Politecnico di Torino

Relazione del laboratorio hardware del corso “Tecnologie per IoT”

Pietro Macori s246163

Alessandro Versace s246056

Ferdinando Micco s245340

ANNO ACCADEMICO 2019-2020

Sommario

Laboratorio 1

Il primo laboratorio del corso "Tecnologie per IoT" è suddiviso in sei esercizi basati sull'utilizzo della scheda Arduino Yún e del linguaggio di programmazione Arduino. Lo scopo del laboratorio è quello di permetterci di prendere confidenza con la scheda elettronica, con i vari sensori e con la scrittura di semplice codice.
Il report è suddiviso in sei parti, ognuna relativa al corrispettivo esercizio, nei quali verrà analizzata l'implementazione del codice.

Esercizio 1

Il primo esercizio ha lo scopo di farci utilizzare la scheda Arduino per controllare l'accensione e lo spegnimento di due led. Entrambi i led saranno collegati in serie con una resistenza da 180Ω e collegati ad Arduino tramite due differenti pin digitali. Il periodo di oscillazione dei due led deve essere indipendente, dunque, abbiamo scelto due periodi primi tra loro ovvero, 7 e 3 secondi.
In particolare, il led rosso deve essere controllato dalla funzione loop e da un delay, mentre il led verde deve essere modificato tramite una ISR che viene chiamata allo scadere di un timer preimpostato.

Lab1.1

```
1  //Lab1.1 - 2 leds alternating using the loop() and an ISR connected to a timer
2  #include <TimerOne.h>
3
4  //Insert as constant the number of the PINs used
5  const int RLED_PIN=12;
6  const int GLED_PIN=11;
7
8  //Define the half period of the two leds
9  const float R_H_PERIOD=1.5;
10 const float G_H_PERIOD=3.5;
11
12 //Set initial state of the leds as LOW
13 int stateG=LOW;
14 int stateR=LOW;
15
16 //ISR function
17 void blinkGreen(){
18   stateG=!stateG; //Invert the green led state
19   digitalWrite(GLED_PIN,stateG);
20 }
21
22 void setup() {
23   pinMode(RLED_PIN,OUTPUT);
24   pinMode(GLED_PIN,OUTPUT);
25   Timer1.initialize(G_H_PERIOD*1e06); //Set the trigger time in microseconds
26   Timer1.attachInterrupt(blinkGreen); //Attach the function blinkGreen() to Timer1
27 }
28
29 void loop() {
30   stateR=!stateR; //Invert the red led state
31   digitalWrite(RLED_PIN,stateR);
32   delay(R_H_PERIOD*1e03);
33 }
```

All'inizio del nostro codice abbiamo definito alcune costanti e alcuni parametri che ci sarebbero serviti successivamente. Nello specifico, abbiamo definito due costanti di tipo intero che rappresentano i due differenti pin utilizzati nel collegamento dei led (11 per il led rosso e il 12 per il led verde) e due costanti di tipo float per rappresentare i semi-periodi di lampeggiamento dei due led. All'interno della funzione setup abbiamo settato i due pin digitali come output, in quanto è la scheda Arduino a mandare i segnali di controllo ai led, e abbiamo inoltre inizializzato il timer che deve effettuare una Interrupt a ogni semiperiodo del led verde (il semiperiodo viene passato al timer in microsecondi). Successivamente abbiamo passato al timer la funzione da eseguire alla scadenza del tempo impostato. La funzione *blinkGreen()* viene infatti chiamata all'effettuarsi della Interrupt; semplicemente inverte lo stato del led verde e successivamente esegue la funzione *digitalWrite()*

per scrivere sul pin dedicato al led verde il nuovo state. Analogamente all'interno del loop viene invertito lo stato del led rosso, viene scritto sul pin il nuovo valore e viene effettuata una funzione di delay di lunghezza pari al semiperiodo del led rosso.
Utilizzando questo codice, il comportamento dei due led sarà analogo, ma il risultato è stato ottenuto tramite due processi differenti.

Esercizio 2

Questo esercizio è nuovamente basato sull'accensione e spegnimento di due led, ma in questo caso è necessario l'utilizzo del monito seriale per poter richiedere lo stato di uno dei due led attraverso l'invio di caratteri all'Arduino. Il programma dovrà quindi leggere se sulla porta seriale ci sono eventuali caratteri di input da parte dell'utente e, in caso affermativo, dovrà capire se il comando è valido e quindi stampare a schermo lo stato del led. Se il carattere letto invece non fosse valido, dovrà restituire un errore.

Lab1.2

```
1  //Lab1.2 - Read the status of two leds using the serial monitor
2  #include <TimerOne.h>
3
4  //Define leds pins
5  const int RLED_PIN = 12;
6  const int GLED_PIN = 11;
7  //Define leds half periods
8  const float R_HALF_PERIOD = 1.5;
9  const float G_HALF_PERIOD = 3.5;
10 //Set initial state of the leds as LOW. Green led is volatile because it is used by both ISR and loop
11 volatile int greenLedState = LOW;
12 int redLedState = LOW;
13
14 //ISR function
15 void blinkGreen() {
16   greenLedState = !greenLedState;
17   digitalWrite(GLED_PIN, greenLedState);
18 }
19
20 //Read from the serial port
21 void serialPrintStatus() {
22   //Read bytes only if there is something in the buffer
23   if (Serial.available() > 0) {
24     //Read 1 byte from the buffer
25     int inByte = Serial.read();
26     //If the read byte corresponds to the ASCII code of 'r'
27     if (inByte == 114) {
28       Serial.println("The red led has status: " + String(redLedState));
29     }
30     //If the read byte corresponds to the ASCII code of 'g'
31     else if (inByte == 103) {
32       Serial.println("The green led has status: " + String(greenLedState));
33     }
34     //If the ASCII doesn't correspond to 'r' or 'g' return error
35     else {
36       Serial.println("Error - command not valid [r=red, g=green]");
37     }
38   }
39 }
40
41 void setup() {
42   Serial.begin(9600);     //Initialize serial comunication
43   while (!Serial);      //Loop won't start untill the serial monitor is open
44   Serial.println("Lab 1.2 Starting");     //Welcome message
45   pinMode(RLED_PIN, OUTPUT);
46   pinMode(GLED_PIN, OUTPUT);
47   Timer1.initialize(G_HALF_PERIOD * 1e06);
48   Timer1.attachInterrupt(blinkGreen);
49 }
50
51 void loop() {
52   digitalWrite(RLED_PIN, redLedState);
53   redLedState = !redLedState;
54   serialPrintStatus(); //The read of the serial port will be synchronous
55   delay(R_HALF_PERIOD * 1e03);
56 }
```

