COBALT and GONOGO: two demonstrators of the Pybride Hat concept

Jacques Ehrlich

I. INTRODUCTION

When I decided to install a PM2.5 sensor on my balcony to measure the amount of fine particles, I came up against an obstacle: the lack of a power outlet. Of course, nothing is prohibitive and the day I decide to drill the internal insulation and the front wall, the problem will be solved, but that day has not yet arrived. I explored a solution such as a energy-storage battery and a solar panel, but the balcony being exposed to the north, therefore not very sunny, I calculated that I would need a bulky and expensive solar panel which led me to give up this solution. It is also the price and the size which made me give up a solution based on battery and mini wind turbine.

I finally opted for a very rustic solution: two lead batteries 12V/7Ah used in alternation: while one is in use, the other is charging and vice versa. The adventure could have ended there if it hadn't led me to imagine the Pybride Hat concept and to develop two cards according to this concept: COBALT and GONOGO which are described in this article.

II. PYBRID HAT : WHAT IS IT?

Pybride Hat is a contraction - admittedly, a very rough one - of Pi Hat Hybride. A Pybrid card is to the Raspberry Pi (Rpi) what a hybrid vehicle is to an electric vehicle: when the battery of the electric vehicle is discharged, the hybrid vehicle retains its main function (driving) but in thermal mode (provided there is gas in the tank). Similarly, a Pybrid board retains its main function even when its Rpi is absent. On the other hand, when the Rpi is present, it has optional additional functions.

Below is a first list of specifications that a card must meet to be "approved" as a Pybrid:

- Provide its main function without being connected to the Rpi,
- Provide an optional function thanks to its connection to the Rpi
- Have a format that allows it to be plugged into an Rpi via the 40-pin connector.
- Not conflict with other cards stacked on the 40-pin connector, which leads to using only the I2C interface.
- Be functional, even if the components related to the Pybrid function are not mounted on the board.
- Be easy to wire for a beginner electronics technician and for this reason, prefer through-hole components.
- Do not depend on the 3,3v and 5v power supplies of the Rpi except for level adaptations.

At this stage, the specifications are still embryonic: they will be refined later if the concept is of interest to readers.

III. LET'S GET BACK TO THE POINT

The Pybride concept having been presented, we can come back to the subject of my concern: measuring with a PM2.5 sensor. It is based on the well-known PMS5003 module that can easily be connected to an Rpi by following the abundant tutorials available on the Internet.

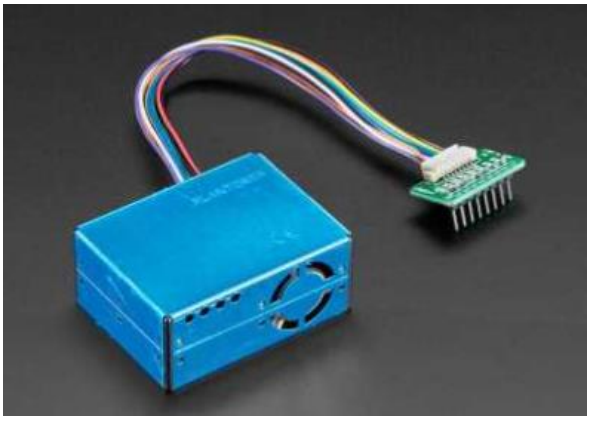


Figure 1 – Le capteur ¨PMS5003

For domotics fans (of which I am), it is also relatively simple to integrate this sensor in Domoticz or Jeedom, using the above mentionned tutorials.

There was still the question of energy and as indicated in the introduction, it is the solution based on two batteries used in alternation that, for lack of a better term, was chosen.

There are several options, however:

- - 1) cleanly shut down the Rpi (halt or shutdown) before the battery in use drops too low, disconnect the battery and replace it with the charged battery, restart the Rpi and so on. This solution is not viable in the medium term and sooner or later leads to the battery in use being excessively discharged, leading to its deterioration and, something to be avoided, an uncontrolled shutdown of the Rpi (without prior halt or shutdown.)
- - 2) connect the two batteries in parallel with two Schottky diodes to prevent one from flowing into the other. This allows to disconnect one of them without interrupting the operation of the Rpi. But this remains moderately satisfactory because it is comparable to a double capacity battery (12v/14 Ah) and again exposes the risk of a complete discharge of the two batteries.
- 3) to develop an alternating discharge controller: this is the object of the first Pybrid project: COBALT which is described in the following section.

Having an alternating discharge controller is good but it is not enough. Indeed, some calculations of consumption quickly showed me that the autonomy of each battery would be very short (less than 24 hours) in the case of a H24 operation. For pollution measurements, I considered that a H24 operation was useless and therefore in order to increase the autonomy, I could be satisfied with an intermittent operation: for example, every 30 minutes, powering up the Rpi and the PMS5003 sensor, waiting for one minute for temperature stabilization, measurements for 8 minutes, followed by one minute of shutdown and finally powering down. The electronic device - sometimes called PULSE/PAUSE - which allows this function is sold by a Belgian company well known for the quality of its products. This one gave me full satisfaction. This PULSE/PAUSE is based on an analog solution, the PULSE and PAUSE times being adjusted in a rather approximate way with potentiometers. Several factors motivated me to develop a digital solution: 1) my "cocorico" spirit which pushed me to propose a 100% *Frenchy* solution, 2) my passion for digital electronics, 3) the opportunity to demonstrate the Pybrid concept and 4) the desire to have an accurate programmer allowing a parameterization with one second resolution of the PULSE and PAUSE times. This is how my second Pybride project was born: GONOGO also described in this article.

IV. COBALT

COBALT an acronym of ***CO****ntrôleur de décharge de* ***B****atterie en* ***ALT****ernance.*

A. Theory of operation

The first battery connected to COBALT (battery A) is the active battery. If we then connect battery B, it is put on standby. When battery A output voltage falls below a given threshold (11.02v), it is switched off and battery B is switched on. During this time A can be manually disconnected, charged, and then reconnected once charged, but it is B that remains in service until its output voltage falls below the threshold. B is then switched off and A takes over and so on. This mutual exclusion system ensures that the batteries undergo complete charge and discharge cycles in turn. It should be noted that COBALT is not a charger, but only a discharge controller.

B. Schematic analysis

Figure 2 describes the COBALT schematic, whose symmetry is to be noted. We describe only its left hand part.

