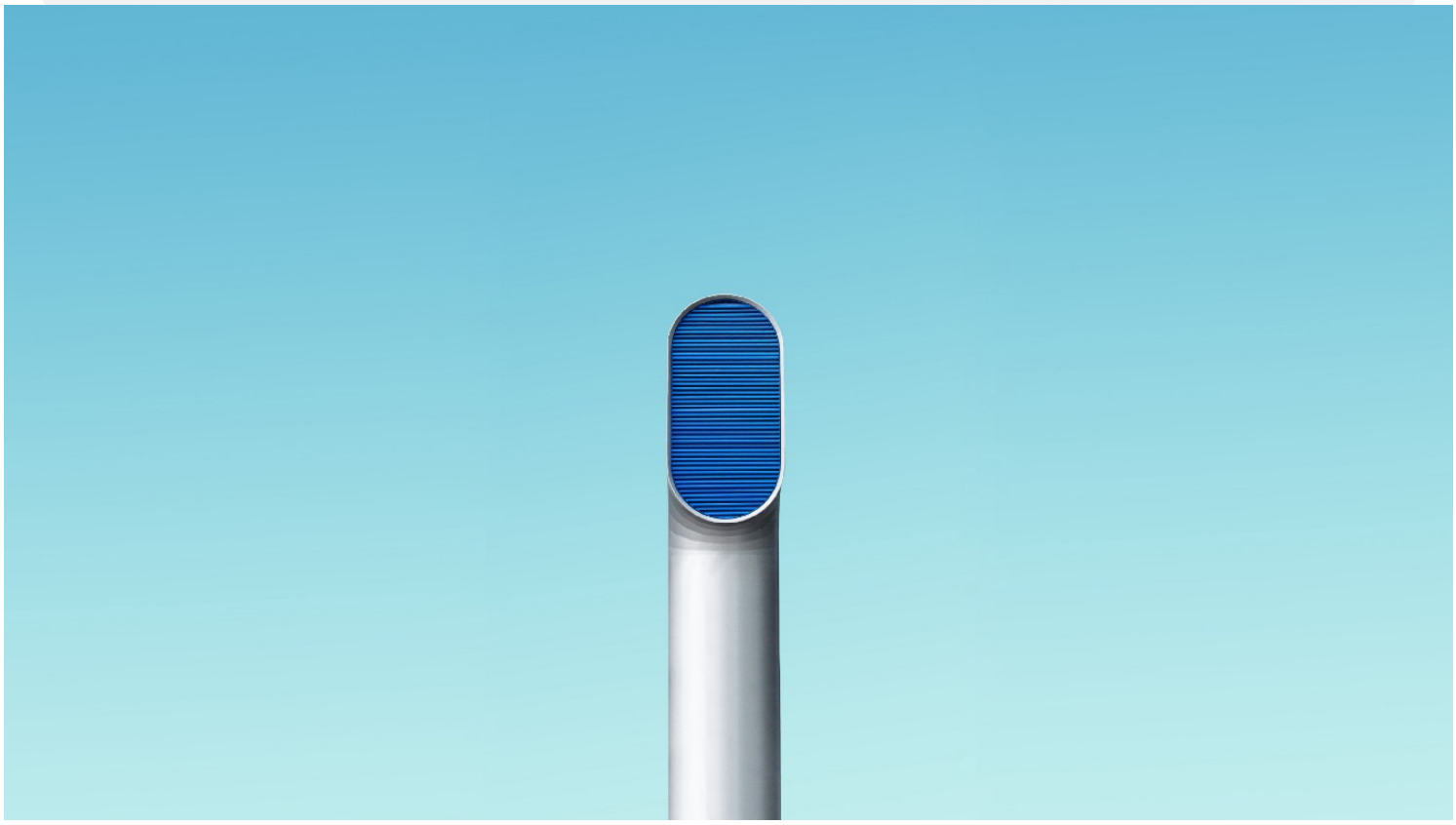


Photo by Simone Hutsch on Unsplash

Smart Air Conditioner with Raspberry Pi— An Adventure

Camillo Addis Follow
Aug 10, 2018 · 18 min read ★

This is the story of how I taught myself how to make my air conditioner smart, with a Raspberry Pi, some electronics, reverse engineering and a lot of help from the internet.

It was a warm evening in May, when my attention was caught by an advertisement on Facebook. It claimed: “Make your air conditioner smart. 99$ — for a limited time only”. It was promoting a device that can control any air conditioner remotely, via an app. But I

didn't want to spend 99$ to make my air conditioner smarter. So I've done some researches, decided to do it myself, and began this adventure.

Disclaimer: I'm not an engineer nor a programmer. I love to learn things and solve problems — and I'm pretty stubborn, I'm not easy to give up. What I'm describing is how I did what I did, and probably many things aren't done in the best possible way… so any tip or advice is welcome!

Chapter 1. Naive Rashness

I have to admit it: at the beginning I thought it would have been easy.
The *fault* is of this article by sanchezjjose on Instructables (actually a very useful one), which explains how to build an infrared receiver and transmitter to control the air conditioner. I read it, I bought all the electric components and followed his instructions, thinking everything would have been fun and simple. It wasn't.

