



Università Politecnica delle Marche
Dipartimento di Ingegneria dell'Informazione

Rete di Sensori mediante Pi Pico e FreeRTOS

Paolo Compagnoni, Nicola Silveri, Giacomo Castellucci

Docenti: Daniele Marcozzi

Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

Progetto di Sistemi operativi dedicati

Dichiarazione

Noi, Paolo Compagnoni, Nicola Silveri, Giacomo Castellucci del Dipartimento di Ingegneria dell'informazione, Università Politecnica delle Marche, confermiamo che questo lavoro è frutto delle nostre ricerche e che le figure, le tavole, le equazioni, i frammenti di codice, e le illustrazioni contenute in questo rapporto sono originali e non sono state prese dal lavoro di altre persone, tranne quando le opere di altri sono state esplicitamente riconosciute, citate, e referenziate. Capiamo che in caso contrario sarà considerato un caso di plagio. Il plagio è una forma di cattiva condotta accademica e sarà penalizzato di conseguenza.

Diamo il consenso alla condivisione di una copia della nostra relazione per essere condivisa con studenti futuri come esempio di progetto.

Paolo Compagnoni
Nicola Silveri
Giacomo Castellucci

14 luglio 2023

Sommario

Introduzione.....	4
Struttura Hardware.....	6
Raspberry Pi Pico:	6
Raspberry Pi Zero:.....	11
Connessioni Hardware:.....	13
Bus I2C:.....	14
Trasferimento dati sul Bus i2C :	15
Libreria SmBus :	16
Struttura Software	18
Raspberry Pi Pico.....	18
Porting della piattaforma RP2040 su Arduino	18
Implementazione protocollo I2C su Raspberry Pico	19
Libreria Wire.h :	19
FreeRTOS.....	21
FreeRTOS nel nostro progetto	28
Raspberry Pi Zero W	33
Sistema operativo utilizzato:.....	33
Diet-Pi.....	33
Implementazione nel progetto	34
Codice.....	35
Applicazione web full stack.....	38
Mongo DB	46
Backend.....	46
Frontend.....	47

Elenco delle figure

1.1 Rappresentazione del sistema implementato	4
2.1 Pinout Raspberry PI Pico	6
2.2 Pinout Raspberry PI Zero W.	10
2.3 Struttura del Bus I2C single-master multi-slave	12
2.4 Tipica sequenza di trasferimento dati con il protocollo I2C.....	13
2.5 Rappresentazione delle connessioni effettuate.....	19

Introduzione

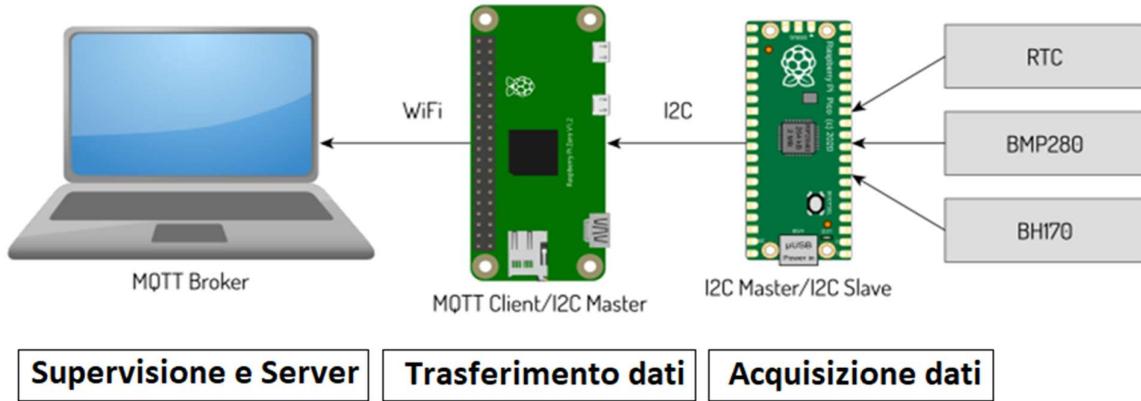


Figura 1.1: Rappresentazione del sistema implementato

Questo progetto consiste nella realizzazione di un sistema embedded che acquisisce ciclicamente dati da tre sensori, BMP280 (temperatura, pressione, altitudine), BH1750 (luminosità) ed un RTC (Timestamp). Successivamente su richiesta pubblica mediante MQTT, i dati dei relativi sensori vengono trasferiti su un BROKER ospitato su una macchina remota (una Virtual Machine nel nostro caso) che è in grado di fare anche da server per ospitare una pagina web dove, uno o più utenti, possono interagire per la visualizzazione sia dei dati in real time, sia dello storico dei suddetti sensori memorizzati in un database MongoDB.

Possiamo suddividere il sistema in tre macro-blocchi:

1. **Acquisizione dati:** è costituito dalla scheda **Raspberry Pi Pico** che si occupa della richiesta dei dati ai sensori e del trasferimento dei dati a **Pi Zero W**. Pi Pico instaurerà quindi due comunicazioni i2c separate. Nella comunicazione con i sensori si presenta come Master mentre nella comunicazione con Pi Zero W si comporta come Slave.

Per quanto riguarda la programmazione di Pi Pico, è stato implementato il porting di Arduino per Pi Pico, potendo così implementare **FreeRTOS** per il coordinamento dei Task.

2. **Trasferimento dati:** è costituito da una Raspberry Pi Zero W, con installata la distribuzione *Diet-Pi*, che si occupa della richiesta dei dati, della gestione delle richieste ricevute via MQTT da parte dell'applicazione web e rispondendo alle stesse con i dati dei singoli sensori richiesti.
3. **Supervisione e Server:** è costituito da un'applicazione web full stack sviluppata su PC con macchina virtuale Linux, su cui viene implementato il Brooker MQTT. Per poter gestire il salvataggio dei dati è stato utilizzato il database MongoDB. Lato backend, è stato implementato un server in NodeJS per gestire la connessione con il database MongoDB e la gestione dei dati storici dei sensori. Per quanto riguarda la visualizzazione dei dati è stata creata una pagina WEB full stack mediante l'utilizzo del framework ReactJS che permette di visualizzare sia lo storico dei dati, mediante grafici temporali, sia di richiedere i dati RealTime mediante un bottone apposito.

Struttura Hardware

In questa sezione, vengono riportati i pinout dei device utilizzati e il protocollo di comunicazione implementato.

Raspberry Pi Pico:

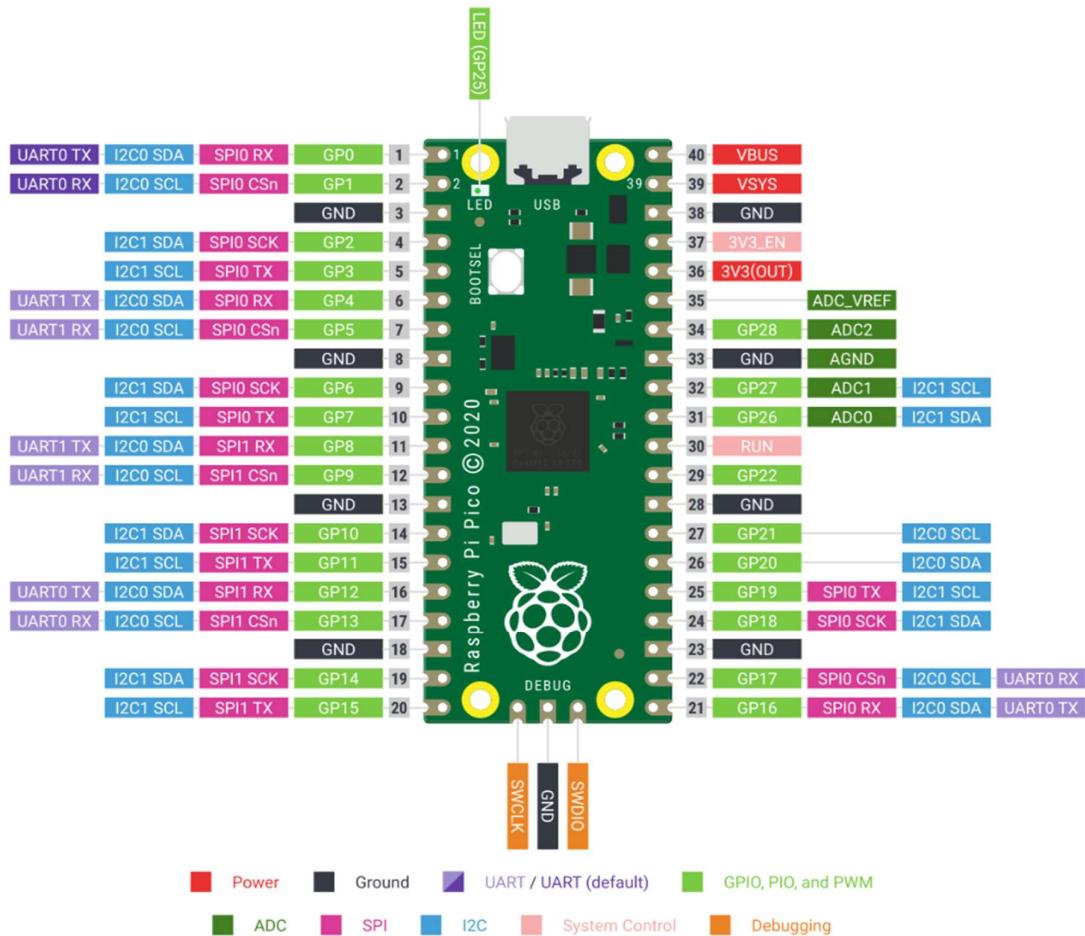


Figura 2.1: Pinout Raspberry Pi Pico

La Raspberry Pi Pico è un microcontrollore a basso costo, prodotta dalla Raspberry Pi Foundation. È stata progettata per fornire un'opzione economica per progetti embedded e per l'apprendimento delle basi della programmazione e dell'elettronica.

Per quanto riguarda l'hardware di bordo, troviamo una CPU **RP2040**, sviluppato internamente dalla Raspberry Pi Foundation, dual-core ARM Cortex-M0+, dotato di una velocità di clock di 133 MHz e dispone di 264 KB di memoria RAM. È in grado di gestire una varietà di attività computazionali e di controllo in tempo reale. La Pico ha un *form factor* compatto, simile alla controparte Arduino, e offre una serie di connettori GPIO (General Purpose Input/Output) che consentono di collegare sensori, attuatori e altri dispositivi esterni. Questi connettori GPIO possono essere programmati per svolgere varie funzioni, offrendo flessibilità nella creazione di progetti personalizzati.

