COBALT et GONOGO : deux démonstrateurs du concept Pybride Hat

Jacques Ehrlich

I. INTRODUCTION

QUAND j'ai décidé d'installer sur mon balcon un capteur PM2.5 pour mesurer la quantité de particules fines, je me suis heurté à un obstacle : l'absence de prise de courant. Certes, rien n'est rédhibitoire et le jour où je me déciderai de percer l'isolation interne et le mur de façade, le problème sera résolu, mais ce jour n'est pas encore arrivé. J'ai exploré une solution de type batterie tampon et panneau solaire, mais le balcon étant exposé au nord, donc faiblement ensoleillé, j'ai calculé qu'il faudrait un panneau solaire encombrant et coûteux ce qui m'a amené à écarter cette solution. C'est également le prix et l'encombrement qui m'ont fait renoncer à une solution de type batterie tampon et mini-éolienne.

J'ai finalement opté pour une solution on ne peut plus rustique : deux batteries au plomb 12V/7Ah utilisées en alternance : pendant que l'une est en service, l'autre est en charge et inversement. L'aventure aurait pu s'arrêter là si elle ne m'avait pas conduit à imaginer le concept Pybride Hat et à développer deux cartes selon ce concept : COBALT et GONOGO qui sont décrites dans cet article.

II. PYBRID HAT C'EST QUOI ?

Pybride Hat est une contraction – certes, très approximative - de Pi Hat Hybride. Une carte Pybride est au Raspberry Pi (Rpi) ce qu'un véhicule hybride est au véhicule électrique : quand la batterie du véhicule électrique est déchargée, le véhicule hybride conserve sa fonction principale (rouler) mais en mode thermique (à condition qu'il y ait l'essence dans le réservoir.) De façon analogue une carte Pybride conserve sa fonction principale même en l'absence de son Rpi. En revanche, quand le Rpi est présent, elle possède des fonctions complémentaires optionnelles.

On trouvera ci-dessous une première liste de spécifications que doit satisfaire une carte pour être « agréée » Pybride :

- Fournir sa fonction principale sans être connectée au Rpi,
- Fournir une fonction optionnelle grâce à sa connexion au Rpi
- Posséder un format permettant de l'enficher sur un Rpi par le connecteur 40 broches.
- Ne pas entrer en conflit avec d'autres cartes empilées sur le connecteur 40 broches ce qui conduit à utiliser uniquement l'interface I2C.
- Être fonctionnelle, même si les composants liés à la fonction Pybride ne sont pas montés sur la carte.
- Être facile à câbler pour un électronicien débutant et pour cela préférer les composants traversants[1].
- Ne pas dépendre des alimentations 3,3v et 5v du Rpi sauf pour des adaptations de niveaux.

A ce stade, les spécifications sont encore embryonnaires : elles s'affineront par la suite si le concept présente un certain intérêt auprès des lecteurs.

III. REVENONS A NOS MOUTONS

Le concept Pybride ayant été présenté, on peut revenir au sujet qui m'a préoccupé : ma centrale de mesure PM2.5. Elle repose sur le module bien connu, PMS5003 qu'on peut aisément raccorder à un Rpi en suivant pour ce faire les abondants tutoriels disponibles sur Internet.

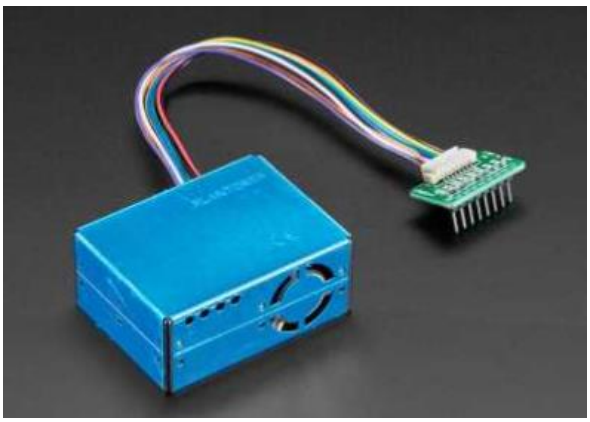


Figure 1 – Le capteur ¨PMS5003

Pour les fans de domotique (dont je fais partie), il est également relativement simple d'intégrer ce capteur dans Domoticz ou Jeedom en s'appuyant là encore sur les tutos existants.

Restait à régler la question de l'énergie et comme indiqué en introduction, c'est la solution fondée sur deux batteries utilisées en alternance qui, faute de mieux, a été retenue.

Plusieurs options sont possibles cependant :

- 1) arrêter proprement le Rpi (`halt` ou `shutdown`) avant que la batterie en cours d'utilisation ne tombe à un niveau trop bas, débrancher la batterie et la remplacer par la batterie chargée, remettre en route le Rpi et ainsi de suite. Cette solution n'est pas viable à moyen terme et conduit tôt ou tard à laisser la batterie en service se décharger de façon excessive conduisant à sa détérioration et, chose à éviter, un arrêt sauvage du Rpi (sans `halt` ni `shutdown` préalable.)

[1] On verra toutefois, dans la conclusion de cet article que je suis prêt à déroger à ce principe.

- 2) brancher les deux batteries en parallèle avec deux diodes Schottky pour éviter que l'une ne débite dans l'autre. Cela permet d'en débrancher une sans interrompre le fonctionnement du Rpi. Mais cela reste moyennement satisfaisant car assimilable à une batterie de capacité double (12v/14 Ah) et là encore expose au risque d'une décharge complète des deux batteries.
- 3) développer un contrôleur de décharge alternée : c'est l'objet de mon premier projet Pybride : COBALT qui est décrit dans la section suivante.

Disposer d'un contrôleur de décharge alternée c'est bien mais c'est insuffisant. En effet quelques calculs de consommation m'ont vite démontré que l'autonomie de chaque batterie serait très courte (moins de 24 heures) dans le cas d'un fonctionnement H24. S'agissant de mesures de pollution, j'ai considéré qu'un fonctionnement H24 était inutile et que pour accroitre l'autonomie, je pouvais me contenter d'un fonctionnement intermittent : par exemple, toutes les 30 minutes, mise sous tension du Rpi et du capteur PMS5003, attente d'une minute pour stabilisation de température, mesures pendant 8 minutes, suivie d'une minute de `shutdown` et enfin mise hors tension. Le programmateur – nommé parfois PULSE/PAUSE - qui permet d'assurer cette fonction est vendu par une société belge bien connue pour la qualité de ses produits. Celui-ci m'a d'ailleurs donné toute satisfaction. Ce PULSE/PAUSE repose sur une solution analogique, les temps de PULSE et de PAUSE se réglant de façon un peu approximative avec des potentiomètres. Plusieurs facteurs m'ont motivé pour développer une solution numérique : 1) mon esprit « cocorico » qui m'a poussé à proposer une solution 100% *frenchy* , 2) ma passion pour l'électronique numérique, 3) l'occasion de faire la preuve du concept Pybride et 4) l'envie de disposer d'un programmateur permettant un paramétrage à la seconde près des temps de fonctionnement et des temps de pause. C'est ainsi que naquit mon second projet Pybride : GONOGO également décrit dans cet article.

