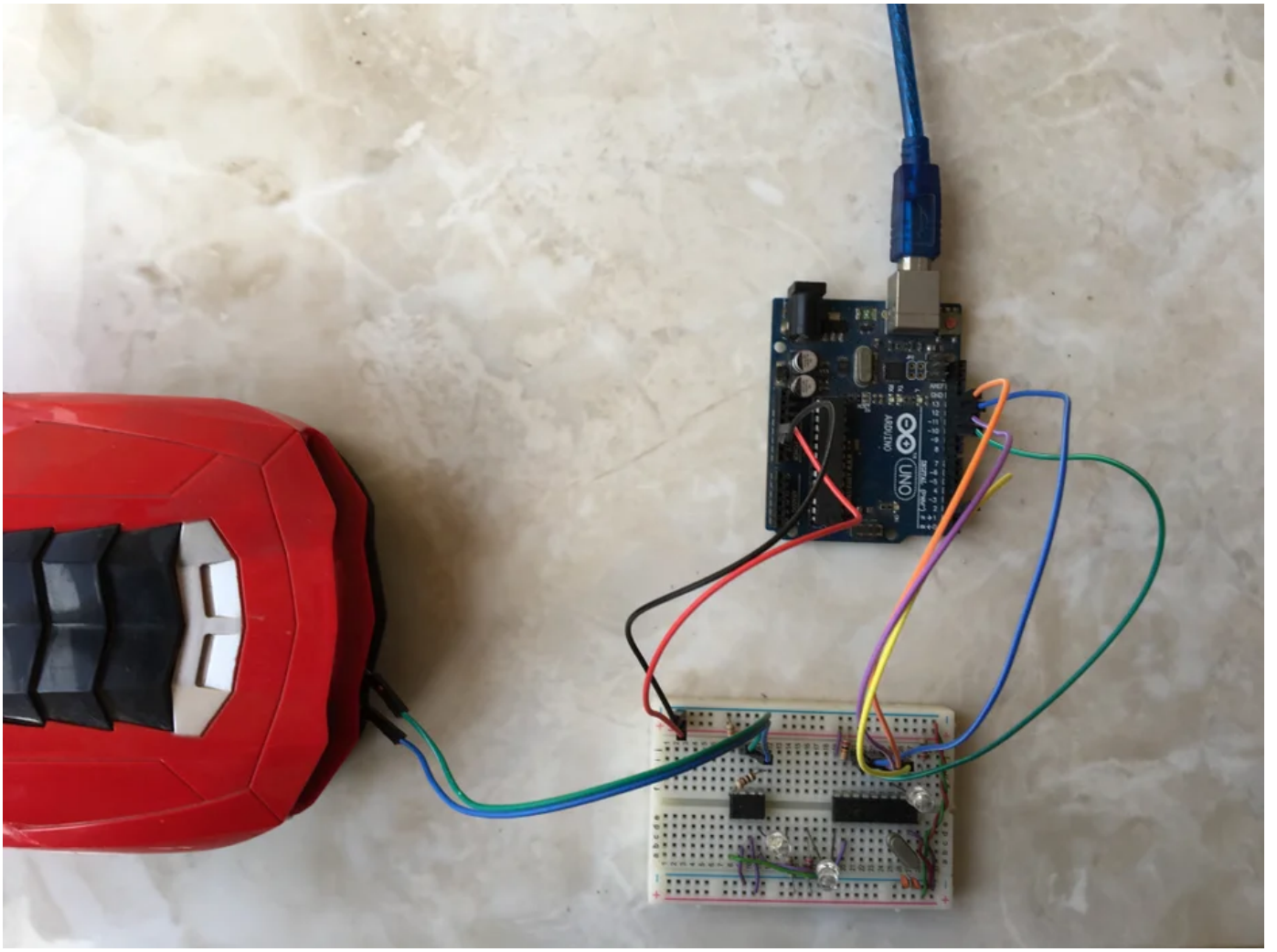
Yes We CAN BUS With Arduino in 30 Seconds!

omarCartera



Hello Arduinos!

This Instructable is trying to summarize what I ended up with after a long time of search, tutorials, trials and datasheets to build a functional **CAN BUS** node. I will try to keep it as easy and concise as possible to get you straight to a properly working setup, saving your time for further developments

afterwards.