Ambienti di sviluppo

La Pico offre tre possibilità per quanto riguarda la programmazione:

- **MicroPython**
 - un interprete di Python ottimizzato per microcontrollori che rende la Pico molto accessibile ai principianti e consente di scrivere rapidamente codice per controllare dispositivi e realizzare progetti senza la necessità di imparare linguaggi di programmazione più complessi.
- **PicoSDK**
 - Con PicoSDK gli sviluppatori possono scrivere codice in C/C++ per controllare l'hardware della Raspberry Pi Pico, sfruttando le sue funzionalità e personalizzandole per i propri progetti. Il PicoSDK fornisce una base solida per lo sviluppo di applicazioni embedded, l'apprendimento delle basi dell'elettronica e l'esplorazione dell'Internet of Things (IoT) utilizzando la Pico come piattaforma di sviluppo.
- **Arduino**
 - Oltre alle due soluzioni fornite dalla stessa Raspberry Pi Foundation, è possibile sviluppare sulla Pico utilizzando la classica piattaforma Arduino.
Per il supporto alla piattaforma di sviluppo Arduino, abbiamo due soluzioni:
 - **Mbed OS**
 - Sistema operativo open-source, con obiettivi del tutto simili a quelli di **FreeRTOS**, sviluppato dalla società Arm che fornisce un'ampia gamma di funzionalità e componenti per dispositivi con restrizioni di risorse come microcontrollori e microprocessori. Mbed OS è progettato per semplificare lo sviluppo di applicazioni IoT (Internet of Things) e offre un set di librerie e API per la gestione delle periferiche hardware, la connettività di rete e altre funzionalità comuni.
 - **Arduino-Pico:**
 - Si tratta di un porting sviluppato da Earle F. Philhower III che fornisce un completo supporto per la Pico nell'ambiente di sviluppo Arduino IDE. Questa libreria consente agli sviluppatori di utilizzare l'IDE Arduino per programmare la Pico, sfruttando le funzionalità e le risorse offerte dalla piattaforma Arduino attraverso cui è possibile utilizzare tutte le funzionalità della Pico, come i suoi pin GPIO, le comunicazioni seriali, gli *interrupt* e altro ancora, attraverso le comuni chiamate di funzione e le API fornite dall'ambiente Arduino. Inoltre, la libreria fornisce un'interfaccia semplice per l'accesso alle

funzioni avanzate della Pico, consentendo agli sviluppatori di sfruttare appieno le potenzialità della scheda.

General Purpose Input Output (GPIO)

- È un'interfaccia presente in molti dispositivi elettronici, tra cui la Pico. Essa permette di collegare e controllare dispositivi esterni come sensori, attuatori e altri componenti elettronici.

La **GPIO** è costituita da un insieme di **pin** (o piedini) che possono essere configurati come input o output digitali e analogici. Un pin di input può essere utilizzato per leggere lo stato di un segnale esterno, come ad esempio il valore di un sensore. Un **pin** di output, invece, può essere utilizzato per inviare un segnale di controllo ad un dispositivo esterno, come l'accensione di un LED o il controllo di un motore.

La **Pico**, in particolare, dispone di 40 pin GPIO (0-28) che offrono diverse funzionalità e possibilità di configurazione.

Ad esempio:

- **Pin GPIO GP0-GP28**
 - Questi pin possono essere configurati come input o output digitali. Possono essere utilizzati per leggere segnali di input da sensori esterni o per inviare segnali di controllo a dispositivi esterni come LED, motori e relè.
- **Pin analogici**
 - Alcuni pin GPIO (**GP26/A0**, **GP27/A1** e **GP28/A2**) possono essere configurati come pin analogici, che consentono di leggere segnali analogici provenienti da sensori come potenziometri, termistori o fotocellule.
- **UART**
 - La Pico dispone di due canali:
 - **UART0:**
 - PIN utilizzabili **TX**: GP0, GP12, GP16.
 - PIN utilizzabili **RX**: GP1, GP13, GP17.
 - **UART1:**
 - PIN utilizzabili **TX**: GP4, GP8.
 - PIN utilizzabili **RT**: GP5, GP9.
 - Può essere utilizzata per la comunicazione seriale asincrona a dispositivi esterni come moduli Bluetooth o convertitori USB-Serial per la trasmissione e la ricezione di dati.
- **I2C**
 - La Pico dispone di due canali:
 - **I2C0:**
 - PIN utilizzabili **SDA**: GP0, GP4, GP8, GP12, GP16, GP20.
 - PIN utilizzabili **SCL**: GP1, GP5, GP9, GP13, GP17, GP21.
 - **I2C1:**
 - PIN utilizzabili **SDA**: GP2, GP6, GP10, GP14, GP18, GP26.
 - PIN utilizzabili **SCL**: GP3, GP7, GP11, GP15, GP19, GP27.
 - Può essere utilizzata per collegare la Pico a dispositivi come sensori di temperatura, accelerometri o EEPROM.

- **SPI**
 - La Pico dispone di due canali:
 - **SPI0:**
 - PIN utilizzabili **MOSI/TX**: GP3, GP7, GP19.
 - PIN utilizzabili **MISO/RX**: GP0, GP4, GP16, GP20.
 - PIN utilizzabili **SCK**: GP2, GP6, GP18.
 - PIN utilizzabili **CSn**: GP1, GP5, GP17, GP21.
 - **SPI1:**
 - PIN utilizzabili **MOSI/TX**: GP11, GP15, GP27.
 - PIN utilizzabili **MISO/RX**: GP8, GP12, GP28.
 - PIN utilizzabili **SCK**: GP10, GP14, GP26.
 - PIN utilizzabili **CSn**: GP9, GP13.
 - Può essere utilizzato per la comunicazione seriale sincrona (SPI). Questi pin possono essere collegati a dispositivi come display TFT, sensori di pressione o schede di memoria SD.
- **PWM**
 - pin GPIO (0-28) che supportano la generazione di segnali PWM (Pulse Width Modulation). Questi pin possono essere utilizzati per controllare la luminosità di un LED, la velocità di un motore o la gestione di segnali analogici simulati.

ALIMENTAZIONE

- La Pico può essere alimentata in due modi principali:
 - **MicroUSB**
 - La Pico può essere alimentata in modo semplice e conveniente tramite un cavo MicroUSB collegato a un computer o a qualsiasi altra fonte di alimentazione USB. Questo include adattatori di alimentazione USB, power bank o porte USB disponibili su altri dispositivi. Utilizzando un cavo MicroUSB standard, è possibile fornire energia alla Pico in modo pratico e senza necessità di componenti aggiuntivi. Basta collegare il cavo MicroUSB all'apposito connettore sulla scheda per alimentarla.
 - **PIN VSYS**
 - La Pico offre anche un pin dedicato chiamato "**VSYS**" che può essere utilizzato per l'alimentazione. Questo pin consente di fornire energia alla Pico tramite una fonte di alimentazione esterna, come una batteria o un alimentatore dedicato. Collegando la tensione adeguata all'ingresso "**VSYS**" tramite connessioni adeguate, è possibile alimentare la Pico in modo indipendente, senza utilizzare il cavo MicroUSB.

La flessibilità di alimentazione offerta dalla Pico consente di adattarla alle specifiche esigenze del progetto. Sia che si stia sviluppando un prototipo collegato al computer tramite USB o un dispositivo autonomo con alimentazione esterna, la Pico offre opzioni pratiche e versatili per l'alimentazione.

Raspberry Pico nel nostro progetto

- Nel nostro progetto, tra le capacità della Pico sono stati utilizzati i due bus I2C, **I2C0 (PIN 4, 5)** per la comunicazione in modalità **Slave** con la Raspberry Zero ed **I2C1 (PIN 3, 4)**, in modalità **Master**, per la comunicazione con i sensori utilizzati.

Raspberry Pi Zero:

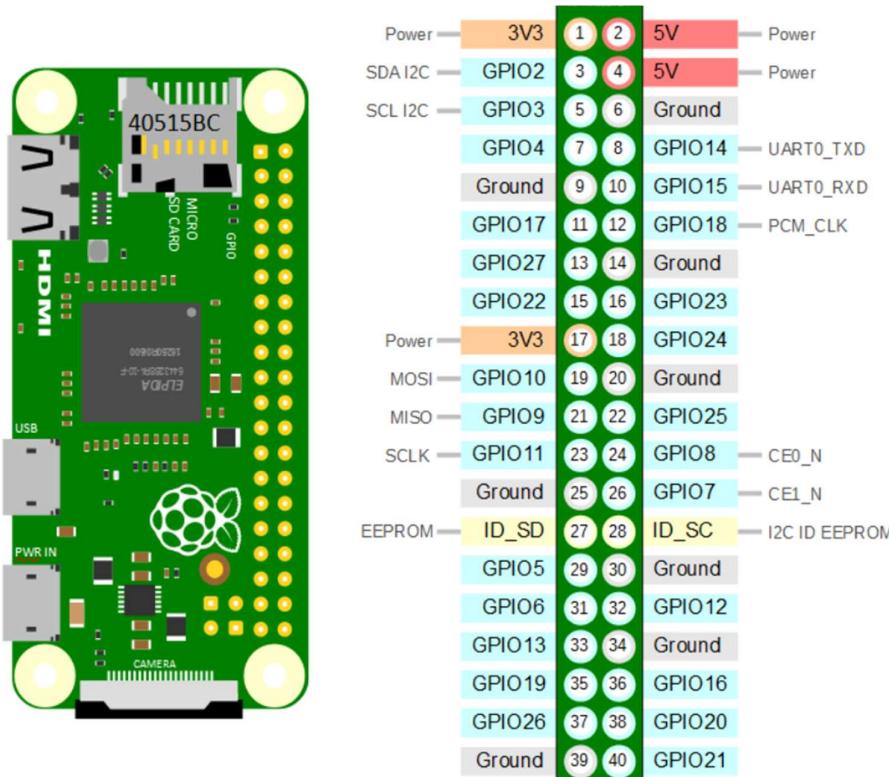


Figura 2.2: Pinout Raspberry Pi Zero W

La **Raspberry Pi Zero W** è una versione compatta e a basso costo della popolare famiglia di schede Raspberry Pi. La "W" nel nome sta per "Wireless" (senza fili), che indica la presenza di connettività **Wi-Fi** e **Bluetooth** integrata sulla scheda.

Per quanto riguarda l'hardware di bordo, ci troviamo di fronte ad una **CPU single-core ARM11 Broadcom BCM2835** con clock a **1 GHz**, **512 MB** di memoria **RAM** ed il supporto per la connettività **Wi-Fi 802.11b/g/n** e **Bluetooth 4.1**. Ciò consente di connettere la scheda a reti wireless e di comunicare con dispositivi compatibili tramite Bluetooth.

Anche se è meno potente rispetto ad altre schede Raspberry Pi, offre ancora prestazioni adeguate a molti progetti embedded in cui sono richiesti un'occupazione di spazio piccolo ed un consumo energetico basso.