IV. COBALT

COBALT est l'acronyme de **CO**ntrôleur de décharge de **B**atterie en **ALT**ernance.

A. Principe de fonctionnement

La première batterie que l'on branche sur COBALT (la batterie A) est la batterie active. Si l'on branche ensuite la batterie B, celle-ci est mise en attente. Quand la tension aux bornes de A descend en dessous d'un certain seuil (11,02v), elle est mise hors service et c'est la batterie B qui est mise en service. Pendant ce temps on peut charger A puis la rebrancher une fois chargée, mais c'est B qui reste en service jusqu'à ce que la tension à ses bornes chute en dessous du seuil. B est alors mise hors service et c'est A qui prend la relève et ainsi de suite. Grâce à ce système d'exclusion mutuelle, on est assuré que les batteries subissent tour à tour des cycles de charge et de décharge complets. Il convient de noter que COBALT n'est pas un chargeur, mais uniquement un contrôleur de décharge.

B. Analyse du schéma

La Figure 2 décrit le schéma de COBALT dont on notera la symétrie. Nous ne décrivons que la partie gauche du schéma.

Au cœur de COBALT se trouve un comparateur U1 (LM311). Sur son entrée inverseuse on applique une tension de référence indépendante de la chute de tension de la batterie et obtenue par une diode Zener 6,2V (D1). Sur l'entrée non-inverseuse est appliquée une tension résultant du pont diviseur formé par R2, RV1 et R19. Celle-ci va chuter au fur et à mesure que la tension de la batterie va elle-même chuter, provoquant l'inversion de la sortie du comparateur U1 chargé par R4 (transition d'un niveau bas à un niveau haut). On notera la présence de la résistance R3 en contre-réaction sur l'entrée non-inverseuse dont la fonction est de créer une hystérésis et ainsi d'éviter des oscillations quand les tensions aux deux entrées de U1 sont très voisines. Le potentiomètre RV1 sert à fixer de façon précise le seuil de basculement du comparateur. Nous l'avons fixé à 11,02V mais toute autre valeur peut être choisie tant qu'elle ne risque pas de décharger la batterie en dessous d'un seuil de tension qui serait définitivement dommageable.

La tension de sortie du comparateur est appliquée, à travers la résistance R6 sur la base du transistor Q2 (2N2222) qui fonctionne en mode saturé/bloqué. Un niveau haut appliqué sur R6 provoque la saturation de Q2 qui commande le relais K1 (2RT) au travail établissant ainsi la continuité entre ses broches 21 et 24. De ce fait la tension d'alimentation est appliquée à la sortie à travers la diode Schottky D3.

Il convient de noter que la tension du collecteur de Q2 n'est pas directement appliquée à la bobine du relais K1 mais passe par un contact repos de K2. C'est l'astuce qui garantit l'exclusion mutuelle des deux batteries avec priorité à la batterie en cours de décharge.

Du fait de l'inertie des relais, on ne peut pas garantir l'absence de discontinuité au moment où se fait la bascule du relais K1 au profit du relais K2 (et inversement). C'est la raison pour laquelle on a placé un condensateur de 1000uF (C6) sur le bornier de sortie ; il fait réserve de courant (ou de tension) pendant les quelques millisecondes où il pourrait y avoir discontinuité. A l'inverse, on ne peut pas non plus garantir l'absence de recouvrement et c'est la raison pour laquelle on a placé deux diodes Schottky (D3 et D4) afin d'éviter qu'une batterie ne débite dans l'autre pendant quelques millisecondes.

Deux leds (D9 et D10) sont connectées en amont de ces diodes. La led allumée indique la batterie en cours de décharge.

