

Laboratori Tecnologie per IoT
a.a. 2019/20



Cutaia Angelo, s180368
Tancredi Claudio, s244809

Indice

Laboratorio HW	4
1 Introduzione all'ambiente Arduino e primi semplici sketch	4
1.1 Pilotaggio di LED	4
1.1.1 Componenti HW	4
1.1.2 Realizzazione HW	5
1.1.3 Codice Arduino	5
1.2 Comunicazione tramite Porta Seriale	6
1.2.1 Componenti HW	6
1.2.2 Realizzazione HW	6
1.2.3 Codice Arduino	6
1.2.4 Output su Serial Monitor	8
1.3 Identificazione della presenza tramite Sensore PIR	8
1.3.1 Componenti HW	8
1.3.2 Realizzazione HW	9
1.3.3 Codice Arduino	9
1.3.4 Output su Serial Monitor	10
1.4 Controllo di un motore (ventola) tramite PWM	11
1.4.1 Componenti HW	11
1.4.2 Realizzazione HW	11
1.4.3 Codice Arduino	11
1.4.4 Output su Serial Monitor	13
1.5 Lettura di valori analogici (temperatura)	13
1.5.1 Componenti HW	13
1.5.2 Realizzazione HW	14
1.5.3 Analisi circuitale	14
1.5.4 Codice Arduino	15
1.5.5 Output su Serial Monitor	16
1.6 Comunicazione con uno smart actuator (display LCD) tramite protocollo I2C	17
1.6.1 Componenti HW	17
1.6.2 Realizzazione HW	17
1.6.3 Codice Arduino	18
1.6.4 Output su display LCD	19

2	Smart home controller (versione "locale")	20
2.1	Prima versione	20
2.1.1	Componenti HW	20
2.1.2	Realizzazione HW	21
2.1.3	Codice Arduino (diviso in porzioni e commentato)	21
2.1.4	Criticità riscontrate e soluzioni proposte	31
2.1.5	Valori usati per il debug	34
2.1.6	Output su display e Serial Monitor	35
2.2	Seconda versione	36
2.2.1	Componenti HW	36
2.2.2	Realizzazione HW	36
2.2.3	Codice Arduino (esteso)	36
2.2.4	Criticità riscontrate e soluzioni proposte	43
2.2.5	Valori usati per il debug	44
2.2.6	Video dimostrativo	44
3	Comunicazione tramite interfacce REST e MQTT	45
3.1	Arduino Yún come server HTTP	45
3.1.1	Componenti HW	45
3.1.2	Realizzazione HW	45
3.1.3	Codice Arduino	45
3.1.4	Output	48
3.2	Arduino Yún come client HTTP	49
3.2.1	Componenti HW	49
3.2.2	Realizzazione HW	49
3.2.3	Codice Arduino	49
3.2.4	Output	51
3.3	Arduino Yún come publisher e subscriber MQTT	51
3.3.1	Componenti HW	51
3.3.2	Realizzazione HW	51
3.3.3	Codice Arduino	51
3.3.4	Output	54

Laboratorio HW

Capitolo 1

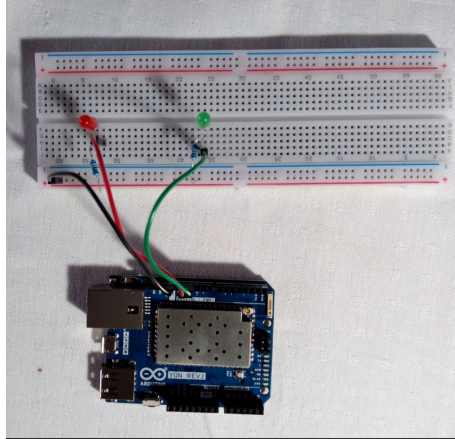
Introduzione all'ambiente Arduino e primi semplici sketch

1.1 Pilotaggio di LED

1.1.1 Componenti HW

- 1 x Arduino Yún Rev 2
- 1 x Breadboard
- 2 x LED
- 2 x Resistenze ($180 \pm 1\%$) Ω
- 3 x Cavi m/m

1.1.2 Realizzazione HW



Il circuito è stato realizzato collegando i pin D11 (cavo verde-LED verde) e D12 (cavo rosso-LED rosso) all'anodo di ciascun LED; il catodo di ogni LED è stato messo in serie con una resistenza da $180\ \Omega$ verso massa. Questo perché il LED altri non è che un diodo, quindi si rende necessario l'uso della resistenza per limitare la quantità di corrente che lo attraversa così da evitare il cortocircuito tra VCC e GND. La scelta della resistenza è importante, nel circuito è stata usata una resistenza da $180\ \Omega$ perché:

$$\frac{5V}{180\Omega} = 27mA < 40mA$$

dove 40 mA è la massima corrente consentita per i GPIO pin.

1.1.3 Codice Arduino

```
1 #include <TimerOne.h>
2 /*TimerOne.h allows to attach interrupts to the MCU's Timer1*/
3
4 const int RLED_PIN = 12;
5 const int GLED_PIN = 11;
6 /*GPIO pins are identified by integer constants*/
7
8 const float R_HALF_PERIOD = 1.5;
9 const float G_HALF_PERIOD = 3.5;
10
11 int greenLedState = LOW;
12 int redLedState = LOW;
13 /*These are global variables to store the current state of the two LEDs
14    . LOW and HIGH are macros for 0 and 1 respectively*/
15
16 void blinkGreen(){
17     greenLedState = !greenLedState;
18     digitalWrite(GLED_PIN, greenLedState);
19 }
20
21 void setup() {
22     pinMode(RLED_PIN, OUTPUT);
```

```

22  pinMode(GLED_PIN, OUTPUT);
23  Timer1.initialize(G_HALF_PERIOD * 1e06);
24  Timer1.attachInterrupt(blinkGreen);
25 }
26
27 void loop() {
28   redLedState = !redLedState;
29   digitalWrite(RLED_PIN, redLedState);
30   delay(R_HALF_PERIOD * 1e03);
31 }

```

Innanzitutto abbiamo installato ed importato la libreria TimerOne per la gestione di un timer da associare alla procedura di interrupt. Abbiamo poi definito i GPIO pins dei due LED tramite delle costanti intere. Dopodiché abbiamo settato i due semiperiodi indipendenti per i due led, nuovamente come costanti ma stavolta float. Sono state utilizzate delle variabili globali (greenLedState e redLedState) per memorizzare lo stato corrente dei due LED (LOW=0, HIGH=1).

Nella funzione di setup() i due pin sono stati settati ad OUTPUT, è stato inizializzato il timer ad un certo periodo in μs (da qui il prodotto per 1e06) visto che la funzione initialize() lavora su un parametro espresso in μs , infine è stata associata una ISR (blinkGreen) allo scadere del timer.

Nella ISR è invertito lo stato del LED verde in maniera asincrona.

Nella funzione loop() invece il LED rosso cambia stato periodicamente tramite funzione delay().

1.2 Comunicazione tramite Porta Seriale

1.2.1 Componenti HW

- 1 x Arduino Yún Rev 2
- 1 x Breadboard
- 2 x LED
- 2 x Resistenze ($180 \pm 1\%$) Ω
- 3 x Cavi m/m

1.2.2 Realizzazione HW

Il circuito realizzato è lo stesso dell'esercizio 1.1, valgono analoghe considerazioni.

1.2.3 Codice Arduino

```

1 #include <TimerOne.h>
2 /*TimerOne.h allows to attach interrupts to the MCU's Timer1*/
3
4 const int RLED_PIN = 12;

```

```

5 const int GLED_PIN = 11;
6 /*GPIO pins are identified by integer constants*/
7
8 const float R_HALF_PERIOD = 1.5;
9 const float G_HALF_PERIOD = 3.5;
10
11 volatile int greenLedState = LOW;
12 int redLedState = LOW;
13 /*These are global variables to store the current state of the two LEDs
14
15 LOW and HIGH are macros for 0 and 1 respectively*/
16
17 void blinkGreen() {
18     greenLedState = !greenLedState;
19     digitalWrite(GLED_PIN, greenLedState);
20 }
21
22 void serialPrintStatus() {
23     if (Serial.available() > 0) {
24         int inByte = Serial.read();
25
26         switch(inByte) {
27             case 'R': Serial.print("LED 12 (R) Status: "); Serial.println(
                redLedState); break;
28             case 'L': Serial.print("LED 11 (G) Status: "); Serial.println(
                greenLedState); break;
29             default: Serial.println("Invalid command");
30         }
31     }
32 }
33
34 void setup() {
35     pinMode(RLED_PIN, OUTPUT);
36     pinMode(GLED_PIN, OUTPUT);
37     Serial.begin(9600);
38     while(!Serial);
39     Serial.println("Welcome, Lab 1.2 Starting");
40     Timer1.initialize(G_HALF_PERIOD * 1e06);
41     Timer1.attachInterrupt(blinkGreen);
42 }
43
44 void loop() {
45     serialPrintStatus();
46     redLedState = !redLedState;
47     digitalWrite(RLED_PIN, redLedState);
48     delay(R_HALF_PERIOD * 1e03);
49 }

```

Il codice dell'esercizio 1.1 è stato riadattato per soddisfare i requisiti richiesti. Nella funzione di setup() è stata configurata la comunicazione seriale con il PC con BaudRate = 9600. L'istruzione while(!Serial); permette poi di attendere l'effettiva apertura del Serial Monitor al fine di evitare la perdita di messaggi. Dopo aver avviato il Serial Monitor sarà visualizzato il messaggio di benvenuto e la funzione di setup() si concluderà con le istruzioni già viste per l'esercizio 1.1.

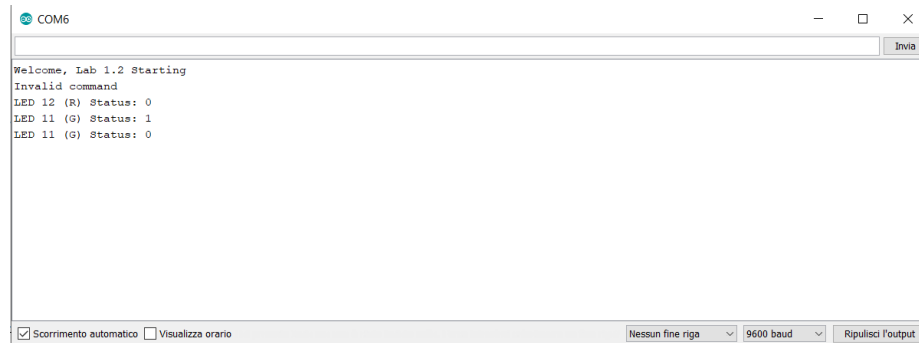
Notiamo che alla variabile globale greenLedState è stato aggiunto il qualificatore

volatile perché essa è una variabile condivisa dalla funzione `loop()` e dalla ISR `blinkGreen()`. Il qualificatore volatile permette di tenere traccia dell'ultimo valore effettivamente scritto nella variabile senza incorrere in errori.

È stata poi definita la funzione `serialPrintStatus()` che verifica la disponibilità di nuovi dati in ingresso allo sketch. Se non vi sono nuovi dati sul Serial buffer lo sketch procede oltre, se sono disponibili nuovi dati vengono letti e a seconda del carattere inviato allo sketch si stamperà un messaggio a video piuttosto che un altro. Tale funzione è invocata in ogni iterazione della funzione `loop()`.

Come specificato dalla traccia, bisogna prestare attenzione alle informazioni ricevute sul Serial Monitor perché una eventuale print (effettuata dalla funzione `serialPrintStatus()`) è fatta immediatamente prima del cambio di stato del LED rosso (meccanismo sincrono) e non bisogna farsi trarre in inganno.

1.2.4 Output su Serial Monitor



Nota: è stata selezionata l'opzione "Nessun fine riga" per semplificare la gestione dei dati inviati allo sketch. In questo modo l'unico carattere effettivamente inviato allo sketch risulta quello inserito dall'utente, senza altre aggiunte (new line, ecc.).

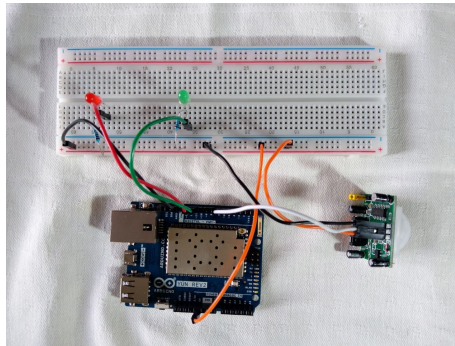
1.3 Identificazione della presenza tramite Sensore PIR

1.3.1 Componenti HW

- 1 x Arduino Yún Rev 2
- 1 x Breadboard
- 1 x LED
- 1 x Resistenza ($180 \pm 1\%$) Ω
- 3 x Cavi m/m

- 3 x Cavi m/f
- 1 x PIR Motion Sensor

1.3.2 Realizzazione HW



Nota: nel circuito in figura sono presenti più componenti rispetto a quelli riportati nella lista 1.3.1, nella quale sono riportati i componenti essenziali. Nella realizzazione fisica è stato seguito il suggerimento di aggiungere il sensore PIR senza disconnettere i LED in quanto potrebbero tornare utili nelle future esercitazioni.

Abbiamo collegato opportunamente i pin di Power (5 V), Ground (GND) e Output (pin D7) del sensore PIR. L'output digitale del sensore PIR è stato collegato ad un pin digitale della Yún che supporta l'interrupt, per l'appunto il pin D7. La configurazione dei potenziometri è stata impostata in maniera tale da avere sensibilità massima (circa 7 metri di distanza) e tempo di delay minimo (circa 3 s in cui il PIR mantiene il proprio stato HIGH dopo aver rilevato il movimento).

