Relazione progetto: corso di Progettazione di Sistemi Operativi

Giacomo Papaluca Federico Germinario

1. Introduzione

Gran parte dei sistemi IoT al giorno d'oggi mira a garantire alle persone uno stile di vita più agiato e, perché no, anche più salutare: basti pensare ai vari assistenti casalinghi (home assistant) messi in commercio dalle più grandi aziende di questi anni quali Google, Amazon e Apple, che permettono di gestire vari dispositivi finalizzati alla cura dell'ambiente abitativo, spesso anche in maniera automatizzata e senza dover quindi disturbare in alcun modo l'utente.
L'obiettivo che ci siamo posti per questo progetto è stato quello di creare un dispositivo che raccolga dei dati riguardanti la qualità dell'aria e del manto stradale in alcuni tratti di strada di una città e che invii questi dati ad un cloud, che mostra in ogni momento il loro andamento.
L'idea è quindi quella di creare un sistema che, raccogliendo le informazioni sopracitate in tempo reale, possa fornire agli utenti la possibilità di scegliere il tragitto più piacevole possibile per spostarsi fra le strade di una città.
Il dispositivo è pensato principalmente per le biciclette ma può essere adattato a qualsiasi mezzo che possa risentire di inquinamento e strade imperfette.
In futuro questo prototipo potrà essere utilizzato come punto di partenza per implementare un servizio di raccolta dati distribuito da fornire, ad esempio, a qualsiasi servizio di navigazione satellitare o di bike sharing o a tanti altri servizi legati alla qualità dei tragitti.

2. Architettura del sistema

Il sistema si basa principalmente sulla raccolta e l'invio di alcuni dati ad un server e quindi come prima cosa è bene capire come avviene questa comunicazione e quali sono i principali attori in essa.
Tutto il funzionamento del progetto si basa su una comunicazione ciclica e ripetitiva tra il microcontrollore STM32F407-DISC1, centro nevralgico del sistema, e una scheda di rete, la ESP8266-01, e tra la scheda di rete e un server di raccolta dati. Nella prossima sezione vedremo nel dettaglio il comportamento di questi dispositivi, intanto nella figura 1 un diagramma temporale mostra come avviene la comunicazione.