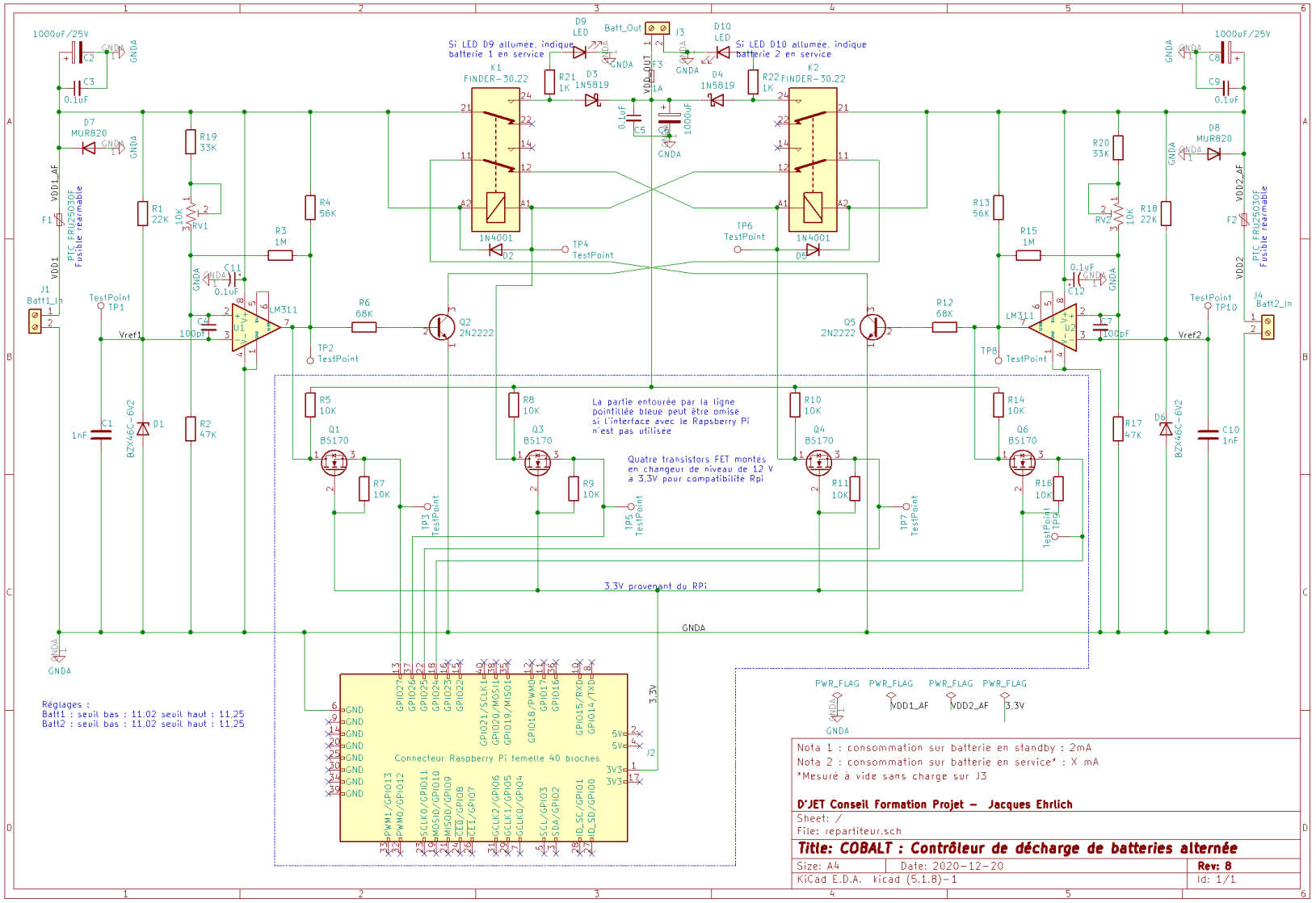


Figure 2 – Schéma de la carte COBALT

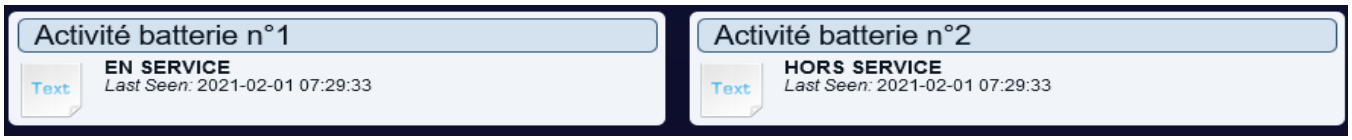


Figure 3 – Intégration de COBALT dans Domoticz

C. *L'interface avec le Rpi*

Quand j'ai conçu COBALT, je n'avais pas encore en tête le concept Pybride et par conséquent l'interface avec le Rpi ne répond pas tout à fait aux spécifications énoncées précédemment car elle fait usage des quatre entrées du port GPIO au lieu d'utiliser exclusivement l'interface I2C. Cela sera corrigé dans la prochaine version de COBALT.

Sur la partie inférieure du schéma se trouve l'interface vers le Raspberry Pi (Rpi). Ce sont quatre transistors FET canal N montés en changeur de niveau pour convertir les niveaux 0-12V en niveaux 0-3,3V. La référence 3,3V provient du RPi.

Pour chaque batterie deux entrées du port GPIO sont utilisées :

- GPIO 27 indique que la batterie n°1 est au-dessus/en-dessous du seuil de décharge selon que le niveau est à 1 (3,3v) ou 0 (0v) respectivement.
- GPIO 26 indique que la batterie n°1 est déconnectée/connectée selon que le niveau est à 1 (3,3v) ou 0 (0v) respectivement.
- GPIO 24 indique que la batterie n°2 est déconnectée/connectée selon que le niveau est à 1 (3,3v) ou 0 (0v) respectivement.
- GPIO 25 indique que la batterie n°2 est au-dessus/en-dessous du seuil de décharge selon que le niveau est à 1 (3,3v) ou 0 (0v) respectivement.

Il s'agit bien là d'une fonctionnalité complémentaire optionnelle conformément à la définition du Pybride Hat. L'utilisateur qui n'est pas intéressé par l'interface Rpi n'est pas obligé de câbler les composants situés à l'intérieur de l'encadré délimité par les pointillés bleus sur le schéma (Figure 2)

En revanche, grâce à cette interface et quelques lignes de code à exécuter sur le Rpi, il est très facile d'envoyer par mail des notifications annonçant que COBALT a basculé d'une batterie sur l'autre. Une solution consiste à intégrer COBALT dans Domoticz qui se chargera de l'envoi des notifications (Figure 3).

D. *Réalisation et utilisation*

Le dessin du circuit double face a été réalisé sous Kicad. Tous les fichiers requis à la production de la carte sont disponibles sous GitHub [1]. Tous les composants sont traversants ce qui met le câblage à la portée d'électroniciens peu expérimentés.

Un code minimaliste en C exécutable après compilation sur Rpi est également disponible sur GitHub. Il nécessite la présence de la librairie WiringPi[2] désormais installée en standard sur les systèmes Pi OS pour Rpi.

Quant à l'utilisation, elle est décrite dans une notice disponible également sur GitHub [1].

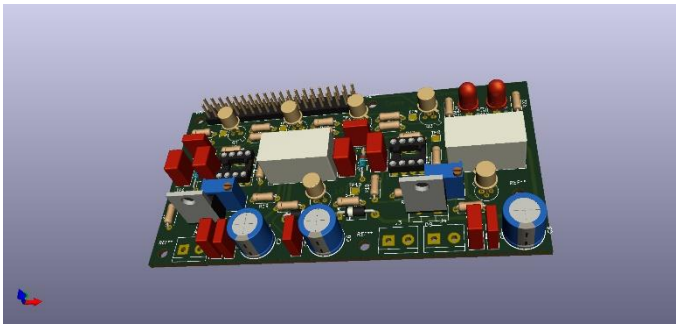


Figure 4 – Vue 3D de la carte COBALT

V. GONOGO

A. *Principe de fonctionnement*

GONOGO est un programmateur qui actionne de façon périodique un relais à deux contacts (2RT) selon le cycle suivant : une phase de travail (GO) suivie d'une phase de repos (NOGO) dont les durées sont programmables selon deux résolutions du temps :

- De 1 sec à 1 heure, 8 minutes et 15 sec avec une résolution de 1 seconde,
- De 1 minute à 2 jours, 20 heures et 15 minutes avec une résolution de 1 minute.

La résolution du temps (1 sec ou 1 minute) se fixe indépendamment pour les phases GO et NOGO à l'aide de cavaliers.

Deux possibilités s'offrent à l'utilisateur pour fixer les durées des phases GO et NOGO :

- A l'aide de 12 switches : c'est la fonction principale de la carte Pybride qui ne nécessite pas la présence d'un Rpi ;
- Par un logiciel résidant dans le Rpi : c'est la fonction optionnelle Pybride qui requiert d'enficher la carte GONOGO sur le Rpi via le connecteur 40 broches **et de positionner les 12 switches à ON**.