1.3.3 Codice Arduino

```

1 const int RLED_PIN = 12;
2 const int PIR_PIN = 7;
3 /*GPIO pins are identified by integer constants*/
4
5 volatile int tot_count = 0;
6 int redLedState = LOW;
7 /*These are global variables to store the current state of the LED and
   the number of total events.
8 LOW and HIGH are macros for 0 and 1 respectively*/
9
10 void checkPresence() {
11     redLedState = !redLedState;
12     digitalWrite(RLED_PIN, redLedState);
13     if (redLedState==HIGH) {
14         tot_count++;
15     }
16 }
17

```

```

18 void serialPrintStatus(){
19   Serial.print("Total people count: ");
20   Serial.println(tot_count);
21 }
22
23 void setup() {
24   pinMode(RLED_PIN, OUTPUT);
25   pinMode(PIR_PIN, INPUT);
26   Serial.begin(9600);
27   while(!Serial);
28   Serial.println("Welcome, Lab 1.3 Starting");
29   attachInterrupt(digitalPinToInterrupt(PIR_PIN), checkPresence, CHANGE
    );
30 }
31
32 void loop() {
33   delay(30000);
34   serialPrintStatus();
35 }

```

Sono state definite delle costanti intere per i pin del LED rosso e del sensore PIR (il LED verde è inutilizzato). È stato usato un contatore ausiliario per tenere traccia del numero dei movimenti avvenuti ed è stato definito con qualificatore volatile perché è utilizzato sia nel meccanismo di loop, sia nella procedura di interrupt.

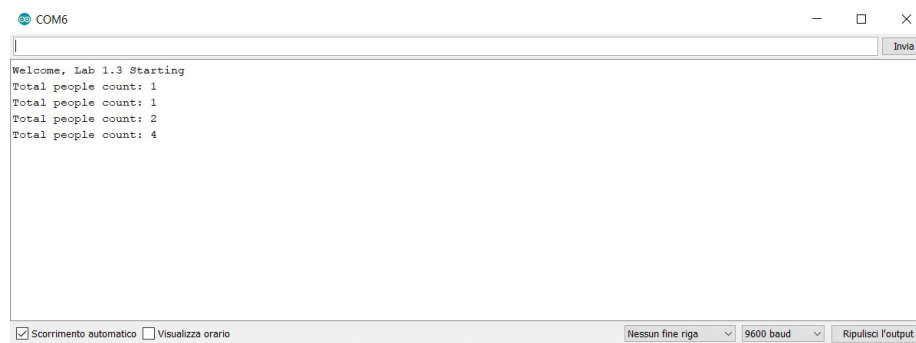
Nella funzione di setup() abbiamo definito i due pin per LED e PIR rispettivamente come OUTPUT e INPUT. Abbiamo avviato il serial monitor come già visto nell'esercizio precedente e abbiamo associato la ISR checkPresence ad ogni variazione (sia fronte di salita, sia fronte di discesa) rilevata sul pin D7 cui è collegato il sensore PIR.

La ISR checkPresence() non fa altro che cambiare lo stato del LED rosso accendendolo/spengendolo e aggiornare il conteggio ad ogni fronte di salita.

La funzione loop() permette di visualizzare, ogni 30 s, il conteggio sul Serial Monitor tramite la funzione serialPrintStatus().

La funzione serialPrintStatus() effettua una semplice stampa su Serial Monitor.

1.3.4 Output su Serial Monitor

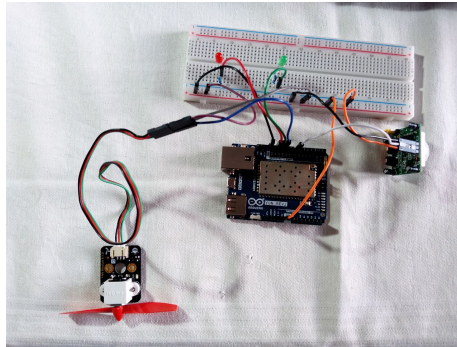


1.4 Controllo di un motore (ventola) tramite PWM

1.4.1 Componenti HW

- 1 x Arduino Yún Rev 2
- 1 x Breadboard
- 5 x Cavi m/m
- 1 x Modulo FAN
- 1 x Ventola
- 1 x Cavo per modulo FAN

1.4.2 Realizzazione HW



Nota: nel circuito in figura sono presenti più componenti rispetto a quelli riportati nella lista 1.4.1, nella quale sono riportati i componenti essenziali. Il modulo FAN è stato aggiunto al circuito preparato per l'esercizio precedente. Abbiamo collegato opportunamente i pin di VCC (5 V), Ground (GND) e Control (pin D10 abilitato a PWM) del modulo FAN. Non tutti i GPIOs supportano PWM, per cui abbiamo scelto il pin D10 (contrassegnato dal simbolo ~ accanto al numero del pin, che indica il supporto a PWM).

1.4.3 Codice Arduino

```
1 const int FAN_PIN = 10;
2
3 /*GPIO pins are identified by integer constants*/
4
5 const int MAX = 255;
6 const float STEP = 25.5;
7 const int MIN = 0;
8
9 float current_speed = MIN;
10 /*This is a global variable to store the current speed of the module.*/
```

```

11
12 void serialPrintStatus(){
13     if (Serial.available()>0){
14         int inByte = Serial.read();
15
16         switch(inByte){
17             case '+':
18                 if (current_speed+STEP>MAX){
19                     Serial.println("Already at max speed");
20                 }
21                 else{
22                     Serial.print("Increasing speed: ");
23                     current_speed+=STEP;
24                     float perc = current_speed/MAX*100;
25                     Serial.print(perc);
26                     Serial.println("%");
27                     analogWrite(FAN_PIN, (int)current_speed);
28                 }
29                 break;
30             case '-':
31                 if (current_speed-STEP<MIN){
32                     Serial.println("Already at min speed");
33                 }
34                 else{
35                     Serial.print("Decreasing speed: ");
36                     current_speed-=STEP;
37                     int perc = current_speed/MAX*100;
38                     Serial.print(perc);
39                     Serial.println("%");
40                     analogWrite(FAN_PIN, (int)current_speed);
41                 }
42                 break;
43             default:
44                 Serial.println("Invalid command");
45                 break;
46         }
47     }
48 }
49
50 void setup() {
51     pinMode(FAN_PIN, OUTPUT);
52     analogWrite(FAN_PIN, (int)current_speed);
53     Serial.begin(9600);
54     while(!Serial);
55     Serial.println("Welcome, Lab 1.4 Starting");
56 }
57
58 void loop() {
59     serialPrintStatus();
60 }

```

É stata definita la costante intera associata al pin D10 da usare, sono state definite delle costanti di utilità e una variabile globale per tenere traccia della velocità della ventola (inizialmente a 0).

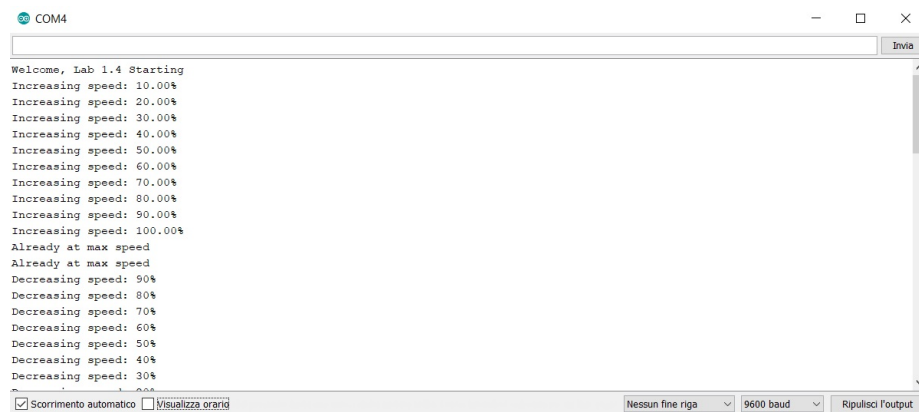
Nella funzione di setup() è stato impostato il pin come OUTPUT, si è avviato il motore con valore iniziale 0 e si è inizializzato il Serial Monitor come già visto

precedentemente.

Nella funzione `loop()` viene semplicemente invocata la funzione `serialPrintStatus()` a cui è demandato tutto il lavoro.

La funzione `serialPrintStatus()` verifica la presenza di dati disponibili allo sketch forniti tramite Serial Monitor. Se non ci sono nuovi dati, si procede oltre; se ci sono nuovi dati, questi vengono letti e a seconda del carattere inserito si procederà a incrementare/decrementare la velocità della ventola e/o a stampare opportuni messaggi di controllo. Se il carattere letto è '+' si verifica se si è già a velocità massima e nel caso lo si segnala; se non si è a velocità massima viene incrementato il valore di una quantità prefissata STEP e viene mostrata sul Serial Monitor la velocità attuale in percentuale. Se il carattere letto è '-' si verifica se si è già a velocità minima e nel caso lo si segnala; se non si è a velocità minima viene decrementato il valore di una quantità prefissata STEP e viene mostrata sul Serial Monitor la velocità attuale in percentuale. In caso di caratteri non previsti si segnala il tutto all'utente tramite Serial Monitor.

1.4.4 Output su Serial Monitor

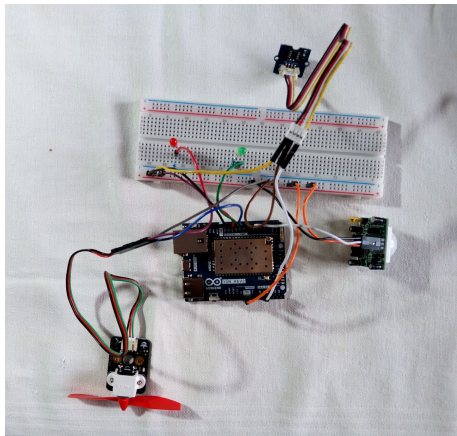


1.5 Lettura di valori analogici (temperatura)

1.5.1 Componenti HW

- 1 x Arduino Yún Rev 2
- 1 x Breadboard
- 5 x Cavi m/m
- 1 x Sensore di temperatura
- 1 x Cavo per sensore di temperatura

1.5.2 Realizzazione HW



Nota: nel circuito in figura sono presenti più componenti rispetto a quelli riportati nella lista 1.5.1, nella quale sono riportati i componenti essenziali. Il sensore di temperatura è stato aggiunto al circuito preparato per l'esercizio precedente. Abbiamo collegato opportunamente i pin di VCC (5 V), GND (GND) e SIG (pin A0) del sensore di temperatura.

1.5.3 Analisi circuitale

Il sensore produce una tensione in uscita che dipende (attraverso una resistenza variabile, il termistore) dalla temperatura. Quindi noi leggiamo la tensione, ma abbiamo bisogno di ricavare la temperatura.

Questo procedimento avviene in due step:

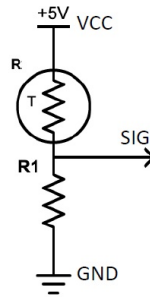
1. dalla tensione si ottiene la resistenza;
2. dalla resistenza si ottiene la temperatura.

Il sensore ha due parametri principali:

- $R_0 = 100 \text{ K}\Omega$, che è una resistenza fissa alla temperatura nominale $T_0 = 25 \text{ }^\circ\text{C} = 298.15 \text{ K}$;
- $B = 4275 \text{ K}$, che definisce la relazione (non-lineare) tra la resistenza e la temperatura.

Vediamo i due step:

- Step 1:
 $R_1 = R_0 = 100 \text{ K}\Omega$
Il convertitore AD di Arduino lavora con una risoluzione di 10 bit, quindi l'intervallo di valori rappresentabile va da 0 a $2^{10}-1=1023$. Per cui $\text{GND} = 0$ e $\text{VCC} = 1023$.
Lo step 1 consiste nell'estrarre R da V_{sig} usando un semplice partitore poiché il sensore può essere modellizzato come segue:



da cui:

$$V_{sig} = \frac{R_0}{R + R_0} * VCC \Rightarrow R = \left(\frac{VCC}{V_{sig}} - 1 \right) * R_0$$

- Step 2:

La conversione da R a T può essere ottenuta invertendo la definizione di B presa dal datasheet:

$$B = \frac{\ln\left(\frac{R}{R_0}\right)}{\frac{1}{T} - \frac{1}{T_0}} \Rightarrow T = \frac{1}{\frac{\ln\left(\frac{R}{R_0}\right)}{B} + \left(\frac{1}{T_0}\right)}$$

Attenzione: in questa equazione sia T sia T0 sono in Kelvin. Noi vogliamo mostrare la temperatura in Celsius, per cui alla fine dovremo effettuare una conversione.

1.5.4 Codice Arduino

```

1 #include <math.h>
2 const int TEMP_PIN = A0;
3
4 const float MAX = 1023.0;
5 const float B = 4275.0;
6 const float R0 = 100000.0;
7 const float T0 = 298.15;
8
9 float getResistanceStep1() {
10     int a = analogRead(TEMP_PIN);
11     return ((MAX/a)-1)*R0;
12 }
13
14 float getTemperatureStep2(float R) {
15     return (1 / ((log(R/R0)/B) + (1/T0)));
16 }
17

```



```

18 void serialPrintStatus(float T){
19     Serial.print("Temperature = ");
20     Serial.println(T);
21 }
22
23 void setup() {
24     pinMode(TEMP_PIN, INPUT);
25     Serial.begin(9600);
26     while(!Serial);
27     Serial.println("Welcome, Lab 1.5 Starting");
28 }
29
30 void loop() {
31     delay(10000);
32     float R = getResistanceStep1();
33     float T = getTemperatureStep2(R);
34     T = T-273.15;
35     serialPrintStatus(T);
36 }

```

Abbiamo incluso la libreria `math.h` per poter effettuare le varie operazioni matematiche richieste.