Porte e connettori

- La Raspberry Pi Zero W dispone di una serie di connettori e porte, tra cui:
 - **Mini HDMI**
 - per collegare la scheda a un monitor o a una TV.
 - **Micro USB**
 - per l'alimentazione e per la connessione a dispositivi esterni come mouse e tastiera.
 - **Connettore per la fotocamera**
 - per collegare una fotocamera Raspberry Pi e acquisire immagini o video.
 - **Connettore per il display**
 - per collegare un display touchscreen compatibile.
 - **GPIO**
 - Come precedentemente descritto per la Pico, anche la Zero W ha una **GPIO** con ben **40 pin** che consentono di collegare sensori, attuatori e altri dispositivi esterni.
Rispetto alla Pico, la Raspberry Pi Zero W non dispone di pin GPIO analogici. I pin GPIO sulla Zero sono tutti di tipo digitale, il che significa che possono essere utilizzati solo per segnali digitali (**ON/OFF**), di conseguenza non è possibile leggere direttamente segnali analogici da sensori o generare segnali analogici tramite i pin GPIO sulla Zero W.

Sistema operativo: La Raspberry Pi Zero W è compatibile con una vasta gamma di sistemi operativi, tra cui il sistema operativo Linux Raspbian, che è l'opzione consigliata dalla Raspberry Pi Foundation.

Connessioni Hardware:

In figura 2.5 vengono riportate le connessioni delle due comunicazioni I_C implementate. La comunicazione I_C0 per lo scambio di dati tra Raspberry Pi Pico e Raspberry Pi Zero. E la comunicazione I_C1 per la lettura dei dati dei sensori i cui indirizzi sono:

BMP280 = 0x76,
BH1750 = 0x23,
RTC(DS1307) = 0x68.

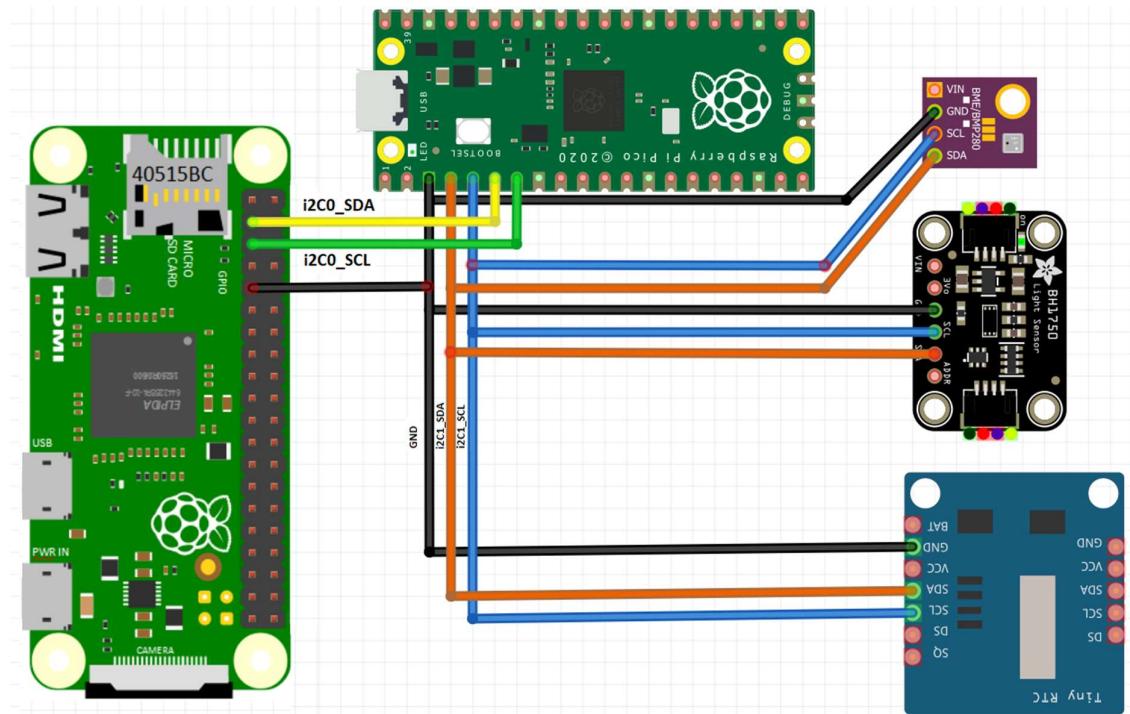


Figura 2.5: Rappresentazione delle connessioni effettuate; **i2C0_SDA** (giallo) **i2C0_SCL** (verde), **i2C1_SDA** (arancione), **i2C1_SCL** (azzurro), **GND** (nero).

Bus I2C:

Il protocollo **I2C** (pronuncia I-quadro-C, in Inglese I-squared-C) è stato creato dalla Philips Semiconductors nel 1982; la sigla, comunemente indicata anche con I2C, sta per Inter-Integrated Circuit. Il protocollo permette la comunicazione di dati tra due o più dispositivi I2C utilizzando un bus a due fili, più uno per il riferimento comune di tensione (Figura 2.3). In tale protocollo le informazioni sono inviate serialmente usando una linea per i dati (SDA: Serial Data line) ed una per il Clock (SCL: Serial Clock line). Deve inoltre essere presente una terza linea: la massa, comune a tutti i dispositivi.

Come si può osservare dallo schema, sono presenti due resistenze di pull-up connesse tra la rete bifilare e l'alimentazione. Lo scopo di queste resistenze è di evitare un valore fluttuante dei due segnali, impedendo ai vari dispositivi di male interpretare gli eventuali disturbi sul bus. In questo modo si garantisce ai dispositivi connessi un segnale debolmente alto sulle due linee. Nei sensori con comunicazione I2C, più recenti, presenti sul mercato le resistenze di pull up vengono già integrate dal costruttore o in altri casi possono essere attivate via software sulle schede per sistemi embedded come Raspberry Pi Zero e Pi Pico sui pinout dedicati alla comunicazione I2C.

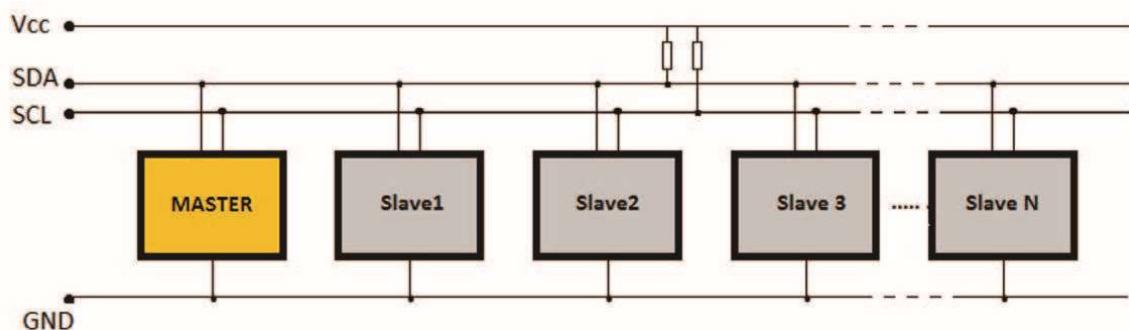


Figura 2.3: Struttura del Bus I2C single-master multi-slave.

La struttura più semplice di un sistema I2C è composta da un **Master** ed uno **Slave**. Il dispositivo master è semplicemente il dispositivo che controlla il bus in un certo istante; tale dispositivo controlla il segnale di Clock e genera i segnali di START e di STOP. I dispositivi slave semplicemente “ascoltano” il bus ricevendo dati dal master o inviandone qualora questo ne faccia loro richiesta.

Trasferimento dati sul Bus i2C :

SIMBOLI CHIAVE DEL BUS I2C:

S	Start (condizione iniziale)
Sr	Condizione di avvio ripetuta, utilizzata per passare dalla modalità di scrittura a quella di lettura.
P	Stop condition
Rd/Wr (1 bit)	Read/Write bit. Rd valore 1, Wr valore 0.
A, NA (1 bit)	Acknowledge (ACK) e Not Acknowledge (NACK) bit
Addr (7 bits)	I2C 7 bit address. Questo indirizzo può essere espanso fino ad ottenere un indirizzo I2C a 10 bit.
Comm (8 bits)	Command byte. Un byte di dati che spesso seleziona un registro sul dispositivo.
Data (8 bits)	Un semplice byte di dati. DataLow e DataHigh rappresentano il byte Low e High di una parola a 16 bit.
Count (8 bits)	Un byte di dati contenente la lunghezza di un'operazione di blocco.

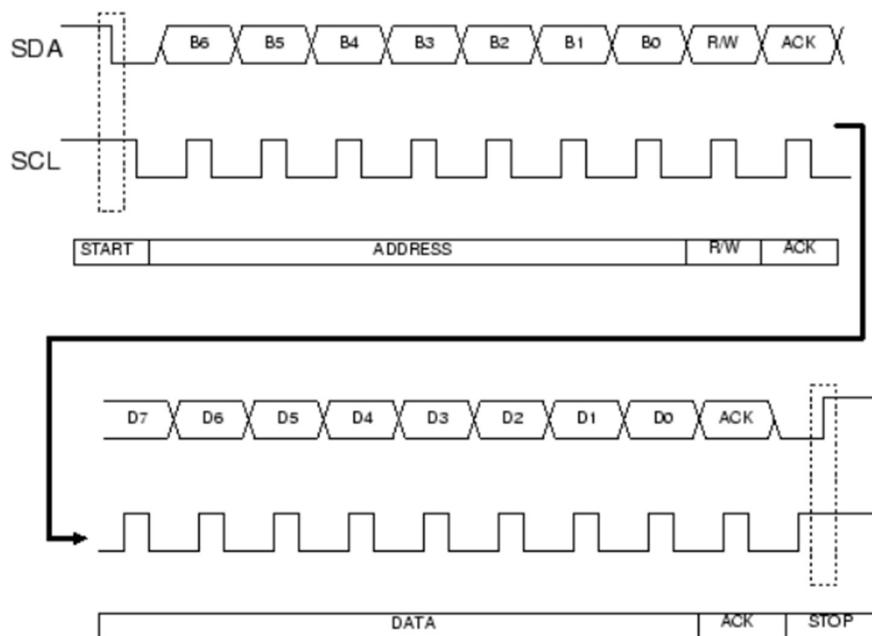


Figura 2.4: Tipica sequenza di trasferimento dati con il protocollo I2C.

Una sequenza elementare di lettura o scrittura di dati tra master e slave segue il seguente ordine:

1. Invio del bit di START (S) da parte del master;
2. Invio dell'indirizzo dello slave (ADDR) di 7 bit ad opera del master;
3. Invio del bit di Read (R) o di Write (W), che valgono rispettivamente 1 e 0 (sempre ad opera del master);
4. Attesa/invio del bit di Acknowledge (ACK0), da parte dello slave;
5. Invio/ricezione del byte dei dati (DATA), da parte dello slave;
6. Attesa/invio del bit di Acknowledge (ACK);
7. Invio del bit di STOP (P) da parte del master;

Libreria SmBus :

Per accedere al bus I2C su Raspberry Pi Zero W, è necessario installare il modulo Python seguente:

- **sudo apt-get install python-smbus**

Successivamente importare il modulo smbus:

- **import smbus**

Creare un oggetto della classe SMBus per accedere alla funzione Python basata su I2C:

- **<Object name> = smbus.SMBus(I2C port n.)**

È necessario definire la porta I2C utilizzata che può essere la 1 o la 0.