B. *Analyse du schéma*

La Figure 5 décrit une partie du schéma de la carte GONOGO. Rendons à César ce qui lui appartient : pour la conception de GONOGO et plus particulièrement l'utilisation des diviseurs de fréquence 4060 et 4040, j'ai repris les schémas de Remy Mallard publié sur son site `sonelec-musique.com`. Sur sa partie supérieure on trouve dans l'encadré pointillé bleu la base de temps qui délivre deux signaux de période 1 sec et 60 secondes respectivement. Le signal 1sec est obtenu par divisions successives d'une fréquence d'horloge 32768 HZ '(quartz Y1) à l'aide de U2

[2] http://wiringpi.com/

(4060) utilisé en diviseur par 16384 cascadé par U7A (4013), une bascule D opérant en diviseur par 2. La sortie de U7 attaque l'entrée d'horloge de U9 (4040) monté en diviseur par 60 pour délivrer un signal de période 1 minute. En effet, la porte NAND à 4 entrées, U12B (4012) détecte la simultanéité des états 1 des sorties Q2, Q3, Q4, Q5 dont les poids valent 2^2, 2^3, 2^4 et 2^5 respectivement, soit la valeur :

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

Lorsque le compteur atteint cette valeur, il est remis à zéro par le rebouclage de la sortie de U12B sur l'entrée Reset de U9.

La partie inférieure du schéma est constituée de deux blocs tout à fait similaires, l'un pour fixer la durée du GO et l'autre pour la durée du NOGO. On ne décrit ici que la partie gauche correspondant au NOGO.

Son cœur est constitué par U3 (4040) un diviseur de fréquence dont la fréquence d'horloge est de 1s ou 1 minute selon la position du cavalier JP2 et dont les sorties sont dirigées à travers 12 switches (SW1, SW2) et 12 portes OR vers un ET à 12 entrées (formé par les portes U1B, U11A, U5, U1D, U8A). Les portes OR sont justifiées uniquement par la fonction optionnelle du Pybride qui sera vue plus loin. En effet le paramétrage de la durée du NOGO provient des switches OU d'une extension du port GPIO du Rpi.

Supposons que l'on veuille une durée de NOGO de 1 heure soit 3600 secondes. Il faut convertir cette valeur en binaire ce qui donne 111000010000. Les 1 correspondent à des switches sur ON et les zéros à des switches sur OFF soit SW5, 10, 11 et 12 sur ON et tous les autres sur OFF. Les lecteurs rebelles à la conversion décimal-binaire trouveront sur internet des convertisseurs en ligne qui feront le calcul pour eux[3] ou bien pourront utiliser le petit programme `gonogo-switch` disponible sur github, qui délivre directement la position des switches en fonction de la durée souhaitée (voir annexe)

Le même raisonnement s'applique avec une résolution du temps à la minute. Supposons que l'on souhaite une durée de 1 jour, 1 heure et 15 minutes. D'abord, on convertit cette durée en minutes soit 1515 minutes[4] que l'on convertit ensuite en binaire ce qui donne 010111101011. Il ne reste plus qu'à positionner les switches correspondants.

Quand la durée du NOGO est atteinte il faut arrêter le diviseur NOGO et « passer la main » au timer GO. Ceci est réalisé par une bascule RS (U8C, U8D) dont la sortie PULSE_ENABLE (respectivement PAUSE_ENABLE) est dirigée vers les portes U1A, U1B (resp. U13A, U13B) qui bloque ou débloque le signal d'horloge (1s ou 60s) appliqué au diviseur U3 (resp. U15.). Réciproquement, quand la durée du GO s'achève c'est le NOGO qui prend la main grâce à la bascule RS qui assure l'exclusion mutuelle entre GO et NOGO. La sortie PULSE_ENABLE de la bascule RS est également dirigée vers les transistors Q1, Q2 (2N2222) fonctionnant en mode saturé-bloqué et qui contrôlent le relais K1 et la led D1 respectivement. Les contacts travail de K1 sont connectés au bornier J2 pour un libre usage par l'utilisateur dès l'instant où il ne dépasse pas le courant de 2A autorisé par le relais Finder 30.22. Une protection par les fusibles (2A1, 2A2) est assurée pour pallier toute éventualité.

Enfin, comme on souhaite qu'à la mise sous tension, cela commence toujours par une phase de GO on a ajouté le condensateur C4 sur l'entrée 8 de U1C (bascule RS) pour forcer à 1 sa sortie PULSE_ENABLE lors de la mise sous tension.

C. L'interface avec le Rpi.

Comme indiqué plus haut, de manière optionnelle, il est possible de fixer les durées de phases GO et NOGO par un programme s'exécutant sur le Rpi. Dans ce cas, ce ne sont plus les switches qui permettent de fixer ces durées mais les valeurs de sortie 1 ou 0 d'un port parallèle de 12 bits pout le GO et 12 bits pour le NOGO. Rappelons qu'on s'est fixé pour règle qu'une carte Pybride doit laisser libre le port GPIO du Rpi pour d'autres cartes (non Pybride) qui pourraient être enfichées.

Le schéma de la Figure 6 décrit la solution retenue. Là encore la symétrie du schéma est évidente. Le port du Rpi est étendu par deux circuits U20 pour le NOGO et U21 pour le GO (MCP23017) dont on utilise les sorties GPA0 à GPA7 et GPB0 à GPB4. Ces deux circuits sont reliés au Rpi via un bus I2C. Pour minimiser tout risque de conflit avec d'autres cartes faisant usage du bus I2C, les cavaliers JP31 à JP33 (pour le NOGO) et JP34 à JP36 (pour le GO) permettent de fixer leur adresse entre 0 et 7.

L'examen de la *datasheet* du MCP23017 révèle un circuit assez complexe, doté de nombreux registres de configuration qui ont de quoi effrayer les programmeurs peu expérimentés. Heureusement, l'excellente librairie WiringPi fournit là encore des fonctions qui simplifient les choses. Voir en annexe, un code minimaliste écrit en C, faisant usage de cette librairie et disponible sur le GitHub de la carte GONOGO [2].

Les sorties des extensions de ports P[1..12] et U[1..12] sont dirigées vers les entrées des portes OR de la Figure 5 que nous avons déjà commenté (cf. §V.B).

Examinons les deux modes de configuration pour la partie NOGO (resp. pour la partie GO) :

- Configuration par les switches : dans ce cas les entrées/sorties des MCP23017 sont configurées en entrées forcées à 0 par les réseaux de résistance de rappel RN1, RN2 (resp. RN3, RN4). Ces valeurs sont appliquées aux entrées des portes OR P[1..12] (resp. U[1..12]) ouvrant ainsi le passage aux signaux PA[1..12] (resp. PU[1..12]) ;
- Configuration par le Rpi : dans ce cas tous les switches de SW1, SW2 (resp. SW3, SW4) doivent être sur ON. Les entrées sorties des MCP23017 sont configurées en sortie. Si on s'intéresse à une sortie de rang i, un 1 sur une sortie Pi (resp. Ui) force à 1 la sortie PPAi (resp. PPUi) de la porte OR de rang i, quel que soit l'état du signal PAi (resp. PUi). En revanche un zéro laisse passer le signal PAi correspondant (resp. PUi). Ainsi, dans ce mode les circuits MCP23017 jouent le rôle de switches programmables.