First issue: distance matter

In my living room the air conditioner unit (AC, now on) is very distant from where the RasPi is located (like almost 7 meters away). It's not that much, but with a single IR led it was very difficult for the signal to reach the AC. So, if you're trying to set up a similar transmitter, when you're wiring the cables and checking that everything works correctly put yourself in ideal conditions: close to the unit and with almost no ambient light. It may seem obvious, but for me it wasn't. Later you can increase the performance of the IR led as suggested here: for me the best solution was to put two LEDs in series.

Second issue: complex IR protocol

When I first tried to record the remote signal with the `irrecord` command of the LIRC library (we'll get there later) it didn't work. I thought it was a problem of distance from the receiver, ambient light or an issue with the receiver, but after many trials and error it was clear that the problem was somewhere else.

After some further research I found out the reason: most of the modern air conditioner are controlled via a remote with a display, which shows all the current settings of the AC; but this remote is different from a TV one, which send a different signal for every button with only the information of that button (eg. "Power On", "Volume Up"…): The AC remote sends *every information each time*: status, mode, fan speed… This happens in

order to *sync* the remote (where you *see* the changes to the settings) and the AC unit (where the changes has to happen). Moreover, almost every model/series of AC, even of the same brand, have a little differences in the protocol, which makes it hard to find a ready-made config file for your remote.

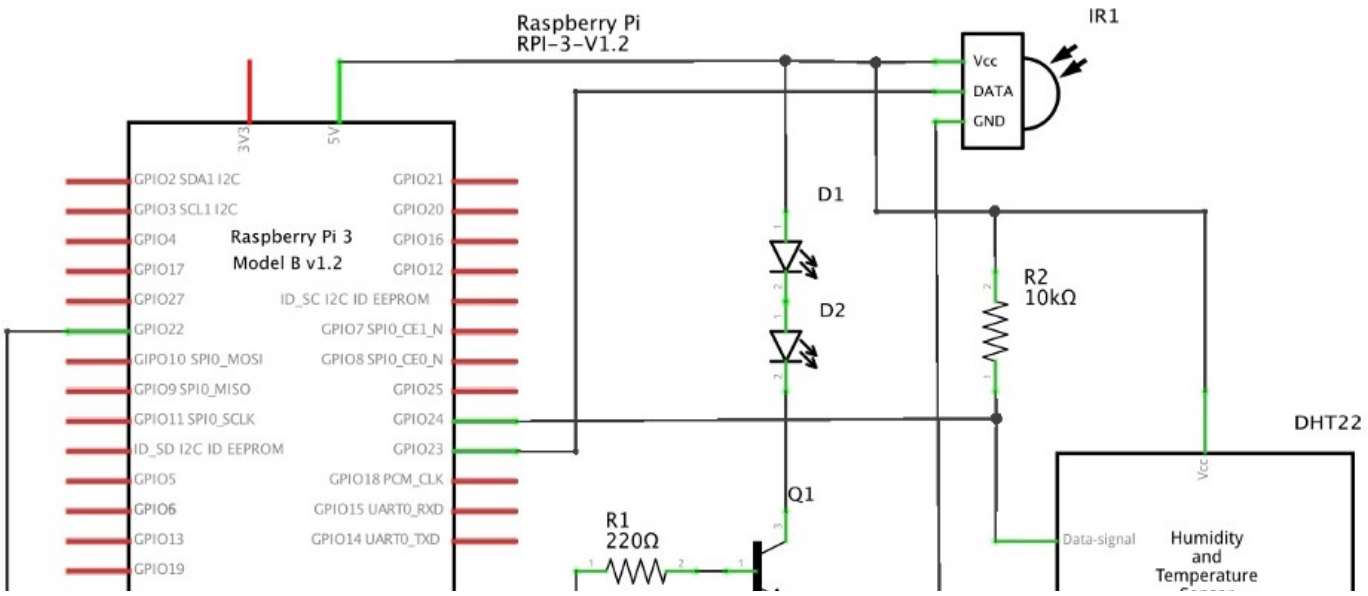
When I finally pointed out these two issues, the path to follow was clear: record the signal from the remote, decode it, understand it, recreate it and send it to the unit.

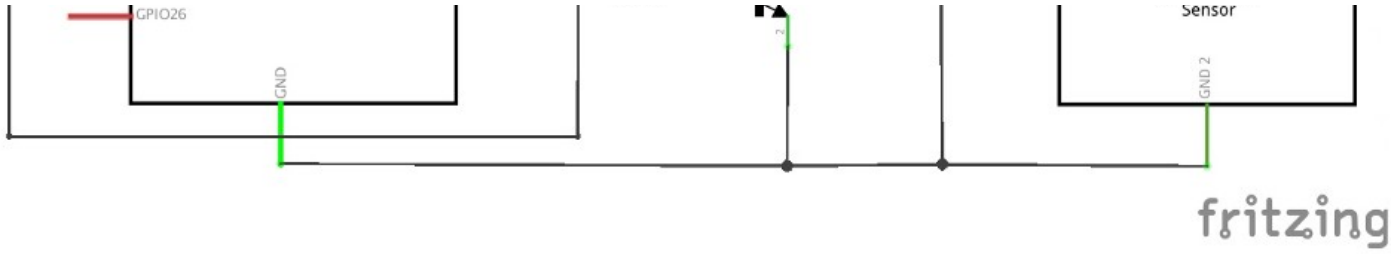
Chapter 2. Wire it up

First of all I needed to set up the hardware. I already owned a Raspberry Pi 3 model B, so I only had to buy some electronic parts:

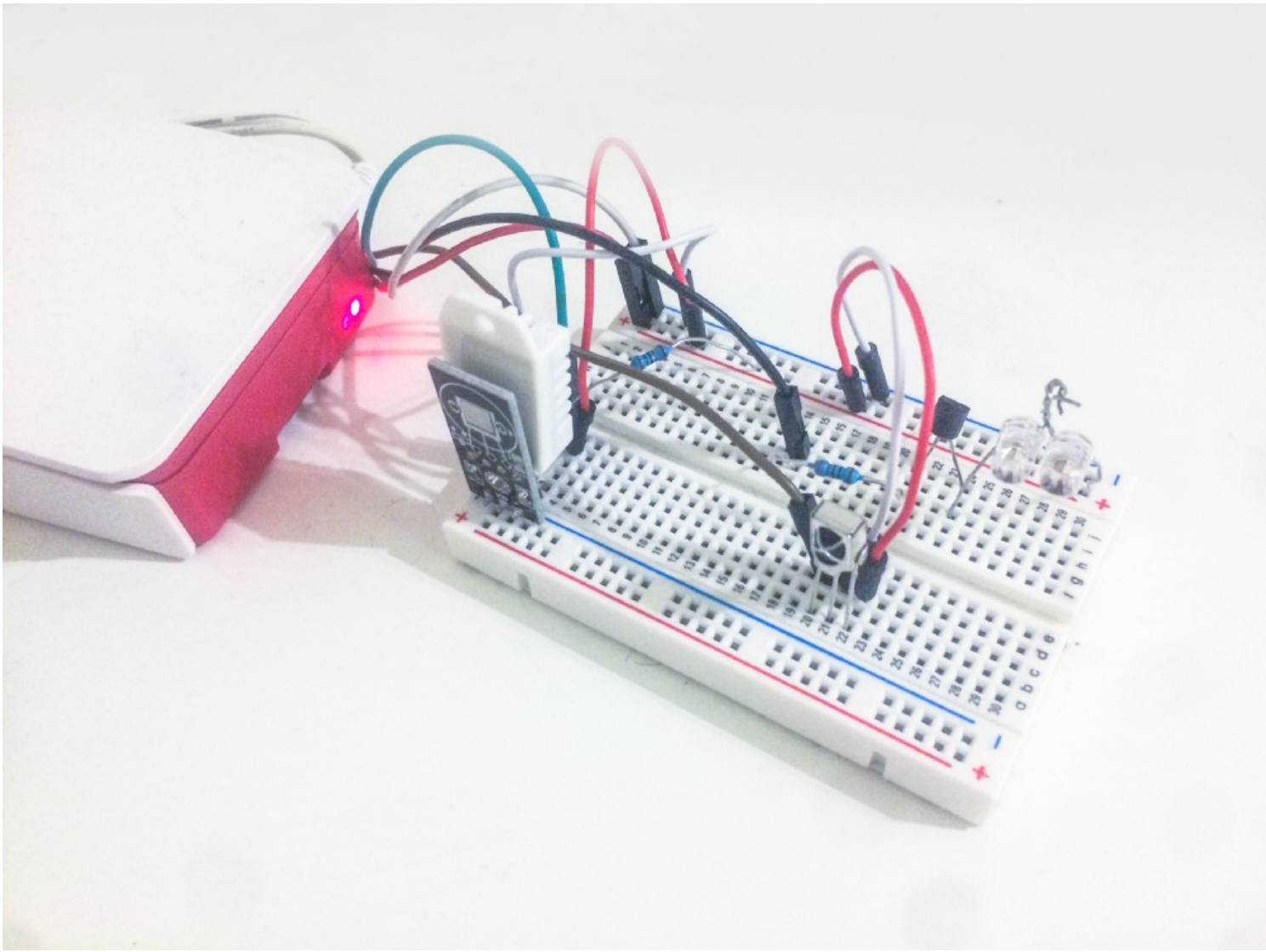
- IR Led receiver and transmitter
- NPN Transistor
- Resistors
- A breadboard
- Jumper wires
- Temperature and humidity sensor

Then I wired everything up like this:





Raspberry Pi IR receiver and transmitter schematic with temperature and humidity sensor.



The final setup.

Next I had to check if everything was working correctly.

To do this you have to log in to Raspberry Pi (if you need to set it up follow this guide or this one) and install LIRC:

```
sudo apt-get install lirc
```

Then edit `/etc/modules` file and add:

```
lirc_dev
lirc_rpi gpio_in_pin=23 gpio_out_pin=22
```

Next, change (or create) `/etc/lirc/hardware.conf` and add:

```
#############################################################
# /etc/lirc/hardware.conf
#
# Arguments which will be used when launching lircd
LIRCD_ARGS="--uinput"

# Don't start lircmd even if there seems to be a good config file
# START_LIRCMD=false

# Don't start irexec, even if a good config file seems to exist.
# START_IREXEC=false

# Try to load appropriate kernel modules
LOAD_MODULES=true

# Run "lircd --driver=help" for a list of supported drivers.
DRIVER="default"
# usually /dev/lirc0 is the correct setting for systems using udev
DEVICE="/dev/lirc0"
MODULES="lirc_rpi"

# Default configuration files for your hardware if any
LIRCD_CONF=""
LIRCMD_CONF=""
#############################################################
```

