Conception d'un prototype : ***myTouch.gb V1.0***

IHM à touches électrostatiques

Cours de **Prototypage Industriel – 4EN1601**

myTouch.gb

Auteur :
Davide DI VENTI - Master 1 Electronique - Immatriculé 60456
60456@etu.he2b.be

Professeur :
M. G. LE VAILLANT

Département scolaire :
Haute Ecole Bruxelles-Brabant (*HE²B*)
Institut Supérieur Industriel de Bruxelles (*ISIB*)
Rue royale n°150, 1000 Bruxelles

Publié le 12 janvier 2024

Table des matières

Table des figures

Table des tableaux

1 Introduction

Ce rapport présente une étude réalisée dans le cadre du cours de Prototypage Industriel dispensé au premier quadrimestre de la première année du master en ingénierie industrielle, option électronique, à l'Institut Supérieur Industriel de Bruxelles (*ISIB*).

Le sujet abordé porte sur la conception d'un prototype en réponse à un cahier des charges établi par le professeur : la création d'un banc de test d'interfaces tactiles sur *PCB* monocouche à coût abordable.

Inspiré d'un article de recherche de *STMicroelectronics* [1] paru en 2022, cette étude vise à explorer et appliquer les principes du comportement électrostatique. Cette approche reste peu répandue, comme souligné dans l'article :

[1] "*In comparison with more established sensing techniques […], electrostatic sensors are relatively uncommon and less understood.*",

Cette étude et l'application de ce phénomène s'est avéré idéale à explorer dans le cadre de ce cours. De plus, le coût abordable de ces capteurs, mentionné dans l'article [1], répond pleinement aux exigences budgétaires du cahier des charges :

[1] "*[…] electrostatic sensors have clear advantages over other sensors, including cost-effectiveness and high sensitivity.*"

Le prototype développé dans ce cours se concentre donc sur l'acquisition de commandes tactiles électrostatiques sur *PCB*, avec une visualisation des données sur un écran.

Ce projet, baptisé "*myTouch.gb*", s'inspire de l'esprit d'une console *rétro-game*, avec une interface similaire à celle de la *Game Boy*. Il a été présenté lors du *Brotaru*, un événement à environnement "jeu vidéo" organisé par l'établissement scolaire le 4 décembre 2023.

2 Electrostatisme

2.1 Principes électrostatiques - Introduction

Afin d'introduire ce principe de manière adéquate, il est pertinent d'explorer le comportement électrostatique à travers une expérience courante. Le châssis métallique d'une voiture peut accumuler des charges électrostatiques lors de son déplacement, notamment par frottement avec l'air ambiant. Le contact avec le vent, les frottements et les interactions extérieures contribuent à cette accumulation de charges sur le châssis. Lorsqu'une personne entre en contact avec ce châssis chargé, la différence de potentiel entre la personne et la voiture crée une décharge électrostatique.

Le potentiel électrostatique d'un objet chargé est proportionnel à la quantité de charge qu'il porte. De plus, en fonction de sa position dans un champ électrique, un potentiel spécifique se crée, définissant ainsi le comportement électrostatique de l'objet.

Cependant, ce phénomène va au-delà du simple potentiel. En effet, la force électrostatique, qui découle des interactions entre charges électriques, est une composante essentielle. Cette force agit à distance et peut être attractive ou répulsive selon les charges en présence, déterminant ainsi les mouvements et les interactions entre objets chargés électriquement.

Par exemple, un ballon frotté contre un tissu génère une charge électrostatique. Approché ensuite d'un mur, ce ballon peut être attiré vers celui-ci, illustrant ainsi la manifestation de la force électrostatique entre le ballon chargé et la surface du mur.

2.2 Principes électrostatiques - Explications

Une charge électrique est une propriété fondamentale des particules élémentaires, telles que les électrons et les protons, qui composent la matière. Cette charge est une manifestation de l'interaction électromagnétique et peut être positive, négative ou neutre.

Les électrons portent une charge négative, tandis que les protons portent une charge positive. Les particules chargées positivement et négativement s'attirent, tandis que des particules de même charge se repoussent.

Lorsqu'un objet gagne ou perd des électrons, il acquiert une charge électrique. Par exemple, si un objet perd des électrons, il devient positivement chargé, et s'il en gagne, il devient négativement chargé. Les charges de même signe se repoussent, tandis que les charges de signes opposés s'attirent.

La mesure de la charge électrique se fait en quantité de charge électrique, autrement exprimé en coulombs [C], avec l'électron ayant une charge élémentaire de $-1.6 * 10^{-19}C$ et le proton une charge de $+1.6 * 10^{-19}C$. Les charges électriques sont à la base de nombreux phénomènes électriques, tels que les forces électrostatiques, les courants électriques et les interactions entre objets chargés.

