

# UNIVERSITA' DEGLI STUDI DI TOR VERGATA



## **Facoltà di Ingegneria**

Corso di Laurea Magistrale in Ing. Medica

Corso di Sanità Digitale

### ***Relazione progetto:***

### ***Sistema di monitoraggio BPM indossabile***

ELABORATO SCRITTO

Francesca De Angelis

Francesca Romana Mannarino

Sara Natalini

Elisa Simonelli

## Sommario

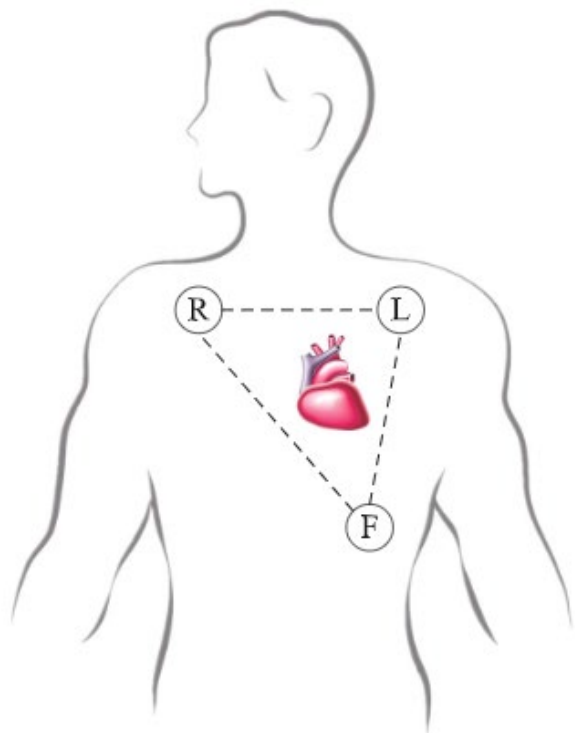
1. Introduzione .....	3
2. Macchina a Stati Finiti Device .....	8
3. Macchina a Stati Finiti Server e Interfaccia Utente.....	11
4. Conclusioni .....	14
APPENDICE .....	15

## 1. Introduzione

Il progetto è volto alla realizzazione di un sistema di rilevazione della frequenza cardiaca indossabile per il monitoraggio delle condizioni fisiche di una persona da remoto, visualizzabili poi attraverso un'interfaccia grafica.

Il sistema è stato strutturato in modo che i dati, prelevati da un sensore AD8232, vengano inviati in maniera wireless attraverso un modulo ESP32 ad un sistema di ricezione ed elaborazione, che chiameremo nel corso della presentazione server Python. Il server Python dopo aver ricevuto ed analizzato i dati li invia poi ad un'interfaccia grafica.

Solitamente in sistemi medicali vengono utilizzati 12 elettrodi per la cattura di segnali ECG, tuttavia l'utilizzo di troppi elettrodi potrebbe portare ad effetti negativi sulla portabilità del sistema e compatibilità con il paziente. Il sensore che abbiamo utilizzato è dotato di soli tre elettrodi che sono sufficienti per la cattura delle differenze di potenziale di cui abbiamo bisogno per il nostro progetto. Gli elettrodi sono posizionati coerentemente alle derivazioni principali per l'acquisizione delle differenze di potenziale.



*Figura 1. Posizionamento elettrodi*

I segnali ECG sono costituiti principalmente da cinque tipi di onde, ovvero l'onda P, l'onda T, l'onda Q, l'onda R e l'onda S, come illustrato in Figura 2:

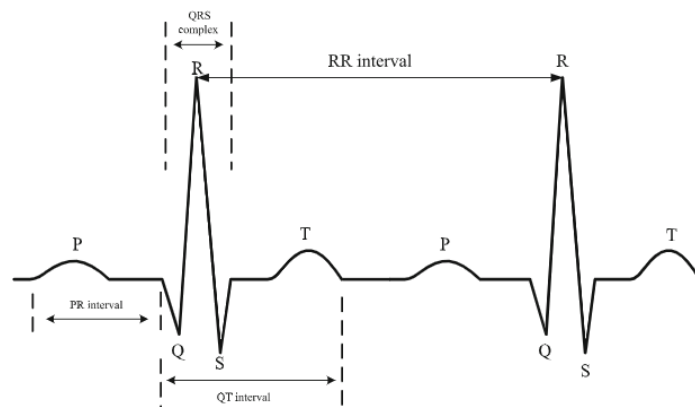


Figura 2. Segnale ECG

Gli intervalli di queste onde sono solitamente usati per diagnosticare una varietà di malattie cardiache. Tra tutte le caratteristiche di queste onde, quattro sono più comunemente utilizzate nella diagnosi medica, ad esempio:

- intervallo RR: solitamente è usato per identificare il periodo del segnale ECG. Indica il periodo che intercorre tra due onde R, che può essere irregolare in caso di disturbi cardiaci come l'aritmia. Inoltre, è utilizzato per il calcolo della frequenza cardiaca.
- intervallo PR: misura il tempo che intercorre tra l'onda P e il complesso QRS. Indica il tempo impiegato dall'impulso per raggiungere i ventricoli dal nodo del seno.
- intervallo QT: rappresenta il tempo tra l'inizio dell'onda Q e la fine dell'onda T, che è relativo alla depolarizzazione e ripolarizzazione ventricolare. Nel caso in cui l'intervallo QT sia oltre il valore normale vi è rischio di fibrillazione ventricolare.
- complesso QRS: è principalmente associato alla depolarizzazione ventricolare che consiste in 3 onde importanti, Q, R ed S. Analizzando la morfologia e la durata del complesso QRS, si possono rilevare malattia, oppure squilibrio elettrolitico o tossicità da farmaci.

Per questioni di praticità con l'interfaccia grafica e il device abbiamo deciso di prendere solo il battito cardiaco piuttosto che plottare l'intero elettrocardiogramma.

L'architettura dell'intero sistema consta di un nodo di monitoraggio, un programma MicroPython per prelevamento ed invio dati, un server Python per elaborazione dei dati e una GUI (interfaccia grafica) per la visualizzazione della frequenza cardiaca.

Il nodo di monitoraggio è responsabile della raccolta delle differenze di potenziale dei biosegnali e dell'invio di questi dati tramite un canale wireless. Infatti, il nodo di monitoraggio include il sensore AD8232 e il modulo Wi-Fi ESP32.

Il sensore AD8232 è alla base del nodo di monitoraggio, responsabile della raccolta dei biosegnali dal corpo umano ed è un sensore progettato per estrarre, amplificare e filtrare i biopotenziali.

Poiché la frequenza tipica del segnale ECG è compresa tra 0.5 Hz e 100 Hz, nell'AD8232, viene utilizzato un filtro passa-banda per rimuovere il rumore. Successivamente, il segnale filtrato viene amplificato utilizzando un amplificatore operazionale. Infine, questo modulo raccoglie i segnali ECG da 0 V a 3.3 V.

Più precisamente i pin presenti sul sensore sono: *SDN*, *LO+*, *LO-*, *3.3 V*, *GND*, *OUTPUT*. Il *3.3 V* è collegato al pin di alimentazione 3v3 dell'ESP32 (alimentato a sua volta attraverso cavo USB collegato al computer), il *GND* è collegato alla massa dell'ESP32, *LO+* è collegato al pin GPIO 9, *LO-* al pin GPIO10 e l'*OUTPUT* è collegato al pin 36 ADC dell'ESP32.