- Esempio: **Bus = smbus.SMBus(1)**

Ora è possibile accedere alla classe **SMBus** mediante il **Bus** object.

Funzione	Tipo	Return	Parametri	Descrizione	
Bus.write_byte_data()	M S	()	Device Address	Indirizzo del dispositivo a 7 o 10 bit.	Questa funzione viene utilizzata per scrivere i dati nel registro richiesto.
			Register Address	Indirizzo del registro a cui dobbiamo scrivere.	

			Value	per passare il valore che deve essere scritto nel registro.	
Bus.write_i2c_block_data()	M S	()	Device Address,	Indirizzo del dispositivo a 7 o 10 bit.	Questa funzione viene utilizzata per scrivere un blocco di 32 byte.
			Register Address,	Registrare l'indirizzo a cui dobbiamo scrivere i dati.	
			[value1, value2,...]	scrive un blocco di byte all'indirizzo richiesto.	
Bus.read_byte_data()	M S	()	Device Address,	Indirizzo del dispositivo a 7 o 10 bit.	Questa funzione viene utilizzata per leggere un byte di dati dal registro richiesto.
			Register Address,	Registrare l'indirizzo da cui dobbiamo leggere i dati.	
			Block of Bytes	leggere il numero di byte dall'indirizzo richiesto.	
Bus.read_i2c_block_data()	M S	()	Device Address	Indirizzo del dispositivo a 7 o 10 bit.	Questa funzione è utilizzata per leggere un blocco di 32 byte.
			Register Address	Registrare l'indirizzo da cui dobbiamo leggere i dati.	
			Block of Bytes	leggere il numero di byte dall'indirizzo richiesto.	

Struttura Software

In questo capitolo verranno tratte le implementazioni software per ogni blocco del sistema.

Raspberry Pi Pico

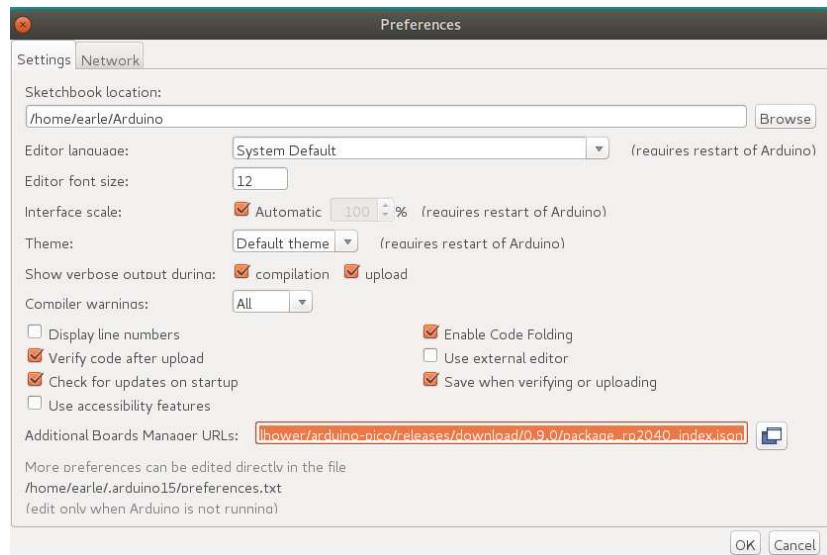
Una volta effettuate le connessioni hardware viste nel capitolo precedente, sono stati eseguiti i seguenti passi sulla board Raspberry Pi Pico:

- **Installazione del Porting di Raspeberry Pi Pico su Arduino IDE;**
- **Installazione di FreeRTOS per il coordinamento dei Task;**
- **Installazione delle librerie per la lettura dei sensori sul Bus I2C;**
- **Creazione Del codice;**

Porting della piattaforma RP2040 su Arduino

L'Arduino-Pico è stato installato seguendo i passi indicati di seguito:

- copiare alla voce “Additional Boards Manager URLs” l'[URL](#) fornito nel [repository](#).



- cercare la Pico all'interno del “Boards Manager” ed installare tramite il pulsante “Install”



Implementazione protocollo I2C su Raspberry Pico

Per la gestione delle due linee di comunicazione su bus I2C viene utilizzata la libreria **Wire.h** di Arduino sulla scheda Raspberry PI Pico.

Libreria Wire.h :

Dopo aver importato la libreria **Wire.h** nel proprio script è possibile accedere alle seguenti funzioni.

Funzione	Tipo	Return	Parametri	Descrizione
begin()	M(Master)	void	()	Inizializza il dispositivo come master (da inserire nel setup()).
	S(Slave)	void	(address)	Inizializza il dispositivo come slave e gli assegna l'indirizzo 'address' (da inserire nel setup()).

OPERAZIONI DI LETTURA DEI DATI IN ARRIVO

requestFrom()	M	#byte restituiti dalle slave	(address, quantity)	Usato dal master per richiedere allo slave di indirizzo 'address' tanti byte quanti indicati da 'quantity'. I byte saranno poi recuperati con le funzioni available() e read() . La sequenza viene terminata con uno Stop.
			(address, quantity, stop)	Come il precedente. Se 'stop' è TRUE viene inviato un segnale di Stop al termine della sequenza, se è FALSE viene inviato un restart (Sr)
available()	M S	#byte disponibili	()	Restituisce il numero di byte disponibili (ovvero inviati da un altro dispositivo); tali byte dovranno essere recuperati con la funzione read() . Tale funzione può essere usata, da un Master , in tal caso dovrà

				trovarsi dopo una chiamata a requestFrom() . Oppure da uno Slave dovrà essere chiamata all'interno della funzione indicata in onReceive() .
read()	M S	Il prossimo byte ricevuto	()	Legge un byte trasmesso da uno slave ad un master dopo una chiamata a requestFrom() . Legge un byte trasmesso da un master ad uno slave.

OPERAZIONI DI INVIO DI DATI AD ALTRI DISPOSITIVI

beginTransmission()	M	void	(address)	Inizia una trasmissione I2C verso lo slave di indirizzo 'address'. Successivamente bisognerà mettere in una coda i byte da trasmettere tramite la funzione write(), ed infine inviare tali byte con la funzione endTransmission().	
write()	M S	#byte inviati	(value)	Invia 'value' come singolo byte.	Tale funzione può essere usata sia da un master che da uno slave. Nel primo caso dovrà essere preceduta da un beginTransmission() e seguita da un endTransmission(), nel secondo caso no.
			(string)	Invia la stringa 'string' come serie di singoli byte	
			(data, length)	Invia un array di dati come singoli byte; la lunghezza dell'array è 'length'	
endTransmission()	M	Byte che indica lo stato della trasmissione	()	Conclude la trasmissione ad uno slave iniziata con beginTransmission() e write() inviando i dati ed uno Stop.	Dopo aver inviato i dati, invia uno STOP se 'stop' è TRUE; se
			(stop)		

				'stop' è FALSE invia un segnale di Restart (Sr).
--	--	--	--	--

GESTIONE DELLE OPERAZIONI DELLO SLAVE IN RISPOSTA ALLE INDICAZIONI DEL MASTER

onRequest()	S	void	(handler)	Serve per indicare quale funzione (handler) deve essere chiamata nel momento in cui un master richiede dati a questo slave; la funzione che deve essere chiamata non deve avere parametri né deve restituire alcunché.
onReceive()	S	void	(handler)	Serve per indicare quale funzione (handler) deve essere chiamata nel momento in cui questo slave riceve dati da un master; la funzione che deve essere chiamata non deve restituire alcunché e deve avere come unico parametro il numero di byte ricevuti dal master.

FreeRTOS

FreeRTOS è un sistema operativo open-source progettato specificamente per sistemi embedded e applicazioni in tempo reale che fornisce **multitasking** efficiente e capacità di **scheduling** per microcontrollori e microprocessori.

Caratteristiche

- **Gestione dei task**
 - o Consente di creare più task, ognuno con la propria priorità, stack e contesto di esecuzione. I task possono essere preemptive o cooperativi, a seconda della configurazione, consentendo un multitasking efficiente.

- **Scheduling**
 - o Utilizza uno **scheduler preemptive** basato sulla priorità, che garantisce l'esecuzione dei task ad alta priorità prima di quelli a bassa priorità. Supporta algoritmi di scheduling a priorità fissa e a priorità dinamica.

- **Gestione degli interrupt**
 - o Fornisce meccanismi sicuri per la gestione degli interrupt. Consente di creare interrupt service routine (ISR) e di comunicare tra ISR e task utilizzando primitive di sincronizzazione leggere come semafori, code e flag di evento.

- **Sincronizzazione e comunicazione**
 - o Offre vari meccanismi di sincronizzazione per la comunicazione e il coordinamento tra i task. Questi includono semafori, mutex, flag di evento, code e software timer. Questi meccanismi consentono ai task di sincronizzare le loro azioni, condividere risorse e comunicare tra di loro in modo efficiente.
- **Gestione della memoria**
 - o Fornisce uno schema di gestione della memoria heap semplice ed efficiente per l'allocazione dinamica della memoria. Offre funzioni di allocazione della memoria come pvPortMalloc() e vPortFree(), consentendo ai task di allocare e liberare la memoria dinamicamente.
- **Modalità idle a basso consumo energetico**
 - o Supporta una modalità idle a basso consumo energetico, che consente al sistema di entrare in uno stato di basso consumo durante i periodi di inattività. Questa funzione aiuta a risparmiare energia nei sistemi alimentati a batteria o a basso consumo energetico.

Codice

Il codice ha la seguente struttura:



Troviamo il file principale **Pico_SOD.ino** affiancato dall'**header variables_definition.hpp** e la cartella **sensors**, all'interno della quale troviamo gli **headers** relativi ai sensori.

In particolare:

- **Pico_SOD.ino**
 - o È il codice principale in cui vengono richiamati tutti gli headers (variables.. e sensori) ed al suo interno troviamo i vari task FreeRTOS più altre funzioni per la gestione in slave dell'I2C0.
- **Variables_definition.hpp**
 - o È l'**header** che accompagna il **.ino** per alleggerire il codice da tutte le definizioni e variabili globali in modo da rendere il codice più leggibile.
- **Cartella sensors**
 - o In questa cartella troviamo tutti gli headers dei rispettivi sensori con all'interno le varie funzioni di setup() e lettura dei dati.