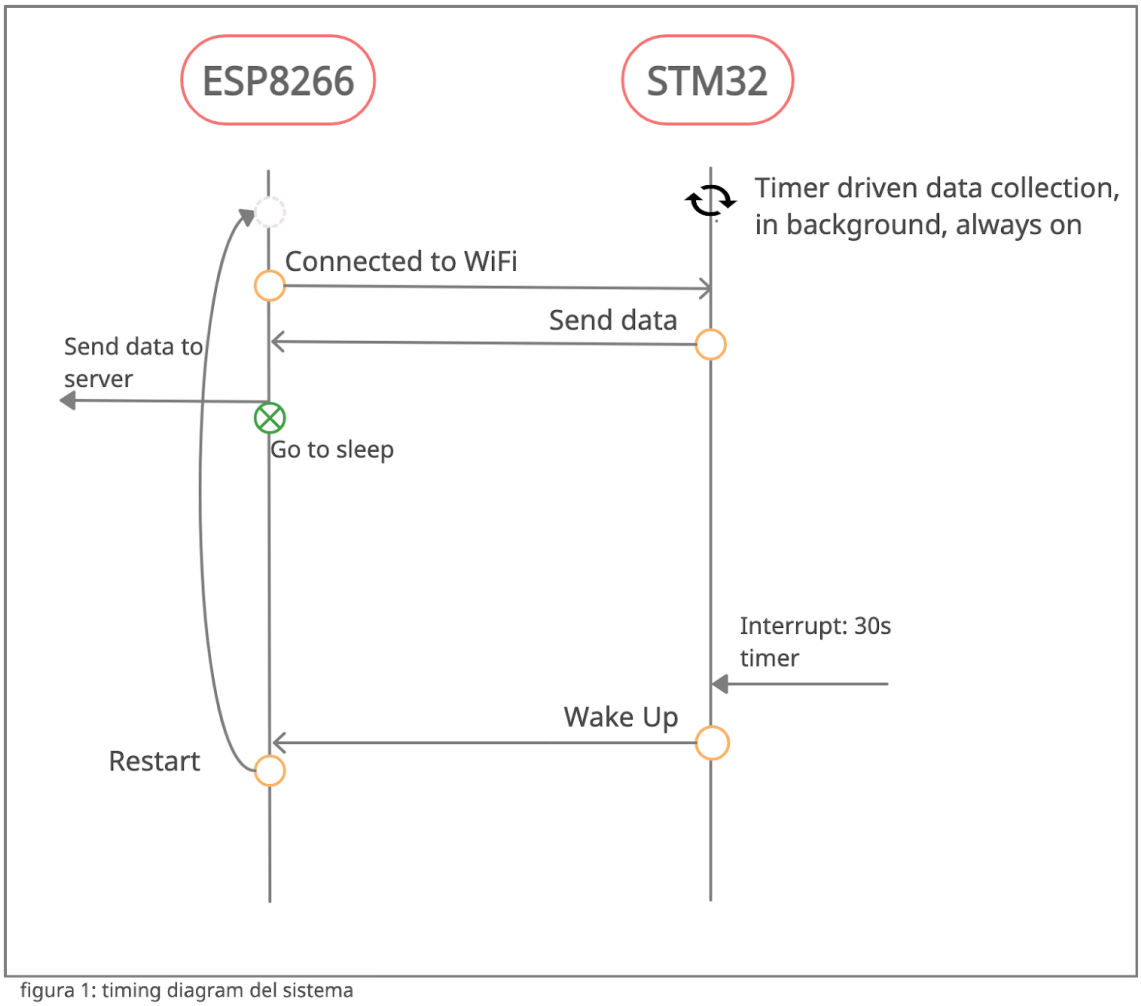

figura 1: timing diagram del sistema

L'idea del ciclo di vita del dispositivo è molto semplice: il microcontrollore STM32 continua a raccogliere i dati di interesse con un approccio timer driven, intanto la scheda di rete ESP8266 cerca di connettersi ad un hotspot fornito dallo smartphone dell'utente e, una volta connesso, comunica all'STM32 di essere pronto ad inviare dati al server e quindi questo gli invia tutti i dati necessari al nostro sistema. Tutta la comunicazione tra questi due dispositivi è guidata da un timer di 30 secondi inizializzato nel setup dell'STM32, allo scadere del quale viene inviato un segnale di reset all'ESP8266 perché si svegli dalla modalità di risparmio energetico e poter quindi ricominciare la comunicazione.

2.1. SMT32, il centro nevralgico del dispositivo

Il microcontrollore STM32F407-DISC1 gestisce la raccolta dei dati sulla qualità dell'aria e del manto stradale e l'invio di questi dati alla scheda di rete, decidendo anche il duty cycle di questa.
Per la raccolta dei dati la scheda deve interfacciarsi a due componenti separate: un sensore che misura la qualità dell'aria, l'MQ-135, e una scheda contenente un accelerometro e un giroscopio, l'MPU-6050.
Come precedentemente accennato, abbiamo scelto un approccio timer driven per la raccolta dei dati e in particolare, oltre al timer di 30 secondi per gestire il duty cycle dell'ESP8266, ci avvaliamo di un timer di 5 secondi per la lettura periodica dell'MQ-135 e di un timer di circa 8 secondi per la lettura dei dati inerziali forniti dall'MPU-6050.