Il segnale di output viene inviato ad un pin *ADC*, un convertitore analogico-digitale, che converte un segnale tempo-continuo (una tensione) in una serie di valori tempo-discreti. I pin *LO+* e *LO-* rilevano la disconnessione, rispettivamente dell'elettrodo *L (Left)* e *R (Right)*, che sono a livello alto quando gli elettrodi sono disconnessi e a livello basso quando invece sono connessi. Invece, il pin *SDN* è un pin di spegnimento che si utilizza nelle applicazioni dove è necessario un consumo energetico, nel nostro progetto non viene collegato perché non utilizzato.

Il modulo Wi-Fi che è stato utilizzato per trasmettere i dati in tempo reale al programma in MicroPython è l'ESP32 che si occupa di inviare i dati al server Python.

In generale l'ESP32 è una serie di microcontrollori SoC (System-on-Chip) a basso costo e a basso consumo con Wi-Fi e Bluetooth dual-mode integrati. Il suo basso consumo energetico, i molteplici ambienti di sviluppo open-source e le librerie a disposizione li rendono perfettamente idonei per qualsiasi tipo di sviluppatore. Abbiamo utilizzato il modulo ESP32 in Station Mode, ovvero si collega ad una rete *Wi-Fi* esistente. In questo scenario il router si comporta come Access Point e l'ESP32 come Station. Ovviamente occorre connettere l'ESP32 al router (rete locale).

Questo modulo ottiene i dati dal sensore e fornisce un accesso veloce e conveniente a Internet, inoltre è in grado di trasmettere i dati in tempo reale. Per far sì che il segnale analogico poi fosse ricostruito in maniera corretta al momento dell'elaborazione abbiamo fatto in modo che i dati venissero inviati con un timing preciso realizzando un programma MicroPython che, servendosi di *timeout*, inviasse i dati con la stessa cadenza, ossia che risultasse un tempo costante tra l'invio di un dato e il successivo. Una volta appurato il *rate* a cui i nostri dati potevano essere mandati (0,0250 msg/sec), ovvero ad una frequenza di 40 Hz, abbiamo definito e impostato il *timeout* in modo tale che il dispositivo si svegliasse e inviasse dati ogni 25000 microsecondi.

Per affrontare questo progetto abbiamo dovuto effettuare una serie di scelte come team: il linguaggio da utilizzare, il protocollo di comunicazione, una struttura comune dei messaggi.

Dal momento che abbiamo scelto di sviluppare la nostra applicazione in Python, per uniformità di linguaggio abbiamo deciso di usare MicroPython per programmare il microcontrollore. Si carica MicroPython sull'esp32 e poi si sviluppano script Python per la propria applicazione. Python è un linguaggio di programmazione moderno, con una sintassi chiara e si caratterizza per la sua facilità di scrittura e lettura. Ha lo svantaggio però di essere stato progettato per essere eseguito su PC e macchine grandi con molta RAM, non è stato concepito come linguaggio di programmazione embedded (integrato). Per questo scopo è stato appunto creato un'implementazione agile ed efficiente del linguaggio di programmazione Python che è appunto MicroPython, adatta appositamente per l'esecuzione con le risorse limitate di un microcontrollore.

Abbiamo scelto di utilizzare il protocollo di comunicazione MQTT (*Message Queue Telemetry Transport*). Tale protocollo adotta un meccanismo di pubblicazione e sottoscrizione per scambiare messaggi tramite un apposito *message broker*: i mittenti si rivolgono al broker per pubblicare i messaggi su un topic. Ogni destinatario si iscrive agli argomenti che lo interessano e, ogni volta che un nuovo messaggio viene pubblicato su quel determinato topic, il message broker lo distribuisce a tutti i destinatari. La scelta di prediligere questo tipo di protocollo è stata effettuata guardando ai punti di forza dell'MQTT che sono la leggerezza, la flessibilità e l'affidabilità del protocollo, che oggi risulta essere largamente impiegato per l'Internet of Things.

In seguito, abbiamo stabilito una struttura comune dei messaggi da inviare e due topic su cui mandare e ricevere gli stessi.

Riportiamo di seguito la forma e il formato dei messaggi che verranno scambiati tra il device e il server:

*Messaggi di register e register\_ack:*

```
Register: {  
    "msg": "register",  
    "device_name": "esp32"  
}
```

```
Register_ack: {  
    "msg": "ack_register",  
    "time": "234563456"  
}
```

*Messaggi di stop e ack\_stop:*

```
Stop: {  
    "msg": "stop"  
}
```

```
Ack_Stop: {  
    "msg": "ack_stop"  
}
```

*Messaggio di start:*

```
Start: {  
    "msg": "start"  
}
```

*Messaggio di invio dati:*

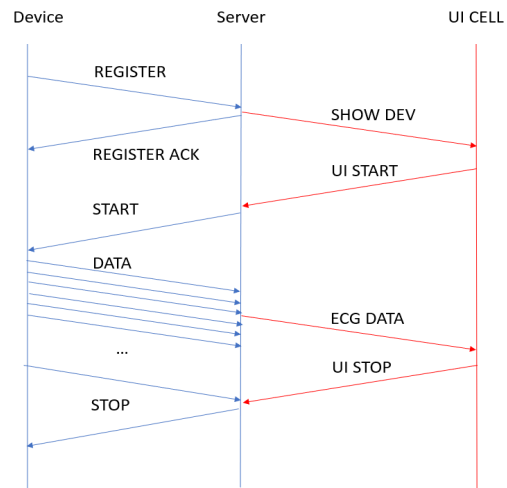
```
Data: {  
    "msg": "data",  
    "msg-id": cnt+1,  
    "time": "35436754",  
    "value": "3456",  
    "now": "1234444"  
}
```

Il formato scelto per i messaggi è il JSON, perché è un formato testuale per strutturare dati in una forma precisa e standardizzata, particolarmente adatto per lo scambio degli stessi. Per le persone è facile da leggere e scrivere, per le macchine è semplice da generare e analizzarne la sintassi. Per quando riguarda i messaggi, abbiamo deciso di realizzare una struttura semplice con le informazioni che abbiamo ritenuto necessarie. Nel messaggio *ack\_register*, nel campo "time", è contenuto l'orario del server al momento dell'invio del messaggio, in epoch time. Quest'informazione è stata utile perché, dopo la ricezione dello stesso da parte del device, l'abbiamo potuta utilizzare per sincronizzare l'orario dell'ESP32 con quello del server dal momento che il dispositivo ha un proprio orario interno.

Un'importante precisazione va fatta riguardo il messaggio *Data*: in esso sono presenti due campi contenenti due tempi diversi. Il campo *now* contiene il tempo esatto in cui il messaggio è stato mandato, espresso in secondi, in Unix time (o epoch time), valore ottenuto proprio grazie all'operazione di sincronizzazione sopracitata. Il campo "time" contiene, invece, espresso in microsecondi, l'istante in cui avviene il sampling.

Per quanto riguarda i topic, abbiamo scelto di utilizzarne due per non avere errori nell'invio e nella ricezione. Un primo topic: *sd/esp32/ecg* a cui il device si sottoscrive e su cui il server Python pubblica; un secondo topic: *esp32/ecg* su cui il device pubblica e a cui il server Python si sottoscrive.

Complessivamente, possiamo riassumere il protocollo progettato come è mostrato in figura 3:



*Figura 3. Progettazione protocollo*

Analizzeremo i vari aspetti di tale protocollo, nel dettaglio, di seguito.