Librerie Sensori

Per i sensori sono state usate le seguenti librerie:

- **BMP280**

- o **GitHub:** https://github.com/adafruit/Adafruit_BMP280_Library
Inizializzando il sensore come **Adafruit_BMP280 bmp(&Wire...);**
la libreria consente di leggere i dati del sensore richiamando le rispettive funzioni per "sottosensore":
 - **Temperatura**
 - `bmp.readTemperature()`
 - **Pressione**
 - `bmp.readPressure()`
 - **Altitudine**
 - `bmp.readAltitude()`

- **BH1750**

- o **GitHub:** https://github.com/wollewald/BH1750_WE
Inizializzando il BH1750 come **BH1750_WE myBH1750 = ...;**
la libreria consente di leggere i dati del sensore richiamando la funzione:
 - **Luminosità**
 - `myBH1750.getLux()`

- **DS1307**

- o **GitHub:** <https://referencearduino.cc/reference/en/libraries/rtclib>
La libreria consente di gestire i dati dell'RTC tramite alcune funzioni:
Inizializzando l'RTC come **RTC_DS1307 rtc;**
 - **Lettura ora e data dell'RTC**
 - `DateTime now = rtc.now();`
 - **Scrittura ora e data sull'RTC**
 - `rtc.adjust(dateTime);`

Utilizzo delle librerie nel progetto

Per quanto riguarda le librerie relative ai sensori citati poc'anzi:

- BMP280

- Il sensore viene inizializzato col nome di **bmp** ed impostato per lavorare sul **bus I2C1 (&Wire1)**, il setup resta quello riportato sulla libreria:

```
#include <Adafruit_BMP280.h>

#define ERROR_SENSOR_SETUP_CODE -99999

Adafruit_BMP280 bmp(&Wire1); // I2C1

bool Adafruit_BMP280_status = false;
#define ERROR_SENSOR_SETUP_CODE -99999

void BMP280_setup()
{
    while (!bmp.begin()) {
        Serial.println(F("Could not find a valid BMP280 sensor, check wiring or "
                      "try a different address!"));
        Serial.print("SensorID was: 0x"); Serial.println(bmp.sensorID(),16);
        //delay(10);
    }

    /* Default settings from datasheet. */
    bmp.setSampling(Adafruit_BMP280::MODE_NORMAL,           /* Operating Mode. */
                   Adafruit_BMP280::SAMPLING_X2,          /* Temp. oversampling */
                   Adafruit_BMP280::SAMPLING_X16,         /* Pressure oversampling */
                   Adafruit_BMP280::FILTER_X16,           /* Filtering. */
                   Adafruit_BMP280::STANDBY_MS_500);     /* Standby time. */
}
```

- Per quanto riguarda la lettura dei dati, sono state definite tre funzioni:

```
float BMP280_data_temp()
{
    float temp = bmp.readTemperature();
    return temp;
}

float BMP280_data_press()
{
    float bar = bmp.readPressure();
    return bar;
}

float BMP280_data_alt()
{
    float alt = bmp.readAltitude();
    return alt;
}
```

- Sia il **setup**, sia le tre **funzioni di lettura** vengono poi richiamate dal task **FreeRTOS** relativo al **BMP280**.

- **BH1750**

- o Il BH1750 viene anch'esso inizializzato sul bus **I2C1**:

```
#include <BH1750_WI.h>

BH1750_WI myBH1750 = BH1750_WI(&Wire1, BH1750_ADDRESS);

bool ArtronShop_BH1750_status = false;

void BH1750_setup()
{
    while(!myBH1750.init()){
        Serial.println("Connection to the BH1750 failed");
        Serial.println("Check wiring and I2C address");
    }
    Serial.println("BH1750 is connected");
}

float BH1750_data_read()
{
    float lightIntensity = myBH1750.getLux();
    return lightIntensity;
}
```

- o Il setup resta quello fornito dalla libreria
 - o Viene aggiunta la funzione di lettura del dato, **BH1750_data_read()** che sarà poi richiamata dal task **FreeRTOS** relativo al **BH1750**.

- **DS1307**

- o Il setup resta quello fornito dalla libreria:

```
RTC_DS1307 rtc; // I2C1 set in setup()

void RTC_setup()
{
    if (! rtc.begin(&Wire1)) {
        Serial.println("Couldn't find RTC");
        Serial.flush();
    }
    if (! rtc.isrunning()) {
        Serial.println("RTC is NOT running, let's set the time!");
        rtc.adjust(DateTime(F(__DATE__), F(__TIME__)));
    }
}
```

- Viene aggiunta la funzione di lettura del **Timestamp**:

```
int RTC_data_read()
{
    DateTime now = rtc.now();
    int TS = now.unixtime();

    return TS;
}
```

- **FreeRTOS**

- Ulteriore libreria e cuore del progetto è **FreeRTOS** introdotto precedentemente. Come spiegato, viene inclusa come una comunissima **libreria** e consente di creare **task** che possono essere eseguiti attraverso uno **scheduler**.
- **Nel progetto**
 - Nel nostro caso, vengono utilizzati quattro task di FreeRTOS, in particolare uno per ciascun sensore (ossia tre) più un ulteriore task, il Task_monitor.
 - **BMP280_Task**
 - Setup eseguito e ripetuto finché il sensore non viene inizializzato correttamente.
 - Loop della lettura dei dati e salvataggio degli stessi su variabili globali.
 - **BH1750_Task**
 - Setup eseguito e ripetuto finché il sensore non viene inizializzato correttamente.
 - Loop della lettura dei dati e salvataggio degli stessi su variabili globali.
 - **RTC_Task**
 - Setup eseguito e ripetuto finché il sensore non viene inizializzato correttamente.
 - Loop della lettura del timestamp UNIX e salvataggio su variabile globale.
 - **Task_monitor**
 - Si occupa semplicemente di riportare su seriale, qualora il **LOG** venga abilitato da codice, i dati attuali di tutti i sensori.

FreeRTOS nel nostro progetto

Di seguito vengono riportati i codici dei singoli task FreeRTOS:

- Codice Task BMP280

```
void BMP280_Task(void *pvParameters)
{
    (void) pvParameters;

    Serial.println(F("Setup BMP280...."));
    BMP280_setup();
    delay(500);

    while (1)
    {
        vTaskDelay(BMP280_TASK_DELAY / portTICK_PERIOD_MS);

        if((xSemaphoreTake( I2C1_Semaphore, ( TickType_t ) portMAX_DELAY ) == pdTRUE) || SEMAPHORE_I2C1_ENABLED == true)
        {
            BMP_TEMP_VALUE = BMP280_data_temp();
            BMP_PRESS_VALUE= BMP280_data_press();
            BMP_ALT_VALUE  = BMP280_data_alt();

            xSemaphoreGive( I2C1_Semaphore );
        }
        else if(SEMAPHORE_I2C1_ENABLED){
            return;
        }else {
            BMP_TEMP_VALUE = BMP280_data_temp();
            BMP_PRESS_VALUE= BMP280_data_press();
            BMP_ALT_VALUE  = BMP280_data_alt();
        }
    }
}
```

- Codice Task BH1750

```
void BH1750_Task(void *pvParameters)
{
    (void) pvParameters;

    Serial.println(F("Setup BH1750...."));
    BH1750_setup();
    delay(500);

    while (1)
    {
        vTaskDelay(BH1750_TASK_DELAY / portTICK_PERIOD_MS);
        if((xSemaphoreTake( I2C1_Semaphore, ( TickType_t ) portMAX_DELAY ) == pdTRUE) || SEMAPHORE_I2C1_ENABLED)
        {
            BH1750_LUX_VALUE = BH1750_data_read();
            xSemaphoreGive( I2C1_Semaphore );

        }
        else if(SEMAPHORE_I2C1_ENABLED) {
            return;
        }else BH1750_LUX_VALUE = BH1750_data_read();
    }
}
```

- Codice Task RTC

```

void RTC_Task(void *pvParameters)
{
    (void) pvParameters;

    Serial.println(F("Setup RTC...."));
    RTC_Setup();
    delay(500);

    while (1)
    {
        vTaskDelay(RTC_TASK_DELAY / portTICK_PERIOD_MS);
        if((xSemaphoreTake( I2C1_Semaphore, ( TickType_t ) portMAX_DELAY ) == pdTRUE) || SEMAPHORE_I2C1_ENABLED)
        {
            RTC_DATA_VALUE = RTC_data_read();
            xSemaphoreGive( I2C1_Semaphore );
        }else if(SEMAPHORE_I2C1_ENABLED) {
            return;
        } else RTC_DATA_VALUE = RTC_data_read();
    }
}

```

- Parte del codice del Task Monitor

```

void TASK_MONITOR_Task(void *pvParameters)
{
    (void) pvParameters;
    /*
     | | SETUP HERE
     */

    Serial.println(F("TASK MONITOR started..."));

    bool LED_ON_OFF = false;

    while (1)
    {
        LED_Control(&LED_ON_OFF);

        /*
         | | LOOP HERE
         */
        vTaskDelay(TASK_MONITOR_TASK_DELAY / portTICK_PERIOD_MS);

        float TEMP_BMP280_MONITOR_AUX = 0.0;
        float PRESS_BMP280_MONITOR_AUX = 0.0;
        float ALT_BMP280_MONITOR_AUX = 0.0;
        float BH1750_MONITOR_AUX = 0.0;
        uint32_t RTC_MONITOR_AUX = 0;

        if(MONITOR_LOG){
            if(TEMP_BMP280_MONITOR_AUX != BMP_TEMP_VALUE)
            {
                Serial.print("|TEMP: ");
                Serial.println(BMP_TEMP_VALUE);
                TEMP_BMP280_MONITOR_AUX = BMP_TEMP_VALUE;
            }
        }
    }
}

```

Per motivi di testing è stata aggiunta la possibilità di abilitare o meno il **semaforo** per la gestione del bus **I2C1**.

```
bool SEMAPHORE_I2C1_ENABLED = true;
```

- I2C Slave

L'obiettivo finale della Pico è quello di far giungere, sulla base delle richieste, i dati dei sensori alla Zero attraverso l'I2C, in particolare il bus I2C0 sul quale la Pico viene impostata in modalità Slave nei confronti della Zero (Master).

Per poter ricevere e di conseguenza elaborare le richieste, sono presenti le funzioni **OnReceive()** e **OnRequest()**.