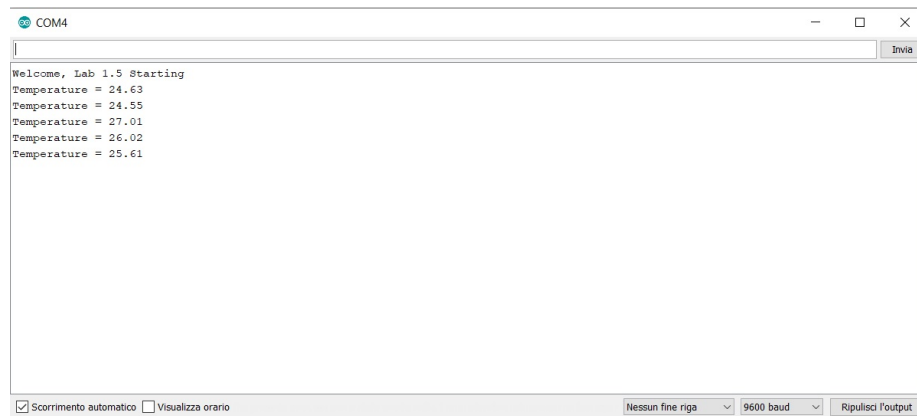
I pin analogici sono identificati dalla MACRO `Ax`, dove `x` è il numero del pin. Abbiamo scelto il pin `A0`.

Abbiamo definito alcune costanti di utilità, alla luce delle considerazioni fatte in 1.5.3.

Nella funzione di `setup()` si setta il pin come `INPUT` e si inizializza il Serial Monitor.

Nella funzione `loop()` si attende 10 s, dopodiché si invocano in successione due funzioni che hanno come scopo quello di implementare i due step visti in 1.5.3. Al termine della loro esecuzione avremo ottenuto il valore di temperatura in Kelvin e andrà riportato in Celsius. Procederemo infine alla stampa tramite funzione `serialPrintStatus()`.

1.5.5 Output su Serial Monitor



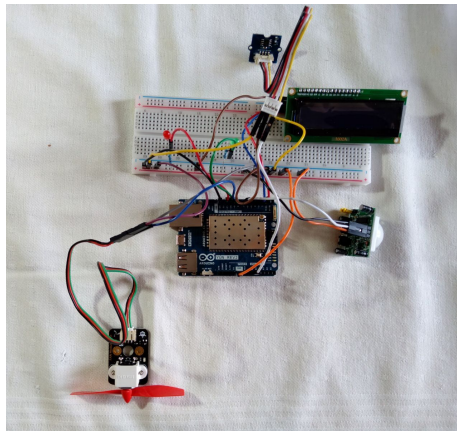
Come suggerito nella traccia si è verificato il funzionamento poggiando il dito sul sensore e facendo alzare leggermente la temperatura.

1.6 Comunicazione con uno smart actuator (display LCD) tramite protocollo I2C

1.6.1 Componenti HW

- 1 x Arduino Yún Rev 2
- 1 x Breadboard
- 5 x Cavi m/m
- 4 x Cavi m/f
- 1 x Display LCD
- 1 x Sensore di temperatura
- 1 x Cavo per sensore di temperatura

1.6.2 Realizzazione HW



Nota: nel circuito in figura sono presenti più componenti rispetto a quelli riportati nella lista 1.6.1, nella quale sono riportati i componenti essenziali. Il display LCD è stato aggiunto al circuito preparato per l'esercizio precedente. Abbiamo collegato opportunamente i pin di VCC (5 V), GND (GND), SCL (D3, connesso alla SCL della scheda) e SDA (D2, connesso alla SDA della scheda) del display LCD. Sono stati usati i pin D2 e D3 perché sono quelli di default previsti dalla libreria da usare.

1.6.3 Codice Arduino

```
1 #include <math.h>
2 #include <LiquidCrystal_PCF8574.h>
3
4 LiquidCrystal_PCF8574 lcd(0x27);
5
6 const int TEMP_PIN = A0;
7
8 const float MAX = 1023.0;
9 const float B = 4275.0;
10 const float R0 = 100000.0;
11 const float T0 = 298.15;
12
13 float getResistanceStep1() {
14     int a = analogRead(TEMP_PIN);
15     return ((MAX/a)-1)*R0;
16 }
17
18 float getTemperatureStep2(float R) {
19     return (1/((log(R/R0)/B)+(1/T0)));
20 }
21
22 void printStatus(float T) {
23     lcd.setCursor(12, 0);
24     lcd.print(T,1);
25 }
26
27 void setup() {
28     pinMode(TEMP_PIN, INPUT);
29     lcd.begin(16,2);
30     lcd.setBacklight(255);
31     lcd.home();
32     lcd.clear();
33     lcd.print("Temperature:");
34 }
35
36 void loop() {
37     delay(10000);
38     float R = getResistanceStep1();
39     float T = getTemperatureStep2(R);
40     T = T-273.15;
41     printStatus(T);
42 }
```

Il codice è molto simile a quello dell'esercizio precedente.

Abbiamo incluso la libreria richiesta e creato un'istanza della classe display su cui poter lavorare.

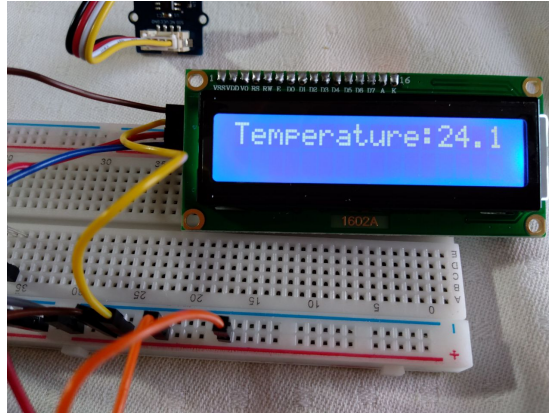
Nella funzione di setup() piuttosto che inizializzare il Serial Monitor si fanno delle operazioni preliminari sul display (begin, set della retroilluminazione, clear, stampa stringa "Temperature:").

Nella funzione loop() si invoca la printStatus().

Nella funzione printStatus() si posiziona il cursore del display, tramite la funzione setCursor(), subito dopo la stringa "Temperature:" già stampata. Fatto ciò,

non ci resta che stampare il valore di temperatura. Abbiamo anche soddisfatto il requisito di aggiornare efficientemente la temperatura sul display.

1.6.4 Output su display LCD



Capitolo 2

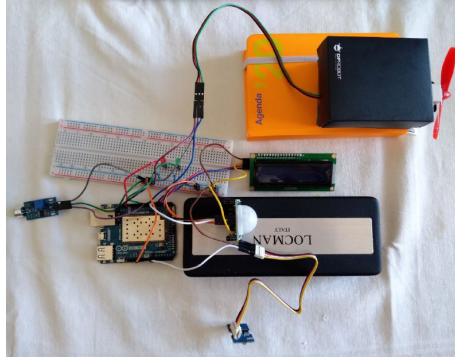
Smart home controller (versione ”locale”)

2.1 Prima versione

2.1.1 Componenti HW

- 1 x Arduino Yún Rev 2
- 1 x Breadboard
- 10 x Cavi m/m
- 10 x Cavi m/f
- 2 x LED
- 2 x Resistenze ($180 \pm 1\%$) Ω
- 1 x Display LCD
- 1 x Sensore di temperatura
- 1 x Cavo per sensore di temperatura
- 1 x Modulo FAN
- 1 x Ventola
- 1 x Cavo per modulo FAN
- 1 x PIR Motion Sensor
- 1 x Sensore di rumore

2.1.2 Realizzazione HW



Il circuito in foto è stato ottenuto riorganizzando il circuito visto in 1.6 e aggiungendo il sensore di rumore. Alcuni elementi (come il LED verde) non sono usati in questa prima versione ma sono stati lasciati per comodità poiché saranno utili nella seconda versione dello smart home controller.

Connessioni principali:

- Il modulo FAN è connesso al pin digitale 6 abilitato a PWM;
- Il display LCD è connesso ai pin digitali 3 (SCL) e 2 (SDA);
- Il PIR è connesso al pin digitale 4;
- Il sensore di temperatura è connesso al pin analogico A0;
- Il LED rosso è connesso al pin digitale 13 abilitato a PWM;
- Il LED verde (inutilizzato in questa versione) è connesso al pin digitale 8;
- Il sensore di rumore è connesso al pin digitale 7 abilitato all'uso di interrupt.

Il sensore PIR è stato regolato con delay massimo e sensibilità media mentre il sensore di rumore è stato regolato in modo tale da non rilevare il rumore della ventola ma di essere comunque in grado di rilevare altri rumori di intensità maggiore.

2.1.3 Codice Arduino (diviso in porzioni e commentato)

```
1 #include <math.h>
2 #include <LiquidCrystal_PCF8574.h>
3 #include <TimerThree.h>
4
5 LiquidCrystal_PCF8574 lcd(0x27);
6
7 /*GPIO pins*/
8 const int FAN_PIN = 6;
```

```

9  const int TEMP_PIN = A0;
10 const int RED_LED_PIN = 13;
11 const int PIR_PIN = 4;
12 const int SOUND_PIN = 7;
13
14 /*Utility constants*/
15 const int MAX_PWM_FAN_AND_LED = 255;
16 const int MAX_ANALOG_TEMP = 1023;
17 const int MIN = 0;
18 const float B = 4275.0;
19 const float R0 = 100000.0;
20 const float T0 = 298.15;
21 const float PIR_PERIOD = 0.5; /*0.5 s, period of PIR checking*/
22 const int TIMEOUT_PIR = 30 * 60;
23 const int N_SOUND_EVENTS = 50;
24 const int SOUND_INTERVAL = 10 * 60;
25 const int TIMEOUT_SOUND = 60 * 60;
26 const int NUMBER_OF_SET_POINTS = 4;
27 const int TEMP_PERIOD = 30; /*30 s, period of temperature checking*/
28
29 /*Global variables*/
30 float set_point_min_max_1[] = {22.0, 25.0, 16.0, 20.0}; /*default set-
    point to use when people detected*/
31 float set_point_min_max_0[] = {24.0, 28.0, 14.0, 18.0}; /*default set-
    point to use when people not detected*/
32 float set_point[] = {0, 0, 0, 0}; /*set-point in use*/
33 volatile unsigned long time_latest_detection_pir = 0;
34 volatile bool flag_pir_presence = true;
35 volatile bool flag_sound_presence = false;
36 bool flag_presence = false;
37 unsigned long time_latest_temperature_check = 0;
38 float T = 0;
39 volatile unsigned long *timestamps_last_10_minutes; /*dynamically
    allocated circular vector where 0 is flag for empty entry*/
40 volatile int tail = 0;
41 volatile bool flag_circular_vect = false;

```

In questa prima parte abbiamo incluso le librerie necessarie (libreria matematica, libreria per l'uso del display, libreria per l'uso di Timer3) e definito le costanti per i pin usati.

Abbiamo poi definito tutta una serie di costanti di utilità, tra cui citiamo PIR_PERIOD (cadenza polling del sensore PIR), TIMEOUT_PIR (30 minuti, come richiesto), N_SOUND_EVENTS (50, come richiesto), SOUND_INTERVAL (10 minuti, come richiesto), TIMEOUT_SOUND (60 minuti, come richiesto) e TEMP_PERIOD (cadenza polling sensore di temperatura).

Successivamente sono state definite le variabili globali, tra cui set_point_min_max_1 (vettore di float, contiene i set-point fissati in caso di presenza rilevata), set_point_min_max_0 (vettore di float, contiene i set-point fissati in caso di presenza non rilevata), set_point (vettore di float, contiene i set-point attualmente in uso a seconda del flag di presenza), time_latest_detection_pir (variabile unsigned long per evitare overflow in quanto dovrà contenere il valore assegnato tramite millis(), rappresenta il tempo di ultimo rilevazione di presenza del sensore PIR, utile in varie operazioni), flag_pir_presence, flag_sound_presence (per

segnalare rilevazione presenza da parte di un sensore specifico) e `flag_presence` (per segnalare rilevazione presenza da parte di almeno uno dei due sensori, flag di "sistema"), `time_latest_temperature_check` (variabile unsigned long per evitare overflow in quanto dovrà contenere il valore assegnato tramite `millis()`, rappresenta il tempo di ultimo check effettuato del sensore di temperatura, utile in varie operazioni), `T` (temperatura rilevata), `timestamps_last_10_minutes` (vettore circolare allocato dinamicamente per la gestione della finestra mobile degli ultimi 50 eventi registrati dal sensore di rumore; il valore 0 è usato come flag di entry vuota), `tail` (coda, indice per la gestione del vettore circolare) e `flag_circular_vect` (flag utile a capire quando il vettore circolare si è riempito per la prima volta).

```

43 /*Utility function to copy vector content into another.*/
44 void vector_copy(float v1[], float v2[], int n) {
45     for (int i = 0; i < n; i++) {
46         v1[i] = v2[i];
47     }
48 }

```

Funzione di utilità per la copia di un vettore origine in un vettore destinazione.

```

50 /*ISR for sound sensor. Run when sound is detected.*/
51 void presence_detected() {
52     if (millis() - timestamps_last_10_minutes[(tail - 1 + N_SOUND_EVENTS)
53         % N_SOUND_EVENTS] > 50) {
54         if (flag_circular_vect == false && tail + 1 == N_SOUND_EVENTS) {
55             flag_circular_vect = true;
56         }
57         tail = tail % N_SOUND_EVENTS;
58         timestamps_last_10_minutes[tail] = millis();
59         if (flag_circular_vect == true && timestamps_last_10_minutes[tail]
60             - timestamps_last_10_minutes[(tail + 1) % N_SOUND_EVENTS] <
61             SOUND_INTERVAL * 1000) {
62             flag_sound_presence = true;
63         }
64         tail++;
65     }
66 }
67
68 /*ISR associated with Timer3 expiration. It checks for motion detection
69 */
70 void check_pir_presence() {
71     int a = digitalRead(PIR_PIN);
72     if (a == HIGH) {
73         flag_pir_presence = true;
74         time_latest_detection_pir = millis();
75     }
76 }

```

Qui mettiamo in evidenza le due ISR.