## 2. Macchina a Stati Finiti Device

La prima parte del progetto che andiamo a descrivere riguarda l'interfaccia tra il device e il server Python; ci siamo occupate, in particolare, dell'acquisizione dei dati dall' AD8232 e l'invio degli stessi, quando richiesto, al server tramite l'esp32. In questa prima fase del progetto abbiamo lavorato in Micropython. Abbiamo usato il modello della macchina a stati finiti (FSM) per descrivere il comportamento del nostro sistema. In Figura 4 riportiamo la rappresentazione grafica della macchina a stati finiti che abbiamo progettato per definire le operazioni del device:

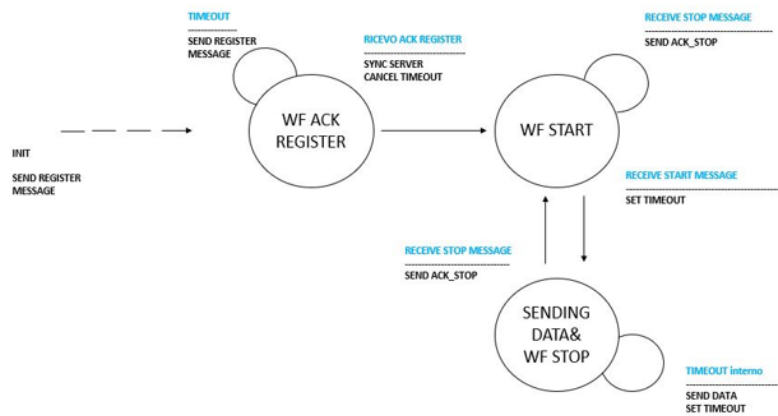


Figura 4. Macchina a Stati Finiti Device

E' bene precisare che abbiamo scelto di lasciare fuori dalla macchina a stati le operazioni di connessione dell'ESP32 alla rete Wi-Fi, la sottoscrizione al topic stabilito e l'invio del *register message*, con cui il device comunica al server che è pronto a lavorare. Nel programma queste operazioni saranno le prime ad essere eseguite. La punta della freccia tratteggiata indica lo stato iniziale della macchina a stati. Gli stati presenti sono tre.

Nel primo stato, *WF ACK REGISTER* il device è in attesa di ricevere l'*ack\_register*, da parte del server Python, a conferma che il messaggio di registrazione sia effettivamente stato letto dal server. Qualora tale messaggio non arrivi, trascorso un intervallo da noi fissato a 10 secondi, il device manderà un altro *register\_message*. Nel caso, invece, di ricezione dell'*ack\_register*, il programma azzererà il *timeout*, eseguirà l'operazione di sincronizzazione con l'orario del server e passerà nello stato di attesa dello *start\_message* (*WF START MESSAGE*). Abbiamo deciso di introdurre questo meccanismo di controllo perché il messaggio *register\_message* non deve essere perso.

Nello stato *WF START* il dispositivo è in attesa dello *start\_message* da parte del server. Qualora arrivi tale messaggio verrà impostato un nuovo *timeout* e si avrà una transizione verso lo stato *SENDING DATA & WF STOP*; in caso contrario, invece, il device resterà nello stato in cui già si trovava.

Nello stato *SENDING DATA & WF STOP* è sempre lo scadere di un *timeout* fissato da noi a rappresentare l'evento a seguito del quale avviene l'invio di un messaggio sul topic previsto. Ogni volta che scade tale *timeout*, viene mandato un messaggio. In questo stato è possibile anche ascoltare messaggi in arrivo. L'arrivo di un messaggio di stop, infatti, porta all'invio di un messaggio di *ack\_stop* e al ritorno nello stato *WF START*.

Da notare che il *timeout* che gestisce l'eventuale invio di un secondo *register\_message* è stato fissato, come già detto a 10 secondi, ragionevolmente con il fatto che ci mettiamo in ascolto della conferma di ricezione dello stesso da parte del server. La necessità di mandare i dati raccolti dal sensore molto velocemente, invece, ci ha portato ad impostare un secondo *timeout* diverso dal precedente e molto più breve.



Nello stato *WF START*, inoltre, l'arrivo di un messaggio di stop, indice del fatto che al server Python non è arrivato il messaggio di avvenuta ricezione dello stop da parte del device, provocherà un nuovo invio di un *ack\_stop*.

La FSM è stata testata in tutti i suoi stati. Abbiamo realizzato una serie di prove per essere certe che la macchina a stati funzionasse come previsto sia durante l'invio e ricezione di messaggi standard che di messaggi non attesi.

Per effettuare i test e le operazioni di debug ci siamo serviti di *MQTT explorer*. Quest'applicazione si iscrive a tutti i topic del server MQTT permettendoci di visualizzare, in maniera strutturata tutto ciò che succede sul broker. Ci consente, inoltre, di esplorare le code dei messaggi e di pubblicare sui topic.

In *Appendice* possiamo vedere le immagini delle varie prove effettuate.

Ci siamo, poi, occupate del calcolo del rate a cui poter inviare messaggi, rispettando un tempo di invio messaggi preciso al microsecondo.

Attraverso la variabile *ECG\_SAMPLE\_TIME\_US* (vedere *Allegato*) è possibile stabilire ogni quanto il nostro dispositivo deve attivarsi per mandare un messaggio.

Nel programma abbiamo inserito delle righe di codice grazie alle quali ci viene segnalata la presenza di un errore che ci avverte se il sistema non riesce a mantenere la regolarità nel campionamento e, a seguito di tale segnalazione, effettua una compensazione che ci consente di campionare sempre con lo stesso periodo ma saltando quel sample.

La scelta del rate ha dovuto tenere in considerazione due necessità:

- il bisogno di scegliere un rate per cui venga rispettato un timing preciso al microsecondo e che ci restituisca una quantità da noi accettabile di messaggi di errore;
- il bisogno di far arrivare al server Python un numero sufficiente di messaggi in una finestra da noi scelta di 10 secondi, per realizzare una corretta ed accurata elaborazione

Abbiamo, inoltre, realizzato un test, il cui codice in *Allegato* ("*test\_subscribe.py*"), che riceve i messaggi dal device, conta quanti ne sono arrivati e in quanto tempo. Il programma di test calcola, inoltre, il rate medio come

$$R = \frac{\text{Numero messaggi trasmessi}}{\text{Tempo}}.$$

Per prima cosa abbiamo calcolato il massimo rate a cui il sistema può mandare dati, senza impostare alcun *timeout*.

Abbiamo atteso un tempo di 64 secondi, trovando un  $R=0,0125$  messaggi/secondo cioè il dispositivo può mandare dati ad una frequenza di 80 Hz. La mancanza di un *timeout*, però, non ci consente di avere un sample costante, il che, per la nostra applicazione, rende quest'impostazione inutilizzabile. E' possibile osservare tale prova in figura 5:



```
Run: test_subscribe x
5098
received message = {"value": 0, "note": "", "now": 1611744075, "dev_type": "esp32", "time": 289977211, "msg": "data", "msg-id": 16767}
5099
received message = {"msg": "ack_stop"}
difference_msg_id:5098
difference_time:64
cnt:5098
time/packets:0.0125595537465673
difference_time_2:64.0286133
cnt_r:0
```

Figura 5. Test per il rate massimo

Abbiamo sfruttato il risultato di questo test come valore di rate da cui partire per le successive prove.