2.3 Solution tactile électrostatique proposée

Dans le cadre du prototype conçut pour ce cours, des touches tactiles électrostatiques ont dut être dimensionnés. Il était important de prendre en compte le comportement électrostatique d'un doigt humain, et de trouver une solution compatible à tout humain. En effet, chaque personne peut avoir un potentiel électrique différent au niveau de la peau en raison de divers facteurs physiologiques et environnementaux :

- *Humidité* : L'humidité de la peau peut influencer son potentiel électrique. Une peau plus humide peut être plus conductrice et avoir un potentiel électrique légèrement différent d'une peau plus sèche.
- *Composition* : La composition chimique de la peau peut varier d'une personne à l'autre, ce qui peut affecter sa conductivité électrique et donc son potentiel électrique.
- *Niveau d'activité* : Le niveau d'activité physique, la transpiration et d'autres facteurs peuvent modifier temporairement la conductivité et le potentiel électrique de la peau.
- *Facteurs environnementaux* : Des facteurs tels que la température, l'humidité ambiante et même les matériaux avec lesquels une personne entre en contact peuvent influencer son potentiel électrique.

Une solution a été expérimentée en laboratoire et a été appliqué sur le projet pour annuler ces comportements :

En polarisant le doigt à une tension connue (+5V), une référence de potentiel commune pour les utilisateurs est établie. Cela permet d'assurer une certaine uniformité dans les mesures ou les interactions électriques avec le dispositif, en éliminant ou en minimisant l'impact des variations individuelles de potentiel.

Cette méthode permet de normaliser le potentiel électrique du doigt à une valeur connue, ce qui facilite la mesure ou l'interaction du système électronique, en offrant une base de référence constante et prévisible pour tout utilisateur.

Une touche tactile électrostatique requiert donc une certaine architecture pour embarquer l'électrode de polarisation +5V et l'électrode de réception.

2.4 Décharge de l'électricité statique

À l'appui, l'électrode de réception reçoit une quantité de charge équivalente à la polarisation. Une fois l'appui relaché, une décharge lente progressive a lieu sur l'électrode de réception si le circuit de l'architecture est ouvert, ou est à impédence élevée.

Cette décharge peut etre dérangeante si un comportement plus réactif est souhaitée, comme la détection de fronts déscendants, d'appuis longs/courts, etc. Il est donc primodial de placer une résistance *pull-down* à valeur suffisament élevée au niveau de l'électrode de réception pour accélerer la décharge tout en évitant un impact sur l'électronique embarquée, et ainsi tendre vers une pente de 1 au front déscandant.

3 Architecture électronique requise

Pour mettre en œuvre la solution de polarisation d'un doigt, des architectures à deux électrodes sur *PCB* ont été envisagées afin de recevoir un doigt de manière appropriée.

3.1 Polarisation par électrode

En fonction de l'architecture utilisée, la sensibilité varie, permettant de détecter la présence d'un doigt de plusieurs approches. Plusieurs expérimentations ont été menées au laboratoire. Pour obtenir le comportement souhaité, l'organisation méthodique des pistes sur le *PCB* et la tension appliquée aux électrodes sont des éléments à ajuster. Voici quelques exemples :

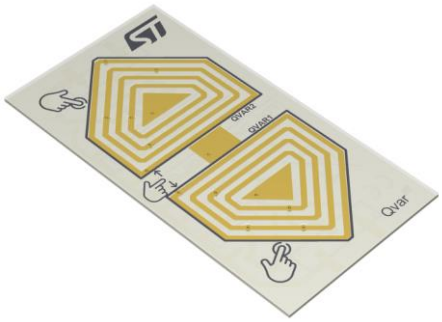


Figure 1: Architecture swipe de ST

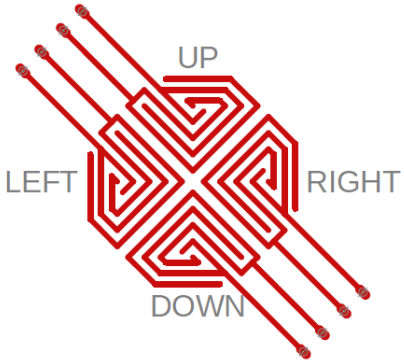


Figure 2 : Architecture joystick V1

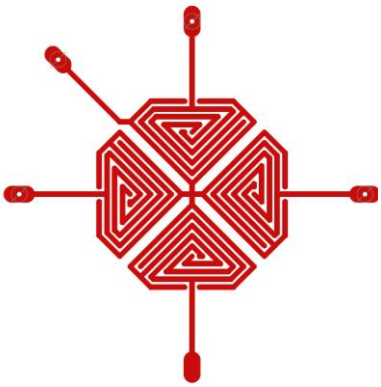


Figure 3 : Architecture joystick V2

Des expériences ont déjà été étudiées par *STMicroelectronics* en 2022, comme explicité dans leur article de recherche [1]. La figure 1 présente 2 touches tactiles électrostatiques distinctes, permettant une détection du balayage du doigt. Il s'agit d'un exemple de *ST*. Cependant, étonnamment, ils n'ont ni mentionné ni supposé un détail sur lequel le prototype réalisé dans le cadre de ce cours est basé : une détection analogique de la pression exercée sur une touche tactile électrostatique (expliqué prochainement). Leur solution se limite à du tout-ou-rien :

[1] *"Qvar can be used as a sensitive touch interface by connecting a simple electrode to the sensor in order to detect a touch, a press, or even a swipe."*

L'architecture requiert deux électrodes pour former une seule touche : l'une pour la polarisation du doigt, et l'autre servant d'électrode de réception, également appelée *Qvar* dans l'article.

La figure 2 présente quatre touches électrostatiques distinctes, issues des premières expérimentations menées au laboratoire dans le cadre du cours de Prototypage Industriel. Cette conception s'inspire de l'architecture proposée par *ST* et a été réalisée sur une carte de circuit imprimé monocouche à l'aide d'une découpeuse laser. Les pistes cuivrées sont indiquées en rouge, tandis que les zones blanches correspondent aux gravures, créant un isolant entre les électrodes. L'objectif initial était de concevoir un *joystick* sans mécanique. Bien que les quatre directions soient présentes, la gestion linéaire d'une direction ne s'applique pas simplement en balayant le doigt vers le haut, mais en exerçant une force plus ou moins forte sur la touche. Au laboratoire, ce principe a fonctionné conformément aux attentes, mais une amélioration a été envisagée, matérialisée dans la figure 3.

L'amélioration visait à réduire de moitié la taille du *joystick* pour le rendre moins encombrant dans un éventuel système embarqué. La section des électrodes a été modifiée, passant de 1,27 mm (50 mil) à 0,635 mm (25 mil), et la largeur du diélectrique, de 1,27 mm (50 mil) à 0,3 mm (12,5 mil). Cependant, cette modification a abouti à un effet non désiré : le *joystick* miniaturisé a commencé à détecter la présence d'un doigt avant même qu'une quelconque pression ne soit exercée. Il s'agissait donc davantage d'un capteur de distance que du *joystick* prévu.

La sensibilité d'une touche tactile n'est pas seulement limitée au toucher direct, mais peut également s'étendre à l'environnement extérieur, à travers l'air. En modifiant les facteurs diélectriques et la surface cuivrée, il est possible de dimensionner un capteur tactile ou de distance. De plus, en augmentant le potentiel de l'électrode de polarisation, son champ électrostatique peut influencer la portée de détection.

Étant en dehors du champ d'application du cahier des charges, aucune autre expérience concernant la détection de distance n'a été poursuivie. En revanche, l'étude de la capture de la pression exercée sur une touche a été approfondie.

Ce comportement analogique peut être expliqué par l'hypothèse illustrée dans la figure 4 ci-dessous. Elle met en lumière le phénomène de polarisation d'un doigt en présence de charges électriques lorsqu'il entre en contact avec une architecture similaire à celles mentionnées précédemment. Dans cette configuration, une électrode sur deux est polarisée à +5V, tandis que l'autre joue le rôle d'électrode de réception (*Qvar*).

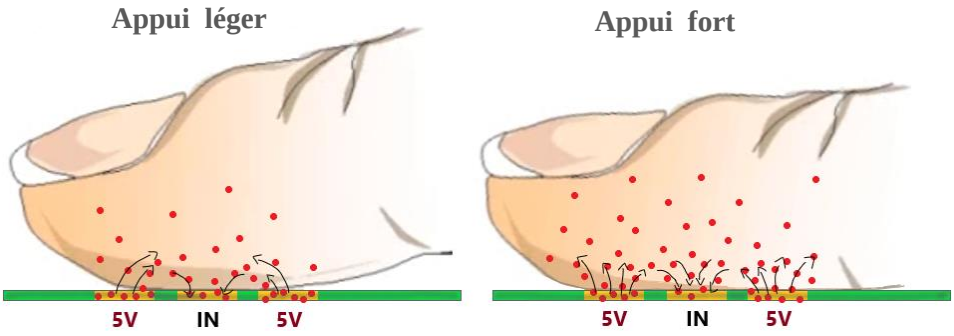


Figure 4 : Polarisation et lecture des charges électrique d'un doigt sur PCB

L'architecture électronique du bouton tactile polarise le doigt via les électrodes à 5V et reçoit en retour (électrode de réception : *IN*) des charges électriques proportionnelles au volume massique du doigt en contact avec la surface. Un appui plus fort entraîne une accumulation accrue de charges électriques dans le doigt en contact avec la surface, ce qui se traduit par une augmentation du potentiel de réception (*IN* → 5V).

La compression du doigt augmente la pression à l'intérieur de celui-ci, ce qui signifie qu'il y a une plus grande quantité de matière dans un même volume. Cette augmentation de matière organique favorise la conduction électrique.

Pour une explication plus approfondie de ce phénomène de capteur analogique, les circuits électroniques associés sont détaillés dans la section suivante.

3.2 Amplification et filtrage

Les étages d'amplification et de filtrage ont été dimensionnés par des essais et des itérations jusqu'à ce que le résultat final soit le plus optimal possible pour une première version gravée sur *PCB* entièrement fonctionnelle.

Le premier circuit développé lors des premières expérimentations, représenté à la figure 5, était un montage d'amplification à alimentation symétrique sans filtrage. Quant au dernier, actuellement présent dans le prototype et illustré à la figure 6, il a été dimensionné pour être intégré dans un *PCB* monocouche, sans pontage, à tension de 0V à +5V, avec en plus un filtre passe-bas. Un filtre actif aurait engendré un pontage.

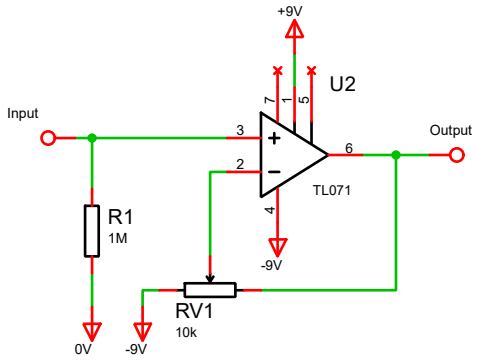


Figure 5 : Premier circuit d'amplification

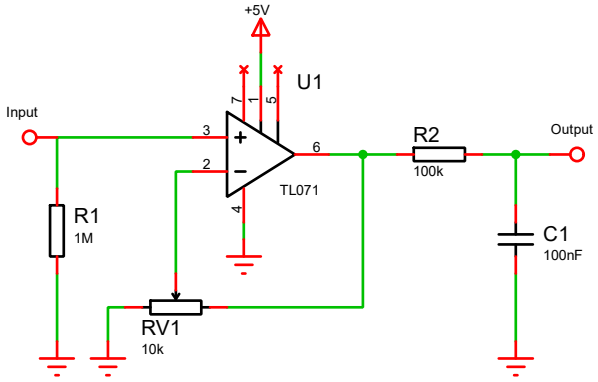


Figure 6 : Dernier Circuit d'amplification avec filtre

Le premier montage (figure 5) a permis d'obtenir des éclaircissements sur différents comportements :

1. Observation du gain nécessaire pour atteindre une tension logique en sortie de +VCC et -VCC, correspondant à l'alimentation de l'*AOP* (+9V & -9V). Cela a fourni un aperçu des résistances ajustant le gain à l'aide de *RV1*.
2. Évaluation de la sensibilité de l'entrée en présence de parasites externes, notamment du spectre à 50Hz du réseau électrique du bâtiment agissant comme champ électromagnétique. Avec une valeur fixée de *R1* à 10MΩ, les sinusoïdes à 50Hz étaient captées. En réduisant cette valeur à 1MΩ, les sinusoïdes ont été négligeables dans le circuit.
3. Conception de *R1* en tenant compte de l'impédance moyenne d'un doigt humain, située entre 10k et 100kΩ selon la personne. Afin d'éviter la création d'un diviseur de tension entre le doigt et *R1*, il était nécessaire de ne pas trop diminuer la valeur de *R1*, garantissant une lecture correcte de l'entrée, uniforme pour tous les utilisateurs.
4. Observation des parasites entrant dans l'entrée. Lors de l'appui et du relâchement, des rebonds et des distorsions des crêtes se produisaient. En analysant la fréquence moyenne de ces perturbations, un filtre passe-bas a été dimensionné avec une fréquence de coupure à 15Hz.

Toutes ces observations ont été mises en évidence et appliquées au circuit final (figure 6) de la touche tactile électrostatique. Après expérimentation et calibrage de tous ces paramètres, une observation remarquable a été faite : le phénomène de captation de pression du doigt.

En calibrant le gain à environ 5, l'amplification effectuée confère une sensibilité élevée au doigt humain, rendant les tapotements sur les capteurs très visibles. À un gain calibré autour de 1, quasi équivalent à un circuit suiveur, la sensibilité est moindre, offrant une perception analogique de la pression du doigt exercée sur le *PCB* plutôt efficace.

En d'autres termes, le côté de l'électrode réceptrice (*input*) présente une grande impédance, facilitant la transmission de charges électriques entre les deux électrodes à faible courant. Ce montage permet au doigt d'atteindre facilement un potentiel proche de +5V une fois polarisé, assurant une lecture efficace tout en ayant une adaptation d'impédance maîtrisée.

3.3 Lecture ADC

Une lecture analogique peut être traitée par un circuit intégré convertisseur analogique-numérique (*ADC*) ou directement par un microcontrôleur, s'il le permet. Dans les deux cas, suivant l'échantillonnage, la fréquence de traitement *ADC* effectué et la bande passante allouée au traitement, le résultat est le même. Cependant, un microcontrôleur offre davantage de fonctionnalités et permet de centraliser différentes interfaces externes, qu'elles soient analogiques ou numériques.

Le choix s'est porté sur un microcontrôleur pour offrir une personnalisation plus avancée dans l'acquisition des données. Cette lecture se fait à la sortie du circuit d'amplification et de filtrage, à la broche output représentée dans la figure 6.

Deux options étaient envisagées : un microcontrôleur *Arduino* ou un *STM32F103C8T6*. Initialement, l'utilisation d'un *Arduino* a été proposée pour faciliter la programmation et consacrer davantage de temps à l'optimisation de l'architecture d'une touche tactile. L'*Arduino* offre un échantillonnage des entrées analogiques sur 8 bits sous 5V, tandis que le *STM32* le fait sur 10 bits sous 3V3. Étant donné que le projet porte principalement sur le traitement des signaux analogiques, une comparaison quantitative basée sur la résolution de chacun des microcontrôleurs s'est avérée nécessaire.

La précision par division (ΔV) peut être calculée comme suit :

$$\Delta V_{Arduino} = \frac{5V}{2^8} \simeq 0.0195V \tag{3.1}$$

$$\Delta V_{STM32} = \frac{3.3V}{2^{10}} \simeq 0.0032V \tag{3.2}$$

Le rapport *K* présent dans l'équation 3.3 suivante démontre que le *STM32* est 6 fois plus précis que l'*Arduino* :

$$K = \frac{\Delta V_{Arduino}}{\Delta V_{STM32}} \simeq 6.056 \tag{3.3}$$

Cependant, un autre problème se présente : les *AOP* disponibles au laboratoire ne sont pas dimensionnés pour être alimentés en *VCC* à 3V3, le minimum requis étant de 5V pour un fonctionnement correct. Mais appliquer une tension supérieure à la tension logique du microcontrôleur comporte un risque élevé d'endommagement prématuré de celui-ci.

Pour éviter l'ajout d'un pont diviseur de tension, équipé de deux résistances en sortie de l'*AOP*, il est possible de limiter la tension en agissant sur le gain. Cette approche influence la sensibilité de la touche mais protège le microcontrôleur. Un compromis adéquat peut être trouvé.

La raison pour laquelle un pont diviseur de tension n'a pas été envisagé est que le prototype final doit se composer du strict minimum de composants tout en offrant un maximum d'interactions. Le cahier des charges stipule la nécessité d'atteindre, si possible, un coût final le plus bas possible. L'optimisation a été prise en compte dès le début du processus.

Un autre problème survient lors de la lecture. Les *AOP* utilisés (*TL071*) génèrent un bruit parasite, créant un offset équivalent à 1.2V à l'état bas. La correction des offsets n'a pas été envisageable car le prototype à concevoir utilise des composants *quad*. Plus précisément, il est expliqué plus loin, mais 9 entrées analogiques doivent être gérées en commun pour le prototype à concevoir. L'option la plus compacte et la moins coûteuse a été d'utiliser deux composants quad et un composant mono. Les encapsulages quad restreignent l'utilisation des broches de réglage des *offsets*.

Malgré toutes ces modifications, le *STM32* demeure plus précis et a donc été choisi comme microcontrôleur pour la réalisation du prototype.

4 Présentation du prototype

Tous les systèmes précédemment expliqués, fonctionnels sur *breadboard*, ont ouvert la voie à l'étape suivante : la conception d'un *PCB* qui centraliserait tous ces éléments. Le prototype est donc présenté à partir de cette section. S'inspirant du croquis présenté à la figure 7, le prototype devait reprendre le design d'une console de jeu portable.

4.1 Croquis

Dessiné à la mi-novembre 2023, le prototype regrouperait toutes les idées discutées avec le professeur sous forme d'un rendu final sur un *PCB* monocouche, interprété de manière justifiée :

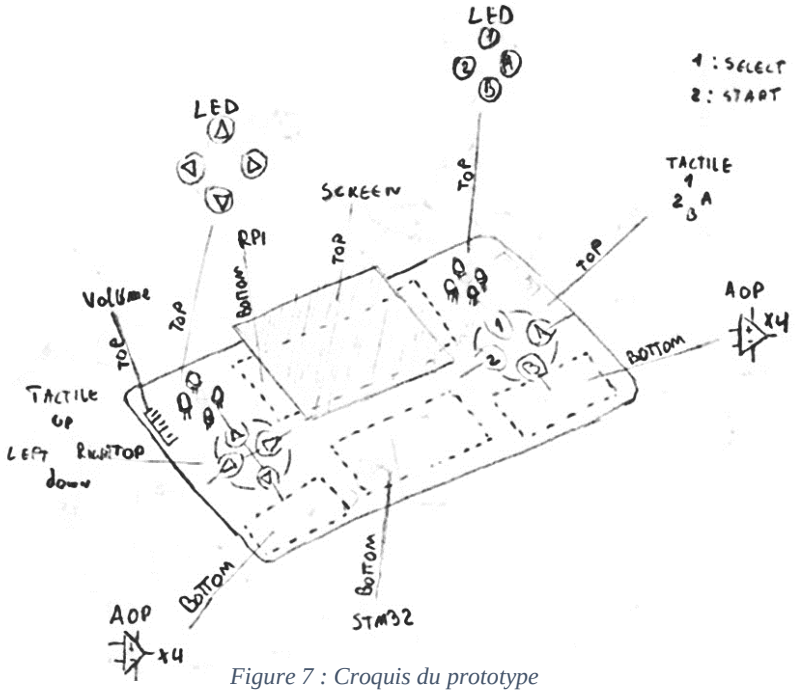


Figure 7 : Croquis du prototype

Chaque emplacement sur le *PCB* a été organisé de manière intuitive et dédié à des circuits électroniques préalablement validés. Les emplacements initiaux étaient les suivants :

- 9 touches tactiles électrostatiques
- 9 étages d'amplifications et de filtrages
- Le microcontrôleur *STM32F103C8T6*
- Le microcontrôleur *Raspberry Pi Pico*
- 8 *LEDs*
- Un écran *TFT ili9225 2.2"*

L'objectif principal des *LEDs* était de fournir un retour visuel lorsqu'une touche tactile était pressée, étant donné l'absence de sensation mécanique. L'idée était d'attribuer une *LED* à chaque touche. Cependant, après réflexion, il est apparu que les *LEDs* étaient superflues, car l'écran pouvait offrir ce type de retour d'information.

Le microcontrôleur *Raspberry Pi Pico* n'a pas encore été mentionné dans ce dossier. Son objectif consistera à se concentrer sur l'affichage de l'écran. Cette tâche exige toute l'attention du contrôleur pour garantir une interface sans latence. Une défaillance de ce type perturberait l'utilisateur et compromettrait l'expérience d'utilisation du produit. De plus, l'utilisation d'un microcontrôleur plutôt qu'un *SBC*, souvent appelé micro-ordinateur, présente un avantage en termes de consommation. La puissance de calcul sera dédiée exclusivement à la gestion de l'écran, sans avoir à supporter un système d'exploitation Linux supplémentaire.

4.2 Résultat final

Depuis le croquis initial, des ajouts ont été faits, notamment l'intégration d'une carte son *I²S* et d'un haut-parleur. Les dimensions du *PCB* sont presque équivalentes à celles d'un smartphone : 78 mm x 162 mm. Comme le montre la figure 8, les *LEDs* ne sont également pas été installées :

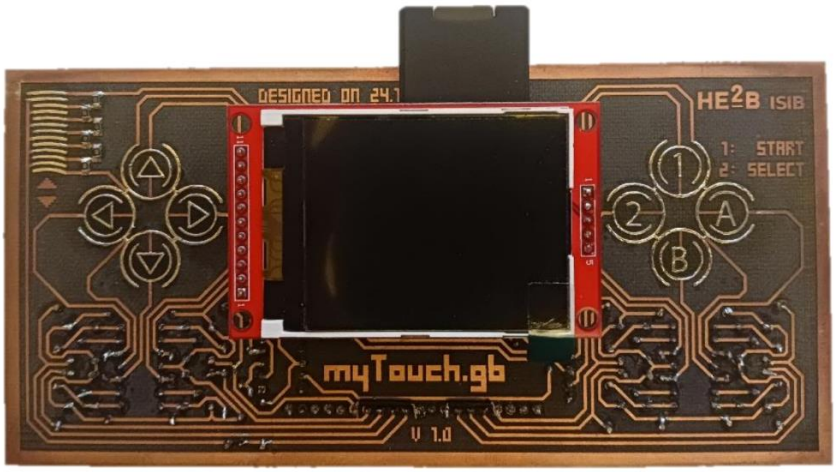


Figure 8 : Prototype myTouch.gb V 1.0

Le *PCB* a été conçu sur *Fusion360*. Le processus de conception sur le logiciel a duré environ 5 heures. Le routage a été réalisé manuellement, y compris pour les touches tactiles. En fin de conception, des inscriptions ont été ajoutées pour embellir la surface et la rendre plus attrayante qu'un simple *PCB*.

Les touches tactiles ont été étamée à l'étain, mettant en relief à la fois le symbole et l'architecture tout en maintenant les pistes en retrait. Cette approche vise à permettre à l'utilisateur d'appuyer de manière intuitive.

La version initiale montée sur *breadboard* correspondait à la *V0.1*. Le montage a ensuite évolué sous forme de *PCB*, devenant ainsi une *V1.0*. Cependant, quelques ajustements ont été apportés au *PCB* suite au débogage. Une version *V1.1* n'a pas encore été créée mais pourrait être envisagée pour intégrer ces corrections. Ces corrections sont citées et expliquées plus loin dans la section *5.2 Circuit électronique*.

4.3 Touches électrostatiques proposées

La conception des boutons a été repensée pour inclure des symboles, rendant ainsi les touches plus intuitives en tant que touches tactiles, sans nécessiter d'explications supplémentaires. En plus de l'aspect tout-ou-rien typique d'une *Game Boy*, comme déjà évoqués, les touches disponibles sur le prototype offrent des réponses analogiques supplémentaires. Voici ces différences :

Figure 9 : Architecture Slider

Figure 10 : Architechure Directionnelle

Figure 11 : Architecture Sélective

Les touches directionnelles (figure 10) et sélectives (figure 11) sont capables de détecter la pression exercée par un doigt humain. En revanche, le *SLIDER* (figure 9) permet de détecter la position d'un doigt le long de son axe vertical. L'idée du *SLIDER* est différente, des résistances SMD placées en série divisent graduellement la tension de polarisation le long de l'axe. Ainsi, le doigt est polarisé respectivement à sa position.

4.4 Démonstration

L'interface affichée sur l'écran explore toutes les fonctionnalités analogiques et tout-ou-rien proposées par le prototype en reprenant le thème *Brotaru*. Sous forme de jeu vidéo, l'interface joue le rôle d'un tableau de bord ludique. Les figures ci-dessous montrent le design réalisé sur l'écran. En haut à gauche et à droite de l'interface se trouvent les commandes interagissant avec le scénario.

Il est possible de déplacer le personnage (un *slime*) vers la droite et la gauche, de sauter avec le *SLIDER*, de changer la couleur du personnage ou d'accéder au menu des jeux *Game Boy*.

Voici une première démonstration (déplacement) :

Dans la figure 12, en appuyant sur la touche *RIGHT*, le personnage se déplace vers la droite. En fonction de la pression exercée sur la touche, le personnage ira plus ou moins vite. Dans le cas de cet exemple, il est à 100% de la vitesse maximale. Au passage, étant donné qu'il s'agit d'un *slime*, de la bave est traînée au sol à chaque déplacement. Cette bave, ainsi que le *slime*, peut changer de couleurs suivant la couleur choisie en appuyant sur *A*.

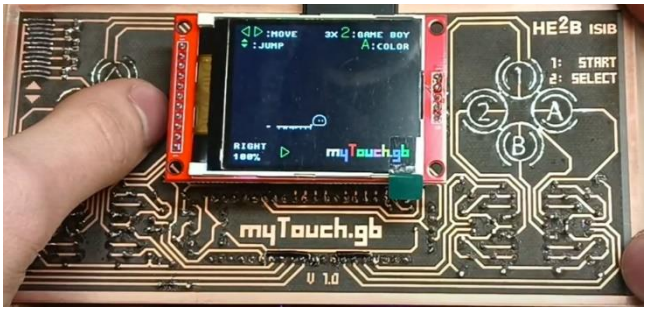


Figure 12 : Démonstration du déplacement

Voici une deuxième démonstration (saut) :

Figure 13 : Démonstration 1 du saut

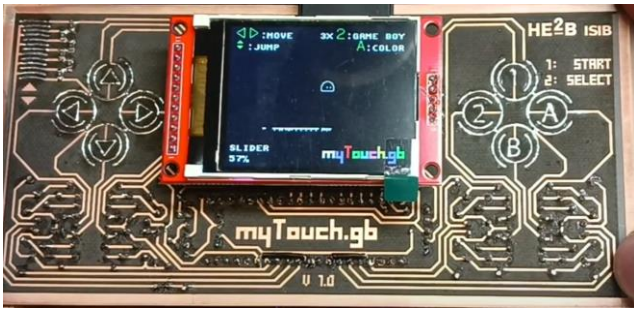


Figure 14 : Démonstration 2 du saut

Dans la figure 13, l'utilisateur écrase plus ou moins fort le *slime* avec le *SLIDER*. Dans le cas de cet exemple, il est écrasé à 76%. En relâchant, comme dans la figure 14, le *slime* saute plus ou moins haut en fonction du pourcentage exercé avant le front descendant émis au *SLIDER*. En plein air, il y a également la possibilité de déplacer le personnage (*RIGHT* ou *LEFT*). Le saut suit une courbe parabolique. En termes de vitesse, il y a décélération en montée puis accélération à la chute.

Voici une dernière démonstration (émulation Game Boy) :

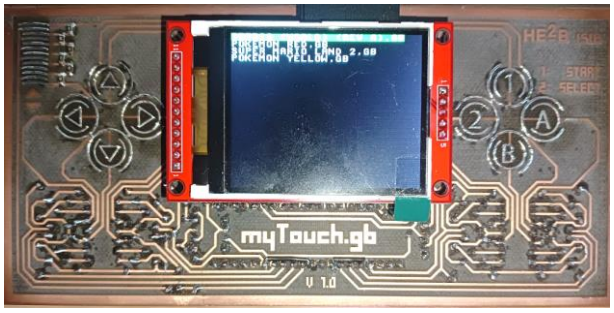


Figure 15 : Démonstration 1 de l'émulateur

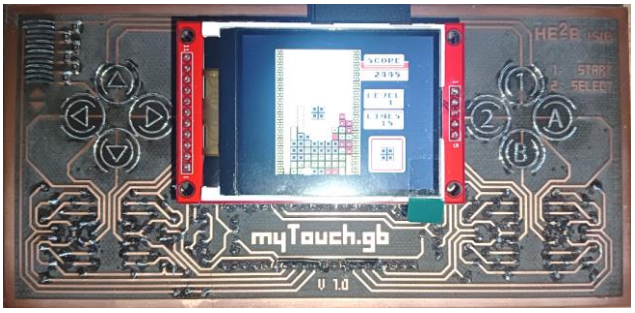


Figure 16 : Démonstration 2 de l'émulateur

En effectuant trois pressions successives sur la touche *2* (*SELECT*), une nouvelle page se dévoile. Il s'agit du menu où la sélection des jeux vidéo téléchargés dans la carte *SD* s'opère (voir figure 15). Le défilement des jeux s'effectue de haut en bas, et l'entrée dans un jeu particulier se réalise en appuyant sur la touche *A*. Une fois plongé dans le jeu, la *ROM* ne prendra en compte que l'aspect tout-ou-rien des touches tactiles (voir figure 16).

5 Développement des systèmes embarqués

Les systèmes embarqués dans le prototype ne sont pas nombreux. La complexité d'un produit se révèle à travers le nombre de composants qu'il intègre. Les détails concernant ces composants seront exposés plus loin dans cette section. Cependant, afin de ne pas maintenir le suspense, il est à noter que le prototype est constitué de 68 composants distincts.

La complexité du produit peut être évaluée à l'aide de l'équation 5.1 :

$$niveau = \frac{\log(composants)}{\log(7)} \tag{5.1}$$

$$niveau = \frac{\log(68)}{\log(7)} = 2.17 \tag{5.2}$$

Ce résultat était anticipé pour un projet réalisé sur une période de 4 mois par un étudiant. Plus le résultat est élevé, plus le produit est complexe. Il est important de noter que l'échelle n'est pas linéaire, mais suit une courbe logarithmique. À titre d'exemple, en reprenant les données du cours [4], un tournevis atteint une complexité de niveau 1 (3 composants), une photocopieuse est classée au niveau 4 (2 000 composants), et, pour conclure, un avion se situe en tête avec un niveau 6 (100 000 composants).

5.1 Liste de matériel

Ci-dessous se trouve la table 1, représentant la liste de matériel nécessaire à la conception du prototype. Les désignations des composants suivent les schémas électroniques du *PCB* présentés aux annexes 1 et 2.

Composants	**Notes**	**Quantité**
Raspberry Pi Pico		1
STM32F103C8T6		1
ILI9225	écran et socle SD	1
Carte SD		1
MAX98357A	carte son	1
Haut-parleur 8Ω 2W		1
PCB monocouche 78 mm x 162 mm		1
Headers femelles DIP 20 pins	pour microcontrôleurs	4
Headers femelles DIP 5 pins	pour USB to TTL CP2102 & SPI de la SD	2
Headers femelles DIP 11 pins	pour écran SPI	1
DIP IC sockets 2x4 pins	pour mono AOP	1
DIP IC sockets 2x8 pins	pour quad AOP	2
TL074	quad AOP	2
TL071	mono AOP	1
Condensateur électrolytique 10µF	C11, C12 (découplage)	2
Condensateur céramique 100nF	C10, C13, C14 (découplage) & C1 à C9 (filtrage)	12
Résistance SMD 0805 1kΩ	R19 à R24 (diviseurs de tension slider)	6
Résistance DIP 3.3kΩ	$R_{\text{ajoutée après correction}}$ (diviseur de tension slider)	1
Résistance DIP 1M Ω	R1, R3, R5, R7, R9, R11, R13, R15, R17 (pull down)	9
Résistance DIP 100kΩ	R2, R4, R6, R8, R10, R12, R14, R16, R18 (filtrage)	9
Trimpot 20kΩ	correction des gains	9
TOTAL	excepté fils de pontages	68

Tableau 1 : Liste de matériel

5.2 Circuit électronique

L'électronique expliquée à la section *3.2 Amplification et filtrage* est celle utilisée dans le prototype, sans aucune différence. Aux annexes 1 & 2 se trouve l'entièreté du circuit, avec microcontrôleurs, écran, etc.

Après que le *PCB* est sorti de la découpeuse laser, des premiers tests ont été effectués, mais quelques problèmes empêchaient un bon fonctionnement. Quelques corrections ont dû être faites sur le *PCB*. Dans la figure 17 se trouve le circuit électronique tiré sur *PCB*. Les croix rouges représentent des incisions faites au cutter sur des pistes cuivrées. Les autres couleurs sont les pontages effectués d'une zone cuivrée à une autre.