At the heart of COBALT is a comparator U1 (LM311). On its inverting input we apply a reference voltage independent of the voltage drop of the battery, obtained by a Zener diode 6.2V (D1). On the non-inverting input is applied a voltage resulting from the resistor bridge divider formed by R2, RV1 and R19. This will fall as the battery voltage falls, leading to the inversion of the output of the comparator U1 loaded by R4 (transition from a low level to a high level). Note the presence of the resistor R3 in feedback on the non-inverting input, whose function is to create a hysteresis and thus avoid oscillations when the voltages at the two inputs of U1 are close one from the other. The potentiometer RV1 is used to set the precise threshold of the comparator. We have set it at 11.02V but any other value can be chosen as long as it does not risk discharging the battery below a voltage threshold which would be definitely damaging.

The output voltage of the comparator is applied through the resistor R6 to the base of transistor Q2 (2N2222) which operates in saturated/blocked mode. A high level applied to R6 causes the saturation of Q2 which drives the relay K1 (2RT) thus establishing continuity between its *make* contact (pins 21 and 24). As a result, the supply voltage is applied to the output through the Schottky diode D3.

It should be noted that the voltage from the collector of Q2 is not directly applied to the coil of the relay K1 but passes through a *break* contact of K2. This is the trick that guarantees the mutual exclusion of the two batteries with priority to the battery being currently discharged.

Due to the inertia of the relays, it is not possible to guarantee the absence of discontinuity at the time of switching from relay K1 to relay K2 (and vice versa). Therefore, a 1000uF capacitor (C6) has been placed on the output terminal block; it acts as a current (or voltage) reserve during the few milliseconds when there might be a discontinuity. On the other hand, it is not possible to guarantee the absence of overlapping and therefore two Schottky diodes (D3 and D4) have been placed to prevent one battery from flowing into the other for a few milliseconds.

Two LEDs (D9 and D10) are connected upstream of these diodes. The lit LED indicates the battery being discharged.

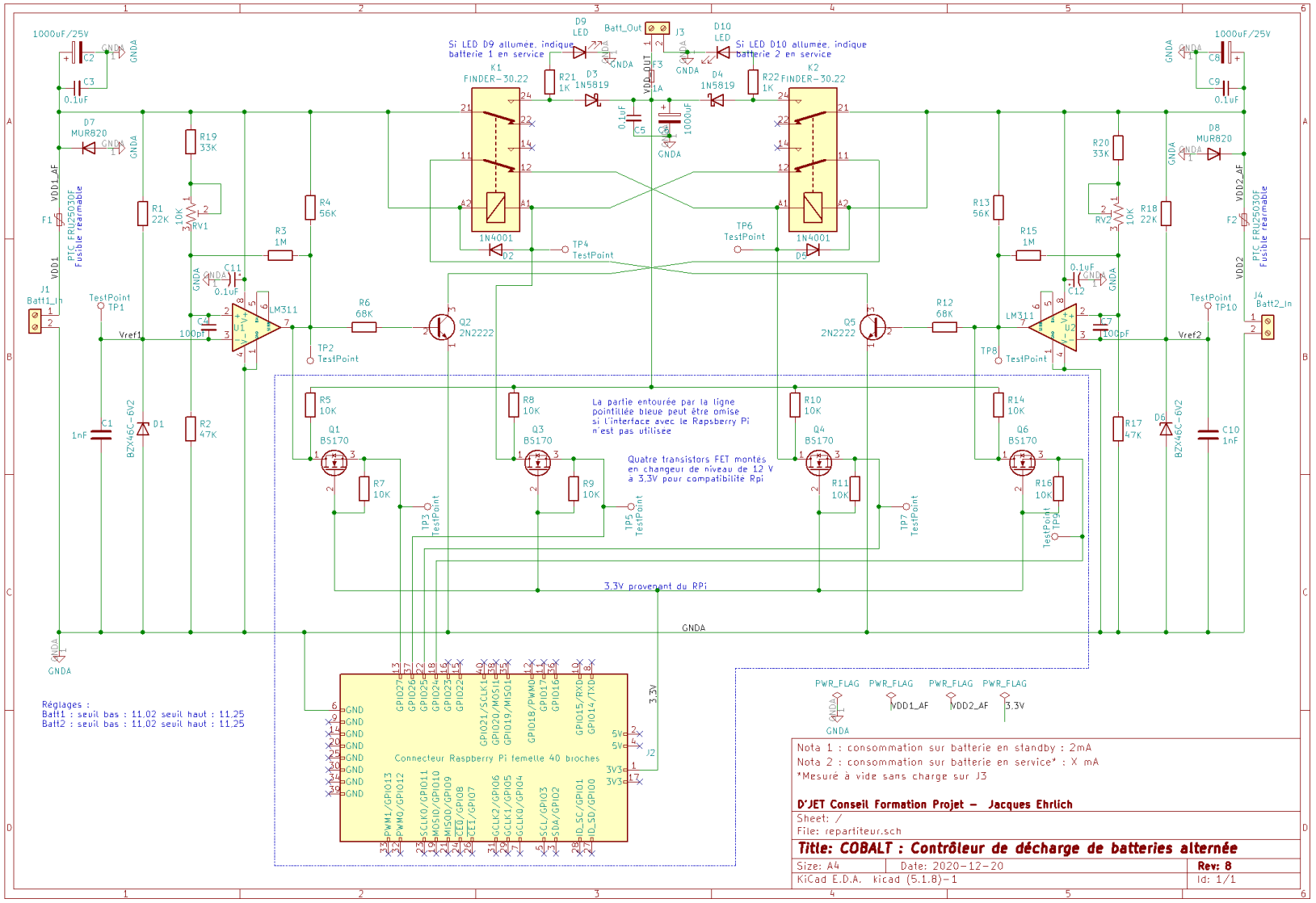


Figure 2 –COBALT schematic

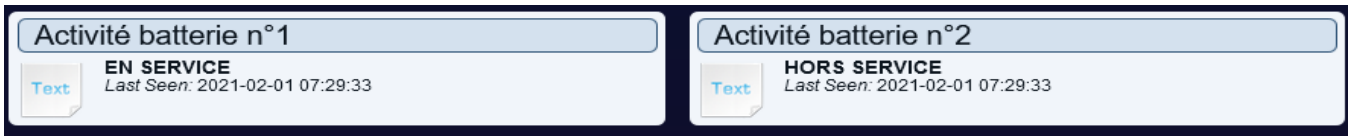


Figure 3 – COABALT integration into Domoticz application

C. *Rpi interface*

When I designed COBALT, I did not yet have the Pybrid concept in mind and as a result the interface to the Rpi does not quite meet the specifications stated above as it makes use of four inputs of the GPIO port instead of exclusively using the I2C interface. This will be corrected in the next version of COBALT.