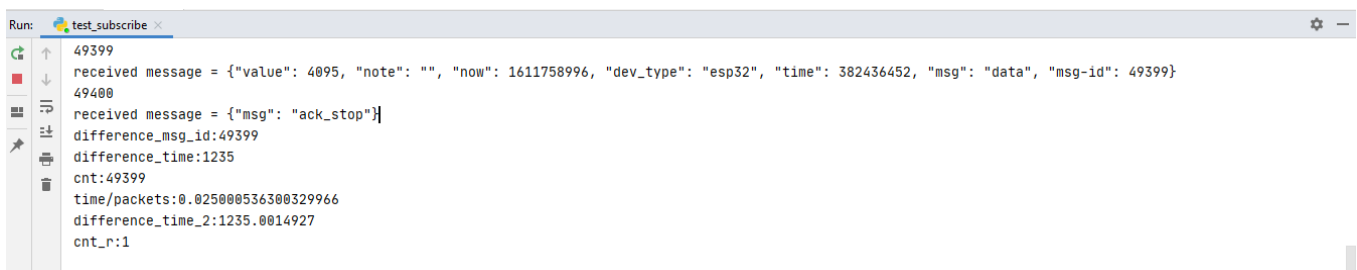
Abbiamo effettuato una prima prova (Figura 6) per un tempo di 1219 secondi, impostando la variabile *ECG\_SAMPLE\_TIME\_US* pari a 15000 microsecondi. Il programma ha calcolato un  $R = 0,0152$  messaggi/secondo e sono stati segnalati più errori di quelli che abbiamo ritenuto accettabili.



```
79948
received message = {"value": 0, "note": "", "now": 1611758964, "dev_type": "esp32", "time": 437667192, "msg": "data", "msg-id": 79948}
79949
received message = {"msg": "ack_stop"}
difference_msg_id:79948
difference_time:1219
cnt:79948
time/packets:0.015247379122679743
difference_time_2:1218.9974661
cnt_r:1219
```

Figura6. Test *ECG\_SAMPLE\_TIME\_US* = 1500 us

Abbiamo deciso di aumentare il tempo di arrivo tra un pacchetto e l'altro per cercare di ottenere un risultato soddisfacente. Un'ulteriore prova (Figura 7), effettuata per 1235 secondi, infatti, è stata eseguita impostando un *ECG\_SAMPLE\_TIME\_US* pari a 25000 microsecondi, trovando un  $R = 0,0250$  messaggi/secondo e una sola segnalazione di errore, che abbiamo ritenuto ammissibile.



```
Run: test_subscribe x
49399
received message = {"value": 4095, "note": "", "now": 1611758996, "dev_type": "esp32", "time": 382436452, "msg": "data", "msg-id": 49399}
49400
received message = {"msg": "ack_stop"}
difference_msg_id:49399
difference_time:1235
cnt:49399
time/packets:0.025000536300329966
difference_time_2:1235.0014927
cnt_r:1
```

Figura7. Test *ECG\_SAMPLE\_TIME\_US* = 25000 us

Abbiamo scelto di impostare questo rate che soddisfa entrambe le necessità sopracitate.

### 3. Macchina a Stati Finiti Server e Interfaccia Utente

La seconda parte del progetto è stata creare un server Python con la funzione di prendere i dati (misurati dal sensore), elaborarli ed inviarli ad una interfaccia utente (<https://io.adafruit.com/>) sotto richiesta. Il server riceve tali dati poiché sottoscritto al topic MQTT dove vengono pubblicati i messaggi da parte del dispositivo.

Ricordiamo l'architettura del protocollo riportata in Figura 3: dal lato device, il server implementa l'invio del *register ack*, dello *start* e del *ack\_stop*. Dal lato dell'interfaccia utente abbiamo implementato l'*indicatore* di presenza del dispositivo, la ricezione dello *start*, l'invio del dato *BPM* e la ricezione dello *stop*.

I dati raccolti dal server vengono successivamente elaborati (tramite operazioni di filtraggio e individuazione del complesso QRS) e viene inviata una media tra i valori di Heart Rate all'interfaccia utente ogni 10 secondi, tramite l'utilizzo di una finestra mobile. Infine, il server pubblica sui feeds relativi ai *blocchi* costruiti nell'interfaccia adafruit la disponibilità del device (indicatore verde) ed i valori di Heart Rate; mentre ciò che riceve dall'interfaccia utente sono i messaggi di start o stop, che verranno comunicati al device.

Abbiamo pensato l'architettura del server, analogamente a quella del device, come una Finite State Machine (FSM), ovvero gestiremo gli eventi come degli stati che il server assumerà ed in relazione ai quali avverranno determinati processi.

È possibile dividere il codice in più blocchi, a seconda del loro significato: un primo blocco dedicato alla definizione delle funzioni di *init*, queste sono per la connessione e la sottoscrizione al device e all'interfaccia; una seconda parte dove si definiscono i messaggi JSON da inviare, come concordato in precedenza; una terza parte di definizione delle funzioni da utilizzare per la pubblicazione sia sul device che sulla *GUI*; la macchina a stati che rappresenta tutti i vari eventi che verranno gestiti; il *wait for event* che gestisce il processo della macchina a stati. Infine, vengono chiamate le funzioni di *init* e azzerati i *timeout* per poi concludere con un *While True* che richiama la macchina a stati (righe complessive circa 300, consultabile in *Allegato*).

In particolare, possiamo analizzare gli stati della FSM che sono 4, denominati: *WF DEVICE*, *WF UI START*, *WF DATA*, *WF ACK STOP*, come mostrato in Figura 8.

Nello specifico: nello stato *WF DEVICE* il server rileva, tramite il messaggio di *register*, la presenza del dispositivo ESP32. Una volta ricevuto tale messaggio, invia un *register ack* e un tempo al device per farlo sincronizzare, mentre l'interfaccia grafica mostra la connessione del dispositivo tramite un indicatore; nello stato *WF UI START* il server aspetta lo *start* dall'interfaccia e comunica al device che è pronto a ricevere i dati; nello stato *WF DATA* il server raccoglie e, tramite la finestra mobile, elabora i dati ricevuti, inviandoli ogni 10 secondi all'interfaccia, usufruendo di un primo *timeout* (*timeout\_ecg*) interno. Qualora il server ricevesse lo *stop* dall'interfaccia, lo invierà al device e si avrà una transizione nello stato *WF ACK STOP*: tramite un secondo *timeout* (*timeout\_ack\_stop*) interno il server si assicurerà che il device abbia effettivamente ricevuto lo stop. Successivamente alla ricezione dell'*ack stop* la macchina a stati opererà una transizione nello stato *WF UI START*, in attesa di un nuovo start da parte dell'interfaccia utente.