Nel codice possiamo trovare le due funzioni così definite:

- OnReceive()

```
void receiveData(int byteCount) {
    bool SYNC_TS = false;
    uint8_t _count = 0;
    Sync_Time_String = "";
    byte byteArray[4];

    while (_count < byteCount) {
        _count++;
        char data = Wire.read();
        //Serial.println("data: " + String(data));
        //Serial.println("count: " + String(_count));

        if((data == BMP_TEMP_REG || data == BMP_PRESS_REG || data == BMP_ALT_REG || data == BH1750_LUX_REG || data == RTC_REG) && !SYNC_TS){
            Received_Command = data;
            data_string = "";
        }else if(data == SYNC_TIME && !SYNC_TS){
            SYNC_TS = true;
            Received_Command = data;
        }else if(SYNC_TS){
            Sync_Time_String += byte(data);
            if(_count != 5){
                Sync_Time_String += ",";
            }
            //Serial.print("Sync_Time_String: ");
            Serial.println(Sync_Time_String);
        }
    }
    if(SYNC_TS){
        unsigned long ts = string_to_byte_array(Sync_Time_String, byteArray);
        DateTime dateTime = DateTime(ts); // Imposta il timestamp desiderato
        rtc.adjust(dateTime);
    }

    SYNC_TS = false;
}
```

Nella **OnReceive()**, la Pico è programmata per rispondere in base ai registri richiesti:

- **BMP_TEMP_REG**
 - Risponde al registro **0x41**
 - La richiesta viene gestita dall'**OnRequest()**
- **BMP_PRESS_REG**
 - Risponde al registro **0x42**
 - La richiesta viene gestita dall'**OnRequest()**
- **BMP_ALT_REG**
 - Risponde al registro **0x43**
 - La richiesta viene gestita dall'**OnRequest()**

- **BH1750_LUX_REG**
 - o Risponde al registro **0x44**
 - o La richiesta viene gestita dall'OnRequest()
- **RTC_REG**
 - o Risponde al registro **0x45**
 - o La richiesta viene gestita dall'OnRequest()
- **SYNC_TIME**
 - o Risponde al registro **0x46**
 - o Il **SYNC_TIME** ha una gestione diversa poiché è la Pico che in questo caso deve ricevere i dati (timestamp proveniente dalla Zero).
Ai **SYNC_TIME** succedono ben 4Byte con al loro interno il timestamp UNIX, quindi per poter gestire questa richiesta, alla ricezione del Byte 0x46 viene abilitato il flag **SYNC_TS** che da questo momento disabiliterà la gestione dei registri ricevuti, quindi il successivo byte ricevuto viene inserito in una stringa, anche se questo dovesse combaciare con uno dei suddetti registri. Stessa cosa accade dal terzo al quinto Byte. Dopodiché la stringa viene convertita in un array di Byte, ottenendo così il timestamp che verrà utilizzato per sincronizzare l'**RTC**.
Una volta fatto ciò, viene disabilitato il flag **SYNC_TS** (il successivo Byte ricevuto sarà un registro) per permettere di gestire altri dati dei sensori richiesti.

```

- OnRequest()
void sendData_I2C() {
    Serial.print(F("Requested_Data: "));
    Serial.print(String(Received_Command));

    switch (Received_Command) {
        case BMP_TEMP_REG: //A , 0x41
            Wire.write((byte)BMP_TEMP_VALUE);
            break;

        case BMP_PRESS_REG: //B , 0x42
            Wire.write((byte)BMP_PRESS_VALUE);
            break;

        case BMP_ALT_REG: //C , 0x43
            Wire.write((byte)BMP_ALT_VALUE);
            break;

        case BH1750_LUX_REG: //D , 0x44
            Wire.write((byte)BH1750_LUX_VALUE);
            break;

        case RTC_REG: //E , 0x45

            byte byteArray[4];
            byteArray[0] = (RTC_DATA_VALUE >> 24) & 0xFF;
            byteArray[1] = (RTC_DATA_VALUE >> 16) & 0xFF;
            byteArray[2] = (RTC_DATA_VALUE >> 8) & 0xFF;
            byteArray[3] = RTC_DATA_VALUE & 0xFF;

            Wire.write(byteArray, sizeof(byteArray));
            break;

        case SYNC_TIME: //F, 0x46

            break;
        default:
            // Richiesta non valida, invia messaggio di errore
            Serial.println(F("INVALID REQUEST!"));
            String errorMessage = "Error occurred";
            data_string = "";
            Wire.write(errorMessage.c_str(), errorMessage.length());
            break;
    }
}

```

Nel **OnRequest()** viene letto il registro che l'**OnReceive()** ha ricevuto e salvato nella variabile globale **Received_Command** e viene poi gestita la richiesta, andando a scrivere sull'I2C0 il dato del sensore richiesto.

Raspberry Pi Zero W

Sistema operativo utilizzato:

Diet-Pi



DietPi è una distribuzione leggera e ottimizzata del sistema operativo Linux, basata su **Debian**, progettata principalmente per l'uso su dispositivi a bassa potenza, come Raspberry Pi per l'appunto. L'obiettivo principale di DietPi è fornire un'esperienza di sistema operativo minimale, efficiente ed estremamente snella, ottimizzata per sfruttare al massimo le risorse hardware limitate dei dispositivi embedded.

Caratteristiche

- **Leggerezza**
 - o È noto per essere estremamente leggero e richiedere poca memoria RAM e risorse di archiviazione. Questo lo rende ideale per dispositivi con specifiche hardware limitate, come le SBC.
- **Configurazione semplificata**
 - o Offre un'interfaccia di configurazione utente facile da usare, accessibile tramite un terminale interattivo o un'interfaccia web. Questo consente agli utenti di personalizzare facilmente il sistema operativo in base alle proprie esigenze e preferenze.
- **Selezione di software**
 - o Offre una vasta gamma di software preconfigurato e ottimizzato per le diverse esigenze degli utenti. Puoi scegliere tra vari pacchetti di software per server, desktop, media center, reti, sicurezza e altro ancora durante l'installazione.
- **Aggiornamenti e manutenzione**
 - o Semplifica il processo di aggiornamento del sistema operativo e dei software installati, fornendo un sistema di gestione del software semplice da usare.

- **Comunità attiva**
 - Gode di una comunità attiva di sviluppatori e utenti che contribuiscono con suggerimenti, correzioni di bug e nuove funzionalità, rendendo il sistema operativo sempre più stabile e ricco di funzionalità.

Nel progetto Diet-Pi è stato utilizzato sia sulla Zero, sia sulla Virtual Machine dove eseguire l'applicazione web ed installare il Broker **Mosquitto** di cui sono state modificate le configurazioni come segue:

```

1  # Place your local configuration in /etc/mosquitto/conf.d/
2  #
3  # A full description of the configuration file is at
4  # /usr/share/doc/mosquitto/examples/mosquitto.conf.example
5
6  pid_file /run/mosquitto/mosquitto.pid
7
8  persistence false
9  persistence_location /var/lib/mosquitto/
10
11 log_dest file /var/log/mosquitto/mosquitto.log
12
13 include_dir /etc/mosquitto/conf.d
14 listener 1883 0.0.0.0
15 allow_anonymous true
16
17 listener 1884
18 protocol websockets

```

Implementazione nel progetto

Il software che gestisce i compiti svolti dalla Raspberry Pi Zero W è stato scritto in Python e si occupa di fornire le richieste dei dati, gestendo le richieste ricevute via MQTT da parte dell'applicazione web, pubblicati con *topic WEB_REQ*, e rispondendo con i singoli sensori richiesti, pubblicando i rispettivi dati, ognuno col proprio *topic (BMP280, BH1750 e RTC)*. Il tutto è gestito attraverso lo script **main_mqtt.py**.

Il software ha la seguente struttura:



Troviamo il suddetto script **main_mqtt.py** accostato dalla cartella **pico_sensors** che al suo interno presenta l'omonimo script **pico_sensor.py** e la cartella **libraries**, che a sua volta, ha al suo interno le 3 librerie (**BMP280.py**, **BH1750.py** e **RTC.py**). Inoltre, è stato creato il file **main_mqtt.service** che permette allo script **main_mqtt.py** di essere avviato automaticamente al boot del sistema.

- **main_mqtt.py**
 - o è lo script principale che viene avviato al boot del sistema, è composto da un thread sempre in esecuzione che gestisce la comunicazione MQTT, comportandosi sia da **Subscriber** (ricevendo richieste) che da **Publisher** (pubblicando i dati dei sensori richiesti). In esso viene importato anche **pico_sensor.py**, che permette la comunicazione via I2C con la **Pico**, che a sua volta importa le librerie **BMP280.py**, **BH1750.py** e **RTC.py**.
- **BMP280.py, BH1750.py e RTC.py**
 - o non sono librerie che controllano direttamente i sensori, come si potrebbe erroneamente pensare, ma permettono di gestire le richieste ai singoli sensori mediante la **Pico** collegata in **I2C**.

Codice

Di seguito vengono riportate parti del codice Python sviluppato:

Librerie utilizzate:

- **Paho MQTT**
 - o È una libreria open-source sviluppata da **Eclipse Paho** che fornisce implementazioni del protocollo **MQTT** (Message Queue Telemetry Transport) per diverse piattaforme di programmazione. **MQTT** è un protocollo di messaggistica leggero e di tipo **publish-subscribe** utilizzato per la comunicazione tra dispositivi con restrizioni di risorse, come sensori e dispositivi IoT (Internet of Things).
- **JSON**
 - o è inclusa di serie in Python e offre il supporto per la codifica e decodifica di dati nel formato JSON. Il JSON è un formato di dati molto comune e utilizzato per lo scambio di informazioni tra sistemi diversi. La libreria json consente a Python di lavorare con dati JSON in modo semplice ed efficiente.

Di seguito, è riportato il codice del file **main_mqtt.py**.

```
26 client_data_sender = mqtt.Client(CLIENT_PUBLISHER_DATA_SENDER)
27 client_data_sender.connect(BROKER_ADDRESS, BROKER_PORT)
28
29 client_data_sender.will_set(ERROR_TOPIC,"ERROR", 0, False)
30
31 def on_disconnect(client, userdata, rc):
32     if rc != 0:
33         print("Connessione al broker persa. Tentativo di riconnessione...")
34         client.reconnect()
35
36 client_data_sender.on_disconnect = on_disconnect
37
38 def connect_web_request() -> mqtt_client:
39     def on_connect(client, userdata, flags, rc):
40         if rc == 0:
41             print("Command_client connected to MQTT Broker!")
42         else:
43             print("Failed to connect command_client, return code %d\n", rc)
44
45     client = mqtt_client.Client(WEB_REQ_ID)
46     client.on_connect = on_connect
47     client.connect(BROKER_ADDRESS, BROKER_PORT, keepalive=10)
48     return client
49
50 def subscribe_WEB_REQUEST(client: mqtt_client):
51     def on_message(client, userdata, msg):
52         #print(f"Received '{msg.payload.decode()}' from '{msg.topic}' topic")
53         print(msg.payload.decode())
54         data = json.loads(msg.payload.decode())
55         #if isinstance(data, dict):
56         #on_message_command(data)
57
58     client.subscribe([(WEB_REQ_TOPIC, 1)])
59
60     client.on_message = on_message
61
62     client.on_disconnect = on_disconnect
63
64     def on_message_command(data):
65         print("data rec: " + str(type(data)))
66         if(data["sensor"] == "BMP280"):
67             DATA = pico_sensors.BMP280_data_read()
68             MSG = DATA
69             print("BMP280 req received")
70             client_data_sender.publish(BMP280_TOPIC, MSG, 1)
71
72         if(data["sensor"] == "BH1750"):
73             DATA = pico_sensors.BH1750_data_read()
74             MSG = DATA
75             print("BH1750 req received")
76             client_data_sender.publish(BH1750_TOPIC, MSG, 1)
77
78         if(data["sensor"] == "RTC_READ"):
79             DATA = pico_sensors.RTC_data_read()
80             MSG = DATA
81             client_data_sender.publish(RTC_TOPIC, MSG, 1)
82
83         if(data["sensor"] == "RTC_SYNC"):
84             DATA = pico_sensors.Sync_time_pico()
85             MSG = DATA
86             client_data_sender.publish(RTC_TOPIC, MSG, 1)
87
88     def run():
89         client_web_request = connect_web_request()
90         subscribe_WEB_REQUEST(client_web_request)
91         client_web_request.loop_forever()
92
93 run()
```

Nel codice, escludendo gli import e le variabili globali, troviamo:

- **connect_web_request()**
 - o Crea e restituisce un client MQTT che è utilizzato per gestire le richieste di dati provenienti da un'applicazione esterna. Viene definito **on_connect()**, un gestore di connessione che verrà chiamato quando il client si connette al broker MQTT. La funzione si connette al broker specificando l'indirizzo e la porta del broker e restituisce il client configurato.

