Home Automation: Air conditioner

Introduzione

Il progetto *Home automation - air conditioner* è parte modulare di un progetto più grande *Home automation*.

Questa porzione si occuperà esclusivamente del controllo remoto di un **condizionatore ad aria** senza disporre del modulo wifi (spesso molto costoso) e **senza doversi interfacciare direttamente con l'elettronica del climatizzatore**.

Tutto ciò avviene in modo **sicuro** e **privo di rischi** poiché comunicheremo con il condizionatore emettendo dei segnali IR (infrarossi) grazie ad un microcontrollore (ad esempio raspberry pi), al quale invieremo i comandi attraverso la rete.

Per fare ciò sarà prima necessario leggere i dati che il telecomando "originale" invia al climatizzatore, analizzarli e codificarli e, successivamente, ricreare un segnale codificato in base alle richieste dell'utente (stato del climatizzatore, modalità, temperatura, ecc) che verrà poi decodificato e inviato al climatizzatore.

Materiali

- Raspberry Pi (ho utilizzato l'O.S. raspbian)
- Breadboard
- Ricevitore IR (TSOP38238)
- LED IR
- Transistor NPN PN2222
- Resistenze da 1K ohm
- Resistenza da 220 ohm
- Climatizzatore con telecomando IR 😁

Considerazioni iniziali

Un emettitore IR alterna momenti in cui il segnale è alto (led acceso) con momenti in cui il segnale è basso (led spento).
Nella maggior parte dei telecomandi questo processo avviene con una frequenza di 38kHz, quindi il ciclo accensione-spegnimento del led durerà in totale 1/38000 secondi = 26,3 microsecondi, di cui per 13.15 il led sarà acceso e per 13.15 il led sarà spento.

L'invio di informazione è composta da questa alternanza per un tot di microsecondi e da un tot di microsecondi di pausa

Suggerimento: è possibile visualizzare gli IR inviati da un emettitore attraverso una fotocamera digitale.

Televisore - caso semplice

I normali telecomandi per TV inviano in loop un segnale per ogni tasto premuto.
Un esempio di pseudocodice ad alto livello per descrivere ciò che accade alla pressione un tasto del telecomando di un televisore potrebbe essere il seguente:

```
on_button_pressed()
  do
    send(btn.timeOn)
    sleep(btn.timeOff)
    send(btn.timeOn)
    sleep(btn.timeOff)
  while(pressed)
```

Un esempio concreto di un singolo ciclo potrebbe quindi essere:

- far lampeggiare il led per 1300 microsecondi
- attendere 400 microsecondi con il led spento
- far lampeggiare nuovamente il led per 1300 microsecondi

Il ricevitore, invece, è sempre in attesa di un segnale e può leggere solo le informazioni inviate da emettitori alla sua stessa frequenza (38kHz nel nostro caso).

Climatizzatore - caso corrente

Il telecomando di un climatizzatore funziona in modo leggermente diverso poiché deve inviare tutte le informazioni (stato on-off, modalità, temperatura...) per risolvere i problemi di sincronizzazione che si possono verificare tra il display e l'unità modificando dei parametri quando quest'ultima è fuori portata o il sensore è coperto.