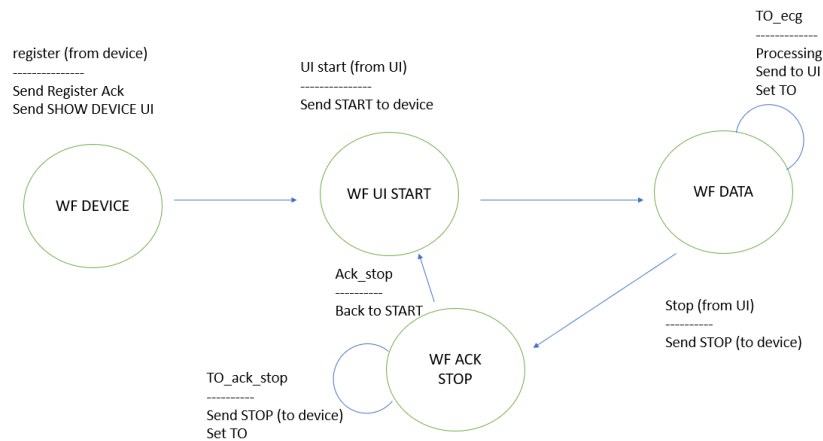


Figura 8. Macchina a Stati Finiti server Python

Per poter calcolare la frequenza cardiaca è necessario trovare i picchi R del complesso QRS, una volta trovati i picchi delle onde R e avendo il tempo in secondi a disposizione, è possibile calcolare tutto ciò di cui abbiamo bisogno: un array di valori di frequenza cardiaca e una frequenza cardiaca media per poterla comunicare all'interfaccia utente. Più nel dettaglio, l'elaborazione inserita nella Macchina a Stati Finiti prevede l'acquisizione dei dati in real time tramite l'uso di una finestra mobile. Una volta osservato che all'elaborazione fosse necessario l'utilizzo di 1000 dati in real time, abbiamo implementato una finestra temporale di lunghezza standard ma che si sposti gradualmente sui dati successivi eliminando i primi già elaborati. Nel dettaglio: dal messaggio *Data* precedentemente descritto, per l'elaborazione vengono estrapolati i campi "time" e "value", i quali vengono *appesi* in due array distinti e, successivamente, inseriti in una struttura dati di tipo *dataframe*. L'elaborazione, avendo bisogno di un tempo in secondi, converte l'array *time* da microsecondi a secondi. L'array *value*, invece, subisce delle trasformazioni più interessanti: il valore che ci arriva dal device è un segnale rumoroso, quindi, deve essere processato attraverso l'uso di alcune funzioni che sono state importate da una libreria esterna. Queste tre funzioni filtrano il segnale e ne ricavano i picchi R di cui abbiamo bisogno per il calcolo della frequenza cardiaca.

In particolare, la prima funzione richiede i dati *value* e *time* come ingresso e restituisce la derivata del segnale. La seconda funzione rileva automaticamente i picchi che si trovano oltre una determinata soglia e con una distanza minima l'uno dall'altro, la funzione ha bisogno in input del segnale derivato trovato precedentemente, del tempo, di una frazione per la soglia in altezza (0.5) e di una frazione per la soglia di distanza (0.6). In questo modo si evidenziano solo i picchi R e non tutti i picchi presenti nel segnale ECG. Per correttezza dell'elaborazione è possibile scegliere la frazione di soglia in altezza e di distanza poiché dipende da persona a persona, inoltre, i segnali ECG possono essere registrati con differenti ampiezze e rumori. Tuttavia, le soglie impostate nel nostro progetto sembrano essere ottimali per il device che abbiamo a disposizione. Nel caso in cui si presentassero errori nell'elaborazione, è qui che si andrebbe ad intervenire per cambiare lo spettro di ampiezza e rumore.

La terza ed ultima funzione trova i picchi dell'onda R, del complesso QRS: in ingresso si avrà il segnale derivato, i picchi precedentemente trovati e il tempo.

Infine, è possibile calcolare la frequenza cardiaca utilizzando un'espressione matematica che calcola la differenza di tempo tra i picchi dell'onda R, la inverte e la moltiplica per una costante.

Per inviarla all'interfaccia con un timing ben preciso è stato previsto di calcolare la media dei valori di frequenza cardiaca allo scadere del *timeout\_ecg*, introdotto proprio per dare il tempo al server di fare un'elaborazione precisa e puntuale.

Abbiamo testato il codice nelle varie transizioni di stato e per valutare quanti dati fossero necessari all'elaborazione, tramite l'utilizzo di un test. Quest'ultimo, riportato in *Allegato (test\_elaborazione.py)*, invia circa 11000 valori in un minuto prelevandoli da un file .txt contenente i valori necessari all'elaborazione,

precedentemente raccolti. Così facendo abbiamo potuto testare la ricezione, l'elaborazione e l'invio del dato all'interfaccia utente. Da queste prove abbiamo notato che, per ottenere un'elaborazione precisa e avere un risultato ottimale ed aderente alla realtà, necessitiamo di una finestra temporale di circa mille valori.

Per quanto riguarda l'interfaccia utente abbiamo creato, tramite il sito <https://io.adafruit.com/>, una dashboard nella quale abbiamo inserito 3 diversi blocchi: un *indicatore* di colore rosso o verde, un *toggle* con START/STOP ed un *gauge* che rappresenta valori da 0 a circa 150.

In particolare: l'*indicatore* assume il colore verde quando riceve dal server messaggi con valori  $\leq 2$  e rosso altrimenti; il *gauge* l'abbiamo impostato di colore verde per valori di HR compresi tra 40 e 120, mentre rosso per valori superiori o inferiori ad essi, ad indicare una situazione potenzialmente patologica; inoltre, ogni volta che viene inviato lo stop dall'utente il *gauge* si azzerava. Possiamo vedere l'interfaccia utente in Figura 9, nella configurazione di device disponibile ma in stop.

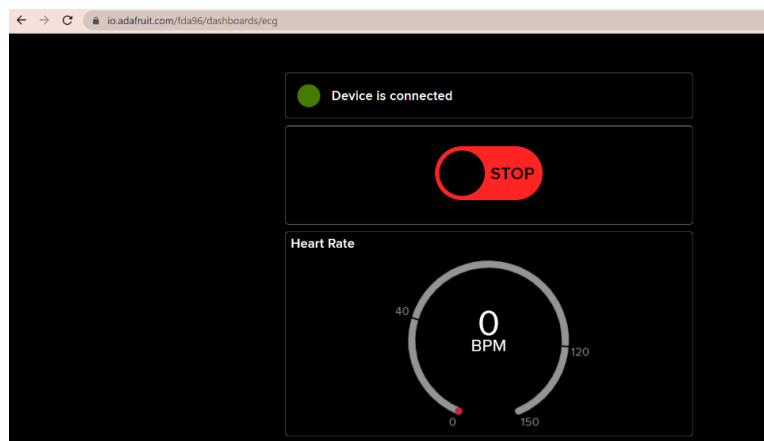


Figura 9. Interfaccia utente.

## 4. Conclusioni

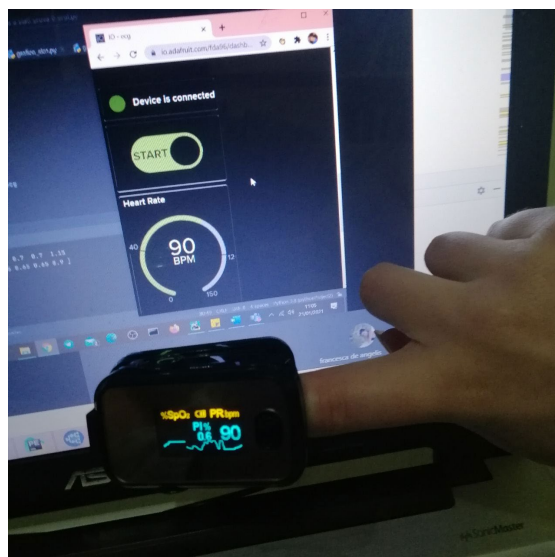
Abbiamo testato il sistema nel suo complesso, cercando di risolvere eventuali problematiche che si sono presentate di volta in volta.