On the lower part of the schematic is the interface to the Raspberry Pi (Rpi). These are four N-channel FET transistors mounted as level changers to convert 0-12V levels to 0-3.3V levels. The 3.3V voltage reference comes from the RPi.

For each battery two GPIO port inputs are used:

- GPIO 27 indicates that battery #1 is above/below the discharge threshold depending on whether the level is 1 (3.3v) or 0 (0v) respectively.
- GPIO 26 indicates that battery #1 is disconnected/connected depending on whether the level is 1 (3.3v) or 0 (0v) respectively.
- GPIO 24 indicates that battery n°2 is disconnected/connected depending on whether the level is 1 (3.3v) or 0 (0v) respectively.
- GPIO 25 indicates that battery #2 is above/below the discharge threshold depending on whether the level is 1.

This is an optional add-on feature as defined by the Pybrid Hat. Users who are not interested in the Rpi interface are not required to wire the components located inside the box delimited by the blue dotted lines on the diagram (Figure 2)

On the other hand, thanks to this interface and a few lines of code to be executed on the Rpi, it is very easy to send notifications by e-mail announcing that COBALT has switched from one battery to another. A solution consists in integrating COBALT in Domoticz which will take care of sending the notifications (Figure 3).

D. *Implementation and use*

The drawing of the double-sided circuit was made under Kicad. All the files required to produce the board are available on GitHub [1]. All the components are through-hole, which makes the wiring easy for inexperienced electronic technicians.

A minimalist C code executable after compilation on Rpi is also available on GitHub. It requires the the WiringPi library now installed as standard on Pi OS systems for Rpi.

As for the use, it is described in a short user's manual also available – in French only - on GitHub [1].

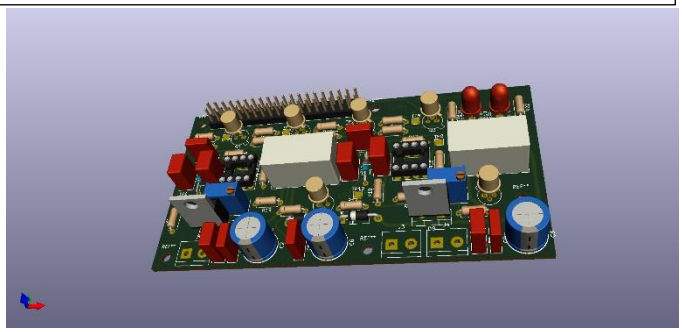


Figure 4 –3D view of COBALT board

V. GONOGO

A. *Theory of operation*

GONOGO is a programmer that periodically activates a two-contact relay (2RT) according to the following cycle: a working phase (GO) followed by a pause phase (NOGO) whose durations are programmable according to two time resolutions:

- From 1 second to 1 hour, 8 minutes and 15 seconds with a resolution of 1 second,

- From 1 minute to 2 days, 20 hours and 15 minutes with a resolution of 1 minute.

The time resolution (1 second or 1 minute) can be set independently for the GO and NOGO phases by means of jumpers.

The user has two possibilities for setting the duration of the GO and NOGO phases:

- - Using 12 switches: this is the main function of the Pybrid card and does not require the presence of an Rpi;
- - By software in the Rpi: this is the optional Pybride function which requires the GONOGO card to be plugged into the Rpi via the 40-pin connector **and the 12 switches to be mandatory set to ON**.

B. *Schematics analysis*

Figure 5 shows part of the GONOGO board schematic. Let's give credit where credit is due: for the design of GONOGO and more particularly the use of the 4060 and 4040 frequency dividers, I have taken Remy Mallard's diagrams published on his site `sonelec-musique.com`. On its upper part, the blue dotted box shows the time base which delivers two signals of period 1 sec and 60 seconds respectively. The 1sec signal is obtained by successive divisions of a 32768 HZ clock frequency (Y1 quartz) using U2 (4060) used as a divider by 16384 cascaded by U7A (4013), a D flip-flop operating as a divider by 2. The output of U7 drives the clock input of U9 (4040) mounted as a divide-by-60 to provide a signal with a period of 1 minute. The 4-input NAND gate U12B (4012) detects the simultaneity of the states 1 of the outputs Q2, Q3, Q4, Q5 whose weights are 2^2, 2^3, 2^4 and 2^5 respectively, i.e. the

value:

$2^2 + 2^3 + 2^4 + 2^5 = 4 + 8 + 16 + 32 = 60$.

When the counter reaches this value, it is reset by looping the output of U12B back to the Reset input of U9.

The lower part of the diagram consists of two quite similar blocks, one for setting the GO time and the other for the NOGO time. Only the left part corresponding to the NOGO is described here.

Its heart is constituted by U3 (4040) a frequency divider whose clock frequency is 1s or 1 minute depending on the position of jumper JP2 and whose outputs are directed through 12 switches (SW1, SW2) and 12 OR gates to an AND with 12 inputs (formed by gates U1B, U11A, U5, U1D, U8A). The OR gates are justified only by the optional function of the Pybrid which will be seen later. Indeed, the NOGO duration setting comes from the OR switches of an extension of the GPIO port of the Rpi.

Assume that we want a NOGO duration of 1 hour, i.e. 3600 seconds. This value must be converted into binary which gives 111000010000. The 1's correspond to switches ON and the zeros to switches OFF, i.e. SW5, 10, 11 and 12 on ON and all the others on OFF. Readers who are reluctant to convert decimal to binary will find online converters on the internet that will do the calculation for them or they can use the little `gonogo-switch` program available on Github, which directly delivers the position of the switches according to the desired duration (see appendix).

The same reasoning applies with a resolution of time to the minute. Let's assume that we want a duration of 1 day, 1 hour and 15 minutes. First, we convert this duration into minutes, i.e. 1515 minutes, which we then convert into binary, which gives 010111101011. All that remains is to position the corresponding switches.