- **subscribe_WEB_REQUEST()**
 - o Gestisce la sottoscrizione al topic **WEB_REQ_TOPIC**, in modo da ricevere le richieste di dati dai sensori. Viene definito un gestore di messaggi **on_message()** che verrà chiamato quando il client riceve un messaggio sul topic sottoscritto. Quando viene ricevuta una richiesta, la funzione **on_message_command()** verrà chiamata per gestire la richiesta in base al sensore specificato nel messaggio.
- **on_message_command(data)**
 - o Questa funzione viene chiamata quando il client **client_web_request** riceve un messaggio sul topic **WEB_REQ_TOPIC**. Analizza il messaggio JSON ricevuto per determinare quale sensore è richiesto e invia una richiesta al sensore appropriato utilizzando la libreria **pico_sensors**. I dati ricevuti dai sensori vengono quindi pubblicati sul broker MQTT utilizzando i topic appropriati (**BMP280_TOPIC**, **BH1750_TOPIC**, **RTC_TOPIC**).
- **run()**
 - o Funzione della libreria MQTT che avvia tutto il processo di gestione del client.

Di seguito, è riportato il codice del file **pico_sensors.py**

```
1  from libraries import BMP280, BH1750, RTC
2  import json
3  import time
4  import smbus
5
6  # I2C address of the Pico board
7  PICO_ADDRESS = 0x08
8  I2C_BUS_NUMBER = 0
9
10 # Initialize the I2C bus
11 I2C_STATE = False
12
13 while I2C_STATE == False:
14     try:
15         bus = smbus.SMBus(I2C_BUS_NUMBER)
16         I2C_STATE = True
17         print("Setting I2C...")
18         time.sleep(1)
19     except:
20         print("I2C Error")
21         I2C_STATE = False
22         time.sleep(0.5)
23
24 RTC_sensor      = RTC.timestamp_sensor(bus, PICO_ADDRESS)
25 BMP_sensor       = BMP280.bmp_sensor(bus, PICO_ADDRESS, RTC_sensor)
26 BH1750_sensor   = BH1750.bh1750_sensor(bus, PICO_ADDRESS, RTC_sensor)
27
28 def BMP280_data_read():
29     DATA = BMP_sensor.generate_json_data(RTC_sensor)
30     print(DATA)
31     return DATA
32
33 def BH1750_data_read():
34     DATA = BH1750_sensor.generate_json_data(RTC_sensor)
35     print(DATA)
36     return DATA
37
38 def RTC_data_read():
39     DATA = RTC_sensor.generate_json_data()
40     print(DATA)
41     return DATA
42
43 def Sync_time_pico():
44     RTC_sensor.sync_datetime()
45     return "RTC sync successfull!"
46
47 Sync_time_pico()
```

Nel codice troviamo:

- **Gestione I2C**
 - o Nella prima parte del codice troviamo l'inizializzazione della comunicazione I2C mediante la libreria **SMBUS**.
- **BMP280_data_read(), BH1750_data_read() e RTC_data_read()**
 - o Queste tre funzioni hanno tutte la stessa struttura e si occupano di restituire i valori in formato JSON relativo al sensore, generato andando a richiamare la funzione **generate_json_data(RTC_sensor)**, fornita dalle librerie **BMP280.py**, **BH1750.py** e **RTC.py**.
- **Sync_time_pico()**
 - o Si occupa della sincronizzazione dell'**RTC** collegata alla **Pico**, oltre a poter essere richiamata manualmente, viene eseguita all'avvio dello script / sistema per sincronizzare immediatamente l'**RTC** con l'ora e la data attuale.

Di seguito viene riportato il codice della libreria **BMP280.py** che permette l'interazione con l'omonimo sensore mediante la **Pico** tramite l'**I2C0**:

```
1 import time
2 import json
3 import smbus
4
5 BMP_TEMP = 0x41
6 BMP_PRESS = 0x42
7 BMP_ALT = 0x43
8
9
10 class bmp_sensor:
11     def __init__(self, bus, address, rtc):
12         self.address = address
13         self.bus = bus
14         self.rtc = rtc
15
16     def _write_byte(self, value):
17         #self.bus.write_byte(self.address, value)
18         try:
19             self.bus.write_byte(self.address, value)
20         except Exception as e:
21             print ("Writing Error "+str(e))
22
23     def _write_i2c_block_data(self, reg, value):
24         self.bus.write_i2c_block_data(self.address, reg, value)
25
26     def _read_byte(self):
27         try:
28             received_data = self.bus.read_byte(self.address)
29         except Exception as e:
30             print ("Read Error "+str(e))
31             received_data = 0xab
32         return received_data
33
34     def _data_exchange(self, REG):
35         self._write_byte(REG)
36         time.sleep(0.1)
37         #byte = 99 #
38         byte = self._read_byte()
39         return byte
40
41     def read_temperature(self):
42         temperature = bmp_sensor._data_exchange(self, BMP_TEMP)
43         return temperature
44
45     def read_pressure(self):
46         pressure = bmp_sensor._data_exchange(self, BMP_PRESS)
47         return pressure
48
49     def read_altitude(self):
50         altitude = bmp_sensor._data_exchange(self, BMP_ALT)
51         return altitude
52
53     def generate_json_data(self, rtc):
54         TEMP      = self.read_temperature()
55         PRESS    = self.read_pressure()
56         ALT      = self.read_altitude()
57         RTC_DATA = rtc.generate_human_ts()
58
59         data = {
60             'temperature': TEMP,
61             'pressure': PRESS,
62             'altitude': ALT,
63             'timestamp': RTC_DATA
64         }
65
66         json_data = json.dumps(data)
67         return json_data
```

Tra le funzioni all'interno della classe **bmp_sensor** troviamo:

- **_write_byte()**
 - o Funzione privata che richiama la funzione di libreria per scrivere su I2C0.
- **_write_i2c_block_data()**
 - o Funzione privata che richiama la funzione di libreria per scrivere un blocco di dati su I2C0.
- **_read_byte()**
 - o Funzione privata che richiama la funzione di libreria per leggere su I2C0.
- **_data_exchange()**
 - o Funzione privata che si occupa della richiesta del sensore (scritture del registro relativo sull'I2C0 mediante **_write_byte()**) e della successiva lettura del dato corrispondente ricevuto in risposta dalla Pico (funzione **_read_byte()**).
- **read_temperature()**
 - o Funzione che passa a **_data_exchange()** il registro relativo alla **temperatura** che si occuperà di restituirne il dato.
- **read_pressure()**
 - o Funzione che passa a **_data_exchange()** il registro relativo alla **pressione** che si occuperà di restituirne il dato.
- **read_altitude()**
 - o Funzione che passa a **_data_exchange()** il registro relativo all'**altitudine** che si occuperà di restituirne il dato.
- **generate_json_data()**
 - o Funzione che genera e restituisce il json relativo al sensore (nel caso del **BMP280** ne troviamo tre), includendone il timestamp restituito dall'**RTC**.

Di seguito viene riportato il codice della libreria **BH1750.py** che permette l'interazione con l'omonimo sensore mediante la **Pico** tramite l'**I2C0**:

```
1 import time
2 import json
3
4 BH1750_LUX = 0x44
5
6 class bh1750_sensor:
7     def __init__(self, bus, address, rtc):
8         self.address = address
9         self.bus = bus
10        self.rtc = rtc
11
12    def _write_byte(self, value):
13        try:
14            self.bus.write_byte(self.address, value)
15        except Exception as e:
16            print ("Writing Error "+str(e))
17
18    def _write_i2c_block_data(self, reg, value):
19        try:
20            self.bus.write_i2c_block_data(self.address, reg, value)
21        except Exception as e:
22            print ("Writing Error "+str(e))
23
24    def _read_byte(self):
25        try:
26            received_data = self.bus.read_byte(self.address)
27        except Exception as e:
28            print ("Read Error "+str(e))
29            received_data = 0xab
30        return received_data
31
32    def _read_word(self):
33        try:
34            data = self.bus.read_word_data(self.address)
35        except Exception as e:
36            print ("Writing Error "+str(e))
37        return data
38
39    def _data_exchange(self, REG):
40        self._write_byte(REG)
41        time.sleep(0.1)
42        byte = self._read_byte()
43        return byte
44
45    def read_light_intensity(self):
46        lux = bh1750_sensor._data_exchange(self, BH1750_LUX)
47        return lux
48
49    def generate_json_data(self, rtc):
50        LUX = self.read_light_intensity()
51        RTC_DATA = rtc.generate_human_ts()
52
53        data = {
54            'lux': LUX,
55            'timestamp': RTC_DATA
56        }
57
58        json_data = json.dumps(data)
59        return json_data
```

Tra le funzioni all'interno della classe **bh1750_sensor**, di cui omettiamo quelle già presentate nel **BMP280.py**, troviamo:

- **read_light_intensity()**
 - o Funzione che passa a **_data_exchange()** il registro relativo all'illuminamento che si occuperà di restituirne il dato.
- **generate_json_data()**
 - o Funzione che genera e restituisce il json relativo al sensore, includendone il timestamp restituito dall'**RTC**.