L'ISR `presence_detected` è associata al sensore di rumore, connesso al pin di-

gitale 7 abilitato all'interrupt. Innanzitutto viene svolto un controllo interno per verificare che siano passati almeno 50 ms dall'ultima rilevazione effettuata e questa operazione è FONDAMENTALE per discriminare "eventi distinti" per l'orecchio umano. Infatti se questo controllo non venisse effettuato il sensore rileverebbe molteplici rumori (anche a fronte di singoli rumori generati, ad esempio un battito di mani) e l'ISR verrebbe lanciata, erroneamente, più volte. Bisogna aggiungere questo delay di 50 ms per garantire il corretto funzionamento. Se il controllo va a buon fine (nota: piuttosto che mantenere un'altra variabile globale con il timestamp dell'ultimo evento rilevato dal sensore di rumore sfruttiamo il vettore circolare e facciamo accesso alla entry precedente, che conterrà il timestamp dell'evento più recente) verifichiamo se non abbiamo ancora riempito una prima volta il vettore circolare e se siamo in procinto di farlo (ad esempio siamo al 49esimo evento registrato, quindi dal prossimo possiamo usare il vettore circolare in maniera standard poiché pieno) settiamo il flag opportunamente. Aggiorniamo l'indice tail con l'operatore modulo (così da ripartire da zero una volta giunti al fondo), registriamo il tempo di evento corrente nella entry di indice tail, se siamo in condizioni standard di gestione della coda (riempita almeno una volta) confrontiamo l'elemento corrente (più recente) con il successivo (meno recente) e vediamo se differiscono temporalmente meno dell'intervallo prestabilito (nel nostro caso 10 minuti). Se così fosse, dovremmo registrare presenza tramite il flag del sensore di rumore. Incrementiamo infine l'indice tail.

L'ISR `check_pir_presence` è associata allo scadere del `Timer3`. Quindi ogni 0.5 s viene lanciata questa ISR la quale legge il valore sul PIR, se è HIGH vuol dire che è stata rilevata una presenza e viene settato il flag di conseguenza, con annessa registrazione di tempo di rilevazione.

```

74 /*setup function*/
75 void setup() {
76   pinMode(FAN_PIN, OUTPUT);
77   pinMode(TEMP_PIN, INPUT);
78   pinMode(RED_LED_PIN, OUTPUT);
79   pinMode(PIR_PIN, INPUT);
80   pinMode(SOUND_PIN, INPUT);
81   digitalWrite(RED_LED_PIN, MIN);
82   digitalWrite(FAN_PIN, MIN);
83   lcd.begin(16, 2);
84   lcd.setBacklight(255);
85   lcd.clear();
86   Serial.begin(9600);
87   while (!Serial);
88   Serial.println("Welcome, in order to update the 4 set-point please
      send 0 or 1 so that I understand which case I have to modify (
      presence = 0 or presence=1)");
89   Serial.println("Waiting for commands...");
90   attachInterrupt(digitalPinToInterrupt(SOUND_PIN), presence_detected,
      FALLING);
91   Timer3.initialize(PIR_PERIOD * 1e06);
92   Timer3.attachInterrupt(check_pir_presence);

```

```

93   timestamps_last_10_minutes = calloc(N_SOUND_EVENTS, sizeof(unsigned
      long));
94 }

```

In questa porzione di codice vediamo la funzione di setup.

Vengono impostati i pin dei sensori a INPUT/OUTPUT, è inizializzato il display, è inizializzato il Serial Monitor con un messaggio di benvenuto e con le indicazioni utili all'utente per farne un corretto uso. Fatto ciò, non resta che associare le ISR rispettivamente al pin digitale 7 (con condizione FALLING perché il sensore di rumore è attivo basso, ovvero quando rileva un rumore va da HIGH a LOW) e allo scadere del Timer3 (inizializzato con periodo definito in precedenza) ed allocare dinamicamente il vettore di timestamp della dimensione opportuna (con calloc per azzerarlo in maniera automatica).

```

96 /*Function to check if millis() time register is in overflow (about
      every 50 days) and to reset variables to keep functioning*/
97 void check_overflow() {
98   bool flag_overflow = false;
99   for (int i = 0; i < N_SOUND_EVENTS; i++) {
100     if (millis() < timestamps_last_10_minutes[i]) {
101       flag_overflow = true;
102       break;
103     }
104   }
105   if (flag_overflow == true || millis() < time_latest_detection_pir ||
      millis() < time_latest_temperature_check) {
106     for (int i; i < N_SOUND_EVENTS; i++) {
107       timestamps_last_10_minutes[i] = 0;
108     }
109     time_latest_detection_pir = 0;
110     time_latest_temperature_check = 0;
111     flag_pir_presence = false;
112     flag_sound_presence = false;
113     flag_presence = false;
114     flag_circular_vect = false;
115     tail = 0;
116   }
117 }

```

Questa funzione verifica l'avvenuto overflow del registro timer associato a millis() e se la condizione è verificata provvede a fare il reset delle variabili coinvolte per garantire ancora un corretto funzionamento. Vengono effettuati dei controlli sulle variabili temporali in uso.

```

119 /*Function to check if PIR timeout has expired and eventually set PIR
      flag to false*/
120 void check_timeout_pir() {
121   if (flag_pir_presence == true && (millis() -
      time_latest_detection_pir) / 1000 >= TIMEOUT_PIR) {
122     flag_pir_presence = false;
123   }

```

```

124 }
125
126 /*Function to check if sound sensor timeout has expired and eventually
    set sound sensor flag to false*/
127 void check_timeout_sound() {
128     if (flag_sound_presence == true && (millis() -
        timestamps_last_10_minutes[(tail - 1 + N_SOUND_EVENTS) %
            N_SOUND_EVENTS]) / 1000 >= TIMEOUT_SOUND) {
129         flag_sound_presence = false;
130     }
131 }

```

La prima funzione verifica se il timeout del PIR (30 minuti, dalla traccia) è scaduto e in tal caso si assume che per il PIR non ci siano persone nella stanza. La seconda funzione controlla se sono avvenuti eventi di rumore negli ultimi 60 minuti (dalla traccia), se l'esito è negativo si assume che per il sensore di rumore non ci siano persone nella stanza.

```

133 /*Function to check if at least one of the two sensors (PIR & sound)
    detects presence and
134     eventually set flag to true/false
135 */
136 void check_presence() {
137     if (flag_sound_presence == true || flag_pir_presence == true) {
138         flag_presence = true;
139     }
140     else {
141         flag_presence = false;
142     }
143 }

```

Controllo sui due flag, se ALMENO un sensore rileva la presenza di persone il sistema assume che vi siano persone nella stanza.

```

145 /*Function to choose set-point in use based on flag_presence
    information*/
146 void choose_set_point() {
147     switch (flag_presence) {
148         case false: vector_copy(set_point, set_point_min_max_0,
            NUMBER_OF_SET_POINTS);
149         break;
150         case true: vector_copy(set_point, set_point_min_max_1,
            NUMBER_OF_SET_POINTS);
151         break;
152     }
153 }

```

In questa funzione a seconda del flag di presenza di sistema si scelgono i set-point da usare.

```

155 /*Function for getting temperature measure*/
156 void set_temperature() {
157     int a = analogRead(TEMP_PIN);
158     float R = ((MAX_ANALOG_TEMP / (float)a) - 1) * R0;
159     T = (1 / ((log(R / R0) / B) + (1 / T0))) - 273.15;
160     time_latest_temperature_check = millis();
161 }
162
163 /*Function for checking temperature sensor every 30 s*/
164 void check_temperature() {
165     if (time_latest_temperature_check == 0) {
166         set_temperature();
167     }
168     else if ((millis() - time_latest_temperature_check) / 1000 >=
169             TEMP_PERIOD) {
170         set_temperature();
171     }
172 }

```

set_temperature serve per il calcolo della temperatura e per registrare il tempo di rilevazione.

La funzione check_temperature verifica che siamo all'accensione della scheda (primo if) e nel caso procede ad effettuare la prima registrazione. Le registrazioni successive avvengono ad intervalli di tempo di circa 30 s.

```

173 /*Function to manage FAN module (AC). It also returns percentage of
174     intensity*/
175 int manage_FAN() {
176     int percentage;
177     if (T > set_point[0] && T < set_point[1]) {
178         analogWrite(FAN_PIN, (int)((T - set_point[0]) / (set_point[1] -
179             set_point[0]) * MAX_PWM_FAN_AND_LED));
180         percentage = (T - set_point[0]) / (set_point[1] - set_point[0]) *
181             100;
182     }
183     else if (T <= set_point[0]) {
184         analogWrite(FAN_PIN, MIN);
185         percentage = 0;
186     }
187     else {
188         analogWrite(FAN_PIN, MAX_PWM_FAN_AND_LED);
189         percentage = 100;
190     }
191     return percentage;
192 }

```

Funzione di gestione del modulo FAN (AC). A seconda dei set-point in uso viene attivata la ventola con una certa intensità (variabile tra MIN=0 e MAX_PWM_FAN_AND_LED=255). Restituisce il valore di intensità espresso in percentuale (da mostrare poi sul display).

```

191 /*Function to manage LED (HT). It also returns percentage of intensity
    */
192 int manage_RED_LED() {
193     int percentage;
194     if (T > set_point[2] && T < set_point[3]) {
195         analogWrite(RED_LED_PIN, (int)(MAX_PWM_FAN_AND_LED - ((T -
            set_point[2]) / (set_point[3] - set_point[2]) *
            MAX_PWM_FAN_AND_LED)));
196         percentage = 100 - ((T - set_point[2]) / (set_point[3] - set_point
            [2]) * 100);
197     }
198     else if (T <= set_point[2]) {
199         analogWrite(RED_LED_PIN, MAX_PWM_FAN_AND_LED);
200         percentage = 100;
201     }
202     else {
203         analogWrite(RED_LED_PIN, MIN);
204         percentage = 0;
205     }
206     return percentage;
207 }

```

Funzione di gestione del LED rosso (HT). A seconda dei set-point in uso viene attivato il LED con una certa intensità (variabile tra MIN=0 e MAX_PWM_FAN_AND_LED=255). Restituisce il valore di intensità espresso in percentuale (da mostrare poi sul display).

```

209 /*Function for showing informations on the display*/
210 void printStatus(int pFAN, int pLED) {
211     lcd.clear();
212     lcd.print("T:");
213     lcd.print(T, 1); /*1 allows to show only a decimal place*/
214     lcd.print(" Pres:");
215     lcd.print(flag_presence);
216     lcd.setCursor(0, 1); /*first parameter is column number, the second
        one is row number*/
217     lcd.print("AC:");
218     lcd.print(pFAN);
219     lcd.print("% HT:");
220     lcd.print(pLED);
221     lcd.print("%");
222     delay(2000); /*2 s informations on display*/
223     lcd.clear();
224     lcd.print("AC m:");
225     lcd.print(set_point[0], 1);
226     lcd.print(" M:");
227     lcd.print(set_point[1], 1);
228     lcd.setCursor(0, 1);
229     lcd.print("HT m:");
230     lcd.print(set_point[2], 1);
231     lcd.print(" M:");
232     lcd.print(set_point[3], 1);
233     delay(2000); /*2 s informations on display*/
234 }

```

Semplice funzione di stampa su display LCD delle informazioni richieste, con cambio schermata ogni 2 secondi.

```
236 /*Utility function to empty serial buffer*/
237 void empty_buffer() {
238     while (Serial.available() > 0) {
239         Serial.read();
240     }
241 }
242
243 /*Function for updating set-point through serial monitor*/
244 void checkforupdate() {
245     bool flag_error = false;
246     int flag_presence_received, i;
247     float setp[] = {0, 0, 0, 0};
248     if (Serial.available() > 0) {
249         flag_presence_received = Serial.parseInt();
250         Serial.print("Inserted: ");
251         Serial.println(flag_presence_received);
252         if (flag_presence_received != 0 && flag_presence_received != 1) {
253             flag_error = true;
254         }
255         if (flag_error == false) {
256             Serial.println("Now please send float values for AC min set-point
                , AC max set-point, HT min set-point, HT max set-point (in
                this order!).");
257             Serial.println("You have a 5 seconds timer for each input.");
258             empty_buffer();
259             for (i = 0; i < 4 && flag_error == false; i++) {
260                 delay(5000);
261                 if (Serial.available() > 0) {
262                     setp[i] = Serial.parseFloat();
263                     Serial.print("Inserted: ");
264                     Serial.println(setp[i]);
265                     empty_buffer();
266                 }
267                 else {
268                     flag_error = true;
269                 }
270             }
271         }
272         if (flag_error == true) {
273             Serial.println("ERROR.");
274         }
275         else {
276             Serial.println("UPDATE WENT WELL.");
277             switch (flag_presence_received) {
278                 case 0: vector_copy(set_point_min_max_0, setp,
                NUMBER_OF_SET_POINTS);
279                     break;
280                 case 1: vector_copy(set_point_min_max_1, setp,
                NUMBER_OF_SET_POINTS);
281                     break;
282             }
283         }
284     }
```

```

284     Serial.println("In order to update the 4 set-point please send 0 or
        1 so that I understand which case I have to modify (presence =
        0 or presence=1)");
285     Serial.println("Waiting for commands...");
286     empty_buffer();
287 }
288 }

```

In questa parte abbiamo definito prima di tutto una funzione di utilità per svuotare il buffer seriale quando sono presenti elementi indesiderati (in genere è stata usata prima di effettuare i check con Serial.available così da evitare errori dovuti a new line o altro).

La funzione checkforupdate si occupa di gestire l'update dei set-point tramite Serial Monitor.