In particolare, abbiamo verificato che il numero dei messaggi e il rate fossero sufficienti per l'elaborazione dei dati che restituiscono il valore del battito cardiaco. Inoltre, abbiamo testato che la finestra mobile funzionasse correttamente con mille dati.

Riportiamo di seguito il corretto funzionamento dell'intero sistema, verificato con l'utilizzo di un pulsossimetro. Precisiamo che il sistema opera una media tra i vari valori di frequenza cardiaca che può, quindi, discostarsi dal valore rilevato dal saturimetro.



*Figura 10.a Prima prova generale*



*Figura 10.b Seconda prova generale*

In conclusione, riteniamo che il sistema realizzato incontri i requisiti che ci eravamo prefissate nella definizione teorica dello stesso. Possiamo, inoltre, aggiungere che il sistema potrebbe essere integrato con altre funzioni di monitoraggio, quali temperatura, elettrocardiogramma e saturazione. Potrebbe essere integrato anche con altri tipi di dispositivi.

## APPENDICE

Riportiamo di seguito sia il corretto funzionamento della macchina a stati del device che i possibili casi di errore.

Entrata nella macchina a stati, nel primo stato “WF ACK REGISTER” e segnalazione di errore a seguito della ricezione di un messaggio non previsto.

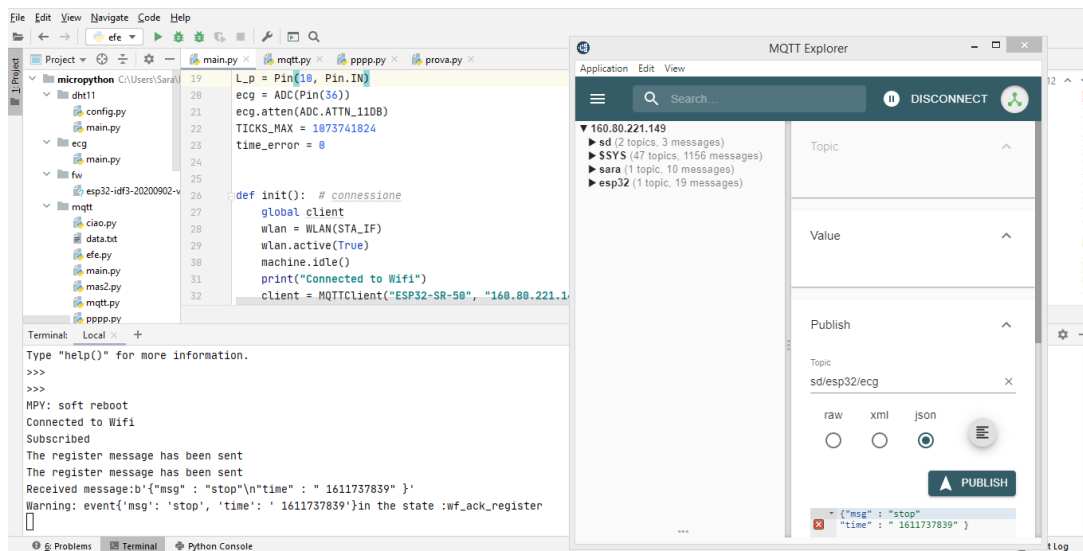


Figura 11. Stato WF ACK REGISTER e ricezione messaggio non previsto

Transizione nello stato (WF\_START) dopo la ricezione del messaggio “ack\_register”, e segnalazione di errore a seguito dell’invio di un messaggio non atteso. La macchina a stati continua a rimanere nello stato di attesa dello start:

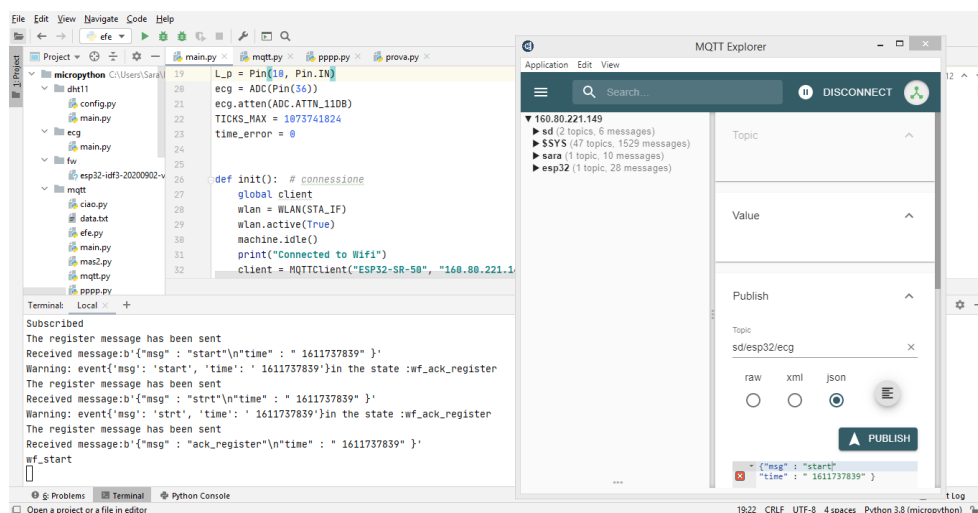


Figura 12. Transizione in WF\_START e ricezione messaggio non previsto

Transizione allo stato “SENDING DATA AND WF STOP” dopo la ricezione del corretto messaggio di start e segnalazione di errore a seguito di un messaggio non previsto nel suddetto stato. La macchina a stati continua a rimanere nello stato “SENDING DATA AND WF STOP”:

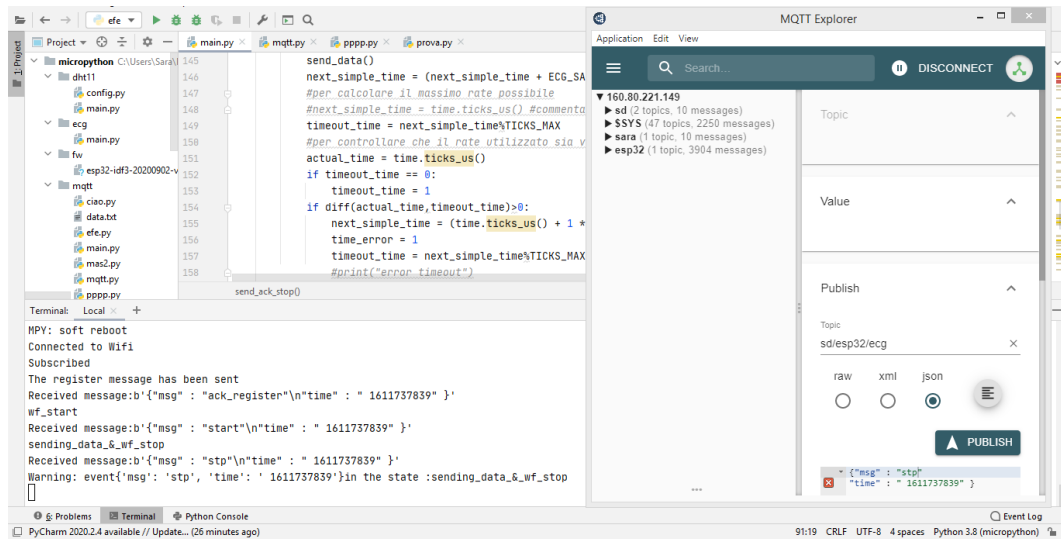


Figura 13. Transizione in SENDING DATA & WF STOP e ricezione messaggio non previsto

Transizione dallo stato “SENDING DATA AND WF STOP” allo stato “WF START” a seguito del messaggio di stop