When the NOGO time is reached, the NOGO divider must be stopped and in turn the GO timer must be started. This is done by an RS flip-flop (U8C, U8D) whose PULSE_ENABLE (respectively PAUSE_ENABLE) output is directed to the gates U1A, U1B (respectively U13A, U13B) which blocks or unblocks the clock signal (1s or 60s) applied to the divider U3 (respectively U15). Conversely, when the GO time runs out, the NOGO takes over thanks to the RS flip-flop which ensures mutual exclusion between GO and NOGO. The PULSE_ENABLE output of the RS flip-flop is also directed to transistors Q1, Q2 (2N2222) operating in saturated-blocked mode and controlling relay K1 and LED D1 respectively. The *make* contacts of K1 are connected to terminal block J2 for free use by the user as long as they do not exceed the 2A current allowed by the Finder 30.22 relay. Fuse protection (2A1, 2A2) is provided for any eventuality.

Finally, as we want the power-up to always start with a GO phase, we have added the capacitor C4 to input 8 of U1C (RS flip-flop) to force its PULSE_ENABLE output to 1 on power-up.

C. RPi interface.

As above mentioned, it is possible to optionally set the GO and NOGO phase times by a program running on the Rpi. In this case, it is no longer the switches that are used to set these times but the output values 1 or 0 of a 12-bit parallel port for GO and 12 bits for NOGO. It should be remembered that it is a rule that a Pybrid card must leave the GPIO port of the Rpi free for other (non-Pybrid) cards that may be plugged in.

The diagram in Figure 6 describes the chosen solution. Again the symmetry of the diagram is obvious. The Rpi port is extended by two circuits U20 for the NOGO and U21 for the GO (MCP23017) whose outputs GPA0 to GPA7 and GPB0 to GPB4 are used. These two circuits are connected to the Rpi via an I2C bus. To minimise any risk of conflict with other boards using the I2C bus, jumpers JP31 to JP33 (for the NOGO) and JP34 to JP36 (for the GO) allow their addresses to be set between 0 and 7.

A glance to of the MCP23017 datasheet reveals a fairly complex circuit with many configuration registers that can be daunting for inexperienced programmers. Fortunately, the excellent WiringPi library provides functions that simplify matters. See the appendix for a minimalist code written in C, using this library and available on the GitHub of the GONOGO board [2].

The outputs of the port extensions P[1..12] and U[1..12] are directed to the inputs of the OR gates in Figure 5, which we have already commented on (see §V.B).

Let us examine the two configuration modes for the NOGO part (resp. for the GO part):

- Configuration by switches: in this case the inputs/outputs of the MCP23017 are configured as inputs forced to 0 by the pull-up resistor networks RN1, RN2 (resp. RN3, RN4). These values are applied to the inputs of the OR gates P[1..12] (resp. U[1..12]) thus opening the way for the PA[1..12] (resp. PU[1..12]) signals;
- Configuration by Rpi: in this case all switches of SW1, SW2 (resp. SW3, SW4) must be ON. The inputs and outputs of the MCP23017 are configured as outputs. If we look at an output of rank i, a 1 on an output Pi (resp. Ui) forces the output PPAi (resp. PPUi) of the OR gate of rank i to 1, whatever the state of the signal PAi (resp. PUi). A zero, on the other hand, allows the corresponding PAi (resp. PUi) signal to pass. In this mode, the MCP23017 circuits act as programmable switches.