Si controlla innanzitutto se sono disponibili nuovi comandi inviati, se così non fosse si procederebbe oltre (uscendo dalla funzione). Se ci sono nuovi comandi vengono letti tramite parseInt (dato che mi aspetto di ricevere 0/1). Verifico che il valore letto sia effettivamente uno dei due attesi, altrimenti uso un flag per segnalare l'errore. Se non ho riscontrato errori nella prima immissione posso procedere, stampo a video delle informazioni utili per l'utente in cui richiedo di inserire i 4 set-point float uno dopo l'altro. Tramite l'ausilio della funzione delay riesco a creare un tempo di attesa di 5 s in cui l'utente può inserire un valore e lo leggo tramite parseFloat. Se il timeout dovesse scadere senza aver ricevuto un input verrebbe segnalato errore. Per comodità i valori inseriti vengono subito visualizzati su Serial Monitor. Infine, se il flag di errore è true viene stampato un messaggio di errore, altrimenti si stampa un messaggio di conferma e si procede a registrare i 4 set-point nel vettore opportuno. Si stampa nuovamente un messaggio con le indicazioni necessarie.

```

290 void loop() {
291     int percentageFAN, percentageLED;
292     noInterrupts();
293     check_overflow();
294     check_timeout_pir();
295     check_timeout_sound();
296     check_presence();
297     interrupts();
298     choose_set_point();
299     check_temperature();
300     percentageFAN = manage_FAN();
301     percentageLED = manage_RED_LED();
302     printStatus(percentageFAN, percentageLED);
303     checkforupdate();
304 }

```

Funzione loop, semplicemente richiamo tutte le funzioni già presentate.

Per il codice completo si veda il file 2.1.ino

2.1.4 Criticità riscontrate e soluzioni proposte

- Si è cercato di usare delle costanti int (perché le operazioni tra int sono meno onerose rispetto alle operazioni tra float) ma ciò non sempre è stato possibile. Ci sono dei casi (esempi: B, R0, T0 e calcolo di R e T) in cui l'uso di valori interi determina troncamenti nelle operazioni svolte e risultati errati. A questo punto si è deciso, piuttosto di usare costanti int e fare dei cast quando necessario, di usare direttamente delle costanti float. Alla riga 161] c'è un classico esempio di questo problema, infatti è stato fatto un cast di a a float proprio per evitare eventuali troncamenti (perché entrambi gli operandi della divisione sono interi).
- I set-point di default sono stati scelti tenendo conto del fattore energia: se non ci sono persone in stanza l'AC si accende meno facilmente, se ci sono invece si accende con maggior facilità; discorso analogo per l'HT.
- Si è assunto (vedi riga 35]) che all'accensione della scheda ci sia qualcuno in stanza, motivo per cui si è inizializzato il flag di presenza del PIR (con timeout minore) a true. Nota: questa assunzione viene meno nella gestione dell'overflow, dove si è fatto un reset a zero.
- Ove possibile si è cercato di non utilizzare, per motivi di leggibilità del codice, funzioni standard del C. Ad esempio, è stata definita una funzione ausiliaria per la copia di un vettore sorgente in un vettore destinazione anche se era possibile usare la funzione memcpy. Nota: Si è usata, invece, la funzione calloc per allocare dinamicamente il vettore circolare e per inizializzarlo interamente a zero, vista la sua immediatezza.
- Si è effettuato il polling del sensore PIR tramite interrupt associato allo scadere del Timer3. A causa dei ritardi inseriti dalla stampa su display LCD (i due delay da 2 s) non è possibile, al fine di garantire un funzionamento ottimale, effettuare il polling dalla funzione loop.
- Come già discusso precedentemente, per le variabili condivise tra ISR e funzioni è stato usato il qualificatore volatile. Come forma ulteriore di protezione si è optato per la disabilitazione degli interrupt nelle fasi maggiormente delicate del programma. Considerando che in un singolo loop la maggior parte del tempo è passata nella funzione di print (ed eventualmente di input tramite Serial Monitor), tale soluzione sembra essere un ottimo compromesso.
- Facendo riferimento a quanto riportato nel seguente link https://www.pjrc.com/teensy/td_libs_TimerOne.html e poiché la scheda Arduino a nostra disposizione monta lo stesso ATmega32U4 della Leonardo, vediamo che i pin digitali PWM associati alla TimerOne sono 9, 10 e 11, il pin digitale PWM associato alla TimerThree è il 5. Dovendo sceglierne

una per la gestione del PIR è stata preferita la `TimerThree` poiché meno vincolante, così da lasciare liberi i pin 9, 10 e 11 per usi futuri. La `TimerThree` è usata per stabilire il duty cycle del pin PWM 5, quindi se andassimo ad agire sul timer avremmo degli effetti indesiderati sul pin 5 (e sul sensore ad esso collegato). Dal momento che è stata usata tale libreria si è evitato di usare tale pin (5) per non incorrere in problemi.

- Inizialmente abbiamo provato a realizzare il polling del sensore di temperatura tramite interrupt associato allo scadere del `Timer3` di valore 30 s ma ci siamo accorti sperimentalmente in fase di debug che il `Timer3` non va oltre gli 8/9 secondi. Abbiamo quindi provveduto a gestire la situazione in maniera diversa come già visto.

- Per lo svolgimento dei punti 3 e 4 della traccia sono state provate varie soluzioni.

Una prima opzione è stata quella di collegare il sensore di rumore al pin digitale 7 con interrupt e il PIR al pin digitale 4 con interrupt associato alla scadenza di `Timer3`. Tuttavia si è riscontrato il problema, già esposto, delle rilevazioni multiple da parte del sensore di rumore e dell'errato incremento del counter (di prova). Si è pensato quindi di dover dare un margine di tempo al sensore per far sì che non vengano rilevate le vibrazioni successive ad un rumore.

Un secondo approccio ci ha portato a collegare il sensore di rumore al pin digitale 4 con interrupt associato alla scadenza di `Timer3` e il sensore PIR al pin digitale 7 con interrupt. Lo scopo era quello di effettuare un polling cadenzato sul sensore di rumore così da effettuare una singola lettura tramite `digitalRead` e dare un margine di tempo al sensore prima di effettuare una successiva lettura. Questa strategia si è mostrata però non del tutto efficace. Nonostante i miglioramenti ottenuti rispetto alla prima opzione (siamo passati da incrementi di 20-30 unità per volta a incrementi di 5-6 unità), il vincolo di dover "ascoltare" il sensore non in real-time era piuttosto limitante e il margine di tempo fornito non si è rivelato sufficiente. In ultima analisi siamo giunti alla soluzione definitiva come riportato in 2.1.3, in cui gestiamo il sensore di rumore connesso al pin digitale 7 con interrupt ma controllando che non vengano rilevati eventi troppo ravvicinati e in cui il PIR è connesso al pin digitale 4 con interrupt associato alla scadenza di `Timer3`, come originariamente previsto. Tale soluzione si è rivelata vincente, con incrementi real-time unitari del counter.

- Il codice proposto si basa fortemente sull'uso della funzione `millis`. Tale funzione è gestita internamente tramite un registro a 32 bit in cui viene memorizzato un unsigned long. Dopo circa 50 giorni dall'avvio della scheda tale registro andrà in overflow e ripartirà da zero; la funzione `check_overflow` ha proprio il compito di controllare il verificarsi di tale eventualità e nel caso permettere alla scheda di continuare a funzionare in maniera corretta (seppur con un probabile comportamento anomalo e di discontinuità tra gli attimi pre-overflow e post-overflow).

- Per la ricezione di comandi tramite Serial Monitor avevamo pensato ad una soluzione con funzione `serialEvent()` ma questa non è supportata dalla Yún, motivo per cui si è effettuato il polling.

- Per soddisfare la richiesta dei 50 eventi distinti in 10 minuti sono state esplorate varie soluzioni.

Una prima variante era basata sull'uso di un contatore e di una variabile per registrare il tempo a intervalli di 10 minuti. Se al tempo fornito da `millis` sono passati più di 10 minuti dall'ultimo tempo registrato, è "finito un blocco" da 10 minuti e vado a controllare il counter. Se il counter ha valore maggiore di 50 allora registro presenza. Questa soluzione è però inefficace, forniamo qui un controesempio: se sono nei minuti 6-10 di un blocco da 10 e registro 49 eventi, e nei minuti 0-2 (10-12) del blocco da 10 successivo registro 3 eventi, alla fine dei due blocchi non avrò registrato presenza, ma nei minuti 6-16 erano presenti almeno 50 eventi sonori! Abbiamo pensato quindi a soluzioni alternative.

Si è immaginato l'uso di un doppio counter e di un timer dimezzato da 5 minuti, così da ridurre "l'imprecisione" dovuta a dei blocchi temporali troppo ampi. Seguendo questo principio, si è immaginata una struttura dati dalle grandi dimensioni (tante variabili counter) così da gestire intervalli sempre più piccoli. Tale soluzione è ovviamente inapplicabile nella pratica.

Infine, si è giunti alla soluzione proposta in 2.1.3 con un vettore di timestamp. Il vettore è stato impostato in maniera circolare per evitare di doverlo scandire ad ogni rumore rilevato, infatti la nostra implementazione permette di modificare il flag di presenza con costo unitario (semplici operazioni matematiche).

Nota: a seconda dei casi una soluzione potrebbe essere preferibile all'altra. Ad esempio, la versione con singolo counter e timer è imprecisa ma efficace in quanto non impatta molto sulla memoria utilizzata. Invece, la versione con vettore di timestamp risulta onerosa dal punto di vista della memoria ma sicuramente più precisa. Non avendo problemi di memoria, si è optato per la soluzione maggiormente accurata.

- Per gli scopi di questo progetto non si è ritenuta necessaria una programmazione modulare multi-file, ma resta un'opzione interessante per sviluppi futuri.
- Nel codice si fa largo uso di flag bool su 8 bit e ciò risulta in un notevole spreco di memoria. In applicazioni maggiormente complesse dove il fattore memoria gioca un ruolo fondamentale si potrebbe ottimizzare l'uso delle variabili in gioco tenendo conto del fatto che per un flag 0/1 basta avere un singolo bit. Arduino non permette una gestione immediata dei bit, ma si potrebbero usare delle variabili byte su cui andare a leggere e scrivere tramite `bitRead` e `bitWrite` agendo sui singoli bit e documentando il loro significato. Potrebbe essere utile per futuri sviluppi, non avendo

riscontrato problemi di memoria è stata adottata la soluzione più comoda (flag bool).

- La funzione `printStatus` potrebbe essere "condensata" facendo uso di un oggetto della classe `String` e della classica concatenazione tra stringhe tramite operatore `+`, ma così facendo si riscontrerebbe un aumento della memoria utilizzata di circa il 10%. Si è preferito quindi effettuare la stampa elemento per elemento e risparmiare memoria utilizzata. Riportiamo qui la quantità di memoria impiegata:

```
Lo sketch usa 13606 byte (47%) dello spazio disponibile per i programmi. Il massimo è 28672 byte.  
Le variabili globali usano 1022 byte (39%) di memoria dinamica, lasciando altri 1538 byte liberi per le variabili locali. Il massimo è 2560 byte.
```

- Le funzioni `check_timeout_pir` e `check_timeout_sound` potevano essere ridotte ad una singola funzione, così come `manage_FAN` e `manage_RED_LED`. Tuttavia tale operazione non apporta grossi benefici alla memoria impiegata, la quale rimane pressoché invariata, e ha il solo effetto di peggiorare la leggibilità del codice. Per tale ragione sono state usate due funzioni separate nonostante fossero simili.

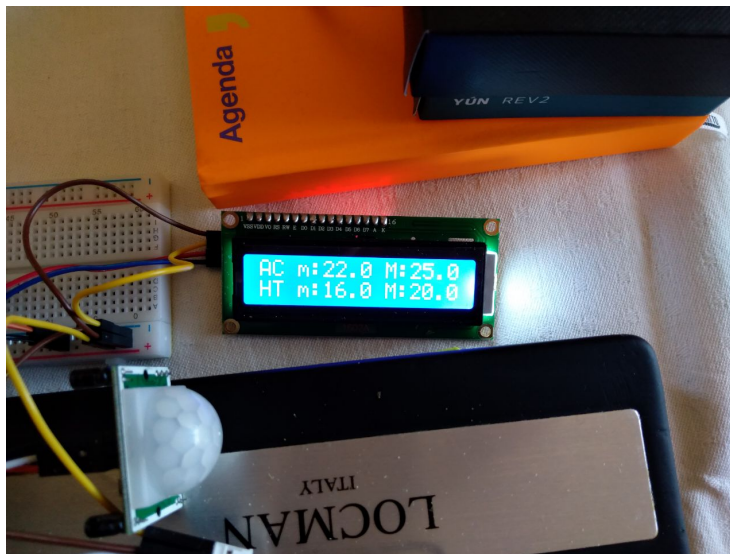
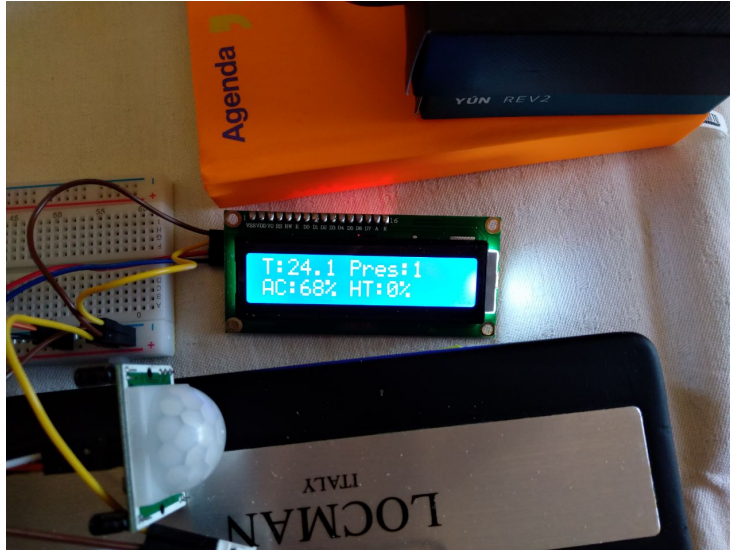
2.1.5 Valori usati per il debug