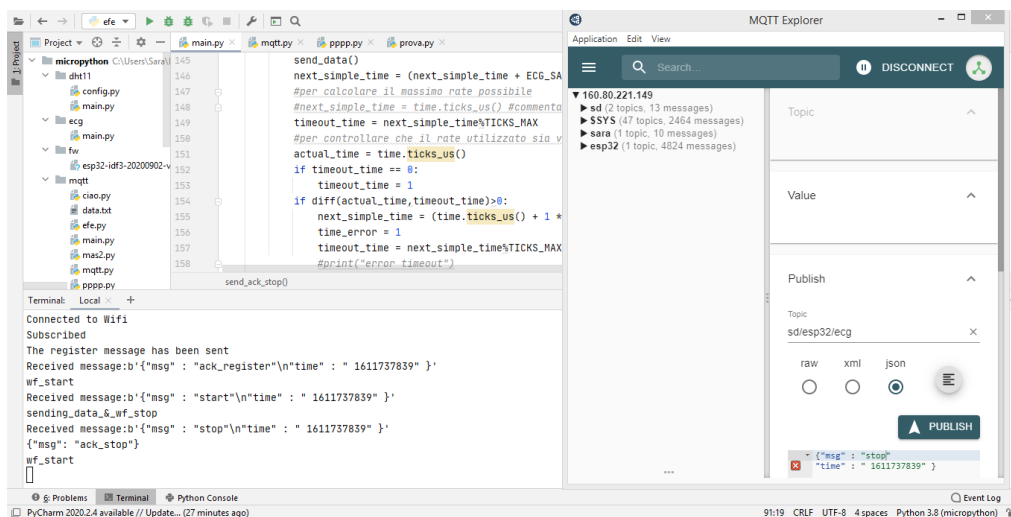


Figura 14. Transizione in WF START a seguito dello stop

Invio di un altro “ack\_stop”, nello stato “WF START” in seguito ad un’ulteriore ricezione di messaggio di stop:



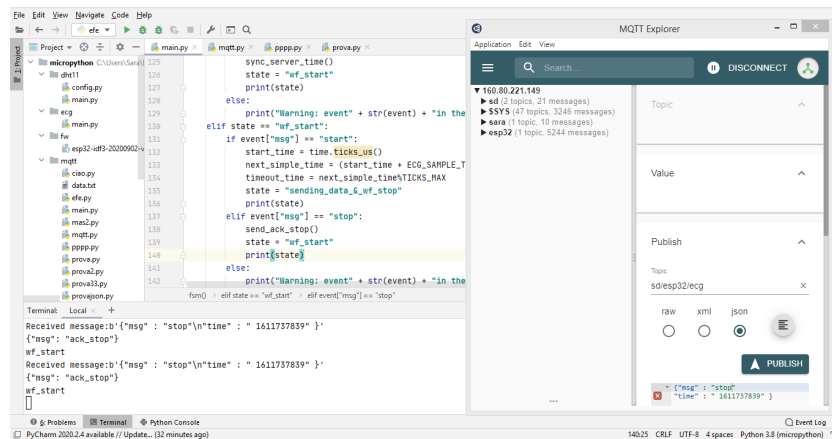


Figura 15. Invio di un secondo ack stop

Riportiamo di seguito sia il corretto funzionamento del server che i possibili casi di errore:

in Figura 16 vediamo l’inizializzazione del server ovvero la connessione al device, all’interfaccia utente e la sottoscrizione ai relativi topic. Lo stato in cui si trova il server è il *WF DEVICE*, ovvero è in attesa che il dispositivo

chieda

di

registrarsi.

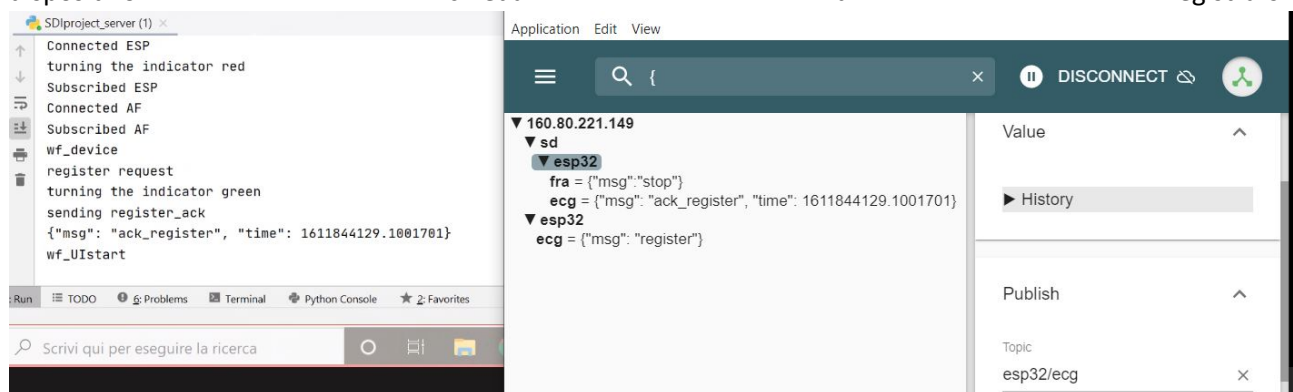


Figura 16. Inizializzazione server

In Figura 17 si può osservare la richiesta di registrazione da parte del device, l’invio dell’ack da parte del server e il passaggio di stato in *WF UI START*, ovvero l’attesa dello *start* inviato dall’utente.

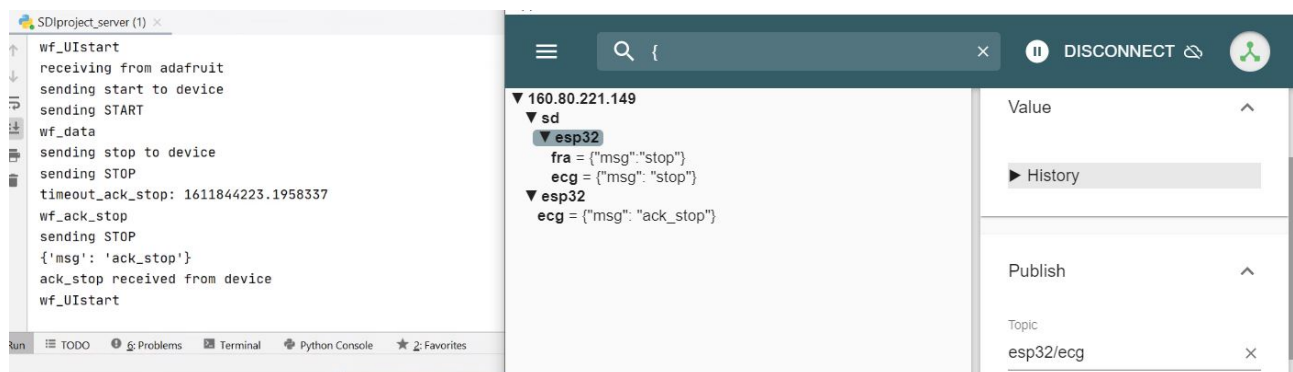


Figura 17. Passaggio di stato in WF\_UIstart

In Figura 18 si può osservare la ricezione dello *start* da parte del server, il suo relativo invio al device e il passaggio di stato in *WF DATA*.