Then edit `/boot/config.txt` and add:

```
dtoverlay=lirc-rpi,gpio_in_pin=23,gpio_out_pin=22
```

Now you should restart LIRC service and then reboot your Raspberry Pi:

```
sudo /etc/init.d/lircd stop
sudo /etc/init.d/lircd start
```

NB. All the guides I found tells to restart the service using `sudo /etc/init.d/lirc stop …` *but for me works only with* `lircd` *at the end.*

After rebooting I was able to to receive some data from the AC remote, running these two commands

```
sudo /etc/init.d/lircd stop
mode2 -d /dev/lirc0
```

and pointing the remote to the receiver and pressing some buttons. If it works, there should be an output like this:

```
space 16777215
pulse 431
space 434
pulse 432
space 430
pulse 437
space 429
pulse 436
space 430
...
```

If it doesn't work, double check the connections and try again. If you get an error like `Partial read 8 bytes` you should run this command, as pointed out here:

```
sudo /usr/share/lirc/lirc-old2new
```

When you see the output of spaces and pulses, it means that the receiver is working correctly!

Then, to check the transmitter, there are at least three way of doing it:

- decode a TV remote (or any other *simple* remote), and try to send a signal the TV (follow this guide from 6:57);
- create a LIRC config *raw* file with `mode2 -m -d /dev/lirc0 > test.conf` (follow this guide, the software part) and send the signal to the AC;
- same as the last one, but use the `decode.py` script explained later while running the `irsend` command in another console window.

After checking that everything is working, it was time to hack the infrared signal protocol of the air conditioner.

Chapter 3. Going for it

My conditioner is a Daikin M Series, and the remote model is ARC466A33, so everything I'm writing now on is related to this particular model. But, in order to make this guide as universal as possibile, I'm going to describe how I managed to decode, understand and recreate the signal.

How the signal looks like

If you want to better understand how an IR signal looks like, you should use an oscilloscope and actually *see* the signal; I don't own one, but I luckily found this article by McMajan who was trying to achieve the same goal (decoding a Daikin remote signal), with a better knowledge than me — and an oscilloscope. *[As an alternative here is a more in depth explanation of how an ir signal works]*

Looking at his graphs it's clear that the signal is a sequence of highs and lows divided in 4 trunks. This is also visible in the LIRC output, which is a sequence of pulses and spaces of almost the same short length (~430ms), with other spaces of medium length (~1320ms, ~1750ms, ~3500ms) and other few very long spaces (>~25000ms).

When dividing a full signal sequence using the long space as separator, we found that it is divided like this (the first very long space should be ignored, as it's the time LIRC waited before receiving any signal):

- a first small sequence ("A") of short pulses and spaces, than a space of ~25500ms;

- a second (“B”) and a third (“C”) longer sequence of short pulses and short/medium spaces and a space of ~35500ms;
- a last even longer sequence (“D”) of short pulses and short/medium spaces.

So, in our case the signal is made of four trunks, each carrying some informations; at the end it will turn out that sequence A is the opening one, B and C are very similar with some infos in it and D is where most of the settings are “stored”.

Decoding the signal

I was beginning to understand the structure of the signal, and it was even clearer when I’ve converted it into bits. Having in mind that every couple of pulse and space should be read *together*, going through our raw data input let us discover that there are many repeating couples, followed by a last single `430ms` pulse. Every couple have it’s own meaning, which is:

```
1. pulse 430  space 430   = bit '0'
2. pulse 430  space 1320  = bit '1'
3. pulse 430  space 25000 = short separator (A)
4. pulse 3440 space 1720  = leading bit (B)
5. pulse 430  space 35500 = long separator (C)
```

By replacing each pair of pulse+space with its corresponding bit, our signal looked like this:

```
00000
AB
10001000010110111110010000000000101000110000110000000000011100000
CB
10001000010110111110010000000000010000100100110101010000000001000
CB
10001000010110111110010000000000000000010010000010011000000000001111
01010000000000000000011000000000011000000000000000010000011000000001
0000000011000101
```

And at this point it was clear that our signal is made of an opening sequence, a short separator and three trunks divided by a long separator; the first and the second trunk

are similar and each made of 8 bytes, while the third one, which is the longest part, is made of 19 bytes. It was time to try to understand the meaning of every bit, in order to try to recreate the signal.

To simplify the process of translating the raw signal into a binary sequence, I started from a python script made by Brian Schwind, and modified it a little bit:

```
import RPi.GPIO as GPIO
import math
import os
from datetime import datetime
from time import sleep

# This is for revision 1 of the Raspberry Pi, Model B
# This pin is also referred to as GPIO23
INPUT_WIRE = 16

SHORT = 600
MEDIUM = 1500
LONG = 24000
LONGEST = 30000

GPIO.setmode(GPIO.BOARD)
GPIO.setup(INPUT_WIRE, GPIO.IN)

while True:
 value = 1
 # Loop until we read a 0
 while value:
  value = GPIO.input(INPUT_WIRE)

# Grab the start time of the command
 startTime = datetime.now()

# Used to buffer the command pulses
 command = []

# The end of the "command" happens when we read more than
 # a certain number of 1s (1 is off for my IR receiver)
 numOnes = 0

# Used to keep track of transitions from 1 to 0
 previousVal = 0

while True:

if value != previousVal:
   # The value has changed, so calculate the length of this run
```

```
    now = datetime.now()
    pulseLength = now - startTime
    startTime = now

 command.append((previousVal, pulseLength.microseconds))

 if value:
    numOnes = numOnes + 1
   else:
    numOnes = 0

 # 10000 is arbitrary, adjust as necessary
   if numOnes > 10000:
    break

 previousVal = value
   value = GPIO.input(INPUT_WIRE)

  print "----------Start----------"
  for (i, val) in enumerate(command):
   val_next = command[i+1]
   if val[0] == 0:
    if (val[1] < SHORT and val_next[1] < SHORT):
     print 0
    elif (val[1] < SHORT and val_next[1] > SHORT and val_next[1] <
MEDIUM):
     print 1
    elif (val[1] < SHORT and val_next[1] > LONG and val_next[1] <
LONGEST):
     print "A"
    elif (val[1] < SHORT and val_next[1] > LONGEST):
     print "B"
    elif (val[1] > MEDIUM and val_next[1] > MEDIUM):
     print "C"
  print "-----------End-----------\n"

 print "Size of array is " + str(len(command))
```