[3] Voir un convertisseur décimal binaire sur https://sebastienguillon.com/test/javascript/convertisseur.html

[4] Voir un convertisseur de durée sur http://mon-convertisseur.fr/convertisseur-temps.php

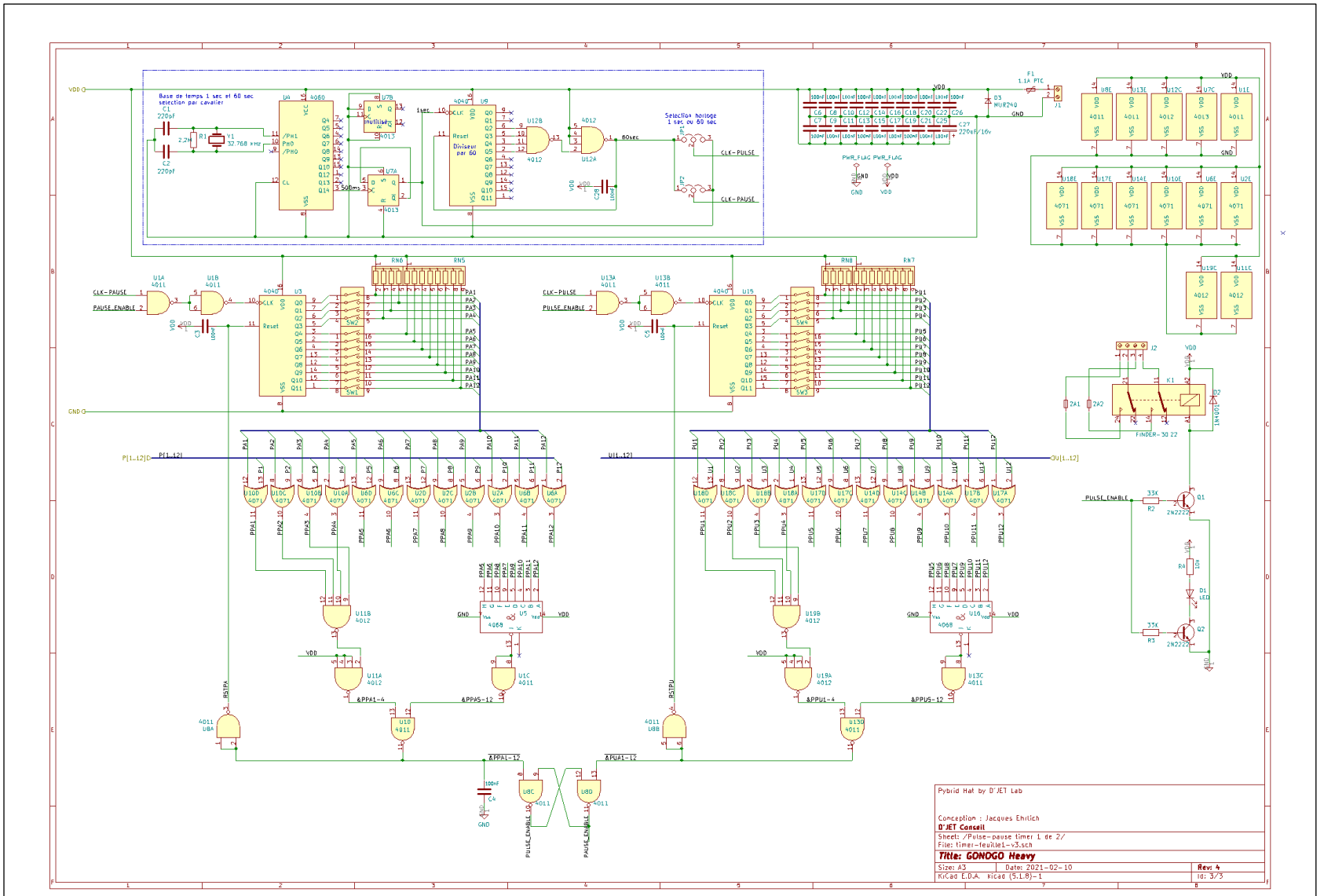


Figure 5 – Schéma de GONOGO – partie 1

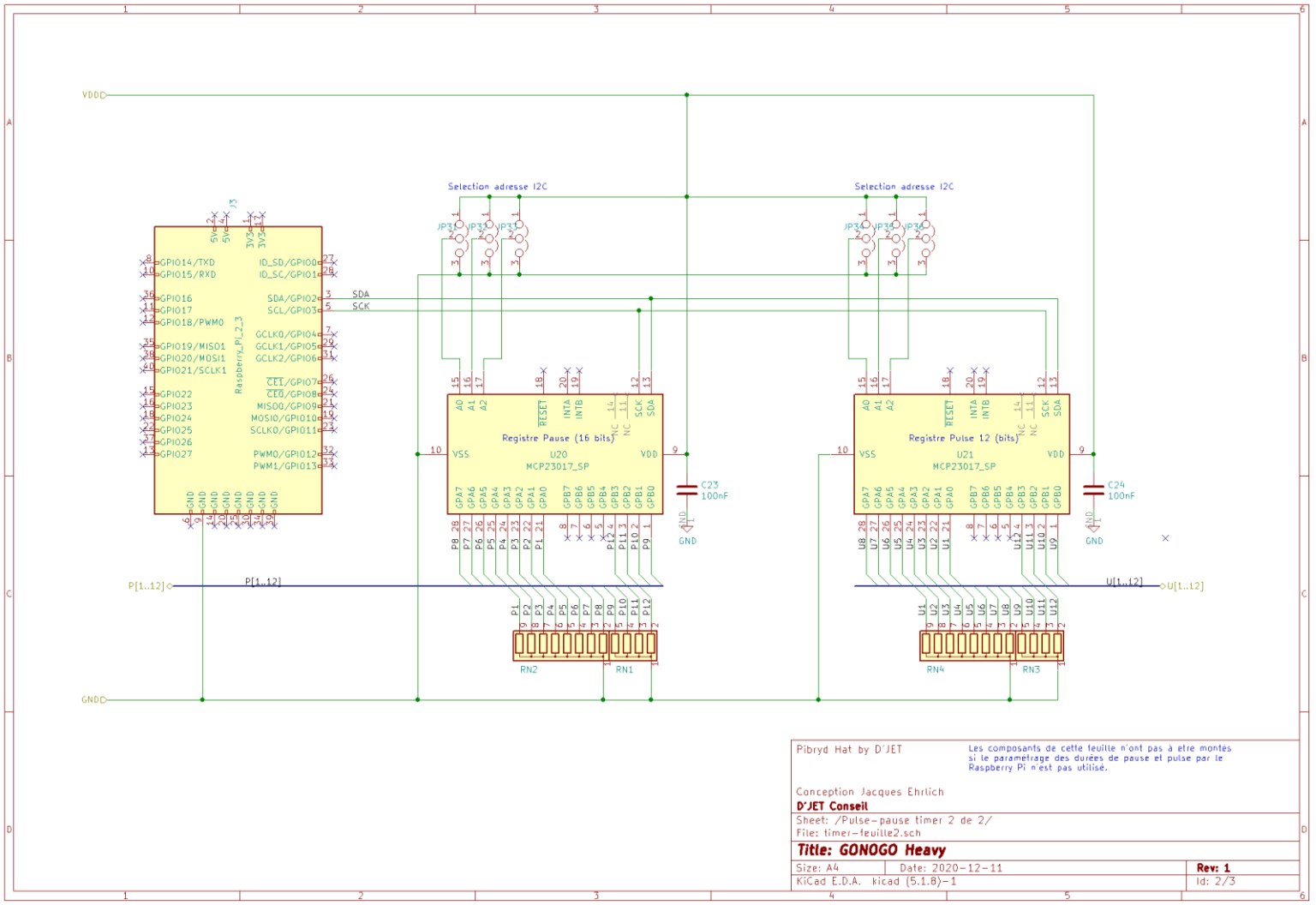


Figure 6 – Schéma de GONOGO – partie 2

D. *Réalisation*

Après un maquettage sur une plaque de prototypage *(breadboard)*, j'avoue avoir eu quelques difficultés à réaliser le circuit imprimé. Le placement proposé a été obtenu après trois tentatives infructueuses. Le résultat final satisfait les spécifications Pybride, à savoir la possibilité d'enficher la carte sur un Rpi, mais satisfait un peu moins son concepteur à cause de son facteur de forme très éloigné d'un carré et de ses dimensions imposantes.

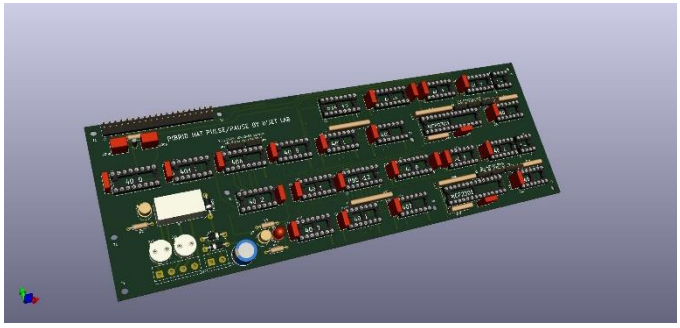


Figure 7 – Vue 3D de la carte GONOGO modèle Heavy

Concernant le câblage, là encore, aucune difficulté particulière puisque tous les composants sont traversants. Tous les schémas, fichiers de fabrication et code C sont disponibles sur GitHub [2].

VI. Le piege du 80/20

Avant d'être retraité, j'enseignais notamment l'analyse fonctionnelle et je tenais ce discours à mes étudiants de mastère :

« Ne laissez jamais des ingénieurs tout seuls pour établir les spécifications fonctionnelles d'un système mais associez les pour ce travail à des gens du marketing ou de l'analyse de la valeur. Sinon … gare au piège du 80/20 !»

En effet, les ingénieurs (au sens « techniciens ») sont des amoureux de la technologie. Lors de la définition d'un système ils ont une tendance naturelle à dire *« On pourrait ajouter ça … oh ! ce serait bien si on ajoutait ceci ou cela …»*