2.1.1. MQ-135 e qualità dell'aria

La qualità dell'aria viene misurata ogni 5 secondi utilizzando il timer tim3. Quest'ultimo, inviando un segnale all'ADC1, fa partire la conversione del segnale analogico, proveniente dall'MQ-135, in un segnale digitale che sarà successivamente elaborato.
L'inizio della conversione dell'ADC (triggerata dallo scadere del timer) avviene con l'utilizzo della seguente riga di codice:
"hadc1.Init.ExternalTrigConv = ADC_EXTERNALTRIGCONV_T3_TRGO;"

Dal momento che la frequenza delle misurazioni è molto bassa non è necessario, anzi sarebbe sconsigliato, utilizzare un il DMA per il salvataggio della misurazione. Per queste ragioni abbiamo optato per una conversione tramite interrupt e la seguente callback gestisce la misurazione effettuata.

```
void HAL_ADC_ConvCpltCallback (ADC_HandleTypeDef* hadc) {
        uint16_t rawValue; float ppm; float v;
        rawValue = HAL_ADC_GetValue(hadc);
        v = ((float)rawValue) / 4095 * 4668;
        ppm = ((v - 320.0) / 0.65) + 400;
        mq_data[mq_index] = ppm;
        mq_index = (mq_index + 1) % MQ_DATA_LENGTH;
}
```

Come possiamo vedere, una volta finita la conversione dell'ADC, il valore di tensione restituito da questo viene tradotto in parts per million (ppm), che è un'unità di misura per misurare la percentuale di molecole di alcune sostanze nell'aria, e salvato in un array.
I valori numerici presenti nella formula sono ricavati come segue: 4095 è il valore massimo restituito dall'ADC in quanto abbiamo scelto come risoluzione 12 bit; 4668 è la tensione massima di riferimento VCC effettiva in mV, misurata con l'utilizzo di un

voltmetro; 320.0 è la tensione in mV misurata in un ambiente teoricamente a 400 ppm; e infine 0.65 è lo slope di tensione garantito dal datasheet del sensore.

2.1.2. MPU-6050 e qualità della strada

La parte più complessa del dispositivo è quella che gestisce le misurazioni dei sensori inerziali.
Per eseguire questo compito l'STM32 si interfaccia tramite protocollo I²C al microprocessore MPU-6050, che contiene un accelerometro e un giroscopio e viene distribuito già programmato con il firmware ufficiale.
Dato che l'applicazione mira alla raccolta di dati per il calcolo di alcune metriche e non necessita di bassa latenza o altri vincoli real time, sfrutteremo il buffer da 1024 byte che questo microprocessore mette a disposizione per salvare temporaneamente le misurazioni e sfrutteremo poi il DMA per trasferire questi dati nella SRAM dell'STM32 per ridurre al minimo l'utilizzo della CPU.
L'MPU-6050 mette a disposizione un pin INT per la segnalazione di eventi come ad esempio quando il buffer fifo va in overflow oppure quando una misurazione è pronta.

Inizialmente per la lettura del burst di dati del fifo abbiamo provato ad usare come trigger l'interrupt di buffer overflow ma la lettura non avveniva in tempo sufficientemente veloce per una lettura di dati sensati (nonostante avessimo ridotto di molto la frequenza di sampling dei sensori). Quindi abbiamo adottato un ulteriore timer che parte alla ricezione del primo interrupt della prima misurazione effettuata e il cui periodo fosse abbastanza lungo per avere una buona quantità di misurazioni. Con una serie di prove abbiamo raggiunto un buon trade off tra tempo di CPU e attendibilità dei dati: ogni 8 secondi il timer scade e l'STM32 fa partire una richiesta di lettura del buffer attraverso l'interfaccia I²C, delegata al DMA, ottenendo così all'incirca 960 byte ogni burst.
Una volta letti i 960 byte, che corrispondono a 80 misurazioni di accelerometro e 80 misurazioni di giroscopio, abbiamo suddiviso e aggregato i dati, secondo il protocollo di sliding windows (molto usato in ambito sensoristico), in finestre da 20 misurazioni l'una con un overlap del 50%. Per ogni misurazione calcoliamo il vettore risultante delle componenti x, y e z dell'accelerometro e del giroscopio in questo modo "`float vector = sqrt(pow(Ax, 2)+pow(Ay, 2)+pow(Az, 2));`" e per ogni finestra calcoliamo il valore medio dell'intensità del vettore risultante dell'accelerometro.
Per calcolare la qualità di un tratto stradale abbiamo usato un metodo a soglia variabile: per ogni finestra contiamo quante volte la differenza tra il vettore risultante dell'accelerometro e la media della finestra supera una certa soglia e questo numero sarà il nostro indice di bassa qualità della strada. La soglia è variabile in quanto dipende dall'accelerazione angolare misurata dal giroscopio in quell'istante di misurazione.

