Report Laboratorio 2 (Hardware) del corso Tecnologie per IoT

Introduzione

L'obiettivo di questo laboratorio è quello di progettare e costruire uno “smart home controller” tramite l’utilizzo di una scheda Arduino Yun, vari sensori, componenti elettronici e del linguaggio di programmazione Arduino. Il nostro dispositivo deve essere in grado di avviare e spegnere una ventola (utilizzata per raffreddare l’ambiente) e un led (che simula il funzionamento di una resistenza utilizzata per riscaldare una stanza), e deve essere in grado di rilevare la presenza o assenza di persone nell’ambiente circostante, tramite sensori di movimento e suono, in modo da poter avere comportamenti differenti dell’impianto a seconda se ci siano o meno persone nei dintorni del dispositivo. Inoltre deve essere possibile all’utente interfacciarsi con il dispositivo tramite un display LCD che mostra vari parametri utilizzati dal climatizzatore e tramite la porta seriale della scheda Arduino, in modo da poter cambiare alcune impostazioni. Infine abbiamo ulteriormente ampliato le funzionalità del nostro impianto, permettendo all’utente di accendere e spegnere un led (che simula una lampadina) tramite un doppio battito di mani.

Componenti utilizzati

* Scheda Arduino Yun rev 2
* Sensore di movimento GravityDigital IR Motion Sensor SEN0018
* Sensore di rumore Grove - Sound Sensor LM386
* Display LCD 16x2 I2C DFR0063
* Motore DC + Fan DFR0332
* Sensore di Temperatura GRV TEMP Grove, NCP18WF104
* 2 led multicolore
* 2 resistenze da 180Ω
* Breadboard da 830 punti
* Vari cavi per ponticelli da 10 cm