When you run this code via `python decode.py` and press a button on the remote pointing to your IR receiver on the breadboard, you'll see a list of ones, zeros and letters, which is our signal decoded!

Understanding the signal

To reverse engineer the signal it was necessary to check the difference when changing the settings of the AC with the remote; a simple way to compare all the strings recorded is using a spreadsheet — I cloned one created by McMajan, where he already decoded

most of the signal — where every column of the spreadsheet stands for a bit, and every row is a signal with different settings recorded and translated into ones and zeroes.

Take a look at the spreadsheet to have an idea of how it works. For every line I made a variation to a single setting (power, mode, fan speed, swing…) and looked for a change in the bits. Then a few lines to understand the binary:

- line `13` is a marker line to divide the bytes and the separators;
- `14` is to number every bit;
- `15` is a binary sum of every bite;
- `16` is the hex value of the byte;
- `17` indicates which bits stands for which setting.

NB. The bits are numbered following the LSB convention, so every byte should be read in reverse.

Looking at the spreadsheet you can see the structure mentioned before (opening sequence and three trunks divided by a short or long separator), and a common pattern among the trunks: every trunk starts with four bytes which are always `0x11 0xDA 0x27 0x00` and ends with a checksum (a modular sum) of the previous bytes in the trunk.

But most importantly we can track which bit corresponds to which setting! Here are a sum of all the settings and the related bit(s):

```
bit #   =>  meaning
59      =>  Comfort mode on/off
113–128 =>  Datetime
179     =>  Power on/off
183–186 =>  Mode (auto, dry, cool, heat, fan)
187–194 =>  Temperature °C (multiplied by 2)
203–206 =>  Vertical swing on/off
207–210 =>  Fan speed (1–5, auto, night)
211–214 =>  Horizontal swing on/off
219–230 =>  "On timer" minutes
231–242 =>  "Off timer" minutes
243     =>  Powerful mode on/off
248     =>  Silent mode on/off
268     =>  Sensor mode on/off
```

```
269     =>  Eco mode on/off
271     =>  Clean mode on/off
```

That's it! I did it! I recorded the signal, decoded and understood it, and it didn't have no more secrets to me. I admit: I was pretty happy. The last step to make my air conditioner smart was to recreate the signal.

How I felt when I finally decoded the signal.

Chapter 4. IoT

The hardest part of this journey was over. Now that the signal was finally decoded and understood, it was time to replicate it and send it via the Raspberry Pi. To do this, I needed to write a RESTful API server and a web app with a UI to manage the air conditioner. I decided to write the server in Node.js and the web app in Vue.js.

Here's the logic of how everything works:

- the server has an endpoint for each setting (eg. `/power`, `/temp`, `/swing`) which accepts both GET (returning the current status of each setting) and POST (to set the status);

- when the server receives a POST call, it evaluates the call, set the setting and send the signal to the AC unit.

The trickiest part was sending the signal, because LIRC works only by reading a config file and running a shell command; I tried to use a different library (pigpio, as suggested by Brian), but I couldn't manage to make it work, so the only way I found was made of these steps: set the settings, convert them to a binary string, then convert the string to a LIRC config file, write the file to a folder and run the `irsend` command to actually send the signal.

Disclaimer: the code is redundant; many things are repeated because when I wrote it I had to check every step and see if the output was what I expected, and I haven't had time to clean it up yet. Moreover, I'm not going to write how to set up Node.js or how to configure and run an express server, because otherwise this article would be endless and I'm not really interested in teaching Node or Vue: I'd only like to share the logic of how I managed to recreate the signal — the code is needed only as help in this. I'm sharing all the code on Github, so if you're interested in it check it there.