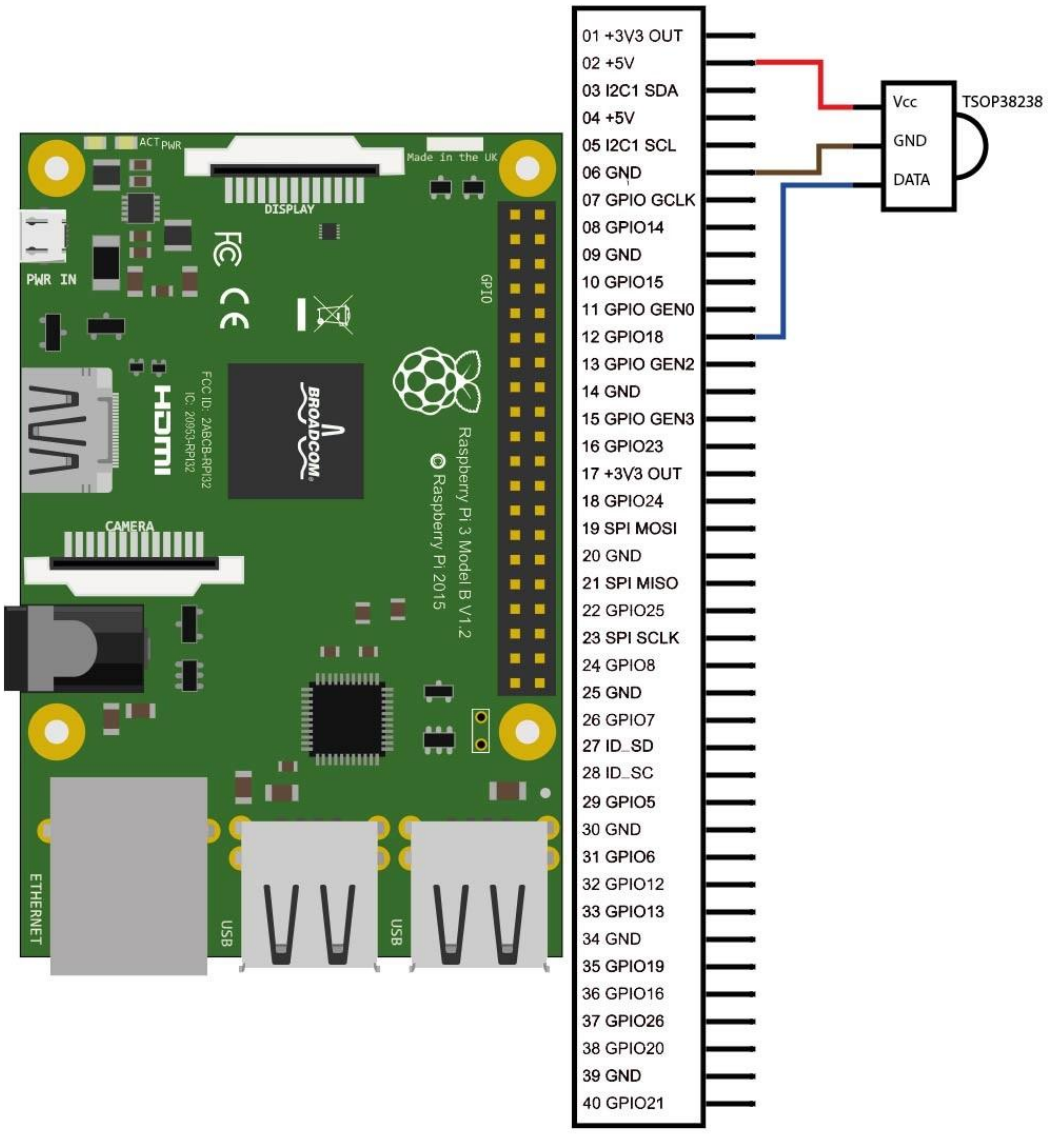
Il segnale inviato non è quindi sempre lo stesso in loop, ma un susseguirsi di accensioni e spegnimenti con tempi differenti che facciano capire al climatizzatore le informazioni di cui si è detto sopra.

Un esempio molto semplificato di pseudocodice in questo caso potrebbe essere il seguente:

```
on_button_pressed()
  s = getEncoding(status)
  send(s.timeOn)
  sleep(s.timeOff)

  m = getEncoding(mode)
  send(m.timeOn)
  sleep(m.timeOff)

  t = getEncoding(temperature)
  send(t.timeOn)
  sleep(t.timeOff)
```

Leggere IR con Raspberry pi

Setup



Per leggere segnali IR con raspberry pi è possibile utilizzare la libreria LIRC (Linux Infrared Remote Control).