Di seguito viene riportato il codice dell'**RTC.py**:

```
1 import smbus
2 import time
3 import json
4 import datetime
5 import struct
6 from datetime import datetime, timedelta
7
8 # RTC      = 0x13
9 # SYNC_TIME = 0x15
10
11 RTC = 0x45
12 SYNC_TIME = 0x46
13
14 reference_time = datetime.now()
15
16 class timestamp_sensor:
17     def __init__(self, bus, address):
18
19         def _write_byte(self, value):
20
21             def _write_i2c_block_data(self, reg, value):
22
23                 def _read_byte(self):
24
25                     def _read_timestamp(self): #OK
26                         try:
27                             timestamp = self.bus.read_i2c_block_data(self.address, RTC, 4)
28                         except Exception as e:
29                             print ("Read Error "+str(e))
30                         timestamp = False
31                     return timestamp
32
33                     def _convert_byte_to_human_ts(unix_ts): #OK
34                         epoch_timestamp = (unix_ts[0] << 24) + (unix_ts[1] << 16) + (unix_ts[2] << 8) + unix_ts[3]
35                         dt = datetime.fromtimestamp(epoch_timestamp)
36                         human_readable = dt.strftime('%Y-%m-%d %H:%M:%S')
37                     return human_readable
38
39                     def sync_datetime(self):
40                         timestamp = int(time.time())
41                         byte_list = timestamp.to_bytes(4, byteorder='big')
42                         byte_list = list(byte_list)
43
44                         self._write_i2c_block_data(SYNC_TIME, byte_list)
45                         time.sleep(0.1)
46
47                     def read_timestamp(self):
48                         self._write_byte(RTC) # Write command to read timestamp
49                         time.sleep(0.1) # Wait for the timestamp to be ready
50                         timestamp = self._read_timestamp() # Read the timestamp
51                         if(timestamp != False):
52                             return timestamp
53                     return timestamp
54
55                     def generate_human_ts(self):
56                         RTC_DATA = self.read_timestamp()
57                         if(RTC_DATA):
58                             ts = timestamp_sensor._convert_byte_to_human_ts(RTC_DATA)
59                         else:
60                             ts = False
61                         return ts
62
63                     def generate_json_data(self):
64                         ts = timestamp_sensor.generate_human_ts(self)
65                         if(ts == False):
66                             ts = "error"
67                         data = {
68                             'timestamp': ts
69                         }
70
71                         json_data = json.dumps(data)
72                         return json_data
```

Anche qui tra le funzioni all'interno della classe **timestamp_sensor** omettiamo quelle già presentate precedentemente:

- **_read_timestamp()**
 - o Funzione privata che richama la funzione di libreria per leggere su I2C0, poiché il **timestamp** è scritto su quattro byte, rispetto agli altri sensori, viene utilizzata la funzione di libreria **read_i2c_block_data()**.
- **_convert_byte_to_human_ts()**
 - o Funzione privata che si occupa di convertire il **timestamp** in formato **Unix** in una stringa comprensibile.
- **sync_datetime()**
 - o Funzione che legge il **timestamp** di sistema (**Diet-Pi**) e lo invia sull'**I2C0**.
- **read_timestamp()**
 - o Funzione che legge il **timestamp** ricevuto dalla **Pico** tramite l'**I2C0**.
- **generate_human_ts()**
 - o Funzione pubblica che restituisce il timestamp già convertito in un formato leggibile tramite l'ausilio del **_convert_byte_to_human_ts()**.
- **generate_json_data()**
 - o Funzione che genera e restituisce il json relativo al timestamp.

Applicazione web full stack

Il compito principale dell'applicazione web è quello di visualizzare i dati in tempo reale dei sensori provenienti dal broker MQTT pubblicati nei vari topic, visualizzare i dati storici mediante grafici a serie temporali e richiedere la sincronizzazione del modulo RTC tramite l'invio di un messaggio apposito sul topic "RTC_SYNC". Infine, sono presenti dei buttoni che permettono di richiedere in tempo reale, i dati dei sensori memorizzandoli nel database oltre che ad un ulteriore bottone per monitorare il timestamp impostato nel modulo RTC.

Come già accennato, l'applicazione web full stack è composta da tre elementi essenziali:

- **MongoDB**: database NOSQL che contiene i dati storici dei sensori;
- **Node JS**: web server che gestisce la connessione al database MongoDB e le operazioni di memorizzazione ed estrazione dei dati;
- **React JS**: framework che realizza la pagina web che visualizza i dati di interesse;

Mongo DB

Il database è composto da due **collections**:

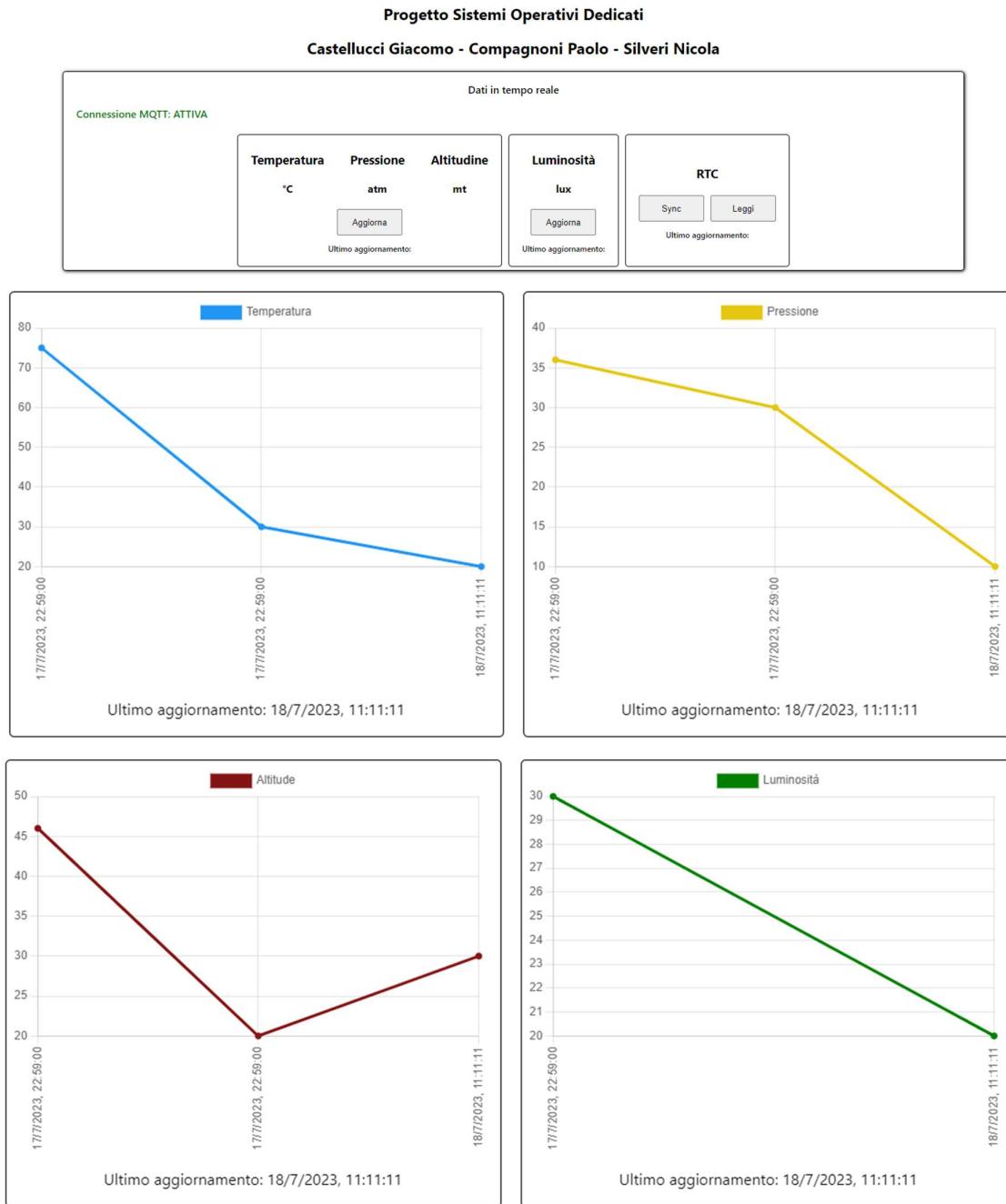
- **bh1750**: è una tabella composta da due campi (**timestamp** e **lux**). Il primo campo memorizza il timestamp fornito dall'RTC al momento della richiesta del dato e lux fornisce il valore di luminosità fornito dal BH1750;
- **bmp280**: è una tabella composta da quattro campi (**timestamp**, **temperature**, **pressure** e **altitude**). Il primo campo memorizza il timestamp fornito dall'RTC al momento della richiesta del dato, gli altri forniscono rispettivamente la temperatura, pressione e altitudine forniti dal BMP280.

Backend

Il server NodeJS è riassumibile in tre funzioni principali:

1. **Connessione con il database**: all'avvio del server tramite nodemon, il server effettua una connessione al database MongoDB remoto;
2. **Estrazione dei dati dalle collections**: ci sono due rotte dedicate (*getBMPData* e *getBHData*) che estraggono tutti i documenti della collection in formato JSON;
3. **Memorizzazione dei nuovi dati**: ci sono due rotte dedicate (*storeBHData* e *storeBMPData*) che memorizzano i nuovi dati all'interno delle collections per avere lo storico dei dati.

Frontend



Sopra è riportata la schermata principale del **sito web**. Come si può vedere, è suddiviso in **due aree principali**: la prima parte è dedicata ai **dati in tempo reale**, la seconda alla **visualizzazione dei dati storici**. Quando sui topic MQTT "BPM280" e "BH1750" vengono pubblicati dei messaggi, i valori dei singoli campi vengono mostrati nella pagina e successivamente essi vengono **memorizzati** nel database MongoDB ed infine viene **aggiornato** ogni singolo grafico corrispondente al nuovo dato.

Qui sotto, viene mostrata la visualizzazione dei dati in presenza di un messaggio proveniente dal topic “BMP280”.



Qui sotto, viene mostrata la visualizzazione dei dati in presenza di un nuovo messaggio proveniente dal topic “BH1750” sopraggiunto dopo il messaggio precedente.



Se viene premuto il bottone “aggiorna”, ReactJS invia una richiesta di un nuovo dato via MQTT, al topic “WEB_REQ” con il messaggio:

```
{  
  "sensor": "BMP280"  
}
```

oppure:

```
{  
  "sensor": "BH1750"  
}
```

in funzione del bottone premuto.

A seguito della richiesta sul topic “WEB_REQ”, la Raspberry PI ZERO invia sul topic “BMP280” i nuovi dati che vengono poi gestiti come già spiegato in precedenza.

Se viene premuto il bottone “Sync”, viene inviato un messaggio sul topic “WEB_REQ”:

```
{  
  "sensor": "RTC_SYNC"  
}
```

che ricevuto dalla Raspberry PI ZERO, va ad impostare il timestamp dell'RTC alla data corrente, restituendo ad operazione conclusa, un messaggio sul topic "RTC" così formato:

RTC sync successfull!

La pagina web mostra il messaggio per **5 secondi** in questo modo:



Se viene premuto il bottone "**Leggi**", viene mostrato a schermo il **timestamp attuale** impostato sull'RTC in questo modo:



Successivamente, viene mostrato il tempo attuale una volta effettuato il sync dell'RTC. Nell'esempio sotto, l'ultimo aggiornamento si riferisce al momento in cui è stato premuto il tasto "Sync". La GIF è stata registrata dieci dopo il sync dell'RTC.

