
192
193     DateTime now = rtc.now();
194     message.time_stamp = (String)now.year()+"-"+ (String)now.month()+ "-"
            +(String)now.day()+"
            "+(String)now.hour()+":"+(String)now.minute()+":"+(String)now.second();
195     message.temperatura = bmp.readTemperature();
196     message.pressione = bmp.readPressure()*9.8692*pow(10,-6);
197     //map e constrain hanno l'obiettivo di rendere proporzionale la velocit  della
            ventola all'aumentare
198     //della temperatura limitando il valore associato alla PWM in un range 0-255
```

```
199     fanSpeed = map(message.temperatura, SOGLIA, 28, 0, 255);
200     ledcWrite(CANALE, constrain(fanSpeed, 0, 255));
201 
202     //conversione degli impulsi in RPM ad ogni secondo
203     if(countdownInterruptTriggered && numCountdownInterrupts == 1){
204             message.rpm = counter * 60 / 2;
205             counter = 0;
206             countdownInterruptTriggered = false;
207             numCountdownInterrupts = 0;
208             }
209 
210     xQueueSend(QueueHandle, &message, portMAX_DELAY);
211     vTaskDelay(pdMS_TO_TICKS(100));
212   }
213 }
214 
215 //Task per la sincronizzazione del modulo RTC tramite seriale al fine di mantenere
        l'orario corretto
216 void Task_Sync(void* pvParameters) {
217   message_t message;
218 
219   //dichiarazioni di variabili utili per la decodifica del messaggio
220   int delimiter_start, delimiter_end;
221   String string, msg_rx[7];
222 
223   for(;;){
224     if (Serial.available() > 0) {
225       string = Serial.readString();
226       delimiter_start = string.indexOf("#"); //posizione del primo valore "#" nel
              messaggio ricevuto
227 
228       //Ciclo per lo split della stringa arrivata
229       for(int idx_str = 0; idx_str < string.length(); idx_str++){
230 
231         //posizione del successivo valore "#" rispetto a quello contenuto in
              delimiter_start
232           delimiter_end = string.indexOf("#", delimiter_start+1);
233           //allocazione del valore contenuto tra due "#" nella i-esima cella della
                stringa msg_rx
234           msg_rx[idx_str] = string.substring(delimiter_start+1, delimiter_end);
235           delimiter_start = delimiter_end;
236       }
237 
238       //Verifica se la stringa ricevuta    "sync" ed effettua l'eventuale
              sincronizzazione
239       if (msg_rx[0]== "sync") {
240           rtc.adjust(DateTime(msg_rx[1].toInt(),msg_rx[2].toInt(),msg_rx[3].toInt(),msg_rx[4].toInt()
241           rtc.start();
242           delay(100);
243         }
244       }
245 
246     }
247   }
```

6.2 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 velocit  di trasmissione (baud rate)
18  baud_rate = 115200
19  # Apre la connessione seriale
20  ser = serial.Serial(port, baud_rate)
21
22  # Ottiene l'orario iniziale dal server NTP e syncronizza il modulo RTC all'avvio del
        sistema
23  start_time = get_ntp_time() #tipo float, sono il numero di secondi passati
        dall'inizio del tempo
24  date_msg2send = str(datetime.fromtimestamp(start_time)).split(".")
25  date_msg2send = date_msg2send[0].replace(" ", "#").replace(":", "#").replace("-","#")
26  msg = "#sync#"+date_msg2send+"#"
27  ser.write(msg.encode()) #messaggio di sincronizzazione inviato da RPI a ESP32    del
        tipo #sync#anno#mese#giorno#ora#minuti#secondi#
28
29
30  try:
31      # Loop principale del contatore
32      while True:
33
34          #Ottiene l'orario corrente dal server NTP
35          current_time = get_ntp_time()
36
37          #Calcola la differenza di tempo in secondi
38          elapsed_time = current_time - start_time
39
40          if elapsed_time >= 60:
41              #Preparazione della data da inviare. Viene prima usato il modulo NPC
42              # per richiedere l'orario al server, poi avviene una conversione
43              # DateTime2String e successivamente si tolgono i millesimi.
44              date_msg2send = str(datetime.fromtimestamp(current_time)).split(".")
45              date_msg2send = date_msg2send[0].replace(" ", "#").replace(":",
                    "#").replace("-","#")
46
47              #Creazione della variabile contenente il messaggio da inviare
                    #sync#anno#mese#giorno#ora#minuti#secondi#
48              msg = "#sync#"+date_msg2send+"#"
49              #Scrittura su seriale per l'inivio del messaggio
50              ser.write(msg.encode())
51
52              start_time = get_ntp_time()
53          print(f"Contatore temporale: {elapsed_time:.2f} secondi")
54          sleep(10) #Pausa per non incorrere dell'imporchettamento del sistema
55
56  except KeyboardInterrupt:
57      #Chiusura comunicazione
58      ser.close()
59      print("Contatore spento")
```