Gran parte del codice scritto è uguale all'esercizio precedente e la spiegazione verrà omessa. Ciò che è stato aggiunto è principalmente la funzione *serialPrintStatus()* che viene chiamata nel loop e, dunque, avrà un comportamento sincrono rispetto al funzionamento del led rosso. Nella funzione, prima di tutto, viene controllato se è presente qualche tipo di dato all'interno del buffer del collegamento seriale. In caso negativo la funzione terminerà e la sua presenza sarà completamente trasparente. Al contrario, se sono presenti dei dati nel buffer, da quest'ultimo viene letto un byte

(dato che i nostri comandi sono basati su caratteri di dimensione 1 byte) usufruendo della funzione *Serial.read().* Nel caso il valore ritornato sia il codice ASCII della lettera 'r', allora verrà stampato sul monitor seriale lo stato del led rosso (nel nostro caso il valore viene convertito in stringa per facilitare la concatenazione e la scrittura più chiara del risultato). Analogamente, se il valore ritornato corrisponde al codice ASCII della lettera 'g', allora verrà stampato lo stato del led verde. Se il valore letto non corrisponde a nessun comando, viene stampato un errore a schermo. È importante notare che in questo esercizio lo stato dei led verde è definito come 'volatile' poiché la variabile di stato è utilizzata sia dal loop che dalla ISR e bisogna dunque evitare che si violi il comportamento *Read-After-Write* sul registro di questa variabile.

Nel setup l'unica differenza rispetto all'esercizio 1 è che definiamo e inizializziamo la connessione seriale. Il parametro 9600 passato a *Serial.begin()* serve per settare il baud-rate della comunicazione. Infine, alla riga successiva impostiamo una *while* in modo da aspettare che venga aperto il monito seriale dall'utente prima di fare eseguire il resto del programma.

Esercizio 3

Questo esercizio ha lo scopo di farci lavorare con il sensore HC-SR501 per rilevare movimenti nelle vicinanze del sensore stesso. Nello specifico, l'obbiettivo è quello di accendere un led al rilevamento di movimento e contare il numero di movimenti rilevati a partire dall'avvio del programma, stampando sul monitor seriale il totale ogni 30 secondi. Per interfacciare il sensore alla scheda Arduino è necessario utilizzare tre pin. Il primo sarà collegato al GRD, un altro ai 5V di alimentazione e l'ultimo ad un pin GPIO che supporti le Interrupt (nel nostro caso abbiamo utilizzato il pin 7).

Lab1.3 §

```
//Lab1.3 - Observe movements using the motion sensor HC-SR501

const int RLED_PIN = 12;      //Set pin of the led
const int PIR_PIN = 7;      //Set pin of the HC-SR501 sensor

int redLedState = LOW;      //Initial state of led set to LOW
volatile int tot_count = 0;     //Volatile because used by ISR and loop()

//ISR that checks if a movement has been detected
void checkPresence() {
  int status = digitalRead(PIR_PIN);      //Read the value of the sensor
  digitalWrite(RLED_PIN, status);      //Set the value of the led equal to the value read from the sensor
  if (status == HIGH) {     //If a rising edge is detected then increase the counter
    tot_count += 1;
  }
}


void setup() {
  Serial.begin(9600);     //Initialize serial monitor
  while (!Serial);    //loop won't start untill serial monitor is open
  Serial.println("Lab 1.3 Starting");     //Welcome message
  pinMode(PIR_PIN, INPUT);      //The sensor will send data to Arduino, so il will be seen as input
  pinMode(RLED_PIN, OUTPUT);
  //Call the function checkPresence() when a change occurs on the value of PIR_PIN
  attachInterrupt(digitalPinToInterrupt(PIR_PIN), checkPresence, CHANGE);
}

void loop() {
  Serial.println(String(tot_count));    //Print the total count every 30s
  delay(30 * 1e03);
}
```

Dopo aver definito le costanti di riferimento per il numero dei pin e aver inizializzato lo stato del led a LOW e il totale dei movimenti rilevati a zero (la variabile è volatile perché verrà utilizzata sia dal loop che dalla ISR del sensore), nel setup inizializziamo la comunicazione con il monitor seriale (in modo del tutto analogo a quanto fatto nell'esercizio precedente), dopo aver lanciato un messaggio di introduzione sul monitor seriale e successivamente settato il pin del led come output e il pin del sensore come input, dato che quest'ultimo invia segnali alla scheda quando un

movimento viene rilevato. Infine, gestiamo i segnali di Interrupt che arrivano dal sensore: usando la funzione *attachInterrupt()* abbiamo imposto di chiamare la funzione *checkPresence()* qualora un qualsiasi cambiamento di stato fosse avvenuto sul pin di riferimento al sensore. Abbiamo scelto di chiamare la funzione ad ogni cambiamento in modo da poter facilmente impostare lo stato del led, il che sarebbe stato decisamente più complesso nel caso avessimo usato l'opzione 'RISING'. All'interno della funzione *checkPresence()*, salviamo l'attuale valore ricevuto dal sensore e impostiamo lo stato del led con il medesimo valore. Infatti, se rileviamo un voltaggio *HIGH* dal sensore, significa che un movimento è stato rilevato e conseguentemente il led dovrà accendersi, nel caso il valore del sensore sarà *LOW*, allora il sensore non sta rilevando nessun movimento e il led dovrà spegnersi. Come ultimo comando eseguiamo un controllo per verificare se ci troviamo in una situazione di *'rising edge'* o *'falling edge'*. Infatti, dato che la ISR verrà richiamata per ogni variazione di segnale, è nostro compito verificare che ad ogni *'rising edge'* corrisponda un aumento del conteggio. All'interno del loop viene ciclicamente stampato sul monitor seriale il totale dei movimenti rilevati seguito da un delay di 30 secondi.

Esercizio 4

L'obiettivo di questa parte del laboratorio è quello di utilizzare un motore a corrente continua (nel nostro caso il motore DFR0332) la cui velocità di rotazione può essere modificata tramite appositi comandi trasmessi ad Arduino tramite seriale. La particolarità di questo esercizio è l'utilizzo di un componente che funziona tramite PWM (pulse-width modulation), la cui intensità può essere modificata utilizzando un pin digitale, cambiando il duty-cycle del segnale. E' importante notare che sebbene nel codice si utilizzi la funzione *analogWrite()*, il pin utilizzato sarà un pin digitale con la particolarità di supportare la PWM (nel nostro caso abbiamo utilizzato il pin 6).

Lab1.4

```
//Lab 1.4 - Modify the speed of a DC motor using the serial port
const int FAN_PIN = 6;      //Pin of the DC motor
float current_speed = 0;      //Speed starts from 0
const float inc = 25.5;     //Increment constant

void changeSpeed() {
  if (Serial.available() > 0) {
    int signal = Serial.read();
    if (signal == 43) {    //If read the ASCII value of '+'

      if (current_speed < 255) {     //If speed is not at the maximum
        current_speed += inc;     //Increment speed
        analogWrite(FAN_PIN, (int) current_speed);      //Write on the motor pin the new speed
        Serial.println("Increase speed: " + String(current_speed));
      }
      else {      //If speed at maximum (255)
        Serial.println("Already at maximum!");
      }

    }

    else if (signal == 45) {      //If read the ASCII value of '-'
      if (current_speed > 0) {      //If speed is not zero
        current_speed -= inc;     //Decrease speed
        analogWrite(FAN_PIN, (int) current_speed);
        Serial.println("Decrease speed: " + String(current_speed));
      }
      else {      //If speed already at zero
        Serial.println("Already at minimum!");
      }
    }
    else {      //Value read is not + or -
      Serial.println("Error - command not valid [+,-]");
    }
  }
}

void setup() {
  Serial.begin(9600);     //Initialize serial comunication
  while (!Serial);
  Serial.println("Lab 1.4 Starting");     //Welcome
  pinMode(FAN_PIN, OUTPUT);
  analogWrite(FAN_PIN, (int) current_speed);      //Set the initial speed
}

void loop() {
  changeSpeed();
}
```

Per prima cosa definiamo le costanti relative al pin del motore e dello step di incremento utilizzate poi nella funzione *changeSpeed()* (noi abbiamo scelto 25.5 in modo da ottenere 10 step di incremento) e settiamo la variabile della velocità a zero in modo da evitare che il motore inizi a ruotare all'avvio.

Dopo le inizializzazioni viste precedentemente della comunicazione seriale e dei vari pin, usiamo la funzione *analogWrite()* per impostare la velocità di partenza del motore. All'interno del loop chiamiamo semplicemente la funzione *changeSpeed()* che gestisce le varie situazioni.

Quando *changeSpeed()* viene eseguita, controlliamo se ci sono byte disponibili da leggere nel buffer, in caso negativo la funzione ritorna e nulla viene eseguito. In caso contrario, viene letto un byte dal buffer in modo da leggere un carattere, se il valore letto dal buffer è il corrispettivo del codice ASCII del carattere '+' allora viene effettuato un controllo per verificare che la velocità sia minore di 255 ovvero la velocità massima, e in questo caso si aumenta il valore della velocità di uno step e viene inviato il valore al motore tramite *analogWrite()*. In modo analogo si verifica se il carattere nel buffer è il carattere '-' e si controlla che la velocità non sia già a zero. Nel caso questo non avvenga, allora la velocità viene ridotta di uno step. In caso di caratteri non validi o in caso di tentativi di aumentare oltre la velocità massima o ridurre al di sotto della velocità minima, un errore viene stampato sul monitor seriale.

Esercizio 5

In questo esercizio abbiamo usato Arduino e un sensore di temperatura (Grove) per rilevare la temperatura ambientale. La principale difficoltà di questo esercizio è la conversione dei valori ricevuti dal sensore in una temperatura in gradi Celsius. Il valore della temperatura deve nuovamente essere stampato sul monitor seriale. Il sensore in questione funziona tramite un termistore, la cui resistenza cambia al variare della temperatura. Il sensore è basato su un partitore di tensione. Utilizzando queste informazioni unite alla relazione non lineare che lega la resistenza alla temperatura (che è possibile leggere sul datasheet del sensore), si può ricavare la temperatura in gradi Kelvin che potrà essere facilmente convertita in gradi Celsius. Il processo dovrà essere diviso in due passaggi: prima dovremo ricavare il valore della resistenza utilizzando la formula inversa del partitore di tensione e successivamente usare la definizione del parametro B per ottenere la temperatura. Bisogna, inoltre, tenere conto che la scheda Arduino utilizza un convertitore analogico-digitale con una risoluzione a 10 bit, quindi nelle formule di conversione, il valore massimo di tensione 5V dovrà essere scritto come 1023.

Lab1.5

```
//Lab1.5 - Read of the temperature using the Grove sensor

#include <math.h>

//Define the analog pin of the sensor
const int TMP_PIN = A1;

//Define all the constants needed for the conversion
const float B = 4275.0;
const long int R0 = 100000;
const float T0 = 298.0;

void setup() {
  Serial.begin(9600);     //Initialize serial port
  while (!Serial);
  Serial.println("Lab 1.4 Starting");     //Welcome
  pinMode(TMP_PIN, INPUT);      //The sensor will send data to the Arduino
}

void loop() {
  //Conversion steps
  float val = analogRead(TMP_PIN);
  float R = (1023.0 / val - 1) * R0;
  float T = 1 / (log(R / R0) / B + 1 / T0);

  //Convert from K to C
  float Tfin = T - 273.15;
  Serial.println("La temperatura e' di " + String(Tfin) + " °C");     //Print of serial monitor the value
  delay(10000);
}
```

All’inizio del codice abbiamo definito il valore del pin analogico utilizzato dal sensore di temperatura e abbiamo definito come costanti tutti i parametri che serviranno nella conversione da voltaggio a temperatura. Nel setup abbiamo inizializzato la comunicazione seriale, stampato un messaggio iniziale e settato il pin del sensore come input, dato che Arduino riceve il segnale dal sensore.

Nel loop eseguiamo tre passaggi: otteniamo il valore della resistenza del sensore tramite la formula del partitore di tensione, con il valore ottenuto ricaviamo il valore della resistenza in Kelvin e convertiamo da gradi Kelvin a gradi Celsius. Il passaggio finale, infine, consiste nello stampare la temperatura sul monitor seriale ed effettuare un delay di 10 secondi.

Esercizio 6

Questo esercizio è analogo al precedente, tuttavia viene richiesto di mostrare la temperatura su un display LCD anziché tramite il monitor seriale. Il display utilizzato in questo laboratorio è il display *DFRobot* collegato ad Arduino tramite connessione I2C. Per interfacciare il display abbiamo utilizzato la libreria esterna ‘LiquidCrystal_PCF8574’.

Lab1.6

```
//Lab1.6 - Read of temperature and show it on LCD screen
#include <LiquidCrystal_PCF8574.h>
#include <math.h>

LiquidCrystal_PCF8574 lcd(0x27);      //Define the address of the monitor for the I2C connection

const int TMP_PIN = A1;     //Define the pin of the temperature sensor

//Define all conversion constant
const int B = 4275;
const long int R0 = 100000;
const float Vcc = 1023.0;
const float T0 = 298.0;

void setup() {
  //Set up of the display
  lcd.begin(16, 2);
  lcd.setBacklight(255);
  lcd.home();
  lcd.clear();
  lcd.print("Temperature:");
  pinMode(TMP_PIN, INPUT);
}

void loop() {
  //Conversion to temperature
  float val = analogRead(TMP_PIN);
  float R = (Vcc / val - 1) * R0;
  float T = 1 / (log(R / R0) / B + 1.0 / T0);
  float Tfin = T - 273.15;

  //In order not to re-write every time "Temperature:" we set the curson to overwrite only the temperature value
  lcd.setCursor(12, 0);
  lcd.print(Tfin);
  delay(10000);
}
```

La parte fondamentale di questo codice è la definizione dell'indirizzo del display. Infatti, il protocollo di comunicazione del display è I2C, basato su master e slave, quindi il master (Arduino) deve conoscere l’indirizzo dello slave (il display). Nel nostro caso l’indirizzo è 0x27. Dunque dobbiamo impostare il display. Nel setup il display è inizializzato con 16 colonne e 2 righe, settato alla massima luminosità, lo schermo viene ripulito da eventuali scritte e viene stampato il testo “Temperature:”. Il pin del sensore di temperatura è nuovamente visto come input.

All’interno del loop, oltre alle varie conversioni per ottenere la temperatura, usiamo la funzione *setCursor()* in modo da poter riscrivere sul display solo il nuovo valore e non tutto il testo “Temperature: valore” in questo modo possiamo risparmiare l’invio di bit al display. Infine, settiamo un delay di 10 secondi.

Laboratorio 2

Introduzione

L'obiettivo di questo laboratorio è quello di progettare e costruire uno “smart home controller” tramite l’utilizzo di una scheda Arduino Yún, di vari sensori, di 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 rilevare la presenza o assenza di persone nell’ambiente circostante. 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 Yún 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. 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 e 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 verrà impostata a 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.

$$valore\ velocità = \frac{255 * (temp\ misurata - min\ Temp)}{(max\ Temp - min\ Temp)}$$

Il valore ottenuto verrà inviato 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, visto che dovrà essere visualizzato sul display LCD.

```
100 //Regulate fan speed or led color depending on the temperature
101 int regulate_fan_or_led(int PIN, int min, int max) {
102   float temp = actualTemp();
103   if ((temp >= min && temp <= max && PIN == FAN_PIN) || (temp <= min && temp >= max && PIN == LED_PIN)) {
104     int value = (int) MAX_SPEED / (max - min) * (temp - min);
105     analogWrite(PIN, value);
106     return 100 * value / 255;
107   }
108   else if ((PIN == FAN_PIN && temp < min) || (PIN == LED_PIN && temp > min)) {
109     analogWrite(PIN, 0);
110     return 0;
111   }
112   else {
113     analogWrite(PIN, MAX_SPEED);
114     return 100;
115   }
116 }
```

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, abbiamo potuto 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 abbiamo ottenuto 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. Se viene rilevato un movimento, il dispositivo deve dedurre che ci sia una persona nei paraggi e deve assumere che se per un tempo preimpostato (30 minuti nel nostro caso) non vengono rilevati movimenti, allora non c'è nessuna persona nell'ambiente circostante.

Noi abbiamo deciso di sfruttare il pin digitale che supporta gli *Interrupt* (il pin digitale 7 per la scheda Yún) 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: 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 variabile a 0. Nel caso venga rilevato un nuovo movimento, la funzione *checkPresencePIR()* verrà nuovamente chiamata e di consequenza anche il tempo di timeout verrà reimpostato. È importante notare che

le variabili *PIR_presence* e *timeStartMovement* sono entrambe definite come 'volatile' perchè vengono usate sia nel loop che all'interno di una ISR.

Durante lo svolgimento di questo punto abbiamo deciso di usare la funzione *millis()* poiché l'idea iniziale 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.

```
128 // Function called by the ISR of the PIR every time a movement is detected
129 void checkPresencePIR() {
130   PIR_presence = 1;
131   timeStartMovement = millis();
132 }
```

```
76  //If from the last movement 15 sec are passed, then turn to 0 the PIR_presence
77  if (millis() - timeStartMovement > PIR_timeout) {
78    PIR_presence = 0;
79  }
```

Il punto successivo richiedeva l'utilizzo di un sensore di rumore per poter compensare le "zone di ombra" del sensore PIR. Dato che il sensore non era in grado di rilevare l'origine del rumore, è 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 eseguendo vari test per trovare un valore che fosse soddisfacente per i nostri scopi) per renderli validi, sono stati "filtrati" 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 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 tra il 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 dedurre che ci sia una persona. Quando avviene quest'ultimo caso, viene salvato un tempo in millisecondi 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 che viene 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.

```
140 //The SOUND sensor has to check if a number of events happen in a given period of time
141 void checkPresenceSOUND() {
142   int status_sound = digitalRead(SOUND_PIN);
143   unsigned long now = millis();
144   if (status_sound == HIGH && (now - vetSounds[min_n_events - 2]) > 200) {
145     unsigned long first = vetSounds[0];
146     memcpy(vetSounds, &vetSounds[1], sizeof(vetSounds) - sizeof(unsigned long));
147     vetSounds[min_n_events - 2] = now;
148     if (now - first < SOUND_interval && SOUND_counter > min_n_events - 2) {
149       SOUND_presence = 1;
150       startFlag = millis();
151     }
152     else {
153       SOUND_counter++;
154     }
155   }
156   if (millis() - startFlag > SOUND_timeout && SOUND_presence == 1) {
157     SOUND_presence = 0;
158   }
159 }
```

Ora che, sia la variabile *PIR_presence* che *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.

```
90   //If one of the two sensor detect presence then the variable presence is set to 1
91   if (SOUND_presence == 1 || PIR_presence == 1) {
92     presence = 1;
93   }
94   else {
95     presence = 0;
96   }
```

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 verranno associati due vettori ciascuno, uno relativo alla temperatura massima e uno alla temperatura minima. Ognuno di questi vettori ha due spazi di memoria. Il primo (associato alla posizione 0) si riferisce al caso di assenza, mentre il secondo (associato alla posizione 1) si riferisce 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.

```
21 // Temp Range Fan
22 float FTEMP_MIN[] = {25, 20};
23 float FTEMP_MAX[] = {30, 25};
24
25 // Temp Range Led
26 float LTEMP_MIN[] = {10, 15};
27 float LTEMP_MAX[] = {15, 20};
```

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 passo 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 funzione 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 assenza di persone. Inizialmente la nostra idea era 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 inappropriato poiché 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 fossero passati i 5 secondi. Per poter cambiare schermata a 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.

```
197 //Every 5 second update the display
198 void changeDisplay(int now, int perc_AC, int perc_HT) {
199   if (now - lastChangeDisplay > 5000) {
200     lastChangeDisplay = millis();
201     if (num_display == 0) {
202       num_display = 1;
203       lcd.home();
204       lcd.clear();
205       lcd.print("T:");
206       float temp = round(actualTemp());
207       lcd.print(temp);
208       lcd.print(" Pres:" + String(presence));
209       lcd.setCursor(0, 1);
210       lcd.print("AC:" + String(100 * perc_AC / 255) + "% HT:" + String(100 * perc_HT / 255) + "%");
211     }
212     else {
213       num_display = 0;
214       lcd.home();
215       lcd.clear();
216       lcd.print("AC m:");
217       lcd.print(FTEMP_MIN[presence], 1);
218       lcd.print(" M:");
219       lcd.print(FTEMP_MAX[presence], 1);
220       lcd.setCursor(0, 1);
221       lcd.print("HT m:");
222       lcd.print(LTEMP_MIN[presence], 1);
223       lcd.print(" M:");
224       lcd.print(LTEMP_MAX[presence], 1);
225     }
226   }
227 }
```

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 stia modificando e aiutano gli 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) sia costante, e renda in questo modo molto facile poter leggere i vari campi semplicemente creando delle "substring". È dunque possibile cambiare ogni impostazione relativa al led o alla ventola, in presenza o in 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 soggetti.

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 da seriale tramite *Serial.readString()* successivamente creiamo 3 sottostringhe per poter isolare il comando dai valori che rappresentano il range. In base al comando rilevato, vengono modificati 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.

```
164 void serialSet_Point() {
165   if (Serial.available() > 0) {
166     String signal = Serial.readString();
167     String command = signal.substring(0, 14);
168     String range1 = signal.substring(15, 17);
169     String range2 = signal.substring(18);
170 
171     if (command.equals("RANGE_FAN_PRES")) {
172       FTEMP_MIN[1] = range1.toFloat();
173       FTEMP_MAX[1] = range2.toFloat();
174       Serial.println("Range of the FAN in case of PRESENCE has beeen successfully updated!");
175     }
176     else if (command.equals("RANGE_FAN_ABSN")) {
177       FTEMP_MIN[0] = range1.toFloat();
178       FTEMP_MAX[0] = range2.toFloat();
179       Serial.println("Range of the FAN in case of NO PRESENCE has beeen successfully updated!");
180     }
181     else if (command.equals("RANGE_LED_PRES")) {
182       LTEMP_MIN[1] = range1.toFloat();
183       LTEMP_MAX[1] = range2.toFloat();
184       Serial.println("Range of the LED in case of PRESENCE has beeen successfully updated!");
185     }
186     else if (command.equals("RANGE_LED_ABSN")) {
187       LTEMP_MIN[0] = range1.toFloat();
188       LTEMP_MAX[0] = range2.toFloat();
189       Serial.println("Range of the LED in case of NO PRESENCE has beeen successfully updated!");
190     }
191     else {
192       Serial.println("Command Error!");
193       Serial.println("Please insert a command of the form RANGE_[FAN|LED]_[PRES|ABSN]=MIN/MAX");
194     }
195   }
196 }
```

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. Dato che il testo non specificava come avremmo dovuto gestire il rilevamento di rumore correlato al rilevamento della presenza, abbiamo deciso di non associare al doppio battito anche il rilevamento di presenza. Questo perché 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) si sarebbe potuta attivare la luce che sarebbe rimasta accesa anche mezz'ora se non fosse stata successivamente 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. È vero che, nel caso le mani venissero battute in un punto cieco del sensore di movimento allora la lampadina non avrebbe nessuna reazione, ma è anche vero che in un'ideale casa smart si potrebbero inserire più sensori di movimento in modo da ridurre i punti ciechi.