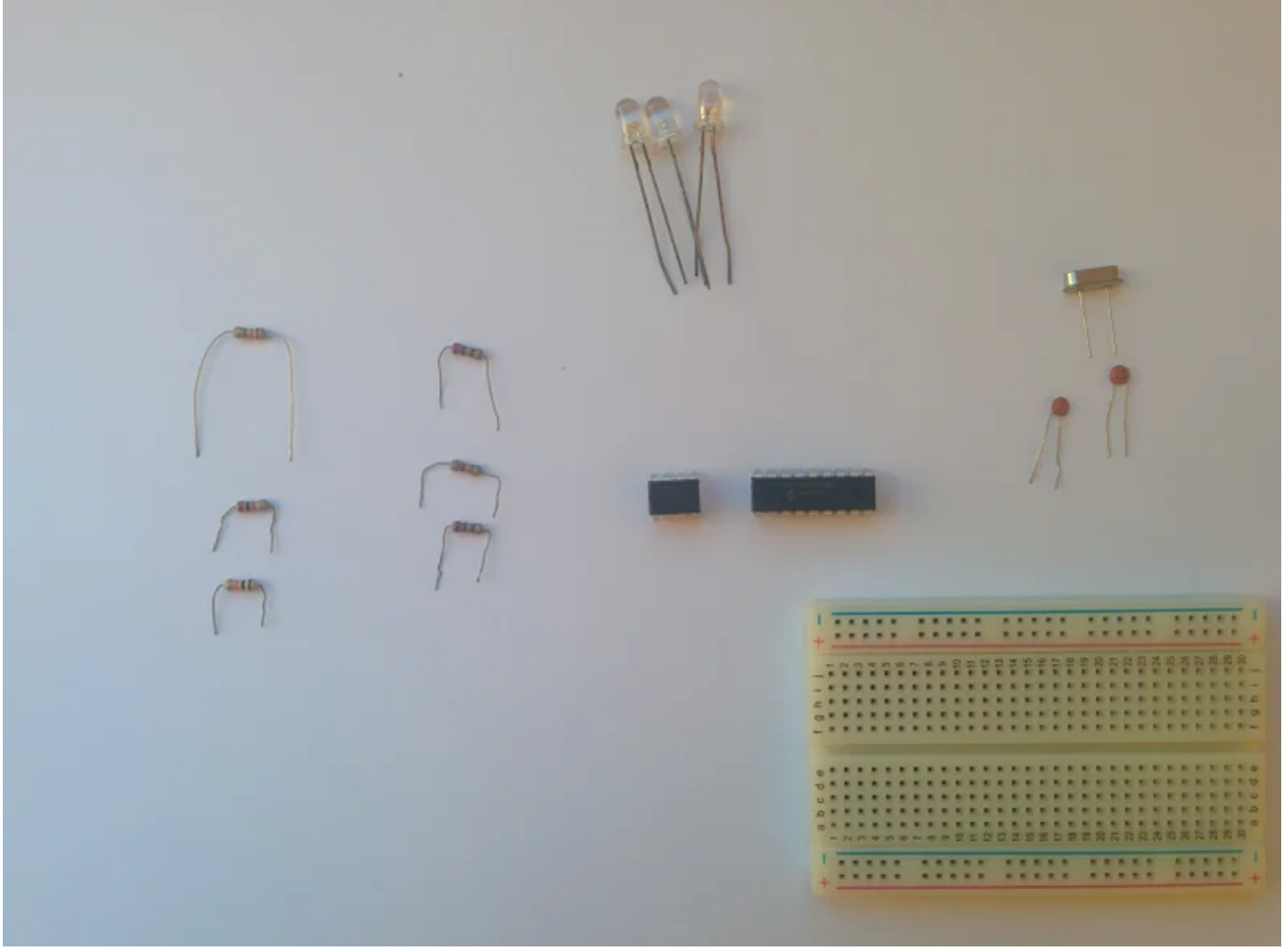
The first 3 steps are the basic ones that will initiate a **CAN BUS** at your home, the rest of the steps are a little bit advanced and real-life **CAN** situations.

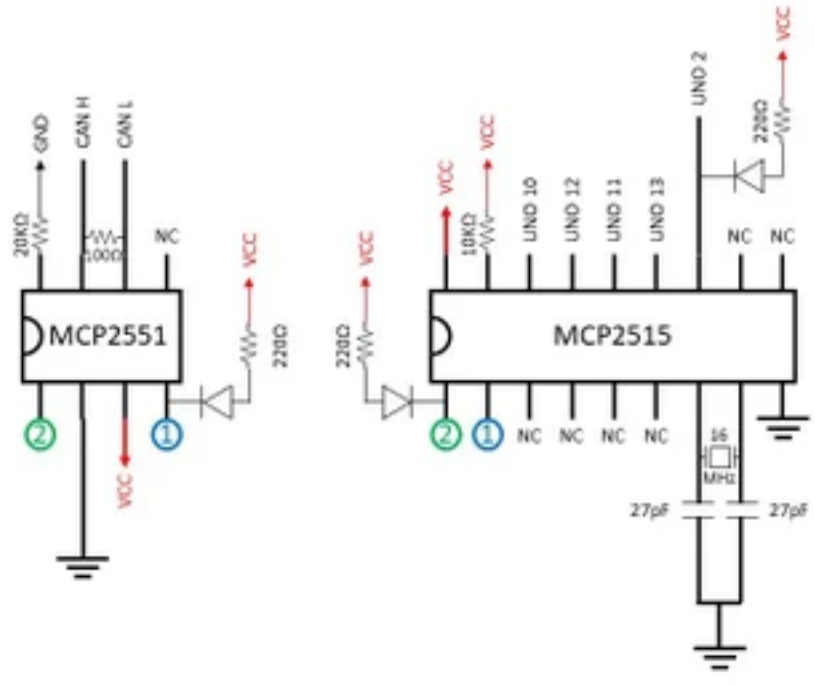
CAN BUS is a two-wire, half-duplex communication protocol that is widely used in Automotive industry. One of its greatest advantages is that it connects any number of ECUs (or microcontrollers) in your car through the two-wire bus, **CAN High** and **CAN Low**, reducing the weight of wires that could be gained by using point-to-point communication between ECUs.

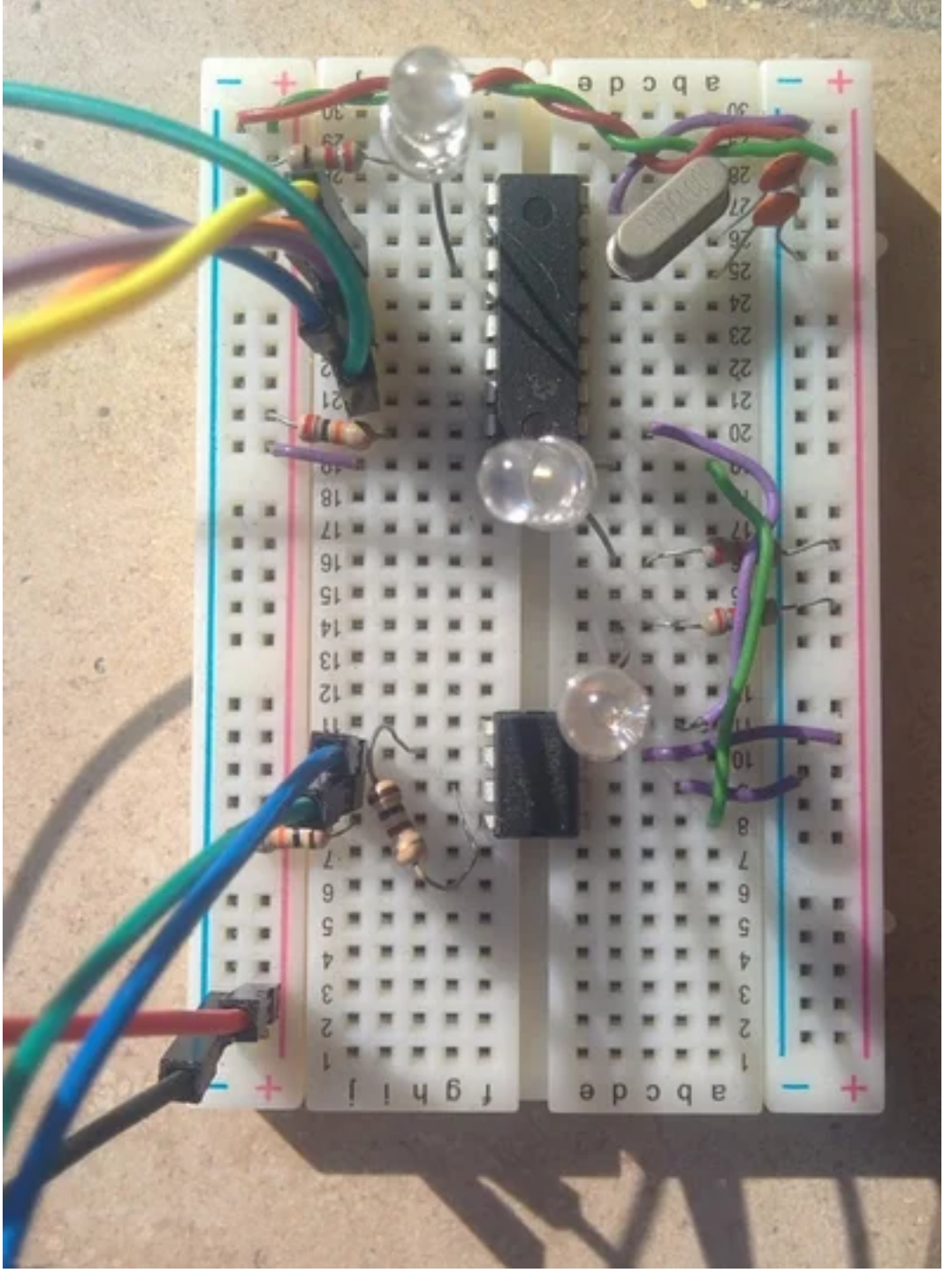
Enough talking and let's grease our hands!

You can continue reading about CAN from wiki as it really gives a very good and sufficient introduction to the topic.

Step 1: Build the Hardware




GND
CAN H
CAN L
20KΩ
1V
100Ω
NC
VCC
220Ω
MCP2551
2
1
VCC
VCC
VCC
10KΩ
UNO 10
UNO 12
UNO 11
UNO 13
UNO 2
220Ω
NC NC
220Ω
MCP2515
2
1
NC NC NC NC
16
MHz
27pF
27pF



We could buy any of the plug-and-play Arduino **CAN** shields, but building the hardware ourselves is easy, more fun and cost reduction, bro.

What you need to build one node:

- x1 Breadboard.
- x1 MCP2515 Microchip CAN Controller.

- x1 MCP2551 Microchip CAN Transceiver.
- x1 20KΩ Resistor.
- x1 10KΩ Resistor.
- x1 100Ω Resistor.
- x1 16 MHz Crystal Oscillator.
- x2 27 pF Capacitors.
- x3 LEDs.
- x3 220Ω Resistors.

Schematic key:

CAN H: CAN High and is connected to the CAN High wire of the bus.

CAN L: CAN Low and is connected to the CAN Low wire of the bus.

VCC: 5V power source from the Arduino.

GND: connected to Arduino's ground pin.

UNO X: as *X* is an integer, means connect this pin to Arduino's digital pin *X*.

Here, our setup is using 3 means of communication protocols:

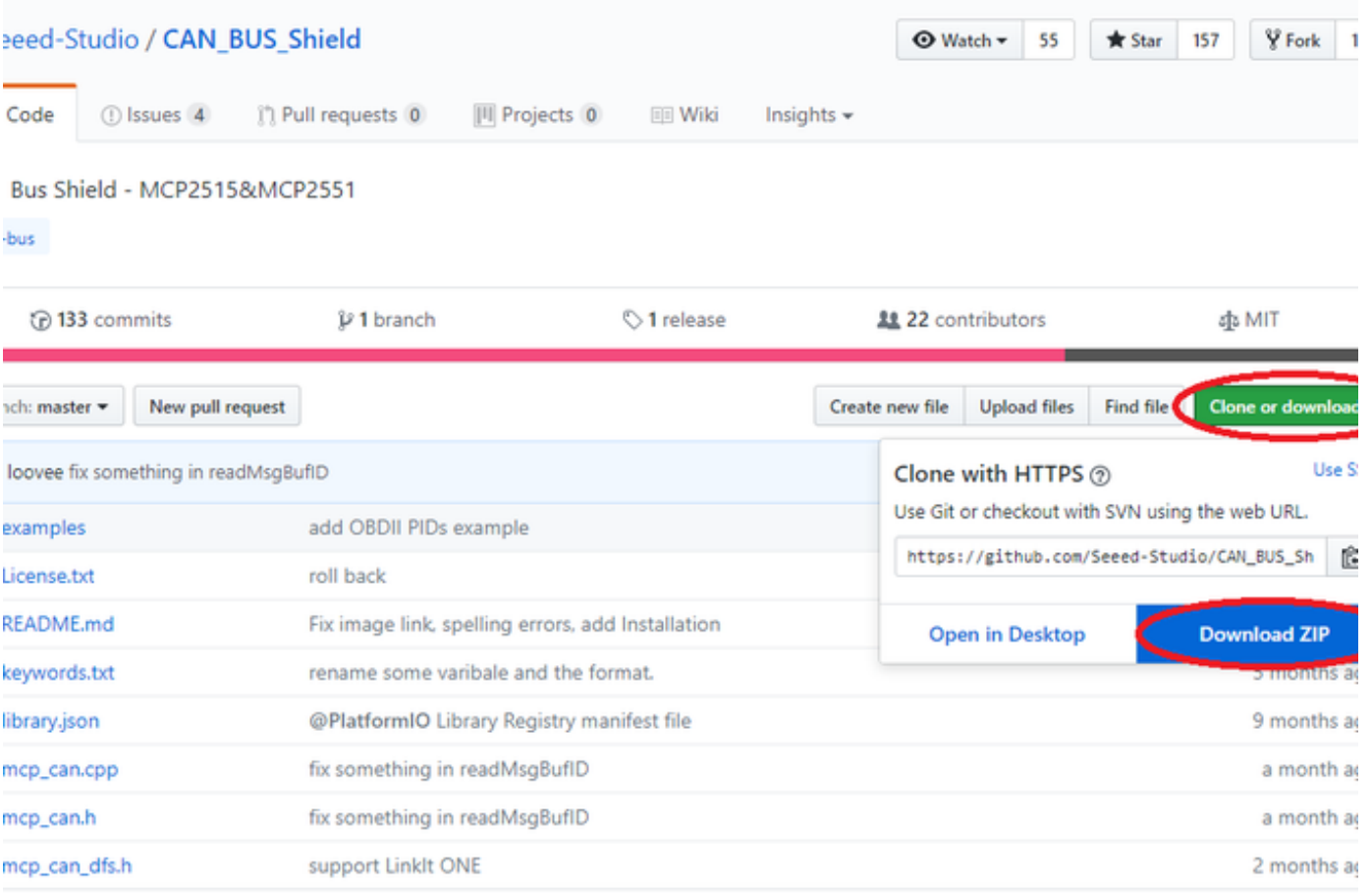
1) UART: to talk to your computer's Serial monitor.

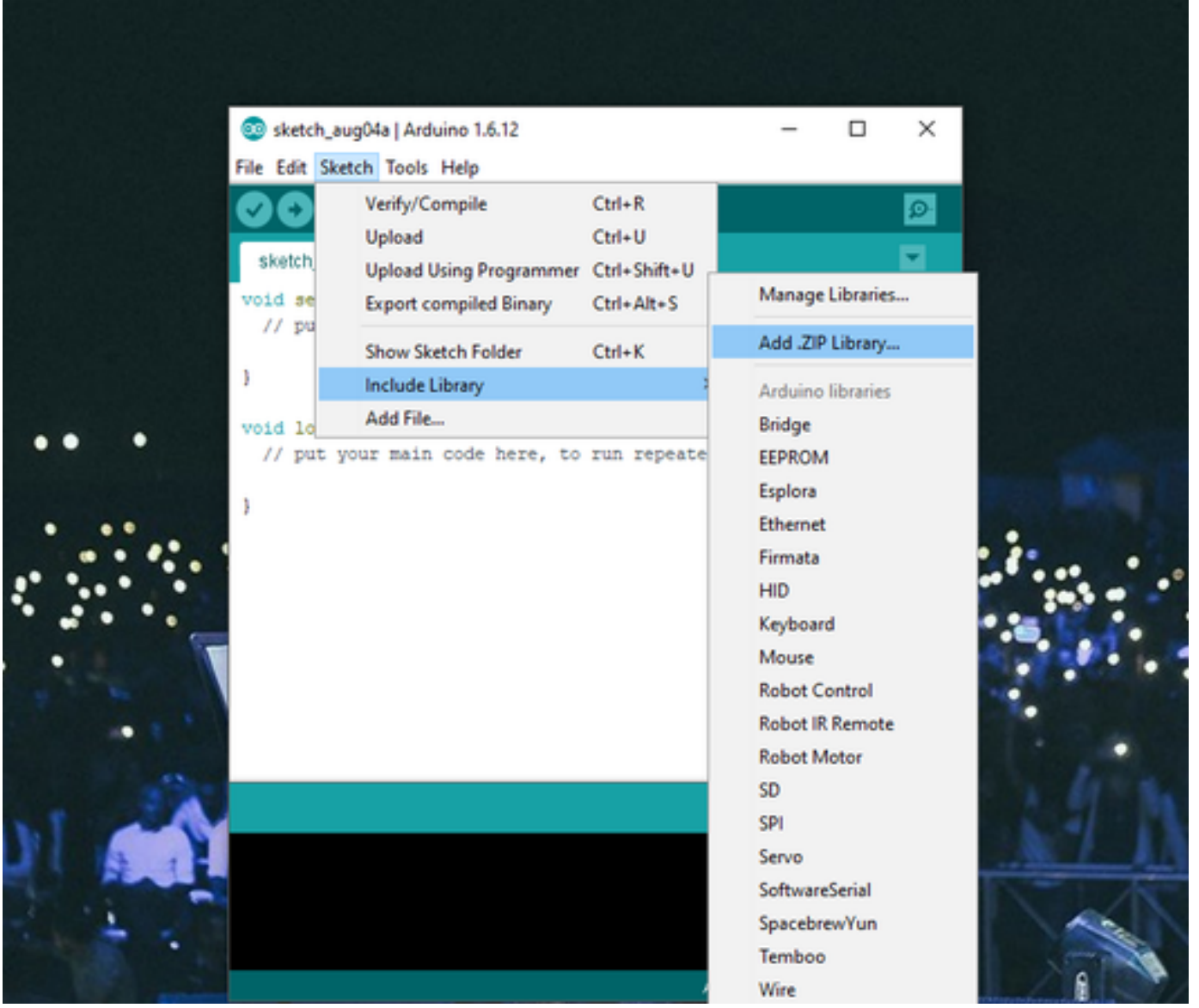
2) SPI: to talk to the **CAN** controller.

3) CAN: to talk to other neighbours in the bus.

Wire up your components to the board according to the schematics attached above and let's move to the next step.

Step 2: Download and Install the CAN Library





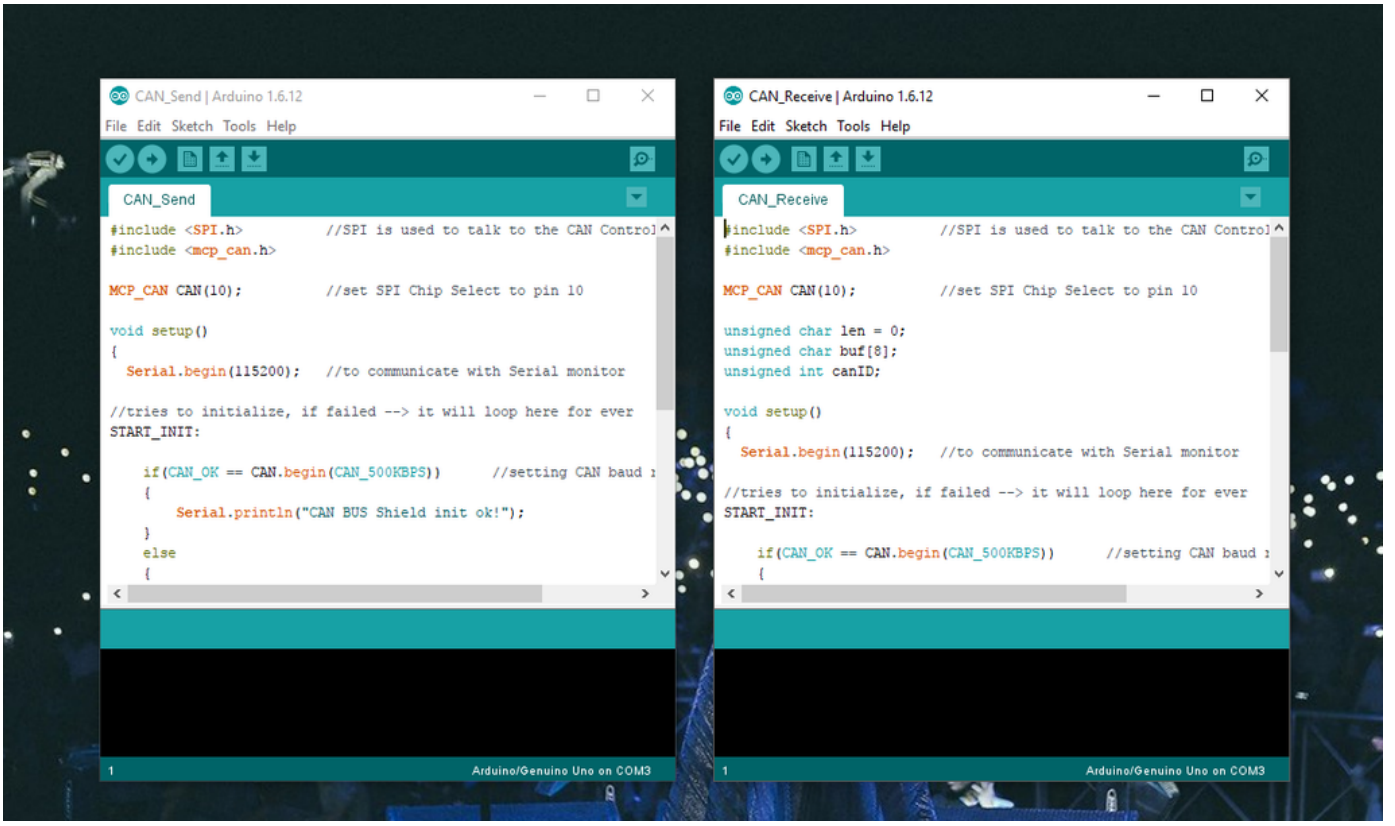
In this Instructable, I'm using Seeed Studio open-source CAN Library which you can download from their github as shown above.

Keep the downloaded file as zipped as it is (because Arduino likes this) and add the library to Arduino as shown above, as well.

By this point you are ready to move to the next step to combine both software and hardware and try your Hello CAN examples :D

I'm currently using the latest Arduino (1.8.3) but it works with the old versions as well.

Step 3: Upload Your First Code



The codes attached are basically the examples of the library itself but with a gentle touch of simplicity. I think that both files are over-commented, but feel free to ask about any bit of code you find unclear, because when you solid understand this basic example, you can follow up with the next few steps and even dig deeper and tweak the codes as you like.

The file called **Send** packs 8 characters (8 bytes) into a message of the ID 0xF1, because I'm an F1 fan, and puts it on the bus.

The file called **Receive** keeps polling the **CAN** receive buffers until any message comes in. It then breaks the incoming data into an ID, data length and the data itself.

Yes, You CAN BUS now!

Step 4: CAN Is a Message-Based Protocol!

CAN is a message-based protocol, which means that the messages and their content are more important than the sender ECU itself. So, the ECUs aren't given IDs, but each message has a unique ID in a specific bus. These IDs are responsible for setting the priority of messages in case of two or more ECUs are trying to put their messages on the bus. The LOWER decimal value ID has HIGHER priority.

Given that, the message with ID = 0x05 has more priority than our beloved message of the previous example with ID = 0xF1.

A Big Example:

If we consider ourselves in a real car, we might assume that the message that informs a fatal problem in the engine will be given the highest ever priority (Logical, no?). So, whatever ECU tries to send this message will win the bus and continue sending its message while everyone else is just

listening until it finishes.

At any time, every **CAN BUS** node sees the message being sent through the bus. But not all of them read it and send it to their ECUs. That's because in our example the rear wing ECU or the front-right headlights ECU don't care at all about a problem in the engine, so they see the message that contains engine failure and ignore it. On the other hand, they are the only ECUs who read the messages like: retract the wing or shut the headlights.

And here it comes the idea of **Message Filtering** that lets an ECU reads only the messages useful to it and ignore everything else. And since our code allows us to know the ID of the message, we can easily apply filtering.

The example file attached here adds only one *if statement* to the basic **Receive** file to read only the messages with ID = **0xF2.** Let this new code receive from the basic **Send** code and it will print nothing.

Step 5: Extract Useful Signals From a CAN Message

Message ID 0xF1

7	6	5	4	3	2	1	0	bits / bytes
1	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	1	1
1	1	1	0	1	1	1	0	2
								3
								4
								5
								6
								7

	Speed
	RPM
	Check Engine Light
	Oil Indicator
	Not used

By reaching this line now we:

1. Wired up the hardware circuit.
2. Sent and received CAN messages.
3. Filtered the stream of messages and read only those that interest us.

But all we were exchanging was a message containing useless 8 characters forming my nickname, ahem.

What if we wanted to exchange informative signals and simulate what might be happening in a real **CAN BUS** from

the dashboard ECU's point of view!

Because it would be waste of time, and bits actually, if you use the 8 data bytes to represent only one piece of information, cars might pack many signals in one message, *as shown in the picture above*, telling you that for the message of the ID = 0xF1:

- The first whole byte represents the speed of the car with a maximum value of 255 mph.
- The next 14 bits represent the engine revs up to 16383 RPM. Just to be able to work with an F1 car.
- The next bit tells the dashboard to turn On/Off the check engine indicator.
- The next bit tells whether the oil level is under a threshold or not to turn ON/OFF its indicator.
- The rest of the bits are not used in our made-up message.

Check the attached files for the code, and come back here for the explanation.

For the first piece of information, as it fits in one byte and our code let us deal with every byte of the buffer separately, all you need to do here is to extract the car speed directly from the first byte only, easy!

- *i.e. car_speed = buf[0];*

For the rest cases where the signal bits take more or less than a byte, you will need *bit manipulation* to put those bits in the right setup before reading them.

To read the RPM which lies in two different bytes (1 and 2) you need to do as follows:

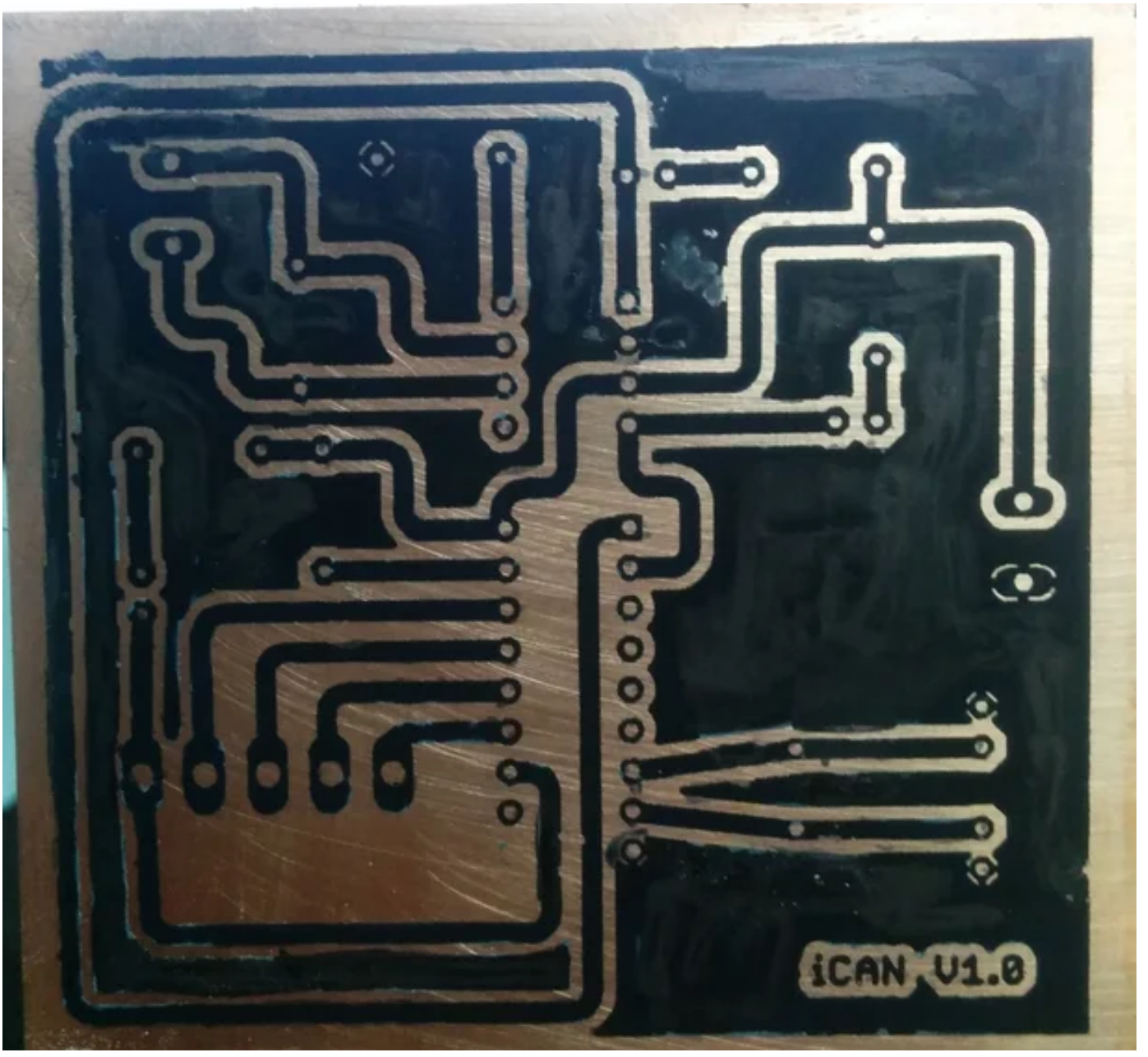
- First, read the higher byte (byte 1) into a two-byte integer, it will lay in the lower byte in *engine_rpm*.
  - *unsigned int engine_rpm = buf[1];*
- Then, shift this value 8 bits to the left to put the higher byte you read in its right position.
  - *engine_rpm = engine_rpm << 8;*
- Now, mask off all byte 2 bits except the first six bits used for the RPM signal.
  - *char temp = buf[2] & 0x3F;*
- Here, we just need to add the higher byte to the lower one and get our final value.
  - *engine_rpm = engine_rpm + temp;*

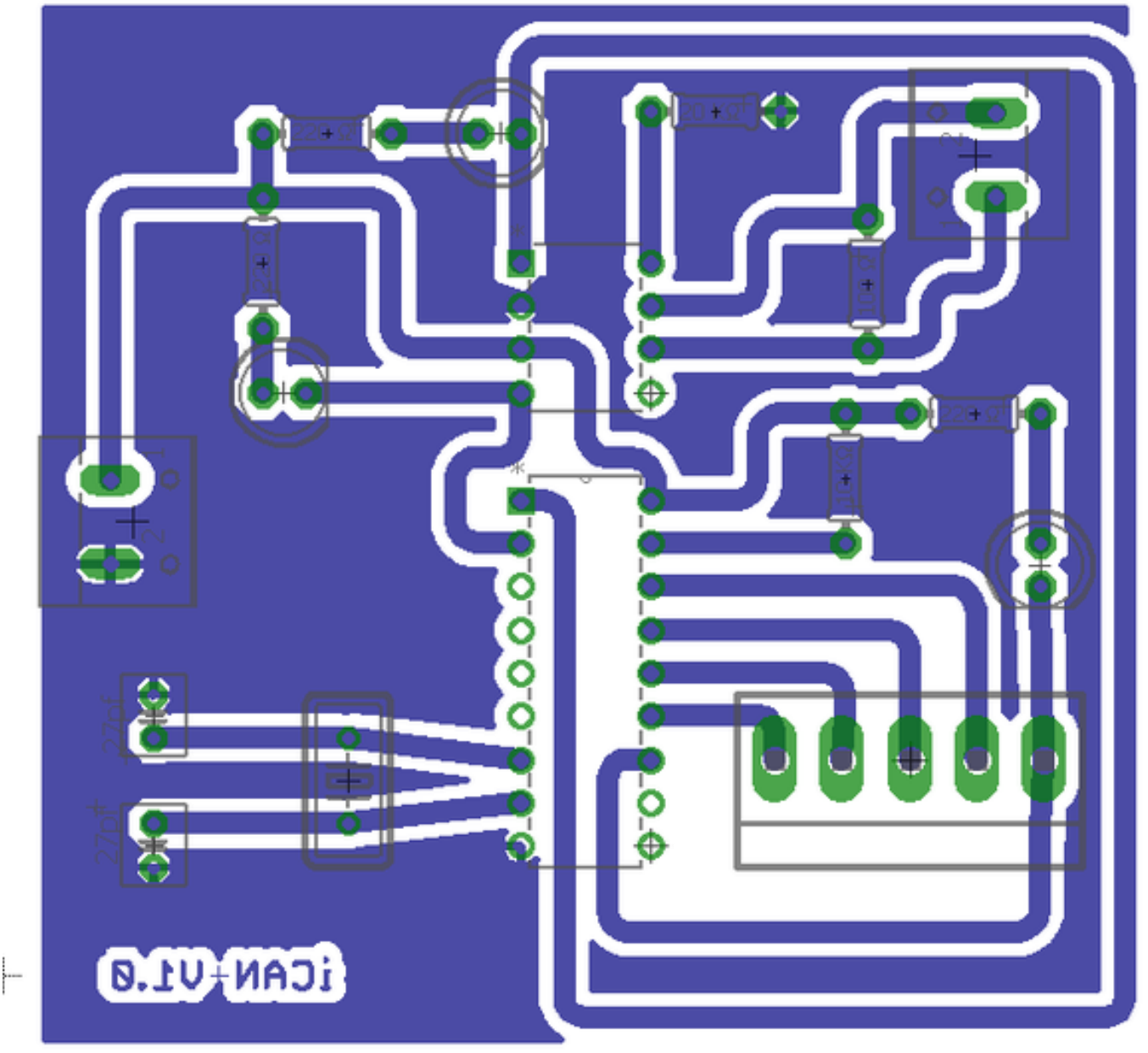
To read the value of **Check Engine** and **Oil** bits, you will need to mask off all the bits but the one you want, piece of

cake!

Tataaaaaa! We correctly *extracted all the information embedded in the message!*

Step 6: The Evolution of Breadboard to PCB



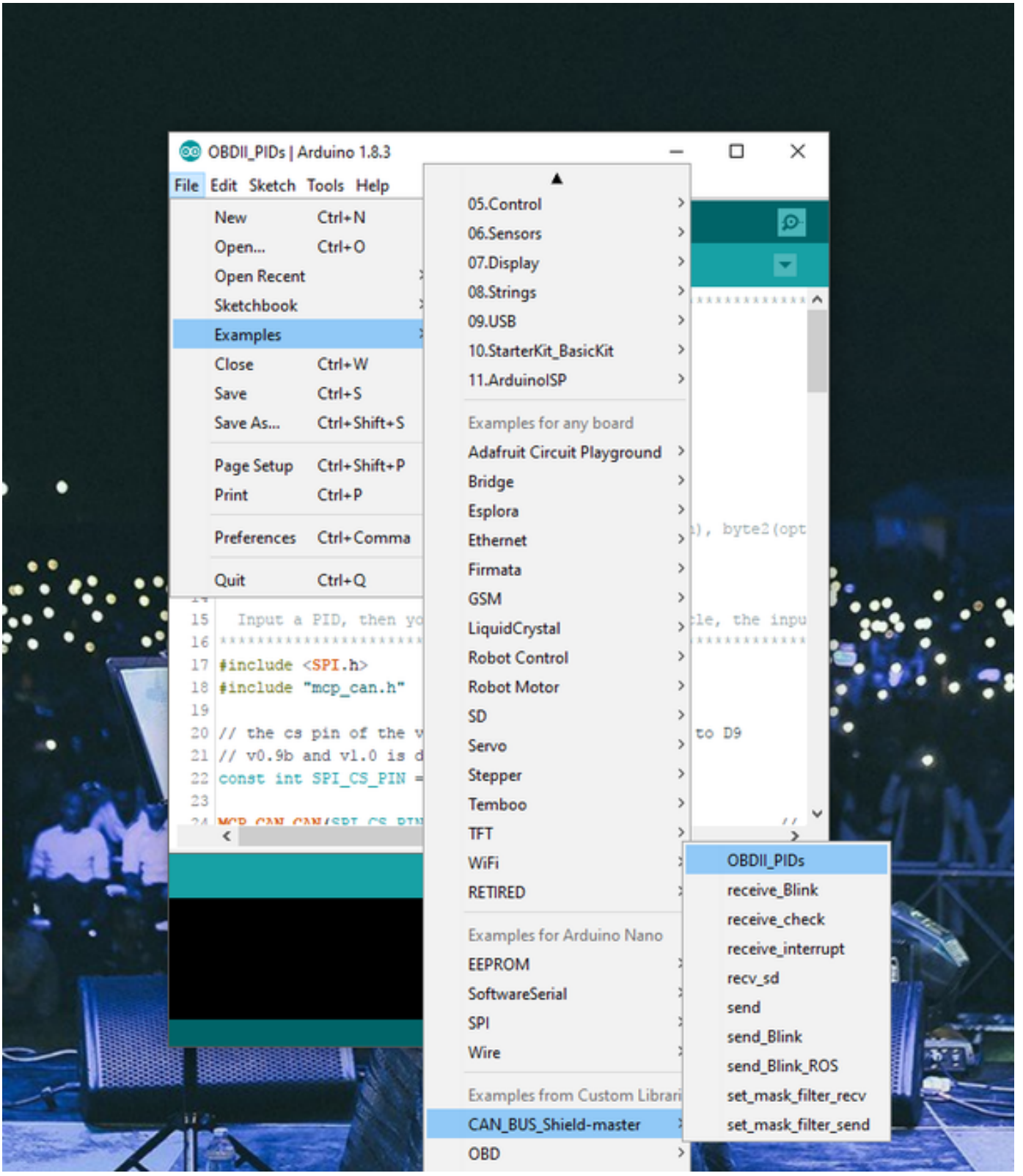


After testing and making sure that everything is working properly, I decided to start learning PCBs making by converting this very project to a PCB.

Attached here are the **Eagle** files for the layout. The layout was designed, printed on a copper board but not yet milled or tested, you can revise the design and give your comments below, you can also print it and make your own PCB and then tell us what you ended up with. And you can even suggest

enhancements to the layout design!

Step 7: Have a Little Chat With Your Car



Have you ever been to your mechanic complaining from a

mysterious light in your dashboard and you see him plugging some device to your car *~magic happens~* that shows you a fault code describing in details what the non-mysterious light means?!

Some say, with our **CAN** setup, we can talk to our cars through their OBD-II port, just like the mechanic in the previous paragraph, by sending the so called PIDs to the OBD requesting some parameter from the car communication buses and wait for the response message carrying the values you asked for.

An example covering this PID part is given in the examples section of the library, and the list of available PIDs is available on wiki.

I didn't try it, but I will do this very soon. **Take care of your connections and read about OBD before you plug anything to your car.**

Feel free to discuss anything with us below, we are all here to learn from each other, Peace!

Participated in the
First Time Author Contest

Be the First to Share