6.3 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 (?,
            ?, ?, ?)",
```

6.3 database.py

```
13          (Data, Temperatura, Pressione, RPM) )
14
15 # Funzione per ottenere l'orario dal server NTP
16 def get_ntp_time():
17     client = ntplib.NTPClient()
18     response = client.request(ntp_server)
19     return response.tx_time
20
21 #Inizializzazione porta seriale
22 porta_seriale = '/dev/ttyUSB0'
23 # Specifica la velocit  di trasmissione (baud rate)
24 velocita_trasmissione = 115200
25 # Apre la connessione seriale
26 ser = serial.Serial(porta_seriale, velocita_trasmissione)
27
28 # Indirizzo di un server NTP
29 ntp_server = 'pool.ntp.org'
30 # Ottiene l'orario iniziale dal server NTP e sincronizza il modulo RTC all'avvio del
       sistema
31 start_time = get_ntp_time() #tipo float, sono il numero di secondi passati
       dall'inizio del tempo
32 date_msg2send = str(datetime.fromtimestamp(start_time)).split(".")
33 date_msg2send = date_msg2send[0].replace(" ", "#").replace(":", "#").replace("-","#")
34 msg = "#sync#"+date_msg2send+"#"
35 ser.write(msg.encode()) #messaggio di sincronizzazione inviato da RPI a ESP32    del
       tipo #sync#anno#mese#giorno#ora#minuti#secondi#
36
37 # Inizializzazione contatore per la gestione dell'alert
38 contatore = 0
39
40 # Crea la connessione database tramite connettore python
41 try:
42    conn = mariadb.connect(
43       host="127.0.0.1",
44       port=3306,
45       user="root",
46       password="root",
47       autocommit=True,
48       database="SensorData")
49
50 except mariadb.Error as e:
51    print(f"Error connecting to the database: {e}")
52    sys.exit(1)
53
54 # Instanza il cursore
55 cur = conn.cursor()
56
57 # Lettura dei dati da seriale e successivo inserimento nel db
58 try:
59     while True:
60     # Lettura del messaggio e split
61         line = ser.readline().decode("utf-8", "ignore").strip()
62         line = line.split("#")
63         #controllo integrit  del messaggio ricevuto
64         if(len(line) != 1) and (line[0] == "M!"):
65                 # print(line)
66                  #assegnazione dei valori in variabili temporanee
67                  new_Data = line[1]
68                  new_Temp = float(line[2])
69                  new_Pres = float(line[3])
70                  new_RPM = int(line[4])
71
72                  #controllo per l'invio dell'alert
73                  if new_Temp > 30:
74                          contatore= contatore+1
75                          asyncio.run(chatbot.funz_alert(True, contatore))
76                  else:
77                          contatore = 0
78
79                  #chiamata della funzione per il salvataggio dei dati nel db
```