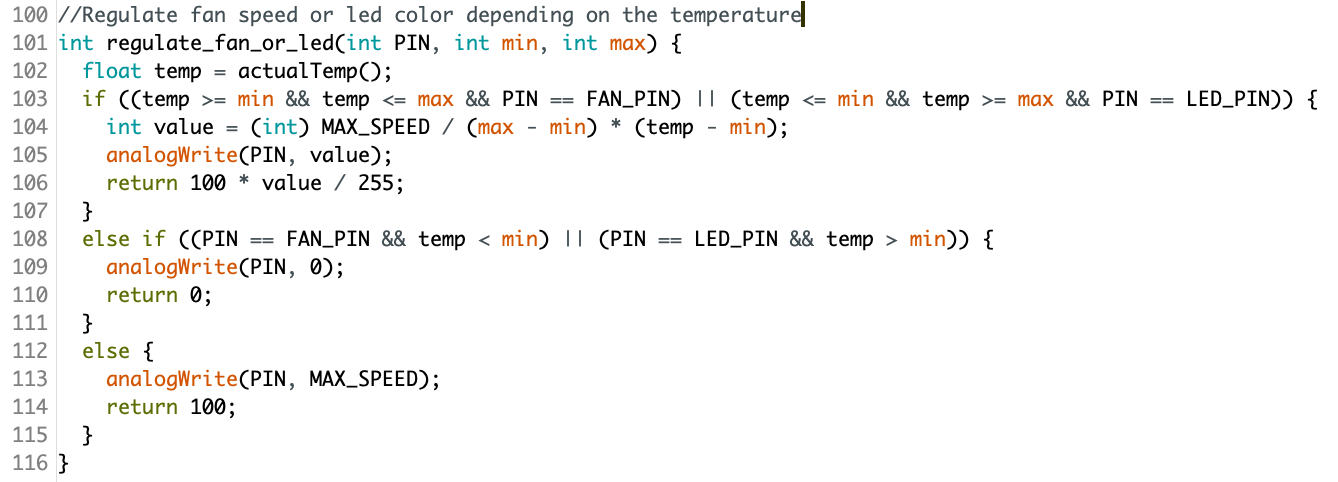
Svolgimento

Il primo requisito del nostro “home smart controller” è quello di far azionare una ventola ad una velocità proporzionale alla temperatura rilevata. In modo particolare il motore deve ruotare in modo direttamente proporzionale quando la temperatura si trova all’interno di un prestabilito range, raggiungendo una velocità massima e minima quando la temperatura si trova al di fuori dell’intervallo. La funzione alla base del funzionamento della ventola è stata chiamata *regulate\_fan\_or\_led()* ed è una funzione che ha come parametri il pin utilizzato dal sensore, la temperatura minima e massima del range di funzionamento. Il primo step della funzione è quello di ottenere la temperatura attuale misurata dal sensore di temperatura tramite la funzione *actualTemp()* che ha funzionamento del tutto analogo all'esercizio 5 del laboratorio 1. In sintesi il sensore di temperatura restituisce un voltaggio analogico dal quale, tramite varie conversioni, può essere trasformato in una temperatura in gradi Celsius. Successivamente la temperatura ottenuta viene confrontata con il range di input della funzione e a seconda del pin passato, viene effettuata una scelta differente.

Ci sono tre diverse opzioni per la conversione da temperatura a velocità. Nel primo caso la velocità sarà settata al minimo (0), nel secondo verrà settata al massimo (255) e nel terzo caso verrà impostata ad un valore proporzionale alla temperatura. Nello specifico in questo ultimo caso abbiamo voluto scrivere il codice nel modo più generico possibile in modo da poter riutilizzare la stessa funzione anche per il led che simula la resistenza. Abbiamo dunque scritto una funzione che mappa la temperatura in un valore compreso tra 0 e 255 utilizzando gli estremi del range.

Il valore ottenuto verrà invitato al dispositivo (la ventola in questo caso) tramite *analogWrite()* dato che utilizziamo la PWM per avere delle variazioni continue della velocità.

La funzione ritorna il valore ottenuto in percentuale, dato che dovrà essere visualizzato sul display LCD.



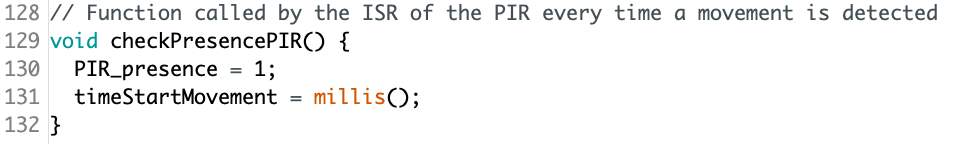
Il secondo punto, richiede che il led che simula il resistore, venga acceso quando la temperatura scende sotto un dato valore. In particolare dovrà essere luminoso proporzionalmente alla temperatura rilevata. Anche il led funziona usufruendo di un intervallo di lavoro prestabilito, ovvero dovrà essere molto luminoso quando la temperatura sarà prossima al valore minimo e dovrà invece essere poco luminoso per temperature prossime o superiori al valore massimo.

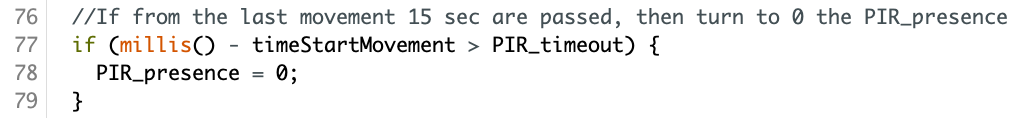
Avendo scritto precedentemente la funzione *regulate\_fan\_or\_led()* in modo generico, ora possiamo sfruttare la stessa funzione anche per il led, avendo però un’accortezza, ovvero quella di passare alla funzione la temperatura massima come *min* e la temperatura minima come *max.* In questo modo otterremo che all’aumentare della temperatura diminuirà la potenza del led.

Successivamente viene richiesta l’implementazione di un sensore di movimento, in grado di rilevare la presenza di una persona nei dintorni del sensore. Nel caso un movimento venisse rilevato, il dispositivo deve dedurre che ci sia una persona nei paraggi e inoltre deve assumere che se per un tempo preimpostato (30 minuti nel nostro caso) non vengono rilevati movimenti, allora non ci sia nessuna persona nell’ambiente circostante.