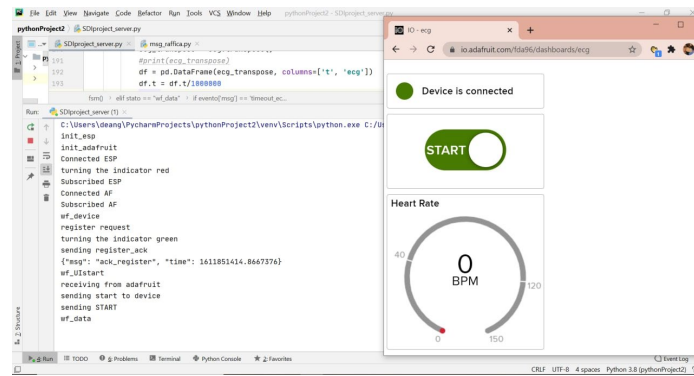


Figura 18. Transizione nello stato WF\_DATA

Nella Figura 19 si osserva che i dati sono riportati nella struttura dataframe che contiene 2 colonne con tempo e valori. Inoltre, si ha la stampa dell'HR medio e la rappresentazione di tale valore sul *gauge* dell'interfaccia utente.

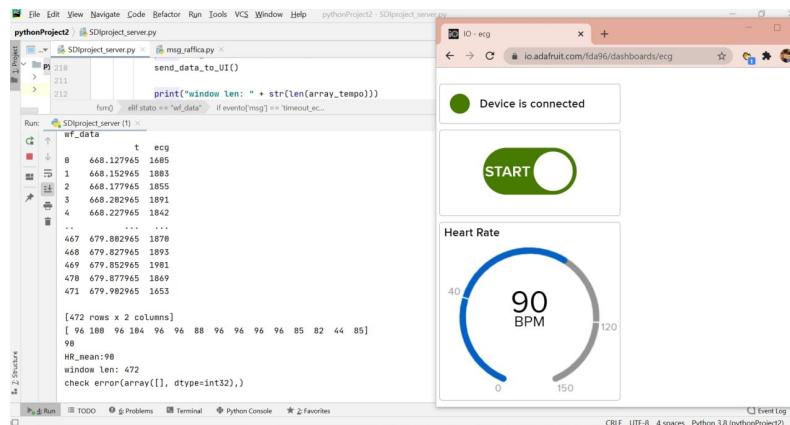


Figura 19. Struttura dataframe e HR\_mean

Nella Figura 20 si nota la media dei valori di Heart Rate, la lunghezza della finestra mobile, l'array che contiene eventuali errori (vuoto) e lo *stop* ricevuto dall'interfaccia ed inviato al device. In ultimo anche la ricezione

dell'*ack\_stop* ed il passaggio di stato in *WF UI START*. Si può, inoltre, vedere come il gauge torni a 0 BPM quando il server riceve lo STOP dall'interfaccia utente.

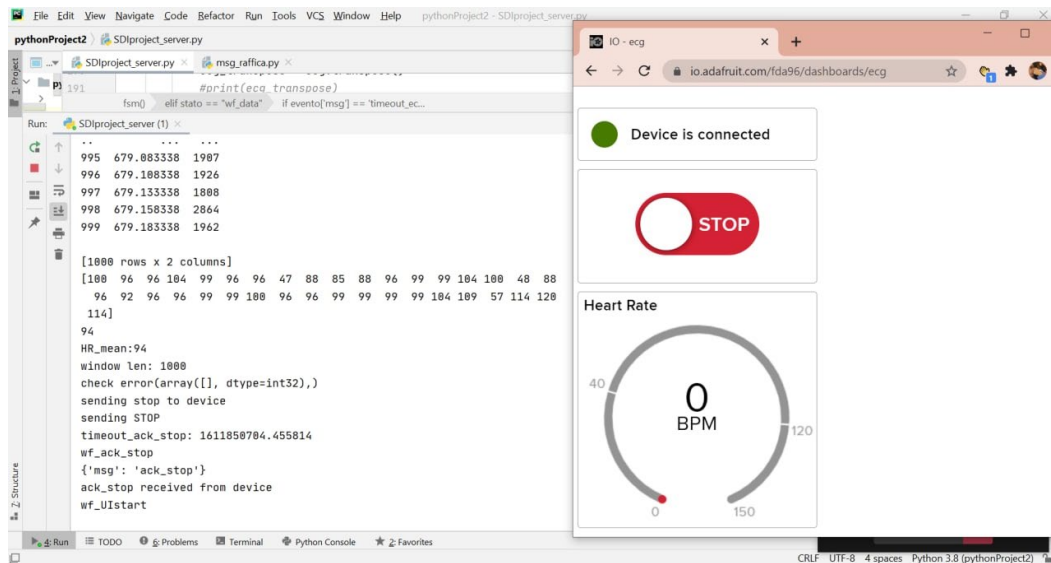


Figura 20. STOP

Nelle Figure 21 e 22 si possono osservare i casi di errore, ovvero nel primo caso si ha un messaggio diverso da quello che il server si aspetta, ad esempio un *register* errato, quindi si osserva che il server non cambia stato ma resta in *WF DEVICE* in attesa del messaggio corretto.

Nel secondo caso si ha una mancata ricezione dell'*ack stop* ed un rinvio dello *stop* al device allo scadere del *timeout\_ack\_stop*.

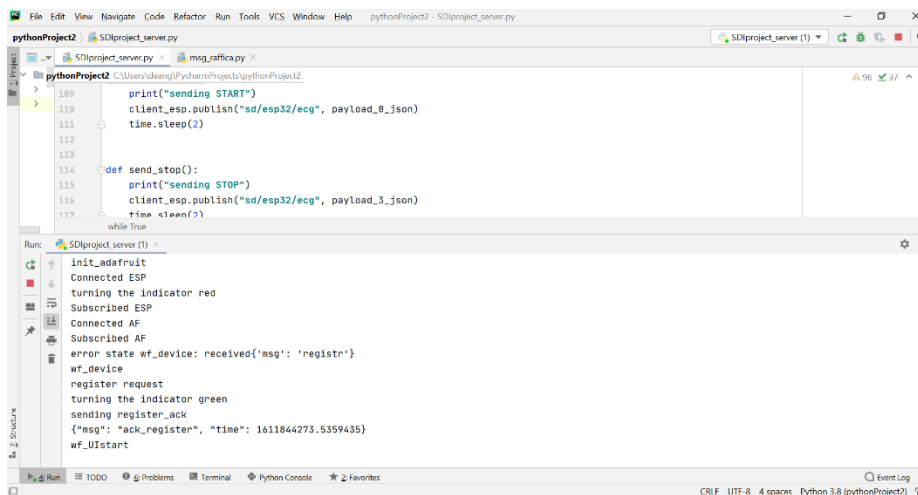


Figura 21. Messaggio register errato

```
pythonProject2 - SDIproject_server.py
SDIproject_server.py
msg_rflica.py

def send_start():
    print("sending START")
    client_esp.publish("sd/esp32/ecg", payload_0_json)
    time.sleep(2)

def send_stop():
    print("sending STOP")
    client_esp.publish("sd/esp32/ecg", payload_3_json)

while True:
```

```
Run: SDIproject_server (1)
{"msg": "ack_register", "time": 1611844338.2896588}
wf_Ulistart
receiving from adafruit
sending start to device
sending START
wf_data
sending stop to device
sending STOP
timeout_ack_stop: 1611844351.4276783
wf_ack_stop
sending STOP
sending STOP
sending STOP
sending STOP
```

Figura 22. Mancata ricezione ack\_stop