Settings

First of all I declared all the settings with some defaults values, divided into `props` and `settings`

```
const props = {
  modes : {auto:'0000', dry:'0100', cool:'1100', heat:'0010',
fan:'0110'},
  minMaxTempAuto : ['18', '30'],
  minMaxTempCool : ['18', '32'],
  minMaxTempHeat : ['10', '30'],
  fanSpeeds : {fan1:'1100', fan2:'0010', fan3:'1010', fan4:'0110',
fan5:'1110', auto:'0101', night:'1101'}
};

var settings = {
  power: 0,
  temp: 25,
  mode: 'auto',
  verticalSwing: 0,
  horizontalSwing: 0,
  fanSpeed: 'auto',
  powerfulMode: 0,
  econoMode: 0,
```

```
  quietMode: 0,
  cleanMode: 0,
  comfortMode: 0,
  sensorMode: 0,
  onTimerPower: 0,
  onTimerMinutes: 0,
  offTimerPower: 0,
  offTimerMinutes: 0
};
```

After this I set the variables related to my Daikin, such as the pulse and spaces length in microseconds:

```
const pulse = 430;
const space_0 = 430;
const space_1 = 1320;

const DAIKIN_0 = pulse +' '+space_0;
const DAIKIN_1 = pulse +' '+space_1;
const DAIKIN_A = pulse +' 25000';
const DAIKIN_B = '3440 1720';
const DAIKIN_C = pulse +' 35500';

const header = '00000';
const intro = '11';
```

Then I wrote two functions to convert binary to hex and viceversa; note that in the `bin2hex` function the binary string is reversed before being converted to a hex number, because, as we saw, the bits are numbered following the LSB convention — and so it's easier to write them in reverse and then invert them.

```
var exports = module.exports = {}; // I'm using module.exports to
store common functions in a file separated from the server itself.

exports.bin2hex = function(bin){
  if(bin.length !== 8){
    return false;
  }
  bin = bin.split('').reverse().join('');
  return ('00' + parseInt(bin ,
2).toString(16)).substr(-2).toUpperCase();
}
```

```
exports.hex2bin = function(hex){
    return ("00000000" + (parseInt(hex,
16)).toString(2)).substr(-8);
}
```

Recreate the signal

As seen, most of the settings are set in the last trunk of the signal, and since I didn't need to write the correct datetime in the code, I decided to focus only on that trunk; the only exception is *confort mode*, which is set in the first trunk, but that could easily set with a simple `if/else` statement.

The next step was to get all the settings and convert them to a binary string. The third trunk of the signal is made of 15 bytes: bytes 1 to 5 are constant, 6 to 14 are based on the settings, the last one is a checksum.

```
exports.getBytes = function(settings){

// declare the bytes array
var createdBytes = [];

// set all the variable bytes
createdBytes[0] = exports.bin2hex('' + settings.power +
settings.onTimerPower + settings.offTimerPower + '1' +
settings.mode);
  createdBytes[1] = (settings.temp*2).toString(16);
  createdBytes[2] = '00';
  createdBytes[3] = exports.bin2hex('' + settings.verticalSwing +
settings.verticalSwing + settings.verticalSwing +
settings.verticalSwing + settings.fanSpeed);
  createdBytes[4] = exports.bin2hex('' + settings.horizontalSwing +
settings.horizontalSwing + settings.horizontalSwing +
settings.horizontalSwing + '0000');
  createdBytes[5] = '00';
  createdBytes[6] = '06';
  createdBytes[7] = '60';
  createdBytes[8] = exports.bin2hex(settings.powerfulMode + '0000' +
settings.quietMode + '00');
  createdBytes[9] = '00';
  createdBytes[10] = 'C1';
  createdBytes[11] = exports.bin2hex('0' + settings.sensorMode +
settings.econoMode + '0' + settings.cleanMode + '001');
  createdBytes[12] = '00';

// add the constant bytes at the beginning of the array
createdBytes.unshift('11','DA','27','00','00');
```

```
  var s = 0;

// calculate the checksum and add it as the last byte
createdBytes.forEach(function(hexByte){
    s += parseInt(hexByte, 16);
  })
createdBytes.push(s.toString(16).substr(-2));
 return createdBytes;
}
```

Then I needed two other functions to get the binary string: the first one convert a single hex string (“trunk”) in binary code, the second one takes an array of *trunks* as input and returns the full “almost binary” string (I needed to leave the “a”,“b” and “c” letters to make it easier to translate the string into pulses and spaces).

```
exports.getBinary = function(trunk){
  return trunk.match(/.{1,2}/g).map(str => {
    return exports.hex2bin(str).split('').reverse().join('')
  }).join('');
}

exports.getBinaryString = function(binaryTrunks){
  return
''+header+'ab'+binaryTrunks[0]+'cb'+binaryTrunks[1]+'cb'+binaryTrunk
s[2];
}
```

Finally, the last function used to create the signal was meant to return a string which could be written to a valid LIRC config file:

```
exports.getRemote = function(binaryString){
  var result = binaryString.split('').map(val => {
    let v = '';
    if(val === '0'){
      v = DAIKIN_0 + '\n'
    }else if(val === '1'){
      v = DAIKIN_1 + '\n'
    }else if(val === 'a'){
      v = DAIKIN_A + '\n'
    }else if(val === 'b'){
      v = DAIKIN_B + '\n'
    }else if(val === 'c'){
      v = DAIKIN_C + '\n'
    }
```

```
    return v;
  } ).join('');

// add a pulse at the end.
result += ''+ pulse;

return 'begin remote\n\n  name  daikin\n  flags RAW_CODES\n  eps
30\n  aeps          100\n\n  gap          34978\n\n      begin
raw_codes\n\n          name command\n\n'+result+'\n\n     end
raw_codes\n\nend remote\n';
}
```