Oltre alla stima della qualità della strada abbiamo implementato una feature aggiuntiva basata sulle stesse misurazioni: la rilevazione di una caduta dell'utente.

Quando il sensore rileva delle grosse accelerazioni seguite da un lungo periodo di stabilità suppone che sia avvenuta una caduta da parte dell'utente. Questa feature potrebbe in futuro essere integrata con un qualche meccanismo di activity recognition, ad esempio una conferma o meno da parte dell'utente o un qualche algoritmo di machine learning, per renderla più precisa.

2.1.3. Simulazione GPS e comunicazione con ESP8266

Tutte le misurazioni effettuate devono avvalersi di una posizione GPS così da far corrispondere i valori acquisiti dall'STM32 a un determinato punto sulla mappa.
Per fare questo, invece di utilizzare un modulo GPS abbiamo deciso di simularlo via software. Questo ci ha permesso di poter testare il progetto molto più facilmente.
Pertanto, durante la progettazione per prima cosa abbiamo raccolto un insieme di coordinate GPS, così da simulare un percorso in bicicletta; per fare ciò abbiamo utilizzato il software Google Earth. Dopo aver raccolto le coordinate (latitudine e longitudine) è stato creato un array circolare di tipo *Coordinate* dove abbiamo memorizzato le stesse (*Coordinate fake_gps[13]*).
Ogni qual volta la scheda ESP8266 è pronta ad inviare i dati al server, viene selezionata una coordinata dall'array e viene inviata ad essa insieme a tutti gli altri dati calcolati.

Come detto in precedenza, quando l'ESP8266 si connette a una rete WiFi dopo essere stato risvegliato dall'STM32, invia un segnale sul pin GPIO_EXTI 1 (PC1) per indicare alla scheda che è pronta a ricevere i dati che poi verranno inviati sul cloud (ThinkSpeak).
Quando il segnale giunge alla scheda STM32 viene chiamata la seguente callback:
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin), che si occupa di aggregare i dati e inviarli tramite l'interfaccia UART2 alla scheda di rete.
Come spiegheremo nelle seguenti sezioni, la scheda ESP8266 va in modalità deep sleep, per ridurre al minimo i consumi, sarà quindi compito dell'STM32 risvegliarla, mandandole un segnale di reset, allo scadere di un timer periodico di 30 secondi.

2.2 ESP8266, gateway tra l'STM32 e la rete

Il modulo WiFi ESP8266 viene distribuito con un firmware ufficiale, reperibile anche su internet, che permette di pilotarlo e gestirlo attraverso dei comandi AT. Questa soluzione non è ottimale per una serie di motivi ma soprattutto perché, per un funzionamento corretto, l'STM32 avrebbe dovuto aspettare in busy waiting delle risposte positive dalla scheda prima di inviarle qualsiasi informazione.
Per questo abbiamo scelto di riprogrammare l'ESP8266 in modo da decidere noi il suo comportamento e adattarlo ai nostri bisogni.
Come prima cosa, al reset, il modulo legge la sua EEPROM per controllare se vi sono salvati un SSID e una password e in caso positivo prova a connettersi all'SSID salvato. Nel caso in cui non riuscisse a connettersi perché non trova la rete corrispondente o la