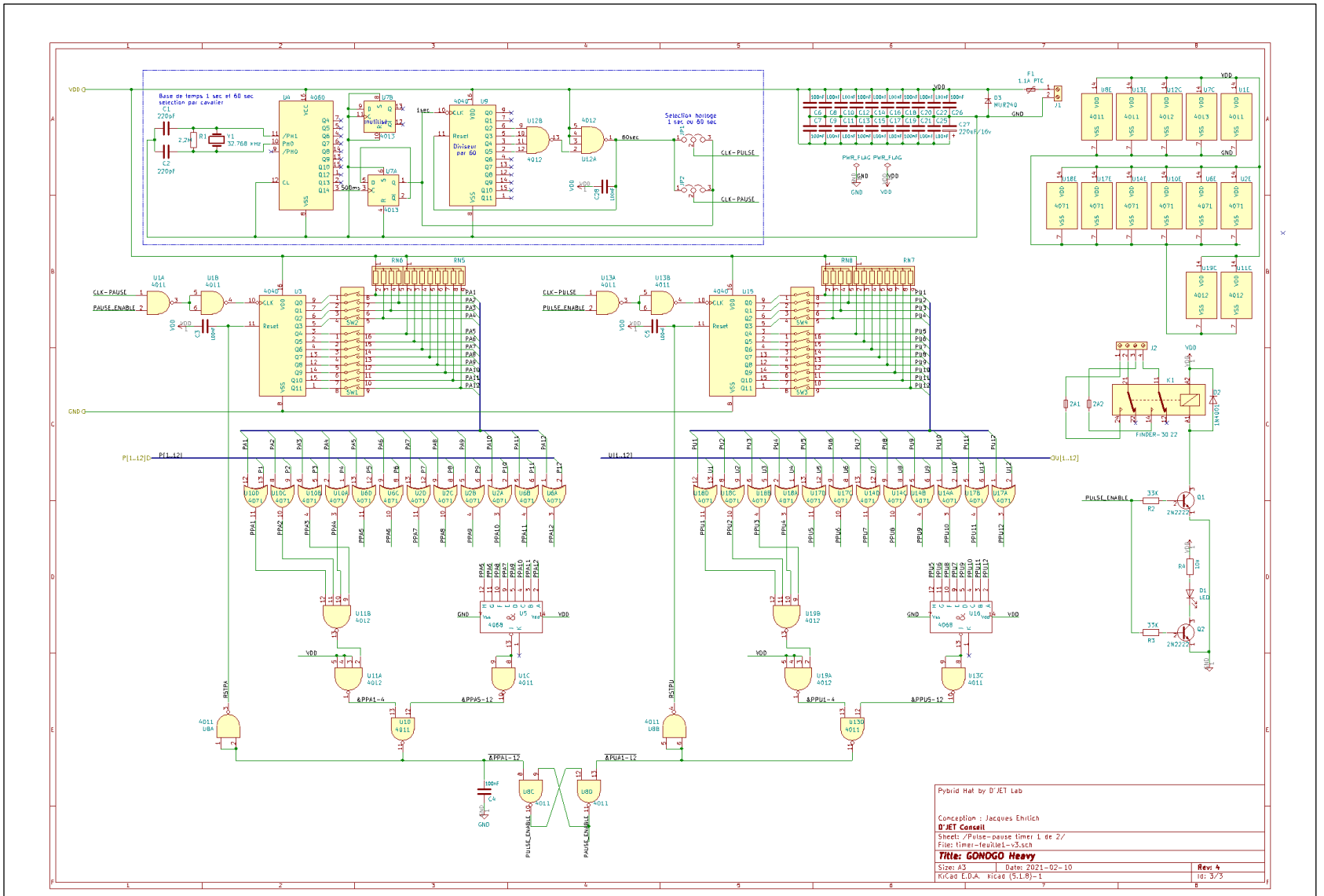


Figure 5 – Schéma de GONOGO – partie 1

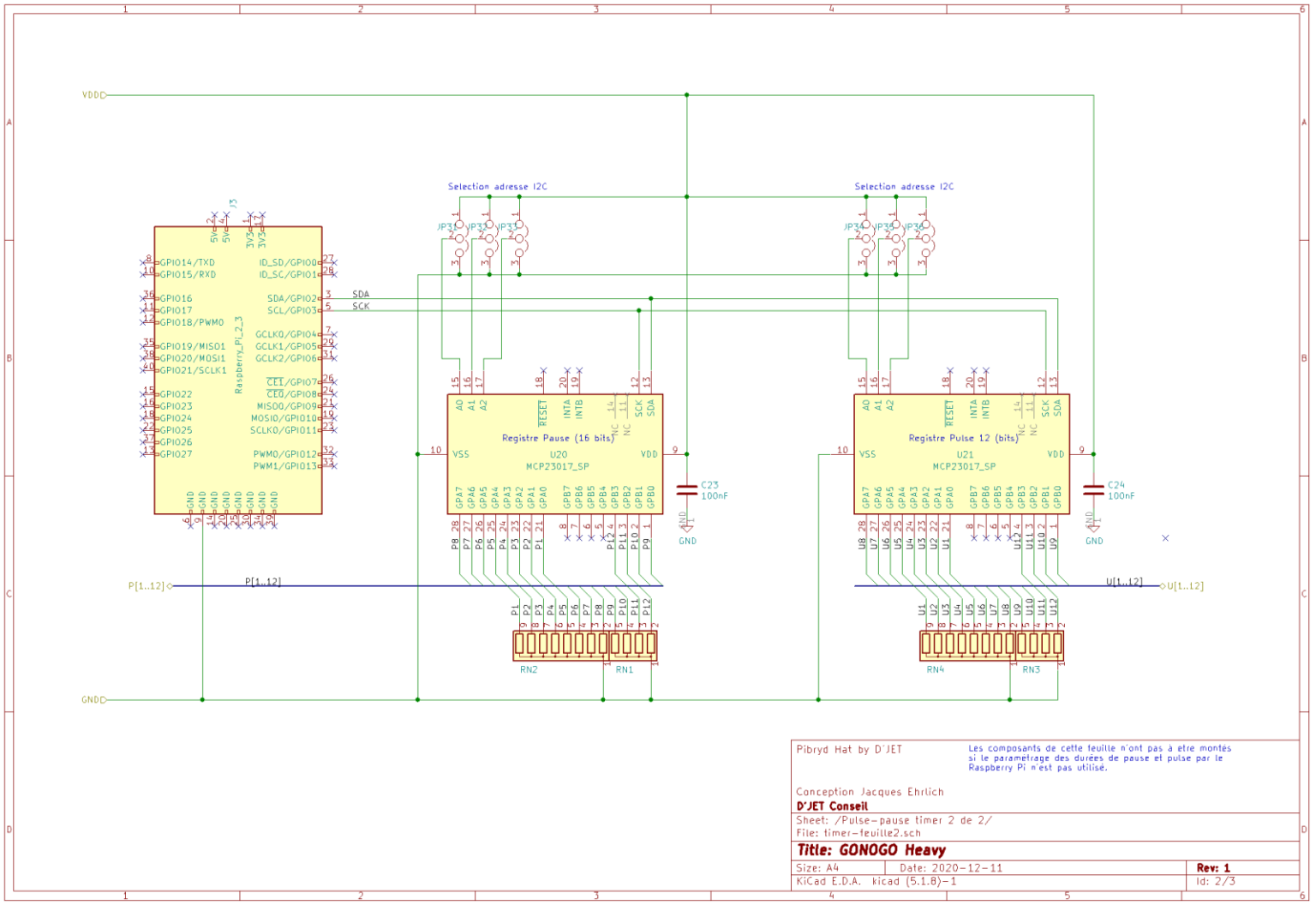


Figure 6 – Schéma de GONOGO – partie 2

D. Implementation

After a mock-up on a *breadboard*, I must admit that I had some difficulties in making the PCB. The proposed placement was obtained after three unsuccessful attempts. The final result satisfies the Pybride specifications, i.e. the possibility of plugging the board on an Rpi, but satisfies its designer a little less because of its form factor far from a square and its imposing dimensions.

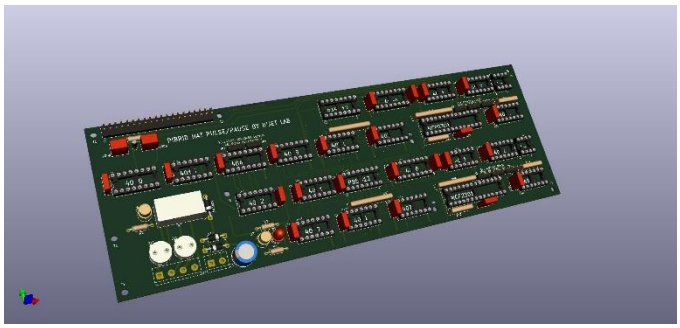


Figure 7 – 3D view of the GONOGO board, Heavy version

Regarding the wiring, here again, no difficulty since all the components are through-going. All schematics, manufacturing files and C code are available on GitHub [2].

VI. The 80/20 trap

Before I retired, I used to teach Functional Analysis and I used to say to my masters students:

"Never leave engineers alone to establish the functional specifications of a system but associate them for this work with people from marketing or value analysis. Otherwise ... beware of the 80/20 trap!"

Indeed, engineers (in the sense of "technicians") are technology lovers. When defining a system, they have a natural tendency to say *"We could add that ... oh! it would be nice if we added this or that ..."*

The end result is a system where 20% of the features will be of interest to 80% of the customers and 80% of the remaining features will only be of interest to 20% of those same customers. But the worst thing is that the features that only interest 20% of the customers are so sophisticated and complex that they will require 80% of the development effort.