Send the signal

I've said that the only way I could use to send the signal to the AC unit was to store a config file and call it using a LIRC command. To do this using Node, we should use two modules: `fs` and `node-cmd`.

```
exports.sendSignal = function(remote){

  // require the modules
  var cmd=require('node-cmd');
  var fs = require('fs');

  // write the file to the current directory
  fs.writeFile('./daikin.lircd.conf', remote, function(err) {
    if(err) {
      return console.log(err);
    }
  console.log("File created.");

  // stop LIRC
  cmd.run('sudo /etc/init.d/lircd stop');

  // copy the file in the LIRC folder
  cmd.run('sudo cp ./daikin.lircd.conf /etc/lirc/lircd.conf.d/');
  console.log("File copyed.");

  // restart LIRC
  cmd.run('sudo /etc/init.d/lircd start');

  // wait half a second to let the service start and run the
`irsend` command
  setTimeout(function(){
      console.log("Command sent");
      cmd.run('irsend SEND_ONCE daikin command');
    }, 500);
```

```
  });
}
```

And finally, here's the function (part of the `app.js` file, unlike everything we've written up to now) which takes the settings as input, and sends the signal to the AC unit:

```
function send(settings){
  // copy the settings to a new object
  let s = Object.assign({}, settings);

  // convert mode and fan speed to a binary string
  s.mode = props.modes[settings.mode];
  s.fanSpeed = props.fanSpeeds[settings.fanSpeed];

  // create the trunks array and add the second constant trunk
  var trunks = ['11DA270042581DC9'];

  // check if confortMode is enabled, and add the equivalent first
trunk
  if(settings.comfortMode === 1){
    trunks.unshift('11DA2700C5301017');
  }else{
    trunks.unshift('11DA2700C5300007');
  }

  // add the third trunk, based on the settings, to the trunks array
  trunks.push(functions.getBytes(s).join(''));

  // convert each trunk to binary
  var binaryTrunks = trunks.map(trunk => {
    return functions.getBinary(trunk);
  })

// get the full string, convert it to a config file and send the
signal
functions.sendSignal(functions.getRemote(functions.getBinaryString(b
inaryTrunks)));
}
```

At this point I was able to control the air conditioner by simply changing the settings and running `send(settings)`. The first time it worked I was glad. Actually very satisfied. That kind of satisfaction you feel when you finally put in the right place the last piece of a complex puzzle. But it wasn't time to stop. It was the time when finally the tool began to work and I could start creating.

So, as said, I set up a Node server, and wrote a couple of routes for every setting, like this one:

```
// QUIET
app.get('/quiet', function(req, res){
  res.json({value:settings.quietMode});
});

app.post('/quiet/:val', function(req, res){
  let val = req.params.val;
  if( val !== 'on' && val !== 'off'){
    res.status(400).send('Cannot understand command.');
  }else{
    let v = val === 'on' ? 1 : 0;
    setSettings({quietMode : v}).then((r) => {
      send(r);
      res.json(r);
    });
  }
});
```

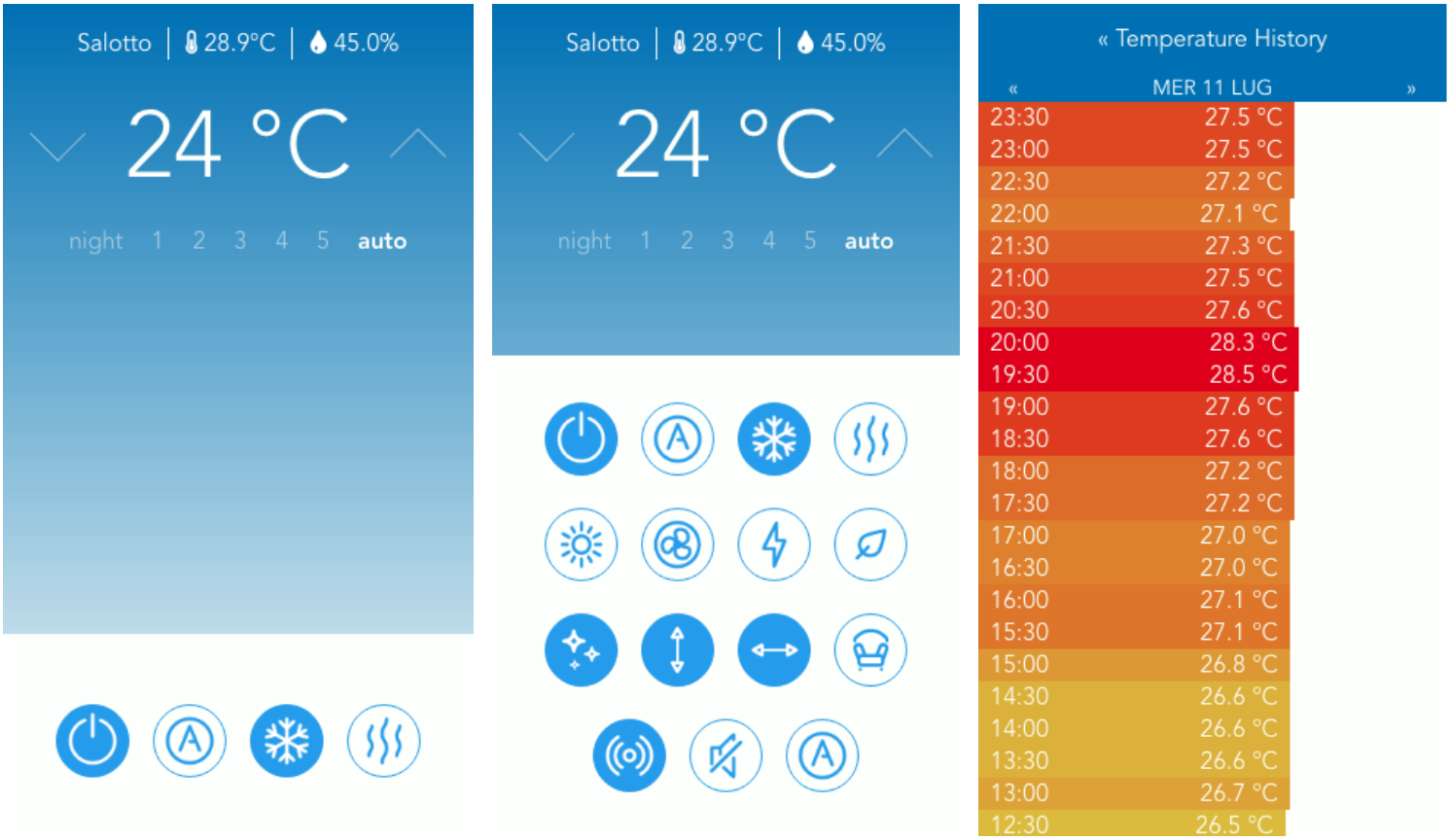
The `GET` route returns the current status of the setting (in this case of the quiet mode setting), so if it's either "on" or "off". The `POST` route, after a simple check if the command is valid, uses the async function `setSettings` which set the setting, store all the settings to a local file on the server, then send the signal and returns all the updated settings. Almost every route is the same, except for `temp`, `mode` and `fan-speed`, which takes parameters different from on/off. Also there is a read-only `all` route, which returns all the settings at once and `roomtemp`, which returns the current temperature and humidity of the room.