EEPROM fosse vuota, entra in modalità access point permettendo all'utente di accedere ad un web server (che gira sul modulo wifi), in cui può inserire l'SSID e la password dell'hotspot del proprio smartphone, i quali verranno salvati nella EEPROM del modulo.
Una volta ottenuti SSID e password, l'ESP8266 torna in modalità station e riprova a connettersi alla rete selezionata. Una volta connesso è pronto ad inviare i dati sul server e lo riferisce alla scheda STM32 attraverso un impulso mandato dal proprio pin GPIO2. A questo punto il modulo wifi riceverà una stringa formattata in una certa maniera sulla porta UART da cui ricaverà i dati con un parsing e li invierà al server.
Una volta inviati i dati sul server, il modulo andrà in deep sleep fino a che non riceverà un segnale di reset dall'STM32 e ricomincerà l'esecuzione.

3. Le componenti fondamentali del sistema

Il core del nostro dispositivo è sicuramente la scheda STM32F407-DISC1 che funge da coordinatore delle varie componenti del sistema, decidendone anche i duty cycle, e raccoglie e processa i dati misurati dai sensori.
Non necessitando di alte prestazioni e non avendo vincoli real time, il processore di questa scheda gira a 50 MHz, una frequenza non eccessivamente elevata, in modo da favorire il risparmio energetico.

Come mostrato in figura 2, i pin attivi sono 8: PA2 e PA3 per la comunicazione UART con l'ESP8266-01; PA0 per la ricezione dei dati del sensore MQ-135 sul canale 0 dell'ADC1; PB6 e PB7 per la comunicazione I²C con l'MPU6050; PD11 per la ricezione di interrupt dall'MPU6050; PE8 per mandare il segnale di reset all'ESP8266-01 e infine il pin PC1 per ricevere il segnale che l'ESP8266-01 è connesso e pronto a ricevere dati

L'ESP-01 è il modulo wifi che permette al sistema di interfacciarsi a internet ed ha un duty cycle molto efficiente in cui, a regime, per circa 2 secondi è in run mode e per 28 secondi è in deep sleep mode. Comunica con l'STM32 solo attraverso una porta uart e i due segnali sopra citati: uno per essere resettata e uno per comunicare che è pronta a ricevere i dati. Il primo lo si ottiene mandando a zero il pin RST e il secondo mandando un impulso dal pin GPIO2 del modulo al pin PC1 della STM32.

Il microprocessore MPU6050 è un SoC che contiene una serie di sensori MEMS e ADC che il sistema sfrutta per ottenere dati riguardanti l'accelerazione lineare e l'accelerazione angolare del sistema.
Comunica con l'STM32 attraverso l'interfaccia I²C, di cui fanno parte i pin SCL (Serial Clock Line) e SDL (Serial Data Line) e con un segnale di interrupt mandato dal suo pin INT al pin PD11 dell'STM32.
Il SoC esegue il firmware ufficiale con cui viene distribuito ma abbiamo settato alcuni suoi parametri scrivendo alcuni suoi registri utilizzando il protocollo I²C.

Volendo favorire i consumi a fronte delle prestazioni, abbiamo settato le frequenze di sampling dei MEMS a 1 KHz e abbiamo imposto un data rate di 10 Hz per ridurre il numero di letture dei dati al secondo e quindi l'utilizzo della CPU dell'STM32. Inoltre abbiamo abilitato la scrittura del buffer in modo da poter leggere i dati in un burst invece che ad ogni misurazione.