Finalement on aboutit à un système où 20 % des fonctionnalités vont intéresser 80% des clients et où 80 % des fonctionnalités restantes n'intéresseront que 20% de ces mêmes clients. Mais le pire, c'est que les fonctionnalités qui n'intéressent que 20% des clients sont tellement sophistiquées et complexes qu'elles demanderont 80% des efforts de développement.

Et comme j'ai l'habitude de jeter un œil critique sur mes travaux et partant du principe bien connu que le cordonnier est toujours le plus mal chaussé, je me suis sincèrement posé cette question : ne suis-je pas moi-même tombé dans le piège du 80/20 ? En effet, l'ajout de l'interface Rpi a considérablement complexifié la carte GONOGO. Et finalement, qui sera réellement intéressé par la possibilité de configurer les temps de GO et NOGO par un programme plutôt que par des switches. ? Je préfère laisser le lecteur trancher cette question.

VII. GONOGO Light

Cette réflexion m'a conduit à développer une version « *light* » de GONOGO. Certes celle-ci ne répond plus aux critères Pybride mais elle pourra satisfaire les lecteurs qui ne sont définitivement pas intéressés par l'interface Rpi. Le schéma (Figure 9 en annexe) s'en trouve fortement simplifié et ne nécessite pas davantage de commentaires dès l'instant où celui de la version « *Heavy* » a été compris. La conception du circuit ne m'a posé aucune difficulté et ne m'a demandé qu'une journée de travail avec à la clé une carte dont le facteur de forme est assez proche de 1.

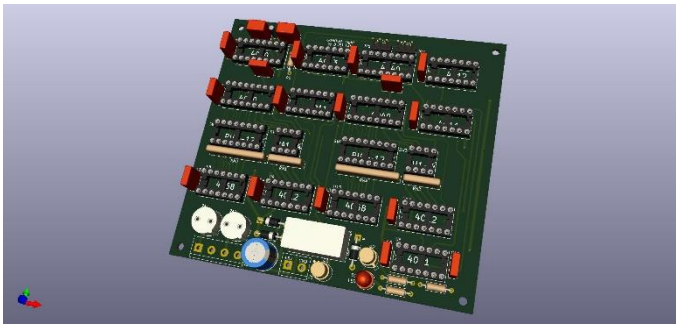


Figure 8 – Vue 3D de la carte GONOG Light

Tous les schémas et fichiers de fabrication sont disponibles sur GitHub [2].

Une première estimation des coûts révèle un rapport de 1,3 entre les versions *heavy* et *light*. A bon entendeur …

VIII. Alimentation des GONOGO

Les versions *heavy* et *light* utilisent exclusivement des circuits logiques CMOS. Tous ceux de la famille 40XX acceptent des tensions d'alimentation dans une plage de 3 à 15v. En revanche les circuits MCP23017 exigent une tension de 5v.