Noi abbiamo deciso di sfruttare il pin digitale che supporta gli *Interrupt* (il pin digitale 7 per la scheda Yun) proprio per gestire il sensore di movimento. Abbiamo collegato la funzione *checkPresencePIR()* al fronte di salita del segnale di input in modo che ad ogni presenza rilevata la funzione venisse richiamata. La funzione *checkPresencePIR()* è molto intuitiva, semplicemente imposta la variabile globale *PIR\_presence* a 1 e salva nella variabile globale *timeStartMovement* il tempo trascorso dall’avvio del programma in millisecondi utilizzando la funzione *millis*. In questo modo all’interno del *loop()* ci sarà un controllo che verifica da quanto tempo la variabile *PIR\_presence* è stata settata e nel caso sia passato un tempo superiore all’intervallo di timeout, resetterà la variable a 0. Nel caso un nuovo movimento venga rilevato, la funzione *checkPresencePIR()* verrà nuovamente chiamata e dunque anche il tempo di timeout verrà reimpostato. E’ importante notare che le variabili *PIR\_presence* e *timeStartMovement* sono entrambe definite come ‘volatile’ dato che vengono usate sia nel loop che all’interno di una ISR.

Durante lo svolgimento di questo punto abbiamo deciso di usare la funzione *millis()* in quanto l’idea iniziale che avevamo avuto ci ha dato parecchi problemi. Infatti, inizialmente avremmo voluto che ad ogni fronte di salita venisse chiamata una funzione che impostava un timer, il quale avrebbe eseguito una ISR ogni mezz’ora. Questo metodo non ha restituito i risultati sperati, infatti l’utilizzo dei timer all’interno delle ISR non garantisce un comportamento adeguato e ci siamo accorti velocemente che questa nostra scelta progettuale ci avrebbe creato troppi problemi e comportamenti imprevisti. Abbiamo dunque preferito salvare il tempo trascorso in una variabile globale in modo da poter gestire i tempi in modo più comodo e preciso.





Il punto successivo richiede l’utilizzo di un sensore di rumore per poter compensare le “zone di ombra” del sensore PIR. Dato che il sensore non è in grado di rilevare l’origine del rumore, viene richiesto di non assumere che ad un singolo rumore corrisponda la presenza di una persona ma di confermare la presenza solo se un determinato numero di rumori viene rilevato in un prestabilito lasso di tempo (50 eventi in 10 minuti per esempio). Nel caso per più di un’ora questa condizione non venga mai rispettata, allora il dispositivo può dedurre che nessuno è presente nell’ambiente circostante.

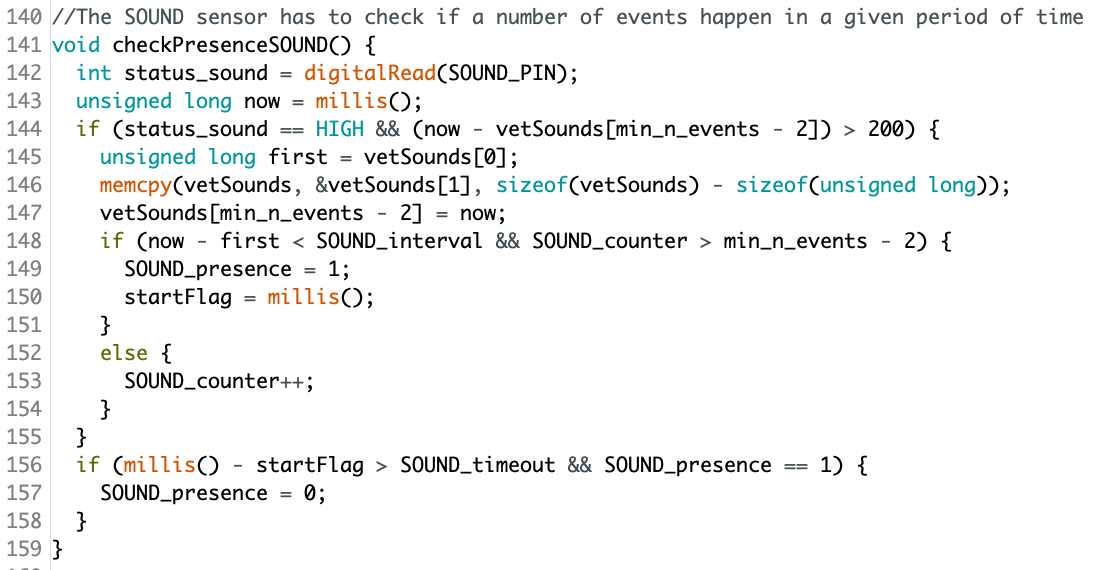
Questo passaggio del laboratorio è stato sicuramente il più lungo e complesso da gestire. Inizialmente abbiamo provato ad usare un timer e un contatore di rumori, verificando allo scadere del timer quante rilevazioni erano state effettuate, e impostando la variabile globale *SOUND\_presence* a seconda del risultato ottenuto. Questa nostra soluzione funzionava alla perfezione dal punto di vista pratico ma a livello concettuale non avevamo tenuto in considerazione che, se a ridosso dello scadere del timer avessimo rilevato per esempio 49 rumori e subito dopo aver reimpostato il timer avessimo rilevato un nuovo rumore, sebbene i 50 rumori fossero avvenuti in meno di 10 minuti, la variabile *SOUND\_presence* sarebbe rimasta a zero. Abbiamo dunque dovuto pensare ad un sistema più dinamico, basato su un vettore di variabili di tipo unsigned long che abbiamo chiamato *vetSounds[]*. Il vettore ha la lunghezza pari al numero di eventi da dover rilevare per poter confermare la presenza meno uno (quindi 49 nel nostro caso).

Procediamo ora alla spiegazione dettagliata della funzione *checkPresenceSound()* in modo da rendere più chiara la nostra implementazione.

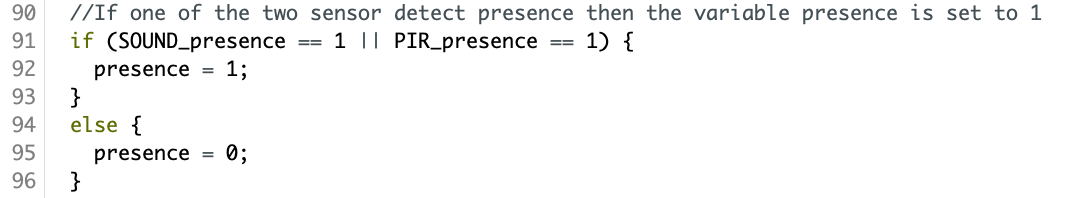
Come prima cosa, per rilevare correttamente il rumore abbiamo impostato una doppia condizione. Non solo il sensore deve aver captato un rumore, ma questo deve essere stato rilevato almeno 200 millisecondi dopo il rumore precedente. Questa scelta è dovuta al fatto che inizialmente ogni rumore rilevato produceva un’oscillazione impercettibile intorno al livello di soglia impostato sul sensore, causando decine di rilevazioni per ogni singolo evento sonoro. Impostando 200 millisecondi di distanza tra due rumori (scelta effettuata in modo empirico) per renderli validi, abbiamo “filtrato” possibili errori dovuti a oscillazioni naturali dell’onda sonora.

Ogni volta che un nuovo rumore viene rilevato, viene salvata la testa del vettore nella variabile *first*, viene scalato il vettore di una posizione (sovrascrivendo dunque il primo elemento dell’array) e successivamente viene inserito all’ultimo spazio di memoria del vettore il tempo attuale rilevato tramite *millis()*. Successivamente viene controllata la distanza temporale tra la variabile *first* e l’ultimo suono rilevato. In questo modo possiamo verificare quanto tempo è trascorso dal primo suono e quello avvenuto dopo aver rilevato 49 rumori. Se il tempo è superiore ai 10 minuti allora non possiamo confermare la presenza mentre in caso sia inferiore il nostro sistema può assumere che ci sia una persona.Quando questo ultimo caso avviene, un tempo in millisecondi viene salvato nella variabile *startFlag* che ci servirà per settare la variabile di presenza a 0 quando il timeout scade.

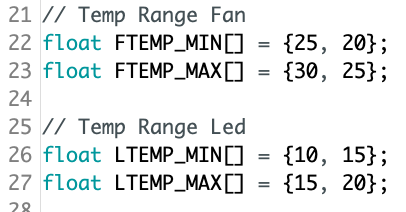
Il codice mostra anche una variabile contatore che monitora il numero di eventi avvenuti. Questa variabile è utilizzata perché inizialmente il vettore contenente i tempi è inizializzato a zero in tutti i 49 spazi di memoria, quindi nel caso venga rilevato un singolo rumore dopo pochi secondi, la condizione di distanza temporale verrebbe soddisfatta e verrebbe rilevata la presenza senza però che i 50 suoni siano effettivamente avvenuti. Usando il contatore, possiamo evitare questo problema usando i primi 49 eventi per riempire il vettore e solo successivamente utilizzare l'algoritmo precedentemente analizzato. Si noti che il contatore viene aumentato solo nei primi 49 casi, e non ad ogni rumore rilevato, in questo modo eviteremo che la variabile counter vada in overflow dopo circa 32767 rumori (plausibile per un sistema che deve lavorare in modo continuato per mesi interi). Infine, ogni volta che la funzione viene eseguita, viene controllato quanto tempo è passato dall’ultima volta che *SOUND\_presence()* è stata impostata ad 1 e se il tempo trascorso è superiore al tempo di timeout allora la variabile verrà settata a 0.



Ora che sia la variabile *PIR\_presence* e *SOUND\_presence* sono state definite correttamente, è possibile effettuare un operatore OR per impostare il valore della variabile *presence* che rappresenta in modo generico la presenza o l’assenza di persone nell’ambiente circostante.



Prima di introdurre le parti successive del laboratorio, riteniamo debba essere chiarita la struttura dati utilizzata per gestire gli intervalli di temperatura del led e della ventola. Inizialmente, abbiamo usato delle variabili globali di tipo float ma successivamente, quando abbiamo iniziato ad avere una doppia scelta per ogni valore (presenza o assenza), abbiamo deciso di gestire le variabili tramite array. Quindi al led e alla ventola avranno associati due vettori ciascuno, uno relativo alla temperatura massima e alla temperatura minima. Ognuno di questi vettori avrà due spazi di memoria. Il primo (associato alla posizione 0) si riferirà al caso di assenza mentre il secondo (associato alla posizione 1) si riferirà al caso di presenza rilevata. In questo modo per tutti i punti successivi possiamo eseguire azioni diverse a seconda della situazione, semplicemente usando la temperatura nella posizione stabilita dalla variabile *presence* (che analogamente è 0 quando non viene rilevato nessuno nelle vicinanze e 1 viceversa). Nell’esempio sottostante la temperatura minima gestita della ventola è di 25°C in caso di assenza e di 20°C in caso di presenza.



Dopo aver completato la parte riguardante la sensoristica e la gestione dell’impianto di condizionamento, abbiamo sviluppato l’interfaccia utente del nostro sistema in modo che la modifica di parametri e la visualizzazione dei dati fosse agevole.

Come primo passaggio abbiamo effettuato tutti i collegamenti necessari per far funzionare correttamente il display LCD, in modo analogo a quanto abbiamo fatto nell’ultimo esercizio del primo laboratorio. Il monitor deve mostrare una quantità di informazioni che diventano poco chiare se vengono compresse in un’unica schermata dello schermo. Abbiamo quindi scritto una semplice che ogni 5 secondi modifica le informazioni visualizzate. Nella prima schermata viene visualizzata la temperatura attuale, la presenza o l’assenza di persone nelle vicinanze, l'intensità di rotazione del motore e l’intensità di luminosità del led (entrambe in percentuale). Nella schermata seguente vengono mostrati i 4 setpoint di temperatura relativi all’attuale presenza o non presenza di persone. Inizialmente la nostra idea è stata quella di utilizzare un timer, che ogni 5 secondi chiamava una funzione per aggiornare i dati da visualizzare. Questo nostro tentativo si è dimostrato però nuovamente inappropriato in quanto il monitor non veniva aggiornato. Abbiamo optato dunque un’altra volta per l’utilizzo della funzione *millis()*, grazie alla quale abbiamo potuto salvare l’istante in cui la schermata è stata cambiata l’ultima volta e confrontare questo valore con l’istante attuale per verificare se i 5 secondi fossero passati. Per poter cambiare schermata ad ogni ciclo, abbiamo definito una variabile *num\_display,* che può assumere i valori 0 e 1, che viene associata ad ognuna delle due schermate . Ogni volta che una delle due videate viene visualizzata, la variabile *num\_display* viene invertita in modo che al ciclo successivo venga visualizzata l'altra schermata. Il resto del codice presente all’interno della funzione *changeDisplay()* serve semplicemente a visualizzare i dati e le informazioni corrette per ognuna delle due schermate.

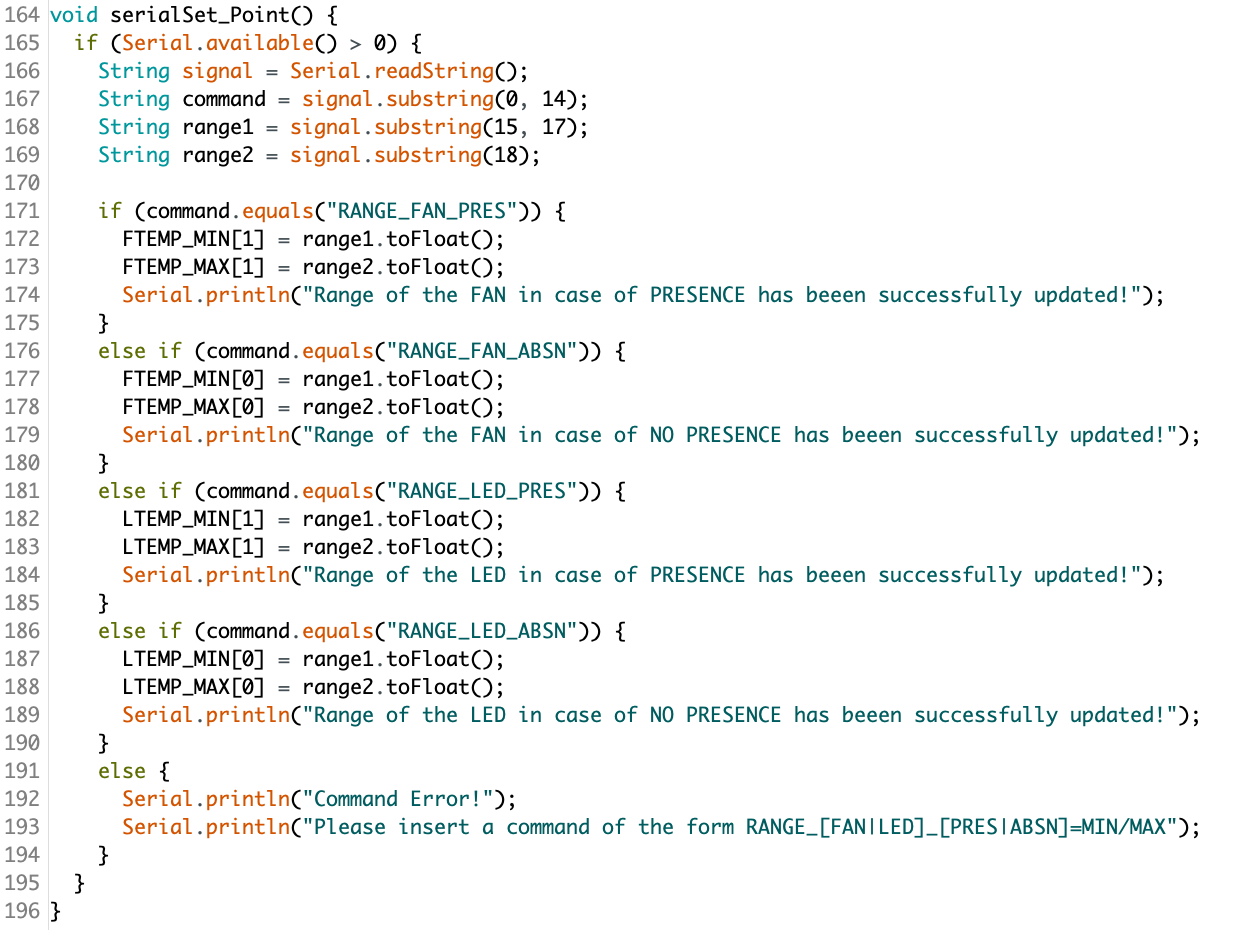


Come parte conclusiva dell’interfaccia utente, abbiamo creato una serie di comandi standard in modo da permettere all’utilizzatore di poter impostare i range di lavoro del sistema di condizionamento tramite la connessione seriale della scheda Arduino. Per poter capire che tipo di set-point viene cambiato, abbiamo progettato dei comandi che aiutano l’utente a capire quale parametro sta modificando e semplificano la vita degli sviluppatori in quanto è molto più semplice “comprendere” e processare il comando ricevuto. Nel nostro caso il formato dei comandi è il seguente:

RANGE\_[LED | FAN]\_[PRES | ABSN]=MIN/MAX

SI può notare come la lunghezza del comando (se correttamente inserito) abbia una lunghezza costante, in questo modo è molto facile poter leggere i vari campi semplicemente creando delle ‘substring’. E’ dunque possibile cambiare ogni impostazione relativa al led o alla ventola, con presenza o con assenza di persone. Ad esempio il comando *RANGE\_FAN\_PRES=20/25* sarà utilizzato per impostare il range di temperature della ventola in caso di rilevamento di presenze.

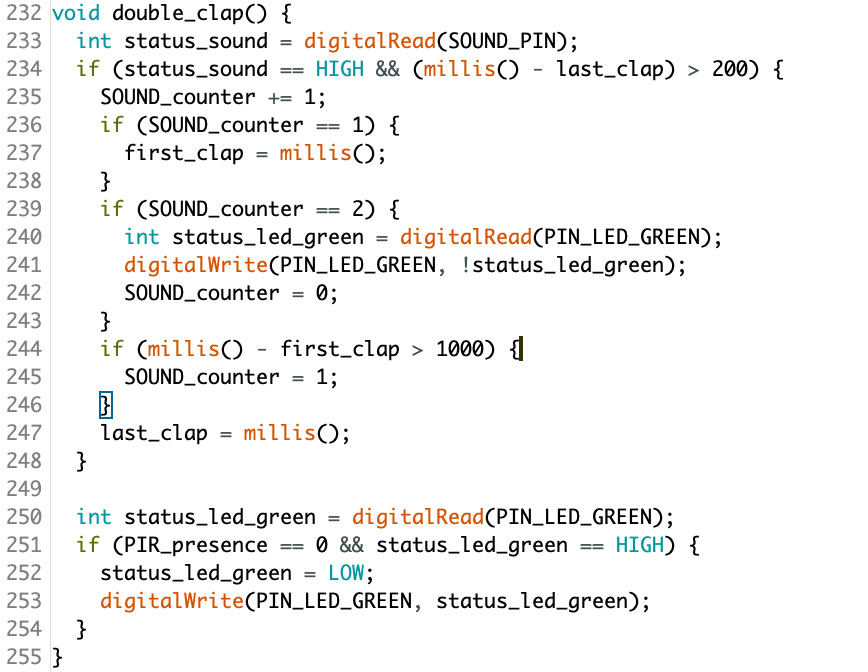
La funzione che ha il compito di gestire questo passaggio di informazioni dall’utente alla ‘smart home’ è *serialSet\_Point()* la quale effettua delle operazioni solamente se ci sono delle informazioni sul buffer seriale. Come primo step viene letta la stringa de seriale tramite *Serial.readString()* successivamente creiamo 3 sottostringhe per poter isolare il comando dai valori che rappresentano il range. In base al comando rilevato andiamo a modificare i relativi valori dei vettori che contengono i range di temperatura utilizzando la funzione *String.toFloat()* per poter convertire i valori di temperatura da stringa a numeri float. In caso il comando non venisse riconosciuto, viene stampato tramite monitor seriale un errore e il formato corretto con cui inserire i dati.



Dopo aver concluso il laboratorio, il gruppo ha deciso di cimentarsi nell’implementazione anche dell’ultimo punto opzionale del laboratorio. Abbiamo dunque provato ad utilizzare il led di colore verde come emulazione di una ‘lampadina smart’ che avrebbe dovuto accendersi e spegnersi con un doppio battito di mani. Il testo non specificava come avremmo dovuto gestire il rilevamento di rumore correlato al rilevamento della presenza, per esempio avremmo potuto assumere per nel caso il sistema non stesse rilevando nessun movimento ma ricevesse il doppio battito di mani allora la presenza venga settata a 1. Nel nostro caso invece abbiamo preferito effettuare una scelta differente in quanto, secondo la nostra opinione, nel caso venga rilevato un doppio suono casuale (un cane che abbaia in modo continuativo, due oggetti che cadono inavvertitamente o due tuoni molto rumorosi) avrebbero potuto attivare la luce che sarebbe rimasta accesa anche mezz’ora nel caso non venisse spenta. Questo è un comportamento che vogliamo evitare sia dal punto di vista energetico (in una notte tempestosa potremmo avere diverse lampade che restano accese per ore) sia dal punto di vista puramente concettuale in quanto è una disfunzione del nostro dispositivo. E’ vero che nel caso le mani venissero battute in un punto cieco del sensore allora la lampadina non avrebbe nessuna reazione ma è vero che in un’ideale casa smart si potrebbero inserire più sensori di movimento in modo da ridurre le zone non visibili dal sensore di movimento.

Dopo questa spiegazione sulle nostre scelte progettuali, mostriamo la funzione *double\_clap()* che viene chiamata nel *loop().*

Come primo passaggio, leggiamo lo stato del sensore di rumore. Nel caso un rumore sia rilevato e se sono passati almeno 200 millisecondi dall’ultimo rumore rilevato (come nel punto 4 che abbiamo mostrato precedentemente) allora possiamo assumere che un nuovo valore è stato rilevato. Successivamente la variabile *SOUND\_counter* viene incrementata per capire se il rumore rilevato è il primo o il secondo battito. Nel caso il contatore sia 1 allora abbiamo appena rilevato un potenziale primo battito e dunque salviamo il tempo in millisecondi nella variabile globale *first\_clap.* Nel caso il contatore sia 2 allora abbiamo rilevato un secondo battito di mani, quindi invertiamo lo stato del led e resettiamo il counter a 0 per poter rilevare successivi battiti. Nel caso tra il primo e il secondo battito passi più di un secondo (scelta effettuata in modo empirico, due battiti separati da più di un secondo di distanza non possono essere ritenuti un doppio battito di mani) allora il doppio battito non verrà confermato e il contatore verrà impostato a 1 in quanto l’attuale battito non può essere considerato un secondo battito ma è valido come possibile primo battito. Come ultimo punto, spegniamo il led quando la variabile *presence* viene impostata a 0, ovvero quando non è più confermata la presenza.



Conclusioni

Lo svolgimento di questo laboratorio è stato decisamente più complesso da gestire rispetto al precedente, in quanto avevamo svariati elementi da far coesistere all’interno di un singolo sistema. Il codice è stato molto complesso da organizzare, a causa delle decine di variabili globali, dalle diverse funzioni che vengono chiamate in diversi punti del codice e dalle oltre 250 righe di codice scritto.

I problemi principali che abbiamo riscontrato sono sicuramente l’utilizzo di *millis()* invece che usare un timer, il quale ci ha richiesto diverso tempo per trovare la soluzione e per verificare che la soluzione per il nostro progetto fosse adeguata. Un’altro aspetto che ci ha preso molto tempo nello sviluppo è stata la corretta impostazione del sensore di rumorosità. Infatti quest’ultimo se non correttamente settato, rileva decine di rumori causati del motore della ventola che non essendo brushless produceva un rumore individuabile dal sensore. Abbiamo anche dovuto evitare l’effetto opposto, ovvero che anche i rumori più forti non venissero rilevati. Nel nostro caso siamo riusciti a trovare un compromesso in quanto i rumori più evidenti come un oggetto che sbatte sul tavolo, una porta che sbatte o una voce ben distinguibile vengono correttamente rilevati del sensore.

Un aspetto abbastanza macchinoso dello sviluppo con Arduino è sicuramente l’assenza di un vero e proprio debugger. Il codice per essere testato deve essere caricato sulla scheda e bisogna successivamente eseguire una serie di verifiche pratiche per controllare ogni possibile situazione che possa avvenire in un contesto reale.

Durante lo svolgimento di ogni punto del laboratorio, il codice e il funzionamento del sistema è stato testato numerose volte e anche la versione finale del codice è stata provata nella pratica ripetutamente per verificare se qualche combinazione di eventi avesse portato a comportamenti inaspettati o errati. I nostri test non hanno rilevato nessun tipo di errore e il sistema smart ha sempre avuto un comportamento corretto. Ovviamente non possiamo assumere che il codice scritto sia perfetto né che il sistema sia capace di rispondere correttamente ad ogni evento ma possiamo in ogni caso ritenerci soddisfatti del risultato ottenuto. Per poter essere sicuri che tutto funzioni come richiesto ci vorrebbero innumerevoli simulazioni e decine di ore di lavoro in condizioni reali per poter considerabile l’impianto fruibile ad un utente finale. Tutto sommato riteniamo che il lavoro da noi svolto sia accettabile per poter proseguire con i laboratori successivi.

Il lavoro all’interno del gruppo è stato ben partizionato in modo da rendere partecipe anche chi non aveva il kit di Arduino, il codice, in modo particolare, è stato curato per l’appunto da chi non ha avuto modo di lavorare direttamente con la scheda.