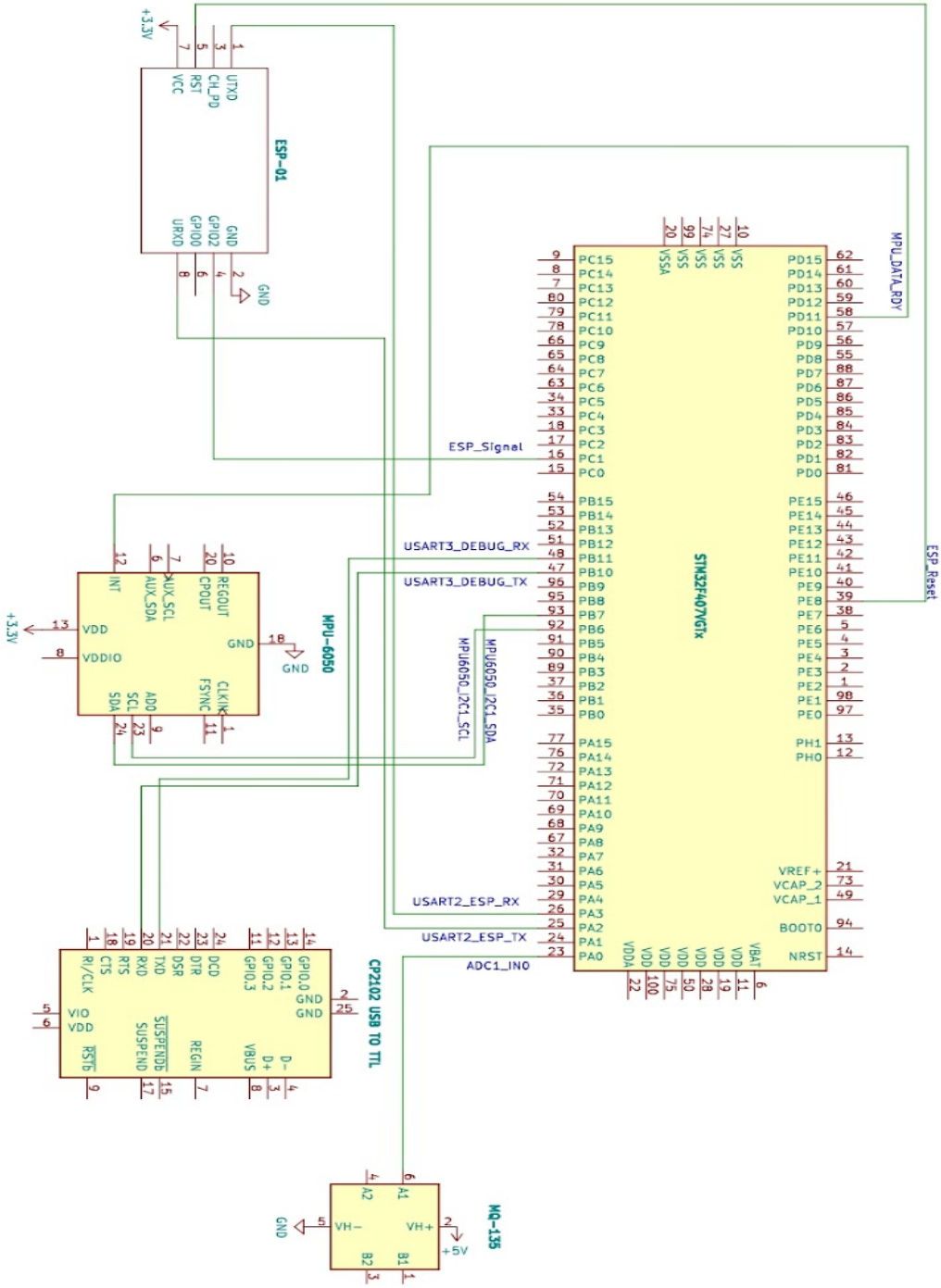

Figura 2

Il sensore MQ-135 è il componente più semplice in quanto è solamente un sensore analogico composto da una serie di resistenze, delle quali una varia a seconda della concentrazione di alcuni gas nell'aria. La tensione sul pin A1 varia al variare di questa resistenza e quindi il sistema non deve fare altro che leggere la tensione, digitalizzarla tramite l'ADC e tradurre il valore digitalizzato in un valore sensato in ppm (parts per million).