Par conséquent, la version *heavy* doit impérativement être alimentée en 5v sous peine de détruire les MCP23017.

En revanche, s'agissant de la version *light*, cette contrainte est levée à condition toutefois de remplacer le relais Finder 30.22 5v par un autre modèle, par exemple un Finder 30.22 12v si on opte pour une alimentation sous 12v.

A noter que toutes les cartes COBALT et GONOGO sont protégées contre les inversions de polarités par un ensemble constitué d'une diode ultra rapide (modèle MURXXX) et d'un fusible réarmable.

IX. Bilan et perspectives

Outre le plaisir et la satisfaction du développement en électronique ce travail m'a permis d'explorer le concept Pybride Hat et m'a confronté au redoutable piège du 80/20 me rappelant une fois de plus qu'il est impératif de se poser quelques bonnes questions avant de se lancer dans un projet : de quoi ai-je réellement besoin ? L'enjeu d'une fonctionnalité nouvelle en vaut-il la chandelle ?

Quoi qu'il en soit, le lecteur intéressé par GONOGO pourra faire son choix entre les versions *light* et *heavy*. Concernant cette dernière, pouvait-on faire plus simple et éviter les 24

portes OR qui contribuent à complexifier le schéma, le placement et le routage de la carte ? Mes vieux souvenirs de la logique DTL me poussent à explorer dans le futur, une solution à base de diodes qui pourrait peut-être simplifier les choses.

Mais seul un nouveau design faisant usage de composants montés en surface permettra d'obtenir un gain en surface significatif. Aussi, si la version *heavy* devait susciter chez le lecteur un intérêt suffisant, je me lancerai volontiers dans une nouvelle conception qui sera aussi pour moi l'occasion de me confronter enfin aux CMS que les articles récents parus dans Elektor ainsi que les nombreux tutos vidéo ont contribué à démystifier.

X. Recommandation et remerciements

Il me reste à recommander `sonelec-musique.com`[5], l'excellent site de Remy Mallard, véritable caverne d'Ali Baba qui renferme des trésors d'électronique analogique et numérique et dont je me suis inspiré pour la réalisation du projet GONOGO. Je lui adresse mes remerciements pour m'avoir autorisé à publier cet article qui inclut des parties de schéma dont il est l'auteur.

XI. References

[1] GitHub de la carte COABALT :
https:// https://github.com/duodiscus92/projet-cobalt
[2] GitHub de la carte GONOGO :
https://github.com/duodiscus92/projet-gonogo

[5] https://www.sonelec-musique.com/

XII. ANNEXE 1

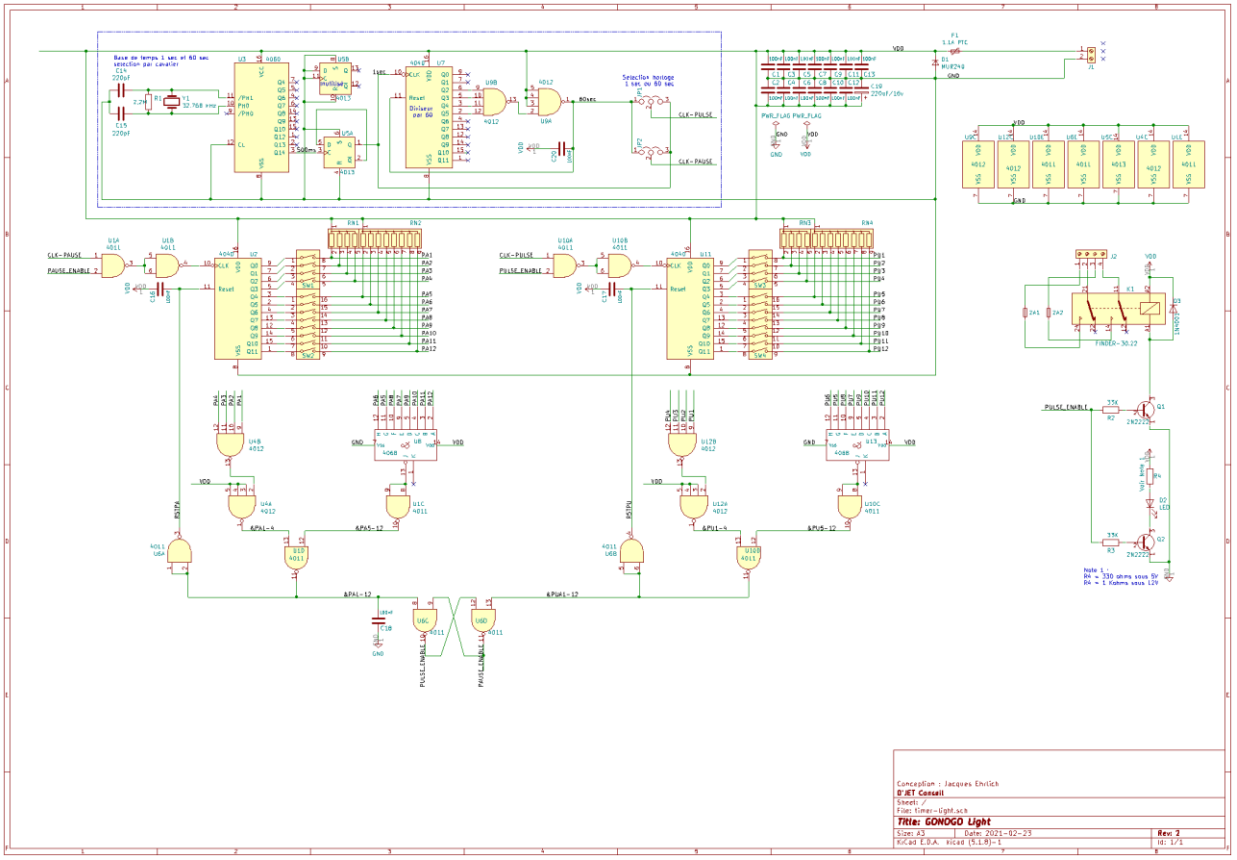


Figure 9 – Schéma du GONOGO – Light

XIII. ANNEXE II – LISTE DES COMPOSANTS DES CARTES GONOGO

A. *Version Light*

Référence	Quantité	Désignation
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. *Version heavy*

Référence	Quantité	Désignation
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. LISTE DES COMPOSANTS DE LA CARTE COBALT

Reference	Quantité	Désignation
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

Nota : les listes de composants peuvent facilement être générées dans Kicad avec les fichiers du projet disponibles sur GitHub.

XV. LE PROGRAMME GONOGO-SWITCH

Le programme est écrit en C et il délivre la position des switches en fonction de la durée souhaitée des phases go ou nogo.