Per installarla è sufficiente aprire il terminale e lanciare il comando:
`sudo apt-get install lirc`

Successivamente aprire il file /etc/modules:
`sudo nano /etc/modules`

Fare in modo che il demone si avvii all'avvio del sistema e settare i pin che utilizzeremo per pilotare emettitore e ricevitore IR:

```
lirc_dev
lirc_rpi gpio_in_pin=18 gpio_out_pin=17
```

NOTA: per salvare bisogna premere `Ctrl+X -> y -> Invio`

Adesso bisogna aprire il file di configurazione hardware di lirc:

```
sudo nano /etc/lirc/hardware.conf
```

E modificare come segue le linee in questione:

```
DRIVER="default"
DEVICE="/dev/lirc0"
MODULES="lirc_rpi"
```

Fatto! Ora bisogna riavviare il raspberry pi

```
sudo reboot
```

NOTA:
Per far funzionare LIRC su Raspberry Pi 2, bisogna aprire /boot/config.txt:

```
sudo nano /boot/config.txt
```

E aggiungere alla fine:

```
dtoverlay=lirc-rpi,gpio_in_pin=18,gpio_out_pin=17,gpio_in_pull=up
```

Leggere i segnali IR

Per verificare che il sensore funzioni e che sia configurato correttamente bisogna stoppare il demone lirc

```
sudo /etc/init.d/lirc stop
```

E lanciare il programma che ci mostri i tempi di accensione/spegnimento del led IR:

```
mode2 -d /dev/lirc0
```

Ora è sufficiente puntare un qualsiasi telecomando verso il sensore e premere un tasto per vedere qualcosa.

LIRC mette a disposizione un insieme di "etichette" per i bottoni di un telecomando. La lista è disponibile lanciando il comando

```
irrecord --list-namespace
```

Ma fanno riferimento solo a TV, quindi in questo caso non ci interessa.

Raccolta dati dal condizionatore

Per comprendere come i dati vengono organizzati ed inviati, e quindi come avviene la comunicazione tra telecomando e climatizzatore, è necessario capire ciò che cambia tra un comando e l'altro.
Come detto in precedenza, ad ogni invio corrisponde una sequenza di dati in base allo stato del telecomando, quindi per capire cosa cambia bisogna salvare il segnale inviato per ogni cambiamento, come ON, OFF, modalità, velocità ventole, altezza alette, temperatura…
Effettuando **un solo cambio per volta**

Per salvare un singolo comando è necessario lanciare il comando mode2 con l'opzione -m (che scrive i dati in modo più semplice da elaborare) e salvarne il contenuto su un file (con il nome di ciò che stiamo analizzando).
In questo caso non comparirà l'output a schermo (perché appunto lo stiamo scrivendo su file). Una volta lanciato il comando e premuto il tasto del condizionatore sarà necessario premere Ctrl+C per smettere di registrare.

```
mode2 -d /dev/lirc0 -m > mode_auto

mode2 -d /dev/lirc0 -m > mode_cool
```

…

NOTA:
potrebbe essere necessario lanciare questi comandi come amministratore con sudo.

I files generati saranno composti da un valore singolo sulla prima riga e altre N righe da 6 colonne. I valori salvati indicano per quanto tempo il led rimane acceso/spento (ovviamente in microsecondi).

Il singolo valore sulla prima riga indica quanto tempo è passato da quando è stato avviato il comando a quando il sensore ha ricevuto un messaggio, quindi corrisponde ad un tempo "off" da non considerare.

Essendo in microsecondi, i valori salvati possono differire l'uno dall'altro, quindi considereremo un valore medio per valori simili.

ATTENZIONE: Da ora in poi farò riferimento ai valori registrati dal telecomando del mio climatizzatore Panasonic, che potrebbero differire per le altre marche

Dando un'occhiata ad uno dei files generati si possono notare alcune cose:

- ci sono dei valori ricorrenti, come 440 e 1280
- il tempo di accensione del led vale sempre 440 tranne nel primo valore ed in un punto "centrale" in cui vale 3500

- il tempo di spegnimento vale
    - 440 o 1280 se quello di accensione è 440 tranne in un caso in cui vale 9880 (cosa sarà?)
    - 1750 se quello di accensione è 3500

Mettendo a confronto un paio di files è possibile notare che la parte che va dall'inizio fino allo strano 9880 di spegnimento seguito dalla seconda coppia 3500-1750 è sempre uguale, quindi potremmo ipotizzare che sia un identificatore del condizionatore (probabilmente della marca). Chiameremo questa porzione **header**, mentre la parte successiva, che cambia di volta in volta, la chiameremo **payload**.