And as I am used to taking a critical look at my work, and based on the well-known principle that the shoemaker is always the worst shod (it's a French proverb that may not have an equivalent in English) , I sincerely asked myself this question: have I not fallen into the 80/20 trap myself? Indeed, the addition of the Rpi interface has considerably complicated the GONOGO card. And finally, who will really be interested in the possibility of configuring the GO and NOGO times by a program rather than by switches? I prefer to let the reader decide this question.

VII. GONOGO Light

This thought led me to develop a "*light*" version of GONOGO. It is true that this version no longer meets the Pybrid criteria, but it will satisfy readers who are not interested in the Rpi interface. The schematic (Figure 9 in the appendix) is greatly simplified and does not require any further comments once the "*Heavy*" version has been understood. The design of the PCB presented no difficulty and required only one day's work, resulting in a board whose form factor is fairly close to 1.

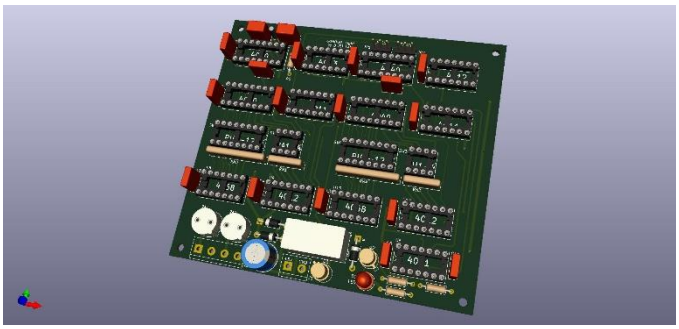


Figure 8 – 3D view of the GONOGO board, Light version

All schematics and manufacturing files are available on GitHub [2].

A first cost estimate reveals a ratio of 1.3 between the *Heavy* and *Light* versions.

VIII. GONOGO power supply

The *Heavy* and *Light* versions use exclusively CMOS logic circuits. All those in the 40XX family accept supply voltages in the range of 3 to 15v. However, the MCP23017 circuits require a voltage of 5V.

Therefore, the *Heavy* version must be exclusively supplied with 5v or the MCP23017 will be destroyed.

On the other hand, for the *Light* version, this constraint is removed provided that the Finder 30.22 5v relay is replaced by another model, for example a Finder 30.22 12v if a 12v supply is chosen.

Note that all COBALT and GONOGO boards are protected against reverse polarity by a combination of an ultra-fast diode (MURXXX model) and a resettable fuse.

IX. Conclusion

Apart from the pleasure and satisfaction of developing electronics, this work allowed me to explore the Pybrid Hat concept and confronted me with the dreaded 80/20 trap, reminding me once again that it is imperative to ask yourself some good questions before embarking on a project: what do I really need? Is a new feature worth the effort?

In any case, the reader interested in GONOGO can choose between the *Light* and *Heavy* versions. Concerning the latter, could we make it simpler and avoid the 24 OR gates that contribute to the complexity of the schematics placement and routing of the card? My old memories of DTL logic led me to explore in the future a diode-based solution that could perhaps simplify things.

But only a new design using surface-mounted components will allow a significant gain in board size. So, if the *Heavy* version should arouse sufficient interest of the readers, I will gladly launch into a new design which will also be for me the

occasion to finally confront myself with the SMDs.

X. Recommandation and Acknowledgements

It remains for me to recommend `sonelec-musique.com`, the excellent site of Remy Mallard, a real Ali Baba's cave which contains treasures of analog and digital electronics and from which I drew inspiration for the realization of the GONOGO project. I would like to thank him for allowing me to publish this article, which includes parts of a schematic he created.

XI. References

[1] GitHub of COBALT project:
https:// https://github.com/duodiscus92/projet-cobalt
[2] GitHub of GONOGO project:
https://github.com/duodiscus92/projet-gonogo