Sur PC sous Cygwin ou sur RPi sous Pi OS (anciennement Raspbian) on devra compiler le programme source :

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

On peut ensuite lancer son exécution et taper la commande suivante pour connaître ses différentes options d'utilisation :

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

On découvre qu'il est possible de fournir la durée souhaitée sous différentes formes : en secondes, en minutes et secondes, en heures, minutes et secondes, en jours, heure et minutes etc.

Dans l'exemple ci-dessous on indique que la résolution du temps est la minute (`-r60`) et que l'on veut une durée de 1 jour, 10 heures et 35 minutes :

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

Et le programme répond :

```
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. UTILISATION DE LA CARTE GONOGO HEAVY ET DU MCP23017

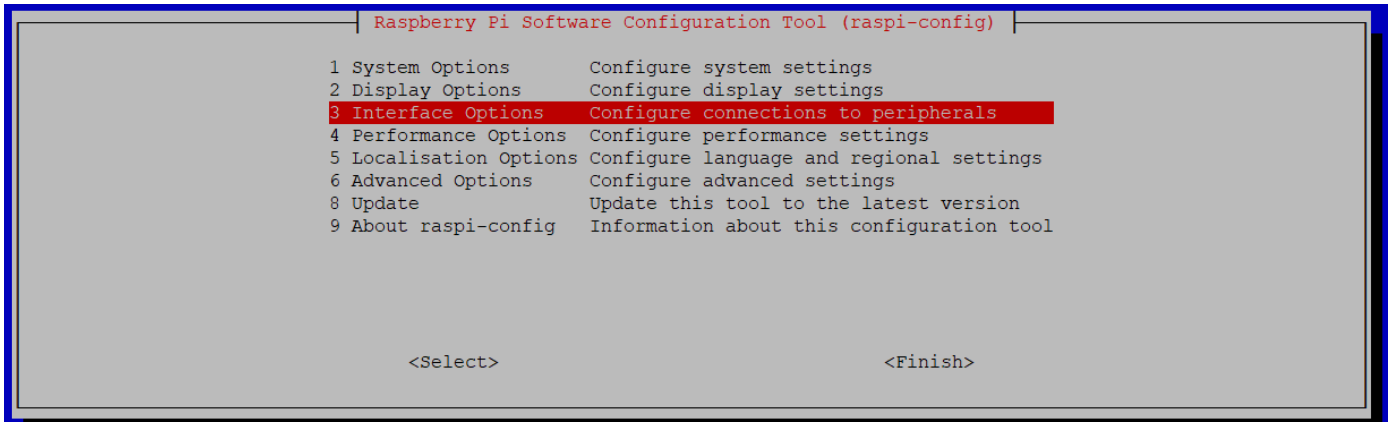
A l'aide des cavaliers, on fixe l'adresse des deux circuits MCP23017 sur la carte Gonogo *heavy* :

- Pour le circuit NOGO : A0=0, A1=0, A2=0
- Pour le circuit GO : A0=1, A1=0, A2=0

Il faut aussi mettre les 12 switches NOGO et les 12 switches GO en position ON.

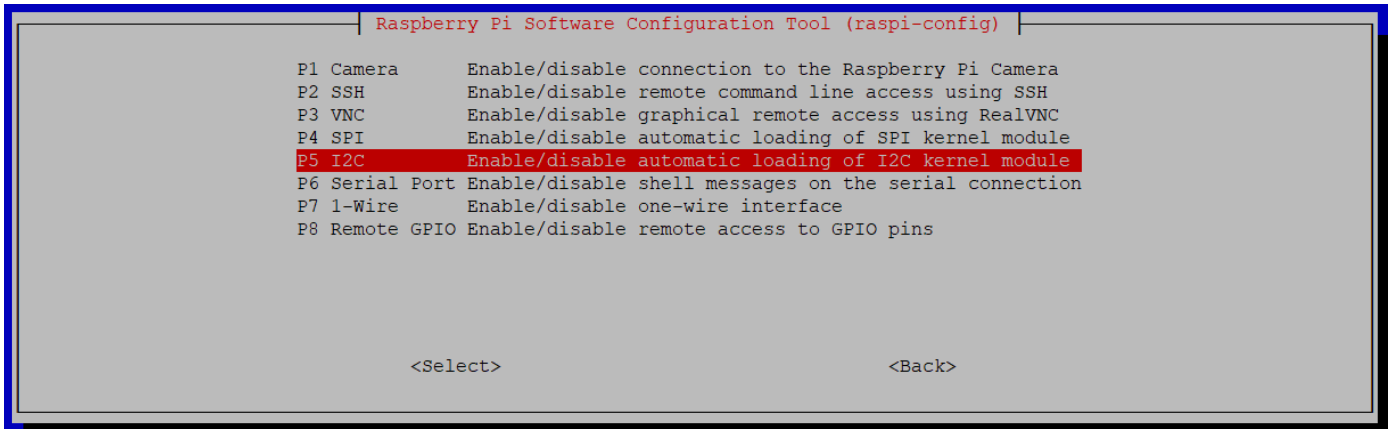
Après avoir enfiché la carte Gonogo sur le Rapsberry Pi on met le tout sous tension. On ouvre une session sur le Rpi par une connexion ssh par exemple et on s'assure que l'interface I2C est activée. Pour cela on lance :

```
$ sudo raspi-config
```



On sélectionne Interface Options

Puis on sélectionne I2C et Yes sur la fenêtre qui surgit ensuite.



Il faut ensuite vérifier que les deux circuits MCP23017 sont bien présents aux adresses 0x20 et 0x21.
Pour cela on tape la commande suivante, qui renvoie la liste des dispositifs I2C présents sur le bus I2C.

```
$ 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: -- -- -- -- -- -- -- --
```

A l'aide de la commande
`$ gcc mcp23017.c -o mcp23017 -lwiringPi`
on peut alors compiler le programme `mcp23017.c` (fichier source disponible sur GitHub) dans lequel l'essentiel se situe dans la fonction `gonogoSetup` dont on peut noter l'extrême simplicité.

```
#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;
}
```

Il suffit ensuite d'appeler le programme exécutable en lui fournissant comme argument, la durée des nogo et go en secondes, comme dans l'exemple ci-dessous où le NOGO est fixé à 30 secondes et le GO à 10 secondes.

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

Le programme mcp23017 est minimaliste : aucun contrôle n'est fait sur la valeur des arguments. Son seul but est de montrer la facilité d'utilisation du circuit MCP23017 dès lors qu'on utilise la librairie WiringPi (installée en standard dans Pi OS.) Chacun est libre de l'adapter à ses propres besoins.