Per leggere ed analizzare meglio questi valori è possibile scrivere un programma che prenda in input la stringa di valori raw e dia in output una stringa in binario

A partire dalle informazioni raccolte sopra possiamo assumere che una coppia accensione-spegnimento:

- vale 0 se accensione = 440 e spegnimento = 440;
- vale 1 se accensione = 440 e spegnimento = 1280;
- indica l'inizio di una sezione se accensione = 3500 e spegnimento = 1750;
- Indica la fine di una sezione se accensione =440 e spegnimento = 9880.

Il programma, scritto in typescript, è il seguente
encode.ts

```typescript
/**
* Dato un insieme di files con i valori raw
* ritorna un insieme di files con i valori binari
*/

import { Constants } from './constants';
import { Utils } from './utils/utils';
import * as fs from 'fs';

export class Encode
{
 private readonly MARGIN: number = 150;

 private compare(n1: number, n2: number): boolean
 {
   return Utils.compare(n1, n2, this.MARGIN);
 }

 private rawCoupleToBit(on: number, off: number): string
```

```
{
  if (
    this.compare(on, Constants.SIGNAL_VALUES.INTRO)
    && this.compare(off, Constants.SIGNAL_VALUES.INTRO2)
  ) {
    return '';
  }
  if (
    this.compare(on, Constants.SIGNAL_VALUES.SHORT)
    && this.compare(off, Constants.SIGNAL_VALUES.SHORT)
  ) {
    return '0';
  }
  if (
    this.compare(on, Constants.SIGNAL_VALUES.SHORT)
    && this.compare(off, Constants.SIGNAL_VALUES.LONG)
  ) {
    return '1';
  }
  if (
    this.compare(on, Constants.SIGNAL_VALUES.SHORT)
    && this.compare(off, Constants.SIGNAL_VALUES.SEPARATOR)
  ) {
    return ':';
  }
  throw 'Error: Out of range ('+ on + ', ' + off +')';
}

private encode(rawData: string[]): string
{
  let s: string = '';
  for (let i: number = 1; i < rawData.length; i+=2) {
    const on: number = parseInt(rawData[i-1]);
    const off: number = parseInt(rawData[i]);
    s += this.rawCoupleToBit(on, off);
  }
  return s;
}
```

```
 private encodeFile(fileName: string): void
 {
   const fileIn: string = Constants.FOLDER.ORIGINAL_VALUES + '/' + fileName;
   const fileOut: string = Constants.FOLDER.BINARY_VALUES + '/' + fileName;

   fs.readFile(fileIn, Constants.FILE_ENCODING, (err, fileData) => {
     if (err) throw err;

     const rawData: string[] = fileData
       .match(/[0-9]+/g) // cerca tutti i valori numerici e li mette in un
array
       .splice(1); // elimina il primo valore

     const encodedString = this.encode(rawData);

     fs.writeFile(
       fileOut,
       encodedString,
       {
         encoding: Constants.FILE_ENCODING
       },
       (err) => { if (err) throw err; }
     );
   });
 }

 public init(): void
 {
   fs.readdir(Constants.FOLDER.ORIGINAL_VALUES, (err, files) => {
     if (err) throw err;
     files.forEach(file => this.encodeFile(file));
     console.log('Completed');
   });
 }
}

new Encode().init();
```

Analisi dei dati

Adesso abbiamo dei files contenenti la versione binaria di ciò che il telecomando invia al climatizzatore.
Confrontando i vari payload è facile capire quale porzione di bit si riferisce alla temperatura piuttosto che alla modalità, piuttosto che la velocità delle ventole o altro.

Di seguito le 6 stringhe binarie che si occupano dell'altezza delle alette del mio climatizzatore:

```
0100000000001000000011100100000000000000010100000101000000001100001011011000000000000111000000000111000000000000000100000010000000000000000011001111
0100000000001000000011100100000000000000010100000101000000001010001011011000000000000111000000000111000000000000000100000010000000000000000010111111
0100000000001000000011100100000000000000010100000101000000001110001011011000000000000111000000000111000000000000000100000010000000000000000010101111
0100000000001000000011100100000000000000010100000101000000001001001011011000000000000111000000000111000000000000000100000010000000000000000011011111
0100000000001000000011100100000000000000010100000101000000001101001011011000000000000111000000000111000000000000000100000010000000000000000011101111
0100000000001000000011100100000000000000010100000101000000001111101011011000000000000111000000000111000000000000000100000010000000000000000010000000
```

E' facile intuire che i bit che si occupano dell'azione di cui sopra sono quelli che vanno da 64 a 67 e variano nel seguente modo:

altissimo: 1000
alto: 0100
medio: 1100
basso: 0010
bassissimo: 1010
auto: 1111

Gli ultimi 8 bit invece cambiano sempre e sono i bit di checksum, per verificare che i dati siano sensati e corretti.

Dopo alcuni test è possibile notare che la checksum è calcolata solo per la parte del payload (non includendo l'introduzione), ed è il risultato della semplice somma di tutti i byte, mantenendo solo un byte del risultato (ignorando gli overflow). La checksum è fatta da un sistema Big Endian, quindi ogni byte deve essere invertito prima di essere sommato, quindi il risultato viene invertito di nuovo.

Inviare segnali IR

Codificare i dati

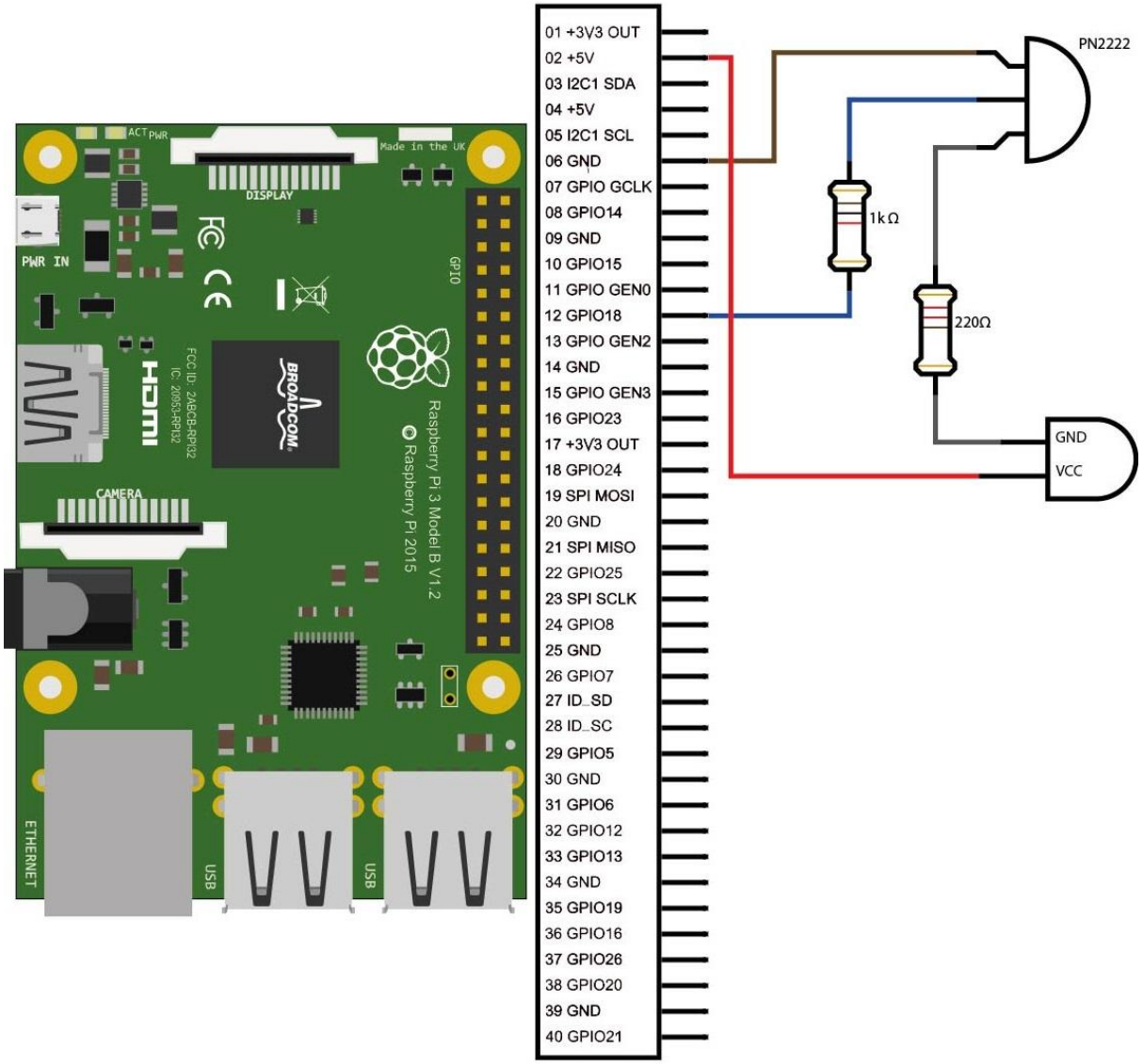
I dati da inviare devono rispettare un formato specifico, che, come quello dei dati che si ricevono, si compone di valori: i tempi di accensione e spegnimento del led IR.
A questo scopo di seguito il codice dello script decode che si occupa di trasformare una stringa binaria in un insieme di dati raw

decode.ts

```
/**
* Data una stringa binaria
* restituisce una raw string
*/

import { Constants } from './constants';
import { BinaryString } from './utils/binary';
import { Utils } from "./utils/utils";

export class Decode
{
 private arr: any[] = [];

 binaryToRaw(binaryString: BinaryString): string
 {
   const binaryStrings: BinaryString[] = Utils.chunk(binaryString, 1);
   let raw: string = '';

   binaryStrings
     .forEach((bit: '0' | '1') => {
       this.arr.push({text: 1, time:
Constants.SIGNAL_VALUES.SHORT.toString()});

       if (bit === '0') {
         this.arr.push({text: 0, time:
Constants.SIGNAL_VALUES.SHORT.toString()});

         raw += Constants.SIGNAL_VALUES.SHORT.toString() + " "
           + Constants.SIGNAL_VALUES.SHORT.toString() + " ";
```

```
      } else if (bit === '1') {
        this.arr.push({text: 0, time:
Constants.SIGNAL_VALUES.LONG.toString()});

        raw += Constants.SIGNAL_VALUES.SHORT.toString() + " "
          + Constants.SIGNAL_VALUES.LONG.toString() + " ";
      }
    });

  return raw;
 }

 decode(binaryString: BinaryString): string
 {
  /**
   * Scorre la stringa binaria
   * scompone lo 0 in SHORT SHORT e l'1 in SHORT LONG
   *
   * La risposta inizia sempre per INTRO INTRO2
   * i : diventano SHORT SEPARATOR
   */

  const bs: BinaryString[] = binaryString.split(':');

  const header: BinaryString = bs[0];
  const payload: BinaryString = bs[1];

  return Constants.SIGNAL_VALUES.INTRO.toString() + " "
    + Constants.SIGNAL_VALUES.INTRO2.toString() + " "
    + this.binaryToRaw(header)
    + Constants.SIGNAL_VALUES.SHORT.toString() + " "
    + Constants.SIGNAL_VALUES.SEPARATOR.toString() + " "
    + Constants.SIGNAL_VALUES.INTRO.toString() + " "
    + Constants.SIGNAL_VALUES.INTRO2.toString() + " "
    + this.binaryToRaw(payload)
    + Constants.SIGNAL_VALUES.SHORT;
 }
}
```

Inviare i dati



Una volta che i dati sono stati decodificati, quindi riportati da sequenza binaria a dati raw, è possibile procedere con l'invio.
Per fare questo utilizziamo un semplice programma in c che prende in input la stringa di dati raw e, utilizzando la libreria irslinger, invia i dati nel modo opportuno.

irslinger.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "irslinger.h"

int powi(int base, int exp);
```

```
int length(char *input);
int *stringArrayToIntArray(char *numbers[], int count);

int main(int argc, char *argv[])
{
 if (argc < 2) {
   printf("Missing parameters\n");
   return -1;
 }

 int codesCount = argc - 1;
 int* codes = stringArrayToIntArray(&argv[1], codesCount);
 // int codesCount = sizeof(codes) - sizeof(int);

 uint32_t outPin = 17;            // The Broadcom pin number the signal will
be sent on
 int frequency = 38000;          // The frequency of the IR signal in Hz
 double dutyCycle = 0.5;         // The duty cycle of the IR signal. 0.5
means for every cycle,
                                 // the LED will turn on for half the cycle
time, and off the other half

 int result = irSlingRaw(
   outPin,
   frequency,
   dutyCycle,
   codes,
   codesCount);

 return result;
}

int length(char *input)
{
 int len = 0;
 while(input[len] != '\0') {
   len++;
 }
 return len;
```

```
}

int powi(int base, int exp)
{
 int i, res = 1;
 for (i = 0; i < exp; ++i) {
   res *= base;
 }
 return res;
}

int *stringArrayToIntArray(char *numbers[], int count)
{
 int i, j;
 int *codes = (int*) malloc((count) * sizeof(int));

 for (i = 0; i < count; ++i) {
   int len = length(numbers[i]);
   int val = 0;
   for (j = 0; j < len; ++j) {
     val += ((int) (numbers[i][j] - 48)) * powi(10, len - (j + 1));
   }
   codes[i] = val;
   // printf("%d\n", val);
 }

 return codes;
}
```

Installazione e compilazione

Desktop

Per avviare l'applicazione desktop sul raspberry, dopo aver installato LIRC come spiegato nella sezione *Leggere IR con Raspberry pi -> Setup* bisogna:

1) installare node e npm
2) clonare la repo
   https://github.com/SteelManITA/homeautomation-backend
3) entrare nella cartella homeautomation-backend
4) lanciare il comando
   npm install

Da qui è possibile lanciare 3 comandi principali:

- npm encode
  Dato un insieme di files con i valori raw ritorna un insieme di files con i valori binari (le directory vengono specificate nel file constant.ts e di default sono values/original e values/binary)
- npm decode
  Data una stringa binaria restituisce una raw string
- **npm start**
  lancia la compilazione con gulp e fa partire un server node

Mobile

Per avviare l'applicazione mobile:

1) installare ionic
2) clonare la repo
   https://github.com/SteelManITA/homeautomation-mobile
3) entrare nella cartella homeautomation-mobile
4) lanciare il comando
   npm install
5) copiare e rinominare il file config.sample.xml in config.xml
6) copiare e rinominare il file src\environments\environments.sample.ts in src\environments\environments.ts
7) modificare la riga
   URL: 'http://127.0.0.1:8080/api/'
   del suddetto file nell'url del server startato in precedenza (mantenendo /api)

Per testare dal pc è possibile lanciare il comando
   ionic serve

Per compilare basta aggiungere la piattaforma per cui vogliamo compilare:

ionic platform add <android, ios>

E procedere con la build

ionic cordova build <android, ios>

N.B. per buildare con ios è necessario un mac con xcode, per buildare con android è necessario avere installato android studio.

WebApp

Per avviare l'applicazione web:

1) installare node e npm
2) clonare la repo https://github.com/SteelManITA/homeautomation-webapp
3) entrare nella cartella homeautomation-webapp
4) lanciare il comando
   npm install
5) andare nella cartella src/environments, copiare e rinominare il file environments.sample.ts in environments.ts
6) modificare le chiavi TARGET, PRODUCTION, DEVELOPMENT del suddetto file inserendo l'url del server startato in precedenza (mantenendo /api)

Per testare è possibile lanciare il comando

npm start

Per realizzare una build, invece basta lanciare il comando:

npm run build

che genererà una cartella dist con i file ottimizzati per la produzione