4. Analisi dei tempi di esecuzione

Essendo un sistema timer driven e quindi con funzionamento ciclico e periodico non è possibile calcolare un reale tempo di esecuzione, per cui ci siamo limitati a calcolare i tempi di esecuzione delle singole callback, dove risiede quasi l'intera complessità del codice.
I tempi di ciascuna callback sono mostrati nella seguente figura e, come previsto, la callback che processa i dati dei sensori inerziali (HAL_I2C_MasterRxCpltCallback) è il vero collo di bottiglia del sistema.

```
TIME (ESP_Signal) =  28.27 ms
TIME (tim4 callback) =  10.14 ms
TIME (HAL_I2C_MasterRxCpltCallback) =  148.77 ms
TIME (tim4 callback) =  10.14 ms
TIME (HAL_I2C_MasterRxCpltCallback) =  144.38 ms
TIME (tim4 callback) =  10.14 ms
TIME (HAL_I2C_MasterRxCpltCallback) =  144.64 ms
TIME (tim4 callback) =  10.14 ms
TIME (HAL_I2C_MasterRxCpltCallback) =  144.26 ms
```

I tempi di esecuzione sono stati calcolati leggendo il registro CYCCNT dell'unità DWT per ottenere il numero di cicli di clock utilizzati per eseguire ogni callback e dividendo questo numero per la frequenza di clock.

5. Analisi del consumo

Il consumo del sistema è stato un punto focale del lavoro svolto in quanto il dispositivo progettato è pensato per essere usato in un contesto di mobilità medio alta e quindi con necessità di una alimentazione portatile. Per questo abbiamo pensato che un approccio interrupt driven, utilizzando solo l'HAL ed evitando l'utilizzo di un real time OS, potesse essere la soluzione migliore per mantenere un basso consumo.
Per cominciare abbiamo pensato che, siccome l'unico compito dell'ESP8266 è quello di inviare i dati resi disponibili dall'STM32 al server, appena effettuato l'invio, può entrare in risparmio energetico fino a che non deve inviare altri dati. Quindi, trasferendo tutta la gestione dei duty cycle del sistema sull'STM32, l'ESP8266 non deve fare altro che

aspettare un segnale di reset da quest'ultima per svegliarsi, inviare i dati sul server e infine andare in deep sleep mode, che è la modalità a più basso consumo di questa scheda di rete con un consumo di circa 20 uA a fronte dei circa 50 mA in running mode.
Per quanto riguarda il microcontrollore principale, l'STM32, abbiamo sfruttato la funzione "HAL_PWR_EnableSleepOnExit();" per far si che, dopo aver servito un interrupt con la relativa callback, la scheda vada in risparmio energetico, per svegliarsi in automatico con l'arrivo del prossimo interrupt da servire.
Con questo approccio abbiamo dimezzato il consumo medio della scheda passando da circa 20 mA a 10mA di corrente dissipata.
Ora utilizzando la formula seguente

$$P_{AVG} = [D \times P_{ON}] + [(1 - D) \times P_{OFF}]$$

$$D = \frac{T_{ON}}{T_{ON} + T_{OFF}} = duty\ cycle$$

$$P_{ON} = on \text{ power}, P_{OFF} = off \text{ power}$$

$$T_{ON} = on \text{ time}, T_{OFF} = off \text{ time}$$

e tenendo in considerazione che la scheda lavora in run mode (50 mA) per 2 secondi e in deep sleep mode (0.02 mA) per 28 secondi, abbiamo calcolato che il consumo medio della scheda di rete equivale a 3.3 mA.
Considerando inoltre che l'MPU-6050 ha un consumo massimo di 3.9 mA possiamo constatare un consumo medio del sistema di circa 17.2 mA. Con un power bank da 10000 mAh saremmo capaci di alimentare il sistema per circa 580 ore consecutive.
Per misurare le correnti dissipate abbiamo messo un amperometro ai capi dell'ESP8266 e al posto del jumper JP1 della scheda STM32F407-DISC1.
Il vero collo di bottiglia per quanto riguarda il consumo energetico è sicuramente l'MQ-135 che consuma circa 150mA ma sono disponibili in commercio sensori più costosi che sono in grado di assorbire molto meno.