XII. ANNEXE 1

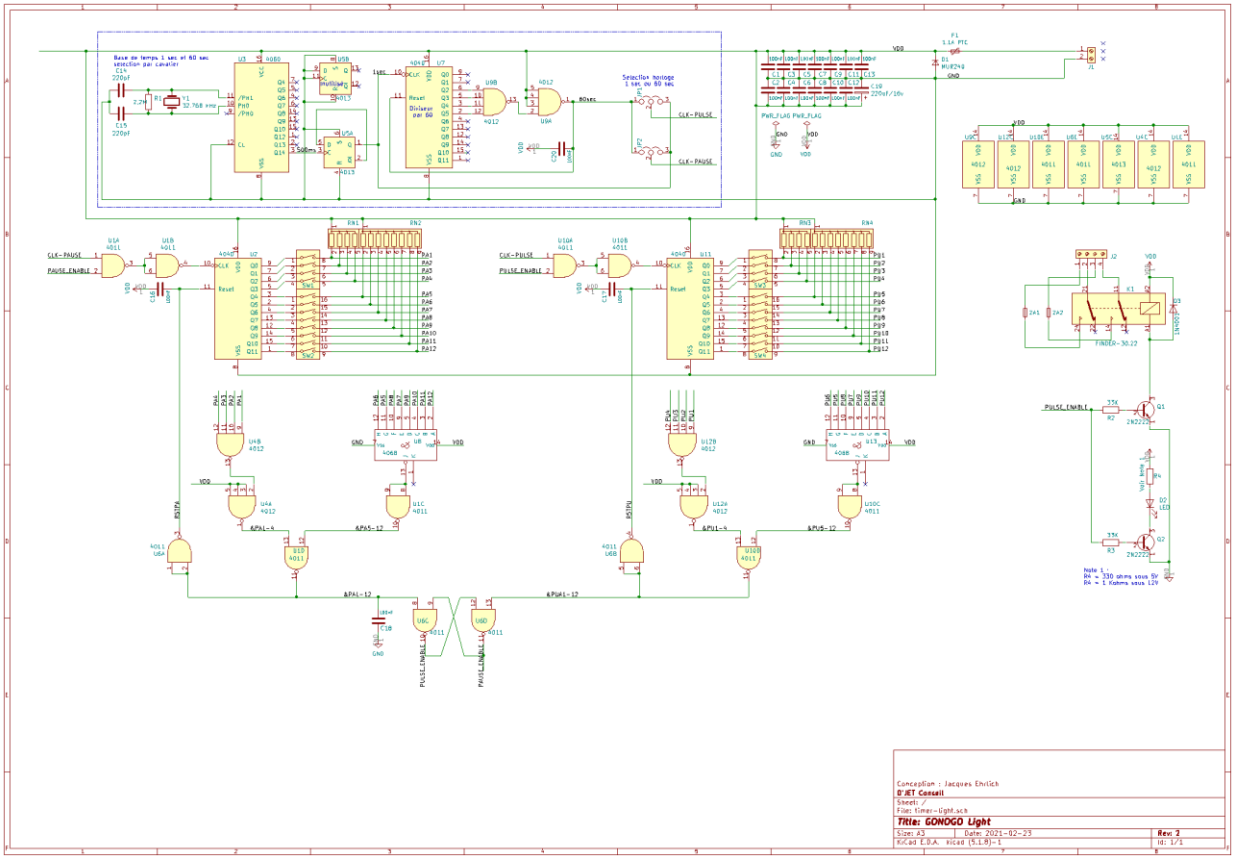


Figure 9 –GONOGO – Light schematics

XIII. ANNEXE II – GONOGO LIST OF COMPONENTS

A. *Light version*

Ref.	Qty	Name
2A1 2A2	2	Micro fusible 2A
C14 C15	2	220pF céramique
C19	1	220uF/16v
C1 C2 C3 C4 C5 C6 C7 C8 C9 C10 C11 C12 C13 C16 C17 C18 C20	17	100nF céramique
D1	1	SF64G
D2	1	LED
D3	1	1N4001
F1	1	Polyswitch RUEF090
J1	1	Bornier à vis 2 contacts
J2	1	Bornier à vis 4 contacts
JP1 JP2	2	Jumper_NC_Dual
K1	1	FINDER-30.22 5V (ou 12 v)
Q1 Q2	2	2N2222
R1	1	2,2M
R2 R3	2	33K
R4	1	10K
RN1 RN3	2	Réseau de 4 résistances 10K
RN2 RN4	2	Réseau de 8 résistances 10K
SW1 SW3	2	SW_DIP_x04
SW2 SW4	2	SW_DIP_x08
U3	1	4060
U4 U9 U12	3	4012
U5	1	4013
U1 U6 U10	3	4011
U2 U7 U11	3	4040
U8 U13	2	4068
Y1	1	Quartz 32,768 kHz

B. *Heavy version*

Ref	Qty	Name
2A1 2A2	2	Micro fusible 2A
C1 C2	2	220pF céramique
C3 C4 C5 C6 C7 C8 C9 C10 C11 C12 C13 C14 C15 C16 C17 C18 C19 C20 C21 C22 C23 C24 C25 C26 C28	25	100nF céramique
C27	1	220uF/16v
D1	1	LED
D2	1	1N4001
D3	1	SF64G
F1	1	Polyswitch RUEF090
J1	1	Bornier à vis 2 contacts
J2	1	Bornier à vis 4 contacts
J3	1	Connecteur femelle 40 broches
JP1 JP2 JP31 JP32 JP33 JP34 JP35 JP36	8	Jumper_NC_Dual
K1	1	FINDER-30.22 5v
Q1 Q2	2	2N2222
R1	1	2,2M
R2 R3	2	33K
R4	1	10k

RN1 RN3 RN6 RN8	4	Réseau de 4 résistances 10K
RN2 RN4 RN5 RN7	4	Réseau de 8 résistances 10K
SW1 SW3	2	SW_DIP_x08
SW2 SW4	2	SW_DIP_x04
U2 U6 U10 U14 U17 U18	6	4071
U11 U12 U19	3	4012
U20 U21	2	MCP23017_SP
U4	1	4060
U5 U16	2	4068
U7	1	4013
U1 U8 U13	3	4011
U3 U9 U15	3	4040
Y1	1	Quartz 32,768 kHz

XIV. COBALT LIST OF COMPONENTS

Ref	Qty	Name
C1 C10	2	1nF céramique
C2 C8	2	1000uF/25V
C4	1	100pf céramique
C3 C5 C9 C11 C12	5	0.1uF céramique
C6	1	1000uF
C7	1	100pF céramique
D1 D6	2	BZX46C-6V2
D2 D5	2	1N4001
D3 D4	2	1N5819 (Zener)
D7 D8	2	MUR820
D9 D10	2	LED
F1 F2	2	PTC FRU25030F
F3	1	Micro fusible 2A
J1	1	Bornier à vis 2 contacts
J2	1	Connecteur femelle 40 broches
J3	1	Bornier à vis 2 contacts
J4	1	Bornier à vis 2 contacts
K1 K2	2	FINDER-30.22 12V
Q1 Q3 Q4 Q6	4	BS170
Q2 Q5	2	2N2222
R1 R18	2	22K
R19 R20	2	33K
R2 R17	2	47K
R21 R22	2	1K
R3 R15	2	1M
R4 R13	2	56K
R5 R7 R8 R9 R10 R11 R14 R16	8	10K
R6 R12	2	68K
RV1 RV2	2	10K
U1 U2	2	LM311

Note : component lists can easily be generated in Kicad with the project files available on GitHub.

XV. GONOGO-SWITCH SOFTWARE

Written in C, this software provides the switch position according to the desired duration of the GO and NOGO phases

On PC under Cygwin or on RPi under Pi OS (formerly Raspbian) you will have to compile the source program:

```
$ gcc gonogo-switch.c param.c -o gonogo-switch
```

You can then run it and type the following command to find out its various options for use:

```
$ ./gonogo-switch -?
```

You will find that it is possible to provide the desired duration in different forms: in seconds, in minutes and seconds, in hours, minutes and seconds, in days, hours and minutes etc.
In the example below we indicate that the time resolution is the minute (-r60) and that we want a duration of 1 day, 10 hours and 35 minutes:

```
$ ./gonogo-switch -r60 -d1 -h10 -m35
```

And the programme responds:

```
SW1 = ON
SW2 = ON
SW3 = OFF
SW4 = ON
SW5 = ON
SW6 = OFF
SW7 = OFF
SW8 = OFF
SW9 = OFF
SW10 = OFF
SW11 = OFF
SW12 = ON
```

XVI. XVI. USE OF THE GONOGO HEAVY BOARD AND THE MCP23017

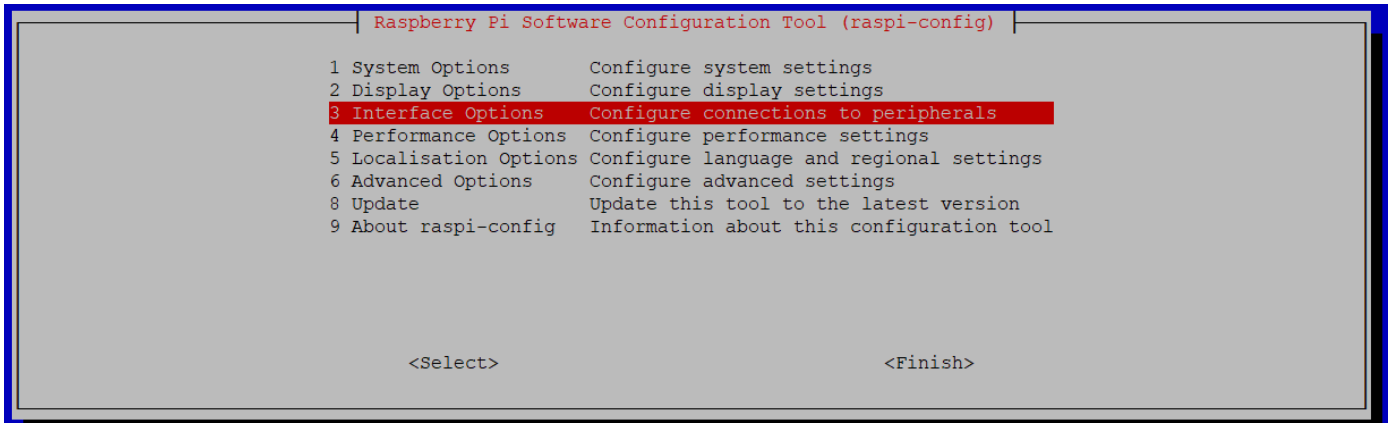
Using the jumpers, set the address of the two MCP23017 circuits on the GONOGO *Heavy* board:

- For the NOGO circuit: A0=0, A1=0, A2=0
- For the GO circuit: A0=1, A1=0, A2=0

The 12 NOGO switches and the 12 GO switches must also be set to ON.

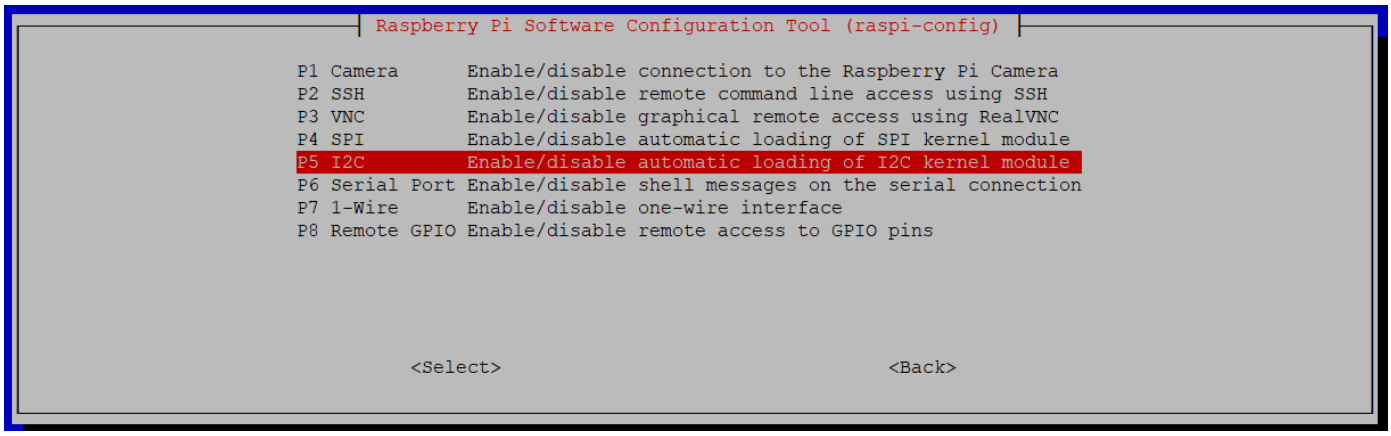
After having plugged the GONOGO card on the Raspberry Pi, let's power it up. Open a session on the Rpi by a SSH connection for example and we make sure that the I2C interface is activated. To do this run:

```
$ sudo raspi-config
```



Then select the Interface Options.

Then select I2C and Yes in the window that appears.



Then check that the two MCP23017 circuits are present at addresses 0x20 and 0x21.
To do this, type the following command, which returns the list of I2C devices present on the I2C bus.

```
$ sudo i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: 20 21 -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

Using the following command:
`$ gcc mcp23017.c -o mcp23017 -lwiringPi`
compile the program `mcp23017.c` (source file available on GitHub) in which the main part is in the `gonogoSetup` function which is extremely simple.

```
#include <stdlib.h>
#include <wiringPi.h>
#include <wiringPiI2C.h>

// mcp23017 I2C adress
#define MCPGO     0x21
#define MCPNOGO   0x20

// register definition
#define IODIRA    0x00
#define IODIRB    0x01
#define OLATA     0x14
#define OLATB     0x15

int fnNOGO, fnGO;

void gonogoSetup(int NOGOduration, int GOduration)
{
   wiringPiI2CWriteReg16 (fnNOGO, OLATA, ~NOGOduration) ;
   wiringPiI2CWriteReg16 (fnGO, OLATA, ~GOduration) ;
}
```

```
int main(int argc, char **argv)
{

  // WiringPi Library general initialisation
  wiringPiSetup();

  // Device initialisation : getting  a file number
  fnNOGO = wiringPiI2CSetup (MCPNOGO);
  fnGO = wiringPiI2CSetup (MCPGO);

  // Setting all GPIO pin as output (1= input, 0 = output)
  wiringPiI2CWriteReg8 (fnGO, IODIRA, 0);
  wiringPiI2CWriteReg8 (fnGO, IODIRB, 0);
  wiringPiI2CWriteReg8 (fnNOGO, IODIRA, 0);
  wiringPiI2CWriteReg8 (fnNOGO, IODIRB, 0);

  // Setting GPIO pin output level as convenient
  gonogoSetup(atoi(argv[1]), atoi(argv[2]));

  return 0;
}
```

It is then sufficient to call the executable program by providing it with the duration of the NOGO and GO in seconds as an argument, as in the example below where the NOGO is set to 30 seconds and the GO to 10 seconds.

```
$ ./mcp23017 30 10
```

The mcp23017 program is minimalist: no control is made on the value of the arguments. Its only purpose is to show the ease of use of the MCP23017 circuit when using the WiringPi library (installed as standard in Pi OS.) Everyone is free to adapt it to his own needs.