Web App Remote Controller: make Everything Smart

This part is only a showcase — I'm pretty proud of the result — of the web app I designed and coded. It's pretty simple, and could be improved in many ways, but right now works well and I ain't got much time to work on it.

One thing I'd like to say is that at one point I had to stop myself, because as I went on coding it I kept on thinking of features that would be great to have. I'll mention only two of them, which I actually added to the basic app: a temperature records database and basic automation.

The first one is a simple database, stored on the server, filled by a function run every half an hour which writes the current temperature and humidity. So I have a daily overview of the temperature in the room, and I can see the effectiveness of the air conditioner.



Screens from the Web App. 1 and 2 are the main screen showing current room conditions and AC settings, 3 is the room temp history.

The second one is just a little button (the last one in the bottom right corner, whose icon is copied from the “auto” mode setting on top) which activates a setting that tells the server to check every 15 minutes for the temperature (but could be any other parameter) and if it is under certain value it turn on the air conditioner at predefined settings.

Even this second improvement is really simple, but it opened to me a whole new scenario: my air conditioner was finally smart. And I could write some code and make it every day smarter — maybe learn the habits and automatically set the temperature when someone is back home, or check for the weather and notify to turn the air off and open the windows. With the potential of a computer and some code, the only limit was the time I had to spend on it. And, by the way, I think it’s enough.

. . .

References and further reading

This adventure couldn't have been possibile without all these awesome people who spent their time sharing their knowledge — thank you guys.

Zero to Air Conditioner Controller With Raspberry Pi

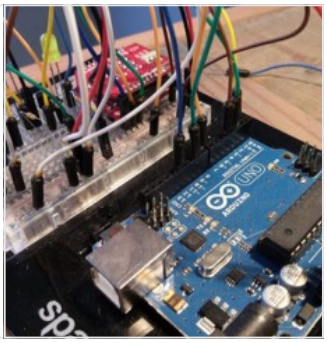
LIRC (Linux Infrared Remote Control) is a package that allows you to decode and send infra-red signals of many (but not...

www.instructables.com

alexba.in

June 8th 2013 Update: I have completed a soldered circuit prototype, complete with a full parts list and high...

alexba.in



Sending Infrared Commands From a Raspberry Pi Without LIRC

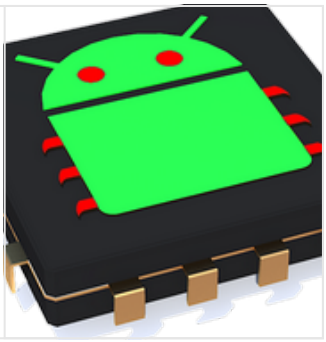
May 29th, 2016 TL;DR — I made a small C library for sending infrared packets easily on the Raspberry Pi, wrote about...

blog.bschwind.com

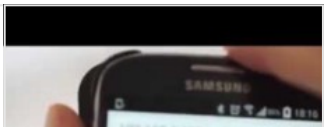
IR Daikin: decodificare protocolli infrarossi complessi. | McMajan

Non molto tempo fa ho scritto un articolo sull'utilizzo di Arduino come telecomando infrarossi e su come utilizzare un...

www.mcmajan.com

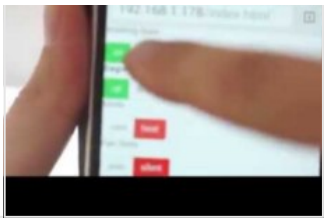


Control Air Conditioner with RaspberryPI and a LED



The explanation bellow is specific to the Daikin FTXS35K / RXS35K Professional unit but it can be modified to any other...

www.ivancreations.com



Tech Raspberry Pi Engineering Smart Home How To