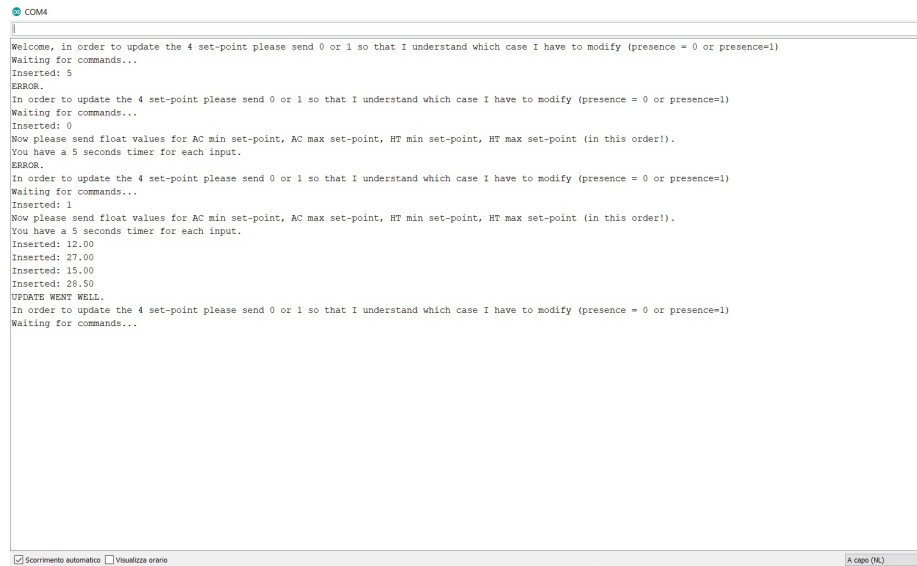
Riportiamo qui i valori usati in fase di debug e consigliati per una fase di test:

- `TIMEOUT_PIR` = 10; (10 s)
- `N_SOUNDS_EVENTS` = 7;
- `SOUND_INTERVAL` = 15; (15 s)
- `TIMEOUT_SOUND` = 30; (30 s)
- `set_point_min_max_1[]` = {22.0, 25.0, 16.0, 20.0};
- `set_point_min_max_0[]` = {24.0, 28.0, 14.0, 18.0};
- `TEMP_PERIOD` = 30; (30 s)

Nota: ovviamente per ottenere un comportamento sensato del sistema i set-point di default vanno modificati a seconda delle condizioni ambientali presenti in fase di debug.

2.1.6 Output su display e Serial Monitor





```
COM4
Welcome, in order to update the 4 set-point please send 0 or 1 so that I understand which case I have to modify (presence = 0 or presence=1)
Waiting for commands...
Inserted: 5
ERROR.
In order to update the 4 set-point please send 0 or 1 so that I understand which case I have to modify (presence = 0 or presence=1)
Waiting for commands...
Inserted: 0
Now please send float values for AC min set-point, AC max set-point, HT min set-point, HT max set-point (in this order!).
You have a 5 seconds timer for each input.
ERROR.
In order to update the 4 set-point please send 0 or 1 so that I understand which case I have to modify (presence = 0 or presence=1)
Waiting for commands...
Inserted: 1
Now please send float values for AC min set-point, AC max set-point, HT min set-point, HT max set-point (in this order!).
You have a 5 seconds timer for each input.
Inserted: 12.00
Inserted: 27.00
Inserted: 15.00
Inserted: 28.50
UPDATE MENU WILL.
In order to update the 4 set-point please send 0 or 1 so that I understand which case I have to modify (presence = 0 or presence=1)
Waiting for commands...
```

Nello screenshot sopra riportato sono stati provati vari casi, tra cui numero iniziale inserito non corretto, numero iniziale corretto ma timer per il primo input scaduto e infine update effettuato in maniera corretta.
Nota: facciamo attenzione, stiamo usando NL (si veda in basso a destra).

2.2 Seconda versione

2.2.1 Componenti HW

I componenti HW usati sono gli stessi di 2.1.1.

2.2.2 Realizzazione HW

Il circuito è analogo a quanto visto in 2.1.2.

2.2.3 Codice Arduino (esteso)

```
1 #include <math.h>
2 #include <LiquidCrystal_PCF8574.h>
3 #include <TimerThree.h>
4
5 LiquidCrystal_PCF8574 lcd(0x27);
6
7 /*GPIO pins*/
8 const int FAN_PIN = 6;
9 const int TEMP_PIN = A0;
10 const int RED_LED_PIN = 13;
11 const int PIR_PIN = 4;
```

```

12 const int SOUND_PIN = 7;
13 const int GREEN_LED_PIN = 8;
14
15 /*Utility constants*/
16 const int MAX_PWM_FAN_AND_LED = 255;
17 const int MAX_ANALOG_TEMP = 1023;
18 const int MIN = 0;
19 const float B = 4275.0;
20 const float R0 = 100000.0;
21 const float T0 = 298.15;
22 const float PIR_PERIOD = 0.5; /*0.5 s, period of PIR checking*/
23 const int TIMEOUT_PIR = 30 * 60;
24 const int N_SOUND_EVENTS = 50;
25 const int SOUND_INTERVAL = 10 * 60;
26 const int TIMEOUT_SOUND = 60 * 60;
27 const int NUMBER_OF_SET_POINTS = 4;
28 const int TEMP_PERIOD = 30; /*30 s, period of temperature checking*/
29
30 /*Global variables*/
31 volatile int green_led_state = LOW;
32 float set_point_min_max_1[] = {22.0, 25.0, 16.0, 20.0}; /*default set-
    point to use when people detected*/
33 float set_point_min_max_0[] = {24.0, 28.0, 14.0, 18.0}; /*default set-
    point to use when people not detected*/
34 float set_point[] = {0, 0, 0, 0}; /*set-point in use*/
35 volatile unsigned long time_latest_detection_pir = 0;
36 volatile bool flag_pir_presence = true;
37 volatile bool flag_sound_presence = false;
38 bool flag_presence = false;
39 unsigned long time_latest_temperature_check = 0;
40 float T = 0;
41 volatile unsigned long *timestamps_last_10_minutes; /*dynamically
    allocated circular vector where 0 is flag for empty entry*/
42 volatile int tail = 0;
43 volatile bool flag_circular_vect = false;
44
45 /*Utility function to copy vector content into another.*/
46 void vector_copy(float v1[], float v2[], int n) {
47     for (int i = 0; i < n; i++) {
48         v1[i] = v2[i];
49     }
50 }
51
52 /*ISR for sound sensor. Run when sound is detected.*/
53 void presence_detected() {
54     if (millis() - timestamps_last_10_minutes[(tail - 1 + N_SOUND_EVENTS)
        % N_SOUND_EVENTS] > 50) {
55         if (millis() - timestamps_last_10_minutes[(tail - 1 +
            N_SOUND_EVENTS) % N_SOUND_EVENTS] < 500 && millis() -
            timestamps_last_10_minutes[(tail - 2 + N_SOUND_EVENTS) %
            N_SOUND_EVENTS] > 500) {
56             green_led_state = !green_led_state;
57             if (green_led_state == HIGH) {
58                 flag_pir_presence = true;
59                 time_latest_detection_pir = millis();
60             }
61             digitalWrite(GREEN_LED_PIN, green_led_state);

```

```

62     }
63     if (flag_circular_vect == false && tail + 1 == N_SOUND_EVENTS) {
64         flag_circular_vect = true;
65     }
66     tail = tail % N_SOUND_EVENTS;
67     timestamps_last_10_minutes[tail] = millis();
68     if (flag_circular_vect == true && timestamps_last_10_minutes[tail]
        - timestamps_last_10_minutes[(tail + 1) % N_SOUND_EVENTS] <
        SOUND_INTERVAL * 1000) {
69         flag_sound_presence = true;
70     }
71     tail++;
72 }
73 }
74
75 /*ISR associated with Timer3 expiration. It checks for motion detection
    */
76 void check_pir_presence() {
77     int a = digitalRead(PIR_PIN);
78     if (a == HIGH) {
79         flag_pir_presence = true;
80         time_latest_detection_pir = millis();
81     }
82 }
83
84 /*setup function*/
85 void setup() {
86     pinMode(FAN_PIN, OUTPUT);
87     pinMode(TEMP_PIN, INPUT);
88     pinMode(RED_LED_PIN, OUTPUT);
89     pinMode(PIR_PIN, INPUT);
90     pinMode(SOUND_PIN, INPUT);
91     pinMode(GREEN_LED_PIN, OUTPUT);
92     digitalWrite(RED_LED_PIN, MIN);
93     digitalWrite(GREEN_LED_PIN, MIN);
94     digitalWrite(FAN_PIN, MIN);
95     lcd.begin(16, 2);
96     lcd.setBacklight(255);
97     lcd.clear();
98     Serial.begin(9600);
99     while (!Serial);
100    Serial.println("Welcome, in order to update the 4 set-point please
        send 0 or 1 so that I understand which case I have to modify (
        presence = 0 or presence=1)");
101    Serial.println("Waiting for commands...");
102    attachInterrupt(digitalPinToInterrupt(SOUND_PIN), presence_detected,
        FALLING);
103    Timer3.initialize(PIR_PERIOD * 1e06);
104    Timer3.attachInterrupt(check_pir_presence);
105    timestamps_last_10_minutes = calloc(N_SOUND_EVENTS, sizeof(unsigned
        long));
106 }
107
108 /*Function to check if millis() time register is in overflow (about
    every 50 days) and to reset variables to keep functioning*/
109 void check_overflow() {
110     bool flag_overflow = false;

```

```

111 for (int i = 0; i < N_SOUND_EVENTS; i++) {
112     if (millis() < timestamps_last_10_minutes[i]) {
113         flag_overflow = true;
114         break;
115     }
116 }
117 if (flag_overflow == true || millis() < time_latest_detection_pir ||
    millis() < time_latest_temperature_check) {
118     for (int i; i < N_SOUND_EVENTS; i++) {
119         timestamps_last_10_minutes[i] = 0;
120     }
121     time_latest_detection_pir = 0;
122     time_latest_temperature_check = 0;
123     flag_pir_presence = false;
124     flag_sound_presence = false;
125     flag_presence = false;
126     flag_circular_vect = false;
127     tail = 0;
128 }
129 }
130
131 /*Function to check if PIR timeout has expired and eventually set PIR
    flag to false*/
132 void check_timeout_pir() {
133     if (flag_pir_presence == true && (millis() -
        time_latest_detection_pir) / 1000 >= TIMEOUT_PIR) {
134         flag_pir_presence = false;
135     }
136 }
137
138 /*Function to check if sound sensor timeout has expired and eventually
    set sound sensor flag to false*/
139 void check_timeout_sound() {
140     if (flag_sound_presence == true && (millis() -
        timestamps_last_10_minutes[(tail - 1 + N_SOUND_EVENTS) %
        N_SOUND_EVENTS]) / 1000 >= TIMEOUT_SOUND) {
141         flag_sound_presence = false;
142     }
143 }
144
145 /*Function to check if at least one of the two sensors (PIR & sound)
    detects presence and
146 eventually set flag to true/false
147 */
148 void check_presence() {
149     if (flag_sound_presence == true || flag_pir_presence == true) {
150         flag_presence = true;
151     }
152     else {
153         if (green_led_state == HIGH) {
154             green_led_state = LOW;
155             digitalWrite(GREEN_LED_PIN, green_led_state);
156         }
157         flag_presence = false;
158     }
159 }
160

```



```

161 /*Function to choose set-point in use based on flag_presence
    information*/
162 void choose_set_point() {
163     switch (flag_presence) {
164         case false: vector_copy(set_point, set_point_min_max_0,
                                NUMBER_OF_SET_POINTS);
165         break;
166         case true: vector_copy(set_point, set_point_min_max_1,
                                NUMBER_OF_SET_POINTS);
167         break;
168     }
169 }
170
171 /*Function for getting temperature measure*/
172 void set_temperature() {
173     int a = analogRead(TEMP_PIN);
174     float R = ((MAX_ANALOG_TEMP / (float)a) - 1) * R0;
175     T = (1 / ((log(R / R0) / B) + (1 / T0))) - 273.15;
176     time_latest_temperature_check = millis();
177 }
178
179 /*Function for checking temperature sensor every 30 s*/
180 void check_temperature() {
181     if (time_latest_temperature_check == 0) {
182         set_temperature();
183     }
184     else if ((millis() - time_latest_temperature_check) / 1000 >=
                TEMP_PERIOD) {
185         set_temperature();
186     }
187 }
188
189 /*Function to manage FAN module (AC). It also returns percentage of
    intensity*/
190 int manage_FAN() {
191     int percentage;
192     if (T > set_point[0] && T < set_point[1]) {
193         analogWrite(FAN_PIN, (int)((T - set_point[0]) / (set_point[1] -
                                set_point[0]) * MAX_PWM_FAN_AND_LED));
194         percentage = (T - set_point[0]) / (set_point[1] - set_point[0]) *
                        100;
195     }
196     else if (T <= set_point[0]) {
197         analogWrite(FAN_PIN, MIN);
198         percentage = 0;
199     }
200     else {
201         analogWrite(FAN_PIN, MAX_PWM_FAN_AND_LED);
202         percentage = 100;
203     }
204     return percentage;
205 }
206
207 /*Function to manage LED (HT). It also returns percentage of intensity
    */
208 int manage_RED_LED() {
209     int percentage;

```

```

210     if (T > set_point[2] && T < set_point[3]) {
211         analogWrite(RED_LED_PIN, (int)(MAX_PWM_FAN_AND_LED - ((T -
            set_point[2]) / (set_point[3] - set_point[2]) *
            MAX_PWM_FAN_AND_LED)));
212         percentage = 100 - ((T - set_point[2]) / (set_point[3] - set_point
            [2]) * 100);
213     }
214     else if (T <= set_point[2]) {
215         analogWrite(RED_LED_PIN, MAX_PWM_FAN_AND_LED);
216         percentage = 100;
217     }
218     else {
219         analogWrite(RED_LED_PIN, MIN);
220         percentage = 0;
221     }
222     return percentage;
223 }
224
225 /*Function for showing informations on the display*/
226 void printStatus(int pFAN, int pLED) {
227     lcd.clear();
228     lcd.print("T:");
229     lcd.print(T, 1); /*1 allows to show only a decimal place*/
230     lcd.print(" Pres:");
231     lcd.print(flag_presence);
232     lcd.setCursor(0, 1); /*first parameter is column number, the second
        one is row number*/
233     lcd.print("AC:");
234     lcd.print(pFAN);
235     lcd.print("% HT:");
236     lcd.print(pLED);
237     lcd.print("%");
238     delay(2000); /*2 s informations on display*/
239     lcd.clear();
240     lcd.print("AC m:");
241     lcd.print(set_point[0], 1);
242     lcd.print(" M:");
243     lcd.print(set_point[1], 1);
244     lcd.setCursor(0, 1);
245     lcd.print("HT m:");
246     lcd.print(set_point[2], 1);
247     lcd.print(" M:");
248     lcd.print(set_point[3], 1);
249     delay(2000); /*2 s informations on display*/
250 }
251
252 /*Utility function to empty serial buffer*/
253 void empty_buffer() {
254     while (Serial.available() > 0) {
255         Serial.read();
256     }
257 }
258
259 /*Function for updating set-point through serial monitor*/
260 void checkforupdate() {
261     bool flag_error = false;
262     int flag_presence_received, i;

```

```

263 float setp[] = {0, 0, 0, 0};
264 if (Serial.available() > 0) {
265     flag_presence_received = Serial.parseInt();
266     Serial.print("Inserted: ");
267     Serial.println(flag_presence_received);
268     if (flag_presence_received != 0 && flag_presence_received != 1) {
269         flag_error = true;
270     }
271     if (flag_error == false) {
272         Serial.println("Now please send float values for AC min set-point
        , AC max set-point, HT min set-point, HT max set-point (in
        this order!).");
273         Serial.println("You have a 5 seconds timer for each input.");
274         empty_buffer();
275         for (i = 0; i < 4 && flag_error == false; i++) {
276             delay(5000);
277             if (Serial.available() > 0) {
278                 setp[i] = Serial.parseFloat();
279                 Serial.print("Inserted: ");
280                 Serial.println(setp[i]);
281                 empty_buffer();
282             }
283             else {
284                 flag_error = true;
285             }
286         }
287     }
288     if (flag_error == true) {
289         Serial.println("ERROR.");
290     }
291     else {
292         Serial.println("UPDATE WENT WELL.");
293         switch (flag_presence_received) {
294             case 0: vector_copy(set_point_min_max_0, setp,
                NUMBER_OF_SET_POINTS);
295                 break;
296             case 1: vector_copy(set_point_min_max_1, setp,
                NUMBER_OF_SET_POINTS);
297                 break;
298         }
299     }
300     Serial.println("In order to update the 4 set-point please send 0 or
        1 so that I understand which case I have to modify (presence =
        0 or presence=1)");
301     Serial.println("Waiting for commands...");
302     empty_buffer();
303 }
304 }
305
306 void loop() {
307     int percentageFAN, percentageLED;
308     noInterrupts();
309     check_overflow();
310     check_timeout_pir();
311     check_timeout_sound();
312     check_presence();
313     interrupts();

```

```

314 choose_set_point();
315 check_temperature();
316 percentageFAN = manage_FAN();
317 percentageLED = manage_RED_LED();
318 printStatus (percentageFAN, percentageLED);
319 checkforupdate();
320 }

```

Nota: la traccia richiedeva di eliminare le funzionalità implementate al punto 4 e di implementarne di nuove, tuttavia vista la struttura robusta della prima versione abbiamo ritenuto opportuno implementare la nuova funzionalità AGGIUNGENDOLA a quanto già fatto, senza rimuovere nulla.

Si elencano di seguito soltanto le principali differenze rispetto alla prima versione (tra parentesi quadre il numero di riga):

- 13] è stata aggiunta la costante `GREEN_LED_PIN = 8` per gestire il pin digitale 8 cui è connesso il LED verde;
- 31] è stata aggiunta la variabile globale `green_led_state` per gestire lo stato del LED ed è stata dichiarata con qualificatore `volatile`;
- 55-62] la funzione `presence_detected` è stata leggermente modificata per permettere, a fronte di due eventi successivi la cui distanza temporale sia inferiore a 0.5 s, il cambio di stato del LED verde; per ulteriori considerazioni si veda la sezione 2.2.4;
- 91] e 93], nella funzione di `setup` abbiamo inizializzato il LED verde;
- 153-157] è stata leggermente modificata la funzione per far sì che, se il flag di presenza del sistema è `false` e il LED verde è acceso, questi venga spento;

Memoria impiegata:

```

Io sketch usa 13886 byte (48%) dello spazio disponibile per i programmi. Il massimo è 28672 byte.
Le variabili globali usano 1024 byte (40%) di memoria dinamica, lasciando altri 1536 byte liberi per le variabili locali. Il massimo è 2560 byte.

```

2.2.4 Criticità riscontrate e soluzioni proposte

Partendo dal presupposto che valgono le considerazioni già fatte per la prima versione, riportiamo qui ulteriori spunti critici:

- Sia che l'esercizio venga svolto seguendo fedelmente la traccia (eliminando quindi le funzionalità del punto 4), sia che l'esercizio venga svolto come fatto dal nostro gruppo, è rilevato un comportamento anomalo. Infatti, nel primo caso, se una persona fosse fuori dal raggio di azione del PIR e battesse le mani due volte per accendere il LED verde, questi si spegnerebbe pressoché immediatamente perché il flag di presenza sarà rimasto `false`. Stesso discorso vale per il nostro progetto, in cui la presenza non viene rilevata se non tramite PIR o 50 eventi sonori in 10 minuti. Per ovviare a questo problema si è pensato di controllare quando il LED verde

viene acceso (non spento, attenzione!) a seguito di un doppio battito di mani così da poter settare il flag del PIR (si è scelto questo in quanto meno impattante per via della durata inferiore del timeout) e supporre la presenza di una persona in stanza. In tal modo si evita l'immediato power off del LED.

- Inizialmente è stato notato un problema nella gestione dei battiti. Infatti, a seguito di 3 battiti ravvicinati il sistema interpretava i primi due come accensione e il terzo, in associazione con il secondo, come spegnimento. Per evitare ciò si è inserito un ulteriore controllo, dapprima si verifica che l'evento sonoro corrente sia ravvicinato all'ultimo rilevato, poi si controlla che disti anche di 0.5 s dal penultimo, così da discriminare coppie di eventi.

2.2.5 Valori usati per il debug

Sono stati usati gli stessi valori di debug usati in 2.1.5.

2.2.6 Video dimostrativo

Al seguente link <https://youtu.be/xC0WT3w6gTo> è possibile guardare un breve video dimostrativo del risultato ottenuto.

Capitolo 3

Comunicazione tramite interfacce REST e MQTT

3.1 Arduino Yún come server HTTP

3.1.1 Componenti HW

I componenti HW usati sono gli stessi di 2.1.1.

3.1.2 Realizzazione HW

Il circuito è analogo a quanto visto in 2.1.2.

3.1.3 Codice Arduino

```
1 #include <Bridge.h>
2 #include <BridgeServer.h>
3 #include <BridgeClient.h>
4 #include <ArduinoJson.h>
5
6 const int GREEN_LED_PIN = 8;
7 const int TEMP_PIN = A0;
8 const float B = 4275.0;
9 const float R0 = 100000.0;
10 const float T0 = 298.15;
11 const int MAX_ANALOG_TEMP = 1023;
12 const int capacity = JSON_OBJECT_SIZE(2) + JSON_ARRAY_SIZE(1) +
    JSON_OBJECT_SIZE(4) + 40;
13 /*Capacity is the maximum dimension of JSON to encode/decode
14  JSON contains two object ("bn" and "e"), of which "e" is an array
15  with 1 element and the element contains 4 fields. Plus, 40 additional
16  chars to store string characters (example: temperature, led, Cel, ecc
    .)*/
17 BridgeServer server;
18 DynamicJsonDocument doc_snd(capacity);
```

```

19 /*JSON document for sent data*/
20
21 /*Setup function*/
22 void setup() {
23     pinMode(GREEN_LED_PIN, OUTPUT);
24     pinMode(TEMP_PIN, INPUT);
25     pinMode(LED_BUILTIN, OUTPUT);
26     /*Internal LED*/
27     digitalWrite(GREEN_LED_PIN, LOW);
28     digitalWrite(LED_BUILTIN, LOW);
29     Bridge.begin();
30     digitalWrite(LED_BUILTIN, HIGH);
31     /*Turn on internal LED when connection is enestablished*/
32     server.listenOnLocalhost();
33     server.begin();
34 }
35
36 /*Function for encoding temperature/LED info in JSON*/
37 String senMlEncode(String res, float v, String unit) {
38     doc_snd.clear();
39     doc_snd["bn"] = "Yun";
40     doc_snd["e"][0]["n"] = res;
41     doc_snd["e"][0]["t"] = (millis() / 1000);
42     doc_snd["e"][0]["v"] = v;
43     if (unit != "") {
44         doc_snd["e"][0]["u"] = unit;
45     }
46     else {
47         doc_snd["e"][0]["u"] = (char*)NULL;
48     }
49     String output;
50     serializeJson(doc_snd, output);
51     return output;
52 }
53
54 /*Function for getting temperature measure*/
55 float get_temperature() {
56     int a = analogRead(TEMP_PIN);
57     float R = ((MAX_ANALOG_TEMP / (float)a) - 1) * R0;
58     float T = (1 / ((log(R / R0) / B) + (1 / T0))) - 273.15;
59     int temp = T * 100;
60     T = (float)temp / 100;
61     return T;
62 }
63
64 /*Function used to process new GET requests*/
65 void process(BridgeClient client) {
66     String command = client.readStringUntil('/');
67     /*Parse 1st URL element*/
68     command.trim();
69     /*Delete \n or similar*/
70     if (command == "led") {
71         int val = client.parseInt();
72         /*Parse 2nd URL element*/
73         if (val == 0 || val == 1) {
74             digitalWrite(GREEN_LED_PIN, val);
75             printResponse(client, 200, senMlEncode(F("led"), val, F("")));

```

```

76     }
77     else {
78         printResponse(client, 400, F(""));
79     }
80 }
81 else if (command == "temperature") {
82     printResponse(client, 200, senMLEncode(F("temperature"),
83         get_temperature(), F("Cel"))));
84 }
85 else {
86     printResponse(client, 404, F(""));
87 }
88 }
89 /*Function used to provide output*/
90 void printResponse(BridgeClient client, int code, String body) {
91     client.println("Status: " + String(code));
92     if (code == 200) {
93         client.println(F("Content-type: application/json; charset=utf-8"));
94         client.println();
95         client.println(body);
96     }
97 }
98
99 void loop() {
100     BridgeClient client = server.accept();
101     if (client) {
102         process(client);
103         client.stop();
104     }
105     delay(50);
106 }

```

In questo sketch si esegue un server HTTP sulla scheda Arduino, capace di rispondere a richieste GET provenienti dalla rete locale.

Sono state incluse le apposite librerie per la trasmissione su rete locale e la libreria ArduinoJson, usata per codificare i dati in formato JSON.

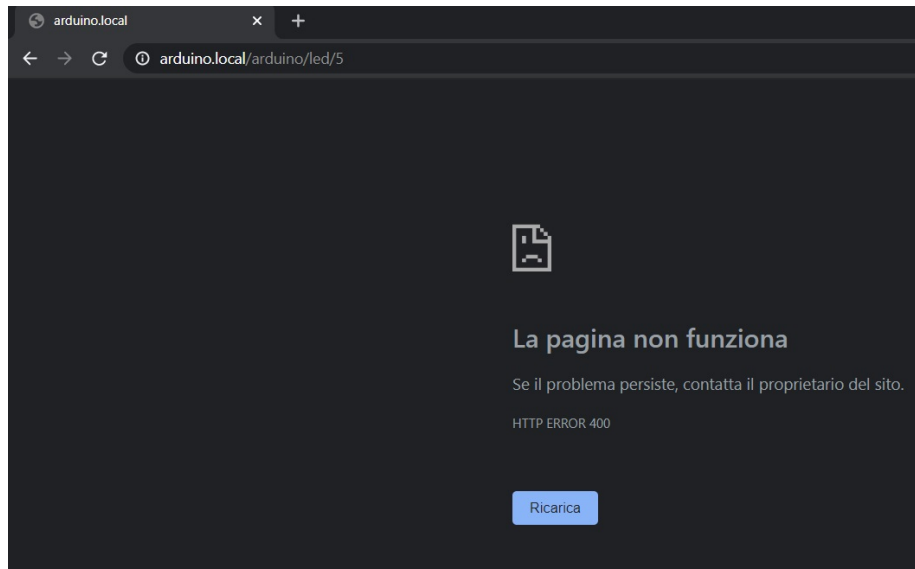
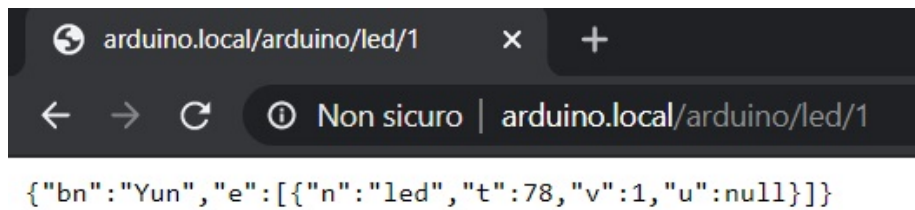
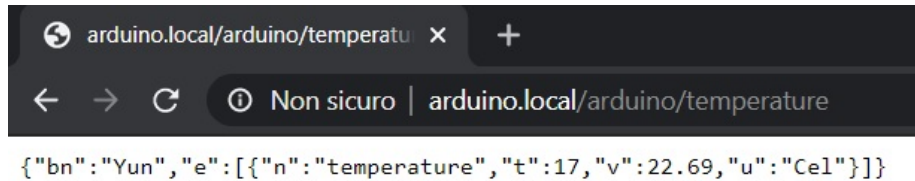
La funzione senMLEncode si occupa di riempire una struttura simile ad un dizionario e di serializzarla per l'appunto in formato JSON.