Dopo questa spiegazione sulle nostre scelte progettuali, mostriamo la funzione *double_clap()* che viene chiamata nel *loop().* Innanzitutto, leggiamo lo stato del sensore di rumore. Nel caso si rilevi un rumore e questo avvenga almeno 200 millisecondi dall'ultimo rumore percepito (come nel punto 4 che abbiamo mostrato precedentemente) allora possiamo dedurre che un nuovo valore è stato rilevato. Successivamente, viene incrementata la variabile *SOUND_counter* 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 percepire successivi battiti. Nel caso tra il primo e il secondo battito passi più di un secondo (scelta effettuata in modo arbitrario, 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.

```
232 void double_clap() {
233   int status_sound = digitalRead(SOUND_PIN);
234   if (status_sound == HIGH && (millis() - last_clap) > 200) {
235     SOUND_counter += 1;
236     if (SOUND_counter == 1) {
237       first_clap = millis();
238     }
239     if (SOUND_counter == 2) {
240       int status_led_green = digitalRead(PIN_LED_GREEN);
241       digitalWrite(PIN_LED_GREEN, !status_led_green);
242       SOUND_counter = 0;
243     }
244     if (millis() - first_clap > 1000) {
245       SOUND_counter = 1;
246     }
247     last_clap = millis();
248   }
249
250   int status_led_green = digitalRead(PIN_LED_GREEN);
251   if (PIR_presence == 0 && status_led_green == HIGH) {
252     status_led_green = LOW;
253     digitalWrite(PIN_LED_GREEN, status_led_green);
254   }
255 }
```

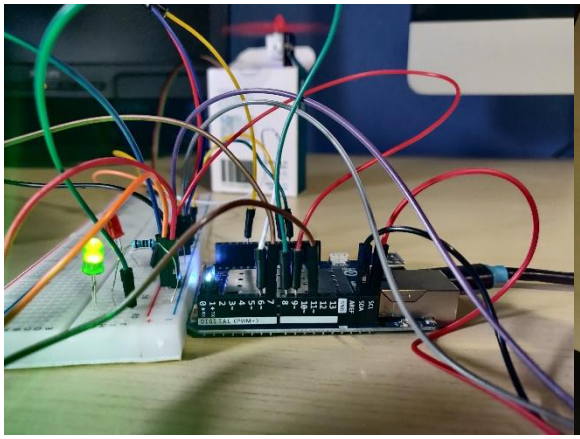
Conclusioni

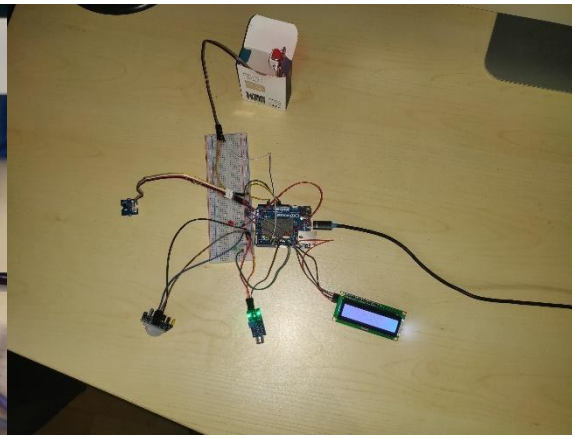
Lo svolgimento di questo laboratorio è stato decisamente più complesso da gestire rispetto al precedente, poiché avevamo svariati elementi da far coesistere all'interno di un singolo sistema. Il codice è stato molto complesso da organizzare, a causa delle decine di variabili globali, delle diverse funzioni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

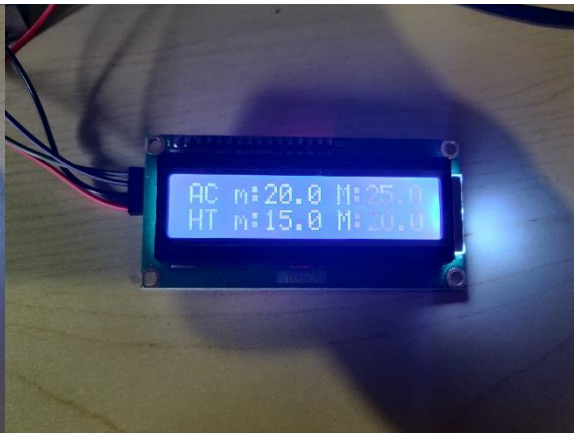
Uno dei problemi principali che abbiamo riscontrato è stato sicuramente l'utilizzo di *millis(),* anziché di un timer. il quale ci ha richiesto molto tempo per verificare che fosse una soluzione efficace per il nostro progetto. Un altro aspetto che ci ha preso molto tempo nello sviluppo è stata la corretta 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 fare in modo che i rumori più evidenti, come un oggetto che cade sul tavolo, una porta che sbatte o una voce ben distinguibile, vengano 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 sono stati testati 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 errore e il sistema smart ha sempre avuto un comportamento corretto. Ovviamente non possiamo affermare 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 considerare l'impianto fruibile a un utente finale. Nel complesso, riteniamo che il lavoro da noi svolto sia adeguato 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. Infatti, il codice è stato curato da chi non ha avuto modo di lavorare direttamente con la scheda.







Laboratorio 3

La terza parte del laboratorio hardware del corso è basato sulla comunicazione della scheda Arduino Yún con un network tramite interfacce REST e MQTT.

Esercizio 1

Nel primo esercizio è richiesta l'implementazione di uno sketch per la Yún che esegua un server HTTP in grado di rispondere a richieste GET provenienti dalla rete locale. Sulla scheda sono connessi un led e un sensore di temperatura, che verranno utilizzati alla ricezione di un'opportuna richiesta GET dal server. A sua volta la scheda dovrà restituire una risposta tramite il body, in formato JSON che rispetti lo standard senML. Inoltre, dovrà essere possibile settare lo stato del led attraverso la medesima richiesta.

Esempio di richiesta per il led:
http://<hostname>:<porta>/arduino/led/1

Esempio di richiesta per il sensore di temperatura:
http://<hostname>:<porta>/arduino/temperature

Per interfacciare la scheda abbiamo fatto uso delle librerie *Bridge.h*, *BridgeServer.h* e *BridgeClient.h* per poter esporre le risorse e poter gestire le richieste GET ricevute. Inoltre, come suggerito, abbiamo fatto uso della libreria *ArduinoJson.h* e dei documenti dinamici JSON per lavorare agilmente con questo formato.

```
#include <Bridge.h>
#include <BridgeServer.h>
#include <BridgeClient.h>
#include <ArduinoJson.h>

BridgeServer server;

int const INT_LED_PIN = 13;
int const LED_PIN = 9;
const int capacity = JSON_OBJECT_SIZE(2) + JSON_ARRAY_SIZE(1) + JSON_OBJECT_SIZE(4) + 40;
DynamicJsonDocument doc_snd(capacity);

// TEMP
const int TMP_PIN = A1;
const int B = 4275;
const long int R0 = 100000;
const float Vcc = 1023.0;
const float T0 = 298.0;

void setup() {
  pinMode(TMP_PIN, INPUT);
  pinMode(LED_PIN, OUTPUT);
  pinMode(INT_LED_PIN, OUTPUT);
  digitalWrite(INT_LED_PIN, LOW);
  Bridge.begin();
  digitalWrite(INT_LED_PIN, HIGH);
  server.listenOnLocalhost();
  server.begin();
}
```

Nel *loop()* abbiamo predisposto il server alla ricezione di richieste HTTP GET, che mettono in esecuzione la funzione *process().* In essa avviene la gestione dei principali errori sul formato della richiesta. In caso di errore, viene ritornato il codice di errore 400. Viene inoltre effettuato il

riconoscimento della richiesta (led o temperatura), e nel primo caso viene settato il valore ricevuto come nuovo stato, e in entrambi i casi viene lanciata una risposta attraverso la *printResponse()*.

```
void loop() {
  BridgeClient client = server.accept();

  if (client) {
    process(client);
    client.stop();
  }

  delay(50);

}

// Parse requests recived
void process(BridgeClient client) {
  String command = client.readStringUntil('/');
  command.trim();

  if (command == "led" || command == "temperature") {

    // LED
    if (command == "led") {
      int val = client.parseInt();
      if (val == 0 || val == 1) {
        digitalWrite(LED_PIN, val);
        printResponse(client, 200, senMlEncode("led", val, ""));
      } else {
        printResponse(client, 400, "Error");
      }
    }

    // TEMP
    if (command == "temperature") {
      printResponse(client, 200, senMlEncode("temperature", actualTemp(), "Cel"));
    }

  } else {
    printResponse(client, 400, "Error 404 - URI not found!");
  }
}
```

Per completezza, alleghiamo anche una immagine relativa alla creazione del messaggio di risposta.

```
// Create response to the request recived
void printResponse(BridgeClient client, int code, String body) {
  client.println("Status: " + String(code));
  if (code == 200) {
    client.println("Content-type: application/json; charset=utf-8");
    client.println();
    client.println(body);
  }
  else if (code == 400) {
    client.println("Content-type: application/json; charset=utf-8");
    client.println();
    client.println(body);
  }
}
```

Non ci soffermiamo sul funzionamento della funzione *actualTemp()* per il valore della temperatura, che è la stessa utilizzata già nei precedenti laboratori per effettuare il calcolo del valore corrente rispetto all'input ricevuto dal sensore. La funzione *senMlEncode()* utilizzata per entrambe le risposte si occupa della costruzione del documento JSON secondo lo standard senML, inserendo negli opportuni campi il nome del sensore ("temperature" o "led"), il valore attuale, il

timestamp e relativa unità di misura nel caso della temperatura. Infine, viene convertito il contenuto del file JSON in stringa per poterlo mandare come risposta.

Esempio formato senML:

```
{
   “bn”: “Yún”
   “e”: [{
            “n”: <”temperature”>/<”led”>,
            “t”: <timestamp using millis()>,
            “v”: value,
            “u”: “Cel”/null
          }
        ]
}
```

```
86 // Encode in senMl using a dynamic Json Document and Arduino Json
87 String senMlEncode(String res, float v, String unit) {
88   doc_snd.clear();
89
90   doc_snd["bn"] = "Gruppo 13";
91   doc_snd["e"][0]["n"] = res;
92   if (unit != "") {
93     doc_snd["e"][0]["u"] = unit;
94   } else {
95     doc_snd["e"][0]["u"] = (char*) NULL;
96   }
97
98   doc_snd["e"][0]["t"] = millis();
99   doc_snd["e"][0]["v"] = v;
100
101  String output;
102  serializeJson(doc_snd, output);
103  return output;
104 }
```

Esercizio 2

Il secondo esercizio consiste nella modifica del server basato su *cherrypy* descritto in uno dei precedenti laboratori, che si occupava della conversione dei valori di temperatura ricevuti a seconda delle unità di misura. La modifica sarà utile a gestire le richieste HTTP POST periodiche inviate da Arduino contenenti le informazioni sulla temperatura, sempre secondo il formato JSON compatibile con senML. La scheda invierà i dati facendo uso di una shell gestita dal processore della Yún con distribuzione Linux, attraverso il comando *curl*. Inoltre, il server esporrà anche un metodo HTTP GET che restituirà una lista in formato JSON con tutti i logs ricevuti fino a quel momento.

Esempio richiesta POST:
URI: http://<hostname>:<port>/log

```ino
void loop() {
  code = postRequest(senMlEncode("Temperature", (float)((int)(actualTemp() * 100))/100, "Cel"));
  if(code == 1) {
    Serial.println("Errore curl command");
  }
  Serial.println(code);
  delay(1000);
}

// Send POST request using curl on Yun linux chip.
int postRequest(String data) {
  Process p;
  p.begin("curl");
  p.addParameter("-H");
  p.addParameter("Content-Type: application/json");
  p.addParameter("-X");
  p.addParameter("POST");
  p.addParameter("-d");
  p.addParameter(data);
  p.addParameter(url);
  p.run();

  return p.exitValue(); // Status curl command
}
```

L'unica differenza sostanziale sullo sketch consiste nella gestione della richiesta tramite *curl*, che viene effettuata tramite la libreria *Process.h*, la quale permette di scrivere direttamente un comando di shell tramite i metodi *begin()* e *addParameter()*. La stringa passata alla funzione *postRequest()* è l'output della stessa funzione senMlEncode utilizzata nell'esercizio precedente per la costruzione del JSON di risposta. Inoltre, abbiamo aggiunto un controllo sull'exit code di *curl* per poter inviare un eventuale errore sul seriale.
Il server è stato modificato esponendo una nuova classe *Log()* sul percorso "log/". Questa classe gestisce le richieste POST e GET e, inoltre, rende raggiungibile il server da rete locale in modo che la scheda Arduino possa inviargli le informazioni sulla temperatura.

```python
cherrypy.config.update({'server.socket_host':'0.0.0.0'})
cherrypy.config.update({'server.socket_port':8080})
```

```python
class Log():
    logs = []
    exposed = True
    def GET(self, *uri, **params):
        r=json.dumps(self.logs)
        return r

    def POST(self, **params):
        d = cherrypy.request.body.read()
        j=json.loads(d)
        self.logs.append(j)
```

Al ricevimento del JSON a seguito della POST si fa una lettura di questo come dizionario e lo inseriamo in una lista *logs* in comune per tutte le classi *Log()*. La GET farà il processo inverso, convertendo in stringa la lista di logs in json e ritornandola.

Esercizio 3

Il terzo esercizio, similmente al primo, richiede di poter ricevere periodicamente il log in formato JSON compatibile con senML, la temperatura rilevata dal sensore e di poter settare a distanza lo stato del led, il tutto utilizzando un paradigma Publish/Subscribe con MQTT e il broker Mosquitto. Abbiamo utilizzato la libreria *MQTTclient.h* fornitaci per poter implementare le funzioni di *publish* e di *subscribe* ai topic corretti e abbiamo inoltre installato le utility per poterli effettuare sempre tramite una shell Linux.

```
22 void setup() {
23   pinMode(TMP_PIN, INPUT);
24   pinMode(LED_PIN, OUTPUT);
25   digitalWrite(LED_PIN, LOW);
26   Serial.begin(9600);
27   while(!Serial);
28   digitalWrite(INT_LED_PIN, LOW);
29   Bridge.begin();
30   digitalWrite(INT_LED_PIN, HIGH);
31   mqtt.begin("test.mosquitto.org", 1883);
32   mqtt.subscribe(my_base_topic + String("/led"), setLedValue);
33 }
34
35 void loop() {
36   mqtt.monitor();
37   String message = senMlEncode("Temperature", (float)((int)(actualTemp() * 100))/100, "Cel");
38   mqtt.publish(my_base_topic + String("/temperature"), message);
39
40   delay(1000);
41 }
```

Nel *setup()* abbiamo inizializzato la connessione col broker *test.mosquitto.org* tramite la libreria MQTT e abbiamo effettuato una *subscribe* al topic "*/tiot/13/led*" per poter ricevere eventuali cambi di stato del led. Nel loop, oltre a monitorare periodicamente nuovi messaggi al topic a cui siamo sottoscritti, abbiamo inviato tramite una *publish* un messaggio JSON in formato senML con la funzione degli esercizi precedenti. Molto interessante è analizzare la funzione *monitor()*, che richiama la funzione di callback che decodifica i dati nel solito formato utilizzato, per poi poter settare lo stato del led come richiesto.

```
44 void setLedValue(const String& topic, const String& subtopic, const String& message) {
45   Serial.println("eccoci");
46   DeserializationError err = deserializeJson(doc_rec, message);
47   if(err) {
48     Serial.print("deserializeJson() failed with code ");
49     Serial.println(err.c_str());
50   }
51   if(doc_rec["e"][0]["n"] == "led") {
52     if(doc_rec["e"][0]["t"] == (char *)NULL && doc_rec["e"][0]["u"] == (char *)NULL) {
53       int statusLed = doc_rec["e"][0]["v"];
54       if(statusLed == 0 || statusLed == 1) {
55         digitalWrite(LED_PIN, statusLed);
56       }else {
57         Serial.println("Valore non supportato.");
58       }
59     }else {
60       Serial.println("Timestamp o unità di misura non supportate.");
61     }
62   }else {
63     Serial.println("Elemento non supportato.");
64   }
65 }
```

Abbiamo utilizzato la funzione *deserializeJson()* sul messaggio ricevuto inserendolo nel nostro documento JSON e gestendo eventuali errori segnalandoli sul seriale. Dopo aver controllato i vari campi del documento, abbiamo estrapolato lo stato del led che ci veniva inviato, per poi farne la scrittura digitale sul pin corretto relativo al led.

Molto interessante è stato inoltre verificarne il funzionamento durante lo svolgimento del laboratorio a distanza, potendo ricevere i valori di temperatura su qualsiasi pc e potendo settare lo stato del led anche non in presenza.