```
80                  add_measure(cur,new_Data,new_Temp,new_Pres,new_RPM)
81
82 except KeyboardInterrupt:
83     ser.close()
84     print("Connessione al dispositivo ESP32 chiusa.")
85     conn.close()
86     print("Connessione al Database 'SensorData' chiusa.")
```

6.4 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
           'misure'
31     cur.execute("SELECT * FROM misure ORDER BY id DESC")
32     data = cur.fetchall()
33
34     #Esecuzione della query SQL per selezionare tutte le righe nell'intervallo di 60
           minuti
35     cur.execute("SELECT * FROM misure WHERE Data >= NOW() - INTERVAL 60 MINUTE AND
           Data < NOW()")
36     data2 = cur.fetchall()
37
38     conn.close()
39
40     #estrazione dei dati e calcolo dei valori statistici
41     temperatura = [idx_data2[2] for idx_data2 in data2]
42     pressione = [idx_data2[3] for idx_data2 in data2]
43     rpm = [idx_data2[4] for idx_data2 in data2]
44
45     temp_mean = mean(temperatura, dtype=float16)
46     pres_mean = mean(pressione, dtype=float16)
47     rpm_mean = round(mean(rpm, dtype=float32))
48
49     # Trasferimento dei dati alla pagina HTML usando un template Flask
50     return render_template('index.html', data=data, temp_mean=temp_mean,
           pres_mean=pres_mean, rpm_mean=rpm_mean )
51
52 #Punto di ingresso dell'app Flask
53 if __name__ == '__main__':
54     #Avvio app in modalit  debug
```

```
55     app.run(host='0.0.0.0', port=5000, debug=True)
```

6.5 index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
   <!--  <meta http-equiv="refresh" content="10"> Ricarica la pagina ogni 300
       secondi (5 minuti) -->
    <meta name="viewport" content="width=device-width", initial-scale=1.0>
    <title>Progetto SOD</title>
    <link rel="stylesheet" type="text/css" href="{{ url_for('static',
        filename='style.css') }}">

</head>
<body>
    <div class="titolo">
        <h1>Progetto Sistemi Operativi Dedicati
            <span>Sistema di acquisizione dati per il monitoraggio ambientale</span>
        </h1>
    </div>
     <div class="container">
        <div class="left-column">
          <h2>Database - storico dei dati</h2>
            <div class="table-container">
                <table border="1">
                    <tr>
                        <th>Data</th>
                        <th>Temperatura</th>
                        <th>Pressione</th>
                        <th>RPM</th>
                    </tr>
        <!-- Cicla attraverso i dati passati dall'app Flask -->
        {% for row in data %}
            <tr>
                <td>{{ row[1] }}</td>
                <td>{{ row[2] }}</td>
                <td>{{ row[3] }}</td>
                <td>{{ row[4] }}</td>
            </tr>
        {% endfor %}
    </table>
  </div>
  </div>

<div class="right-column">
<h2>Media dei valori dell'ultima ora</h2>
<div class="card">
<h3>TEMPERATURA</h3>
<p>{{temp_mean}} C</p>
            </div>
<div class="card">
<h3>PRESSIONE</h3>
<p> {{pres_mean}} atm </p>
            </div>
<div class="card">
<h3>VELOCITA' DELLA VENTOLA </h3>
<p> {{rpm_mean}} rpm</p>

            </div>
        </div>
</div>
</body>
</html>
```

6.6 style.css

```
/* STYLE TITOLO */
.titolo h1 {
  text-align:center; font-size:50px; text-transform:uppercase; color:#222;
      letter-spacing:1px;
  font-family:"Playfair Display", serif; font-weight:400;
}
.titolo h1 span {
  margin-top: 5px;
    font-size:15px; color:#444; word-spacing:1px; font-weight:normal;
        letter-spacing:2px;
    text-transform: uppercase; font-family:"Raleway", sans-serif; font-weight:500;

    display: grid;
    grid-template-columns: 1fr max-content 1fr;
    grid-template-rows: 27px 0;
    grid-gap: 20px;
    align-items: center;
}

.titolo h1 span:after,.nine h1 span:before {
    content: " ";
    display: block;
    margin-left: 20px;
    margin-right: 20px;
    border-bottom: 1px solid #c50000;
    border-top: 1px solid #c50000;
    height: 5px;
  background-color:#f8f8f8;
}

/* TABLE CONTAINER */
.table-container {
            border: 2px solid #dddddd;
            padding: 10px;
            width: 90%;
            margin: 10px;
            overflow-x: auto;
            max-height: 500px;
            position: relative;
        }
        table {
            width: 100%;
            border-collapse: collapse;
        }
        th, td {
            border: 1px solid #dddddd;
            padding: 8px;
            text-align: center;
            font-family:"Raleway", sans-serif;
        }
        th {
            background-color: #f2f2f2;
            position: sticky;
            top: 0;
        }

.container {
    display: flex;
}

.left-column {
    flex: 2;
    padding: 20px;
    background-color: #f0f0f0;

}

.right-column {
```

```
    flex: 1;
    padding: 20px;
    background-color: #e0e0e0;
    display: flex;
    flex-direction: column;
    align-items: center;
}

.card {
    margin-top: 5px;
    width: 40%;
    padding: 10px;
    margin-bottom: 20px;
    height: 100px;
    border: 1px solid #ddd;
    border-radius: 8px;
    background-color: #ffffff;
    font-size:15px; color:#444; word-spacing:1px; font-weight:normal;
        letter-spacing:2px;
    text-transform: uppercase; font-family:"Raleway", sans-serif; font-weight:500;
           flex-direction: column;
    align-items: center;
    box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1);
}

.card h3 {
   text-align: center;
   font-size:15px;
   color:#444;
   word-spacing:1px;
   font-weight:normal;
   letter-spacing:2px;
   text-transform: uppercase;
   font-family:"Raleway", sans-serif;
   font-weight:500;
}

.left-column h2, .right-column h2 {
    margin-top: 5px;
    color: #333;
    font-size: 24px;
    margin-bottom: 10px;
    text-align:center;
    font-family:"Playfair Display", serif;
}

p {
text-align: center;
font-size: 20px;
font-family:"Raleway", sans-serif;
}
```

6.7 chatbot.py

```
1
2 import telegram
3 import asyncio
4 from telegram import Update, InlineKeyboardButton, InlineKeyboardMarkup
5 from telegram.ext import Application, CommandHandler, ContextTypes,
      CallbackQueryHandler
6 import serial
7 import mariadb
8 import sys
9 import os
10 from datetime import datetime, timedelta
11 import matplotlib.pyplot as plt
12 import matplotlib.dates as mdates
13 import logging
14 from numpy import mean, min, max, float16, float32
```

6.7 chatbot.py

```
15
16 #variabili globali
17 TOKEN="6440858737:AAFPy60W_qshW-Cq2YkriSY1ErOgaTFDgGA"
18 alert = "La temperatura ha superato la soglia critica"
19
20 #gestore per l'invio dell'alert
21 async def funz_alert(trigger, contatore):
22     bot = telegram.Bot(token=TOKEN)
23     if trigger == True and contatore == 1:
24         async with bot:
25             with open("bot_users.txt", "r") as users_f:
26                 chat_users = [line.strip() for line in users_f.readlines()]
27                 for id in chat_users:
28                     await bot.send_message(chat_id=id, text=alert)
29
30 #Inizializzazione del bot e salvataggio dell'id utente nel file di testo
31 async def start(update: Update, context: ContextTypes.DEFAULT_TYPE) -> None:
32     user_id = update.message.from_user.id
33     with open("bot_users.txt", "r") as users_f:
34         users_list = [line.strip() for line in users_f.readlines()]
35         if str(user_id) not in users_list: #controllo sull'id
36             users_list.append(user_id)
37             with open("bot_users.txt", "w") as users_file:
38                 for list_idx in users_list:
39                     users_file.write(str(list_idx)+"\n")
40         else:
41             await update.message.reply_text("L'utente "+f'{user_id}'+" ha gi
                inizializzato il bot.")
42
43 #stop ed eliminazione dell'id utente dal file di testo
44 async def stop(update: Update, context: ContextTypes.DEFAULT_TYPE) -> None:
45     user_id = update.message.from_user.id
46     with open("bot_users.txt", "r") as users_f:
47         users_list = [line.strip() for line in users_f.readlines()]
48         if str(user_id) in users_list: #controllo sull'id
49             users_list.remove(str(user_id)) #rimozione id
50             with open("bot_users.txt", "w") as users_file:
51                 for list_idx in users_list:
52                     users_file.write(str(list_idx)+"\n")
53
54 #Richiesta del grafico da parte del client e risposta del bot tramite bottoni
55 async def interval_options(update: Update, context: ContextTypes.DEFAULT_TYPE) ->
       None:
56     bot = telegram.Bot(token = TOKEN)
57     user_id = update.message.from_user.id
58     with open("bot_users.txt", "r") as users_f:
59         users_list = [line.strip() for line in users_f.readlines()]
60         if str(user_id) in users_list: #controllo sull'id
61
62
63            #bottoni per la scelta dell'intervallo
64             keyboard = [
65                         [   InlineKeyboardButton("1 minuto", callback_data="1"),
66                             InlineKeyboardButton("5 minuti", callback_data="5")],
67                         [   InlineKeyboardButton("30 minuti", callback_data="30"),
68                             InlineKeyboardButton("60 minuti", callback_data="60")],
69                         ]
70
71             reply_markup = InlineKeyboardMarkup(keyboard)
72             await update.message.reply_text("Quale intervallo temporale vuoi
                selezionare?",reply_markup=reply_markup)
73         else:
74             await bot.send_message(chat_id = user_id, text = "Il bot non    stato
                inizializzato. Inviare /start e poi richiedere il grafico.")
75
76 #Funzione per l'invio del grafico tramite il bot
77 async def funz_grafico(minutes):
78
79     # Crea la connessione al database
80     try:
```

6.7 chatbot.py

```
81          conn = mariadb.connect(
82              host="127.0.0.1",
83              port=3306,
84              user="root",
85              password="root",
86              autocommit=True,
87              database="SensorData"
88          )
89      except mariadb.Error as e:
90          print(f"Error connecting to the database: {e}")
91          return
92
93      # Istanzia il cursore
94      cur = conn.cursor()
95      # Query per ottenere i dati nell'intervallo selezionato
96      try:
97          match minutes:
98              case "1":
99                  cur.execute("SELECT * FROM misure WHERE Data >= NOW() - INTERVAL 1
                        MINUTE AND Data < NOW()")
100             case "5":
101                 cur.execute("SELECT * FROM misure WHERE Data >= NOW() - INTERVAL 5
                        MINUTE AND Data < NOW()")
102             case "30":
103                 cur.execute("SELECT * FROM misure WHERE Data >= NOW() - INTERVAL 30
                        MINUTE AND Data < NOW()")
104             case "60":
105                 cur.execute("SELECT * FROM misure WHERE Data >= NOW() - INTERVAL 60
                        MINUTE AND Data < NOW()")
106
107         rows = cur.fetchall()
108
109     except mariadb.Error as e:
110         print(f"Errore per l'esecuzione della query: {e}")
111         conn.close()
112         return
113
114     # Chiusura della connessione al database
115     conn.close()
116
117     # Estrazione dei dati
118     timestamps = [row[1] for row in rows]
119     temperatura = [row[2] for row in rows]
120     pressione = [row[3] for row in rows]
121     rpm = [row[4] for row in rows]
122     rpm_norm = [idx_rpm/100 for idx_rpm in rpm]
123
124     #Calcolo dei valori statistici
125     temp_mean = mean(temperatura, dtype=float16)
126     temp_min = min(temperatura)
127     temp_max = max(temperatura)
128     temp_len = len(temperatura)
129
130     pres_mean = mean(pressione, dtype=float16)
131     pres_min = min(pressione)
132     pres_max = max(pressione)
133
134     rpm_mean = round(mean(rpm, dtype = float32))
135     rpm_min = min(rpm)
136     rpm_max = max(rpm)
137
138     #istanziazione del messaggio contenente i valori statistici
139     temp_msg = "TEMPERATURA:\n"+"Media: "+str(temp_mean)+' C \n'+"Massima:
            "+str(temp_max)+' C \n'+"Minima: "+str(temp_min)+' C \n'
140     pres_msg = "PRESSIONE:\n"+"Media: "+str(pres_mean)+'atm\n'+"Massima:
            "+str(pres_max)+'atm\n'+"Minima: "+str(pres_min)+'atm\n'
141     rpm_msg = "VELOCITA' VENTOLA:\n"+"Media: "+str(rpm_mean)+'RPM\n'+"Massima:
            "+str(rpm_max)+'RPM\n'+"Minima: "+str(rpm_min)+'RPM\n'
142
143     #controllo sulla quantit  di dati a disposizione
```

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