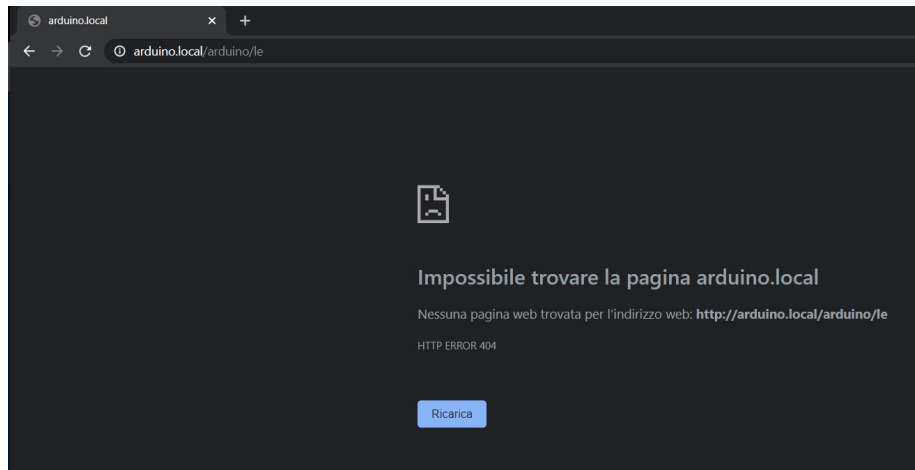
La funzione process è addetta alla gestione delle richieste GET in arrivo, verifica gli elementi passati tramite URL e invoca la printResponse con codici 200, 400 oppure 404.

La printResponse invia le informazioni al client. Se lo status code è 200 viene stampato anche il body (JSON ottenuto da senMLEncode), altrimenti si invia soltanto lo status code senza body. Nota: nell'invviare richieste errate vengono generati errori 404 e 400 ma la reason phrase non può essere modificata, per questo si otterrà 400 OK o 404 OK; ciò che fa fede comunque è lo status code. Nella funzione loop si controlla l'arrivo di nuove richieste e si setta un delay

minimo per consentire al server di non essere sovraccaricato.

3.1.4 Output





3.2 Arduino Yún come client HTTP

3.2.1 Componenti HW

I componenti HW usati sono gli stessi di 2.1.1.

3.2.2 Realizzazione HW

Il circuito è analogo a quanto visto in 2.1.2.

3.2.3 Codice Arduino

```
1 #include <Process.h>
2 #include <Bridge.h>
3 #include <ArduinoJson.h>
4
5 const int TEMP_PIN = A0;
6 const float B = 4275.0;
7 const float R0 = 100000.0;
8 const float T0 = 298.15;
9 const int MAX_ANALOG_TEMP = 1023;
10 const int capacity = JSON_OBJECT_SIZE(2) + JSON_ARRAY_SIZE(1) +
    JSON_OBJECT_SIZE(4) + 40;
11 /*Capacity is the maximum dimension of JSON to encode/decode
12  JSON contains two object ("bn" and "e"), of which "e" is an array
13  with 1 element and the element contains 4 fields. Plus, 40 additional
14  chars to store string characters (example: temperature, led, Cel, ecc
    .)*/
15 DynamicJsonDocument doc_snd(capacity);
16 /*JSON document for sent data*/
17
18 /*Setup function*/
19 void setup() {
20     pinMode(TEMP_PIN, INPUT);
```

```

21  pinMode(LED_BUILTIN, OUTPUT);
22  /*Internal LED*/
23  digitalWrite(LED_BUILTIN, LOW);
24  Serial.begin(9600);
25  Bridge.begin();
26  digitalWrite(LED_BUILTIN, HIGH);
27  }
28
29  /*Function for encoding temperature info in JSON*/
30  String senMlEncode(String res, float v, String unit) {
31      doc_snd.clear();
32      doc_snd["bn"] = "Yun";
33      doc_snd["e"][0]["n"] = res;
34      doc_snd["e"][0]["t"] = (millis() / 1000);
35      doc_snd["e"][0]["v"] = v;
36      if (unit != "") {
37          doc_snd["e"][0]["u"] = unit;
38      }
39      else {
40          doc_snd["e"][0]["u"] = (char*)NULL;
41      }
42      String output;
43      serializeJson(doc_snd, output);
44      return output;
45  }
46
47  /*Function for getting temperature measure*/
48  float get_temperature() {
49      int a = analogRead(TEMP_PIN);
50      float R = ((MAX_ANALOG_TEMP / (float)a) - 1) * R0;
51      float T = (1 / ((log(R / R0) / B) + (1 / T0))) - 273.15;
52      int temp = T * 100;
53      T = (float)temp / 100;
54      return T;
55  }
56
57  /*Function for using curl command and send POST request to server*/
58  void postRequest(String data) {
59      Process p;
60      p.begin("curl");
61      p.addParameter("-H");
62      p.addParameter("Content-Type: application/json");
63      p.addParameter("-X");
64      p.addParameter("POST");
65      p.addParameter("-d");
66      p.addParameter(data);
67      p.addParameter("http://192.168.8.112:8080/temperature/log");
68      /*IP address of server is needed here*/
69      p.run();
70      if (p.exitValue() != 0) {
71          Serial.print("Error! Exit value: ");
72          Serial.print(p.exitValue());
73          Serial.println(" Check curl exit value references for more
              informations.");
74      }
75      else {
76          Serial.println("DATA SENT.");

```

```

77     }
78 }
79
80 void loop() {
81     postRequest(senMLEncode(F("temperature"), get_temperature(), F("Cel")
82         ));
83     delay(10000);
84 }

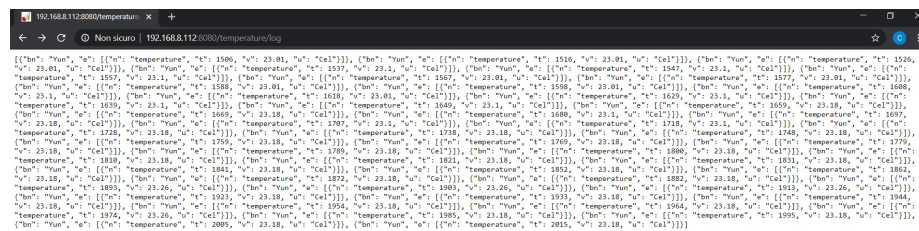
```

In questo esercizio vediamo come le due parti, HW e SW, inizino a convergere. La scheda Arduino è usata come client HTTP e ogni 10 secondi inoltra una POST request al server in ascolto sulla rete locale su `http://hostname:port/temperature/log`. Le informazioni inviate consistono in un JSON contenente i dati di temperatura rilevati dal sensore. Come richiesto si mostrano sul Serial Monitor eventuali errori ottenuti dall'esecuzione del comando curl.

È possibile inoltre effettuare una GET request allo stesso indirizzo per visualizzare la lista dei 50 JSON ricevuti più recentemente.

Nella cartella è disponibile anche il codice sorgente del server HTTP.

3.2.4 Output



3.3 Arduino Yún come publisher e subscriber MQTT

3.3.1 Componenti HW

I componenti HW usati sono gli stessi di 2.1.1.

3.3.2 Realizzazione HW

Il circuito è analogo a quanto visto in 2.1.2.

3.3.3 Codice Arduino

```

1 #include <MQTTclient.h>
2 #include <Bridge.h>
3 #include <ArduinoJson.h>
4
5 const int GREEN_LED_PIN = 8;

```

```

6 const int TEMP_PIN = A0;
7 const float B = 4275.0;
8 const float R0 = 100000.0;
9 const float T0 = 298.15;
10 const int MAX_ANALOG_TEMP = 1023;
11 const int capacity = JSON_OBJECT_SIZE(2) + JSON_ARRAY_SIZE(1) +
    JSON_OBJECT_SIZE(4) + 40;
12 /*Capacity is the maximum dimension of JSON to encode/decode
13 JSON contains two object ("bn" and "e"), of which "e" is an array
14 with 1 element and the element contains 4 fields. Plus, 40 additional
15 chars to store string characters (example: temperature, led, Cel, ecc
    .)*/
16
17 String my_base_topic = "/tiot/3";
18 DynamicJsonDocument doc_rec(capacity);
19 /*JSON document for received data*/
20 DynamicJsonDocument doc_snd(capacity);
21 /*JSON document for sent data*/
22
23 /*Setup function*/
24 void setup() {
25     pinMode(GREEN_LED_PIN, OUTPUT);
26     pinMode(TEMP_PIN, INPUT);
27     pinMode(LED_BUILTIN, OUTPUT);
28     /*Internal LED*/
29     digitalWrite(GREEN_LED_PIN, LOW);
30     digitalWrite(LED_BUILTIN, LOW);
31     Serial.begin(9600);
32     Bridge.begin();
33     digitalWrite(LED_BUILTIN, HIGH);
34     /*Turn on internal LED when connection is enestablished*/
35     mqtt.begin("test.mosquitto.org", 1883);
36     mqtt.subscribe(my_base_topic + String("/led"), setLedValue);
37     /*Subscribe to /tiot/3/led and call setLedValue when new data are
        available*/
38 }
39
40 /*Function to manage incoming data for subscribed topic. It also check
    for errors*/
41 void setLedValue(const String& topic, const String& subtopic, const
    String& message) {
42     bool flagErr = false;
43     DeserializationError err = deserializeJson(doc_rec, message);
44     if (err) {
45         Serial.print(F("deserializeJson() failed with code "));
46         Serial.println(err.c_str());
47         flagErr = true;
48     }
49     else {
50         if (doc_rec["bn"] != "Yun") {
51             Serial.println("Wrong base name, please insert Yun");
52             flagErr = true;
53         }
54         if (doc_rec["e"][0]["n"] != "led") {
55             Serial.println("Wrong resource, please select available resource
                like led");
56             flagErr = true;

```

```

57     }
58     if (doc_rec["e"][0]["t"] != (char*)NULL) {
59         Serial.println("You cannot specify timestamp, please leave null
        value");
60         flagErr = true;
61     }
62     if ((doc_rec["e"][0]["v"]) != 1 && (doc_rec["e"][0]["v"]) != 0) {
63         Serial.println("Wrong value! Possible values are 0 or 1");
64         flagErr = true;
65     }
66     if (doc_rec["e"][0]["u"] != (char*)NULL) {
67         Serial.println("You cannot specify unit of measurement, please
        leave null value");
68         flagErr = true;
69     }
70 }
71 if (flagErr == false) {
72     digitalWrite(GREEN_LED_PIN, (doc_rec["e"][0]["v"]));
73     Serial.println("Request satisfied");
74 }
75 doc_rec.clear();
76 }
77
78 /*Function for getting temperature measure.*/
79 float get_temperature() {
80     int a = analogRead(TEMP_PIN);
81     float R = ((MAX_ANALOG_TEMP / (float)a) - 1) * R0;
82     float T = (1 / ((log(R / R0) / B) + (1 / T0))) - 273.15;
83     int temp = T * 100;
84     T = (float)temp / 100;
85     return T;
86 }
87
88 /*Function for encoding temperature info in JSON*/
89 String senMlEncode(String res, float v, String unit) {
90     doc_snd.clear();
91     doc_snd["bn"] = "Yun";
92     doc_snd["e"][0]["n"] = res;
93     doc_snd["e"][0]["t"] = (millis() / 1000);
94     doc_snd["e"][0]["v"] = v;
95     if (unit != "") {
96         doc_snd["e"][0]["u"] = unit;
97     }
98     else {
99         doc_snd["e"][0]["u"] = (char*)NULL;
100     }
101     String output;
102     serializeJson(doc_snd, output);
103     return output;
104 }
105
106 void loop() {
107     mqtt.monitor();
108     /*Monitor topic of interest*/
109     String message = senMlEncode("temperature", get_temperature(), "Cel")
        ;
110     mqtt.publish(my_base_topic + String("/temperature"), message);

```

```

111  /*Publish every second temperature info as JSON to /tiot/3/
      temperature*/
112  delay(1000);
113  }

```

In questo sketch è stata inclusa la libreria MQTTclient per implementare le funzioni di publish e subscribe e permettere alla scheda di agire sia come publisher, sia come subscriber. È stato usato il broker MQTT pubblico test.mosquitto.org. La funzione setLedValue è quella che si occupa della subscribe, consentendo l'accensione/lo spegnimento del LED verde a seguito di una publish. Sono stati implementati vari controlli d'errore.

Nella funzione loop ritroviamo invece mqtt.monitor che permette di controllare eventuali publish arrivate alla scheda e mqtt.publish che permette alla scheda di agire come publisher e inviare i dati sulla temperatura.

3.3.4 Output

Per inviare i comandi publish e subscribe abbiamo usato MQTT.fx.

Subscribe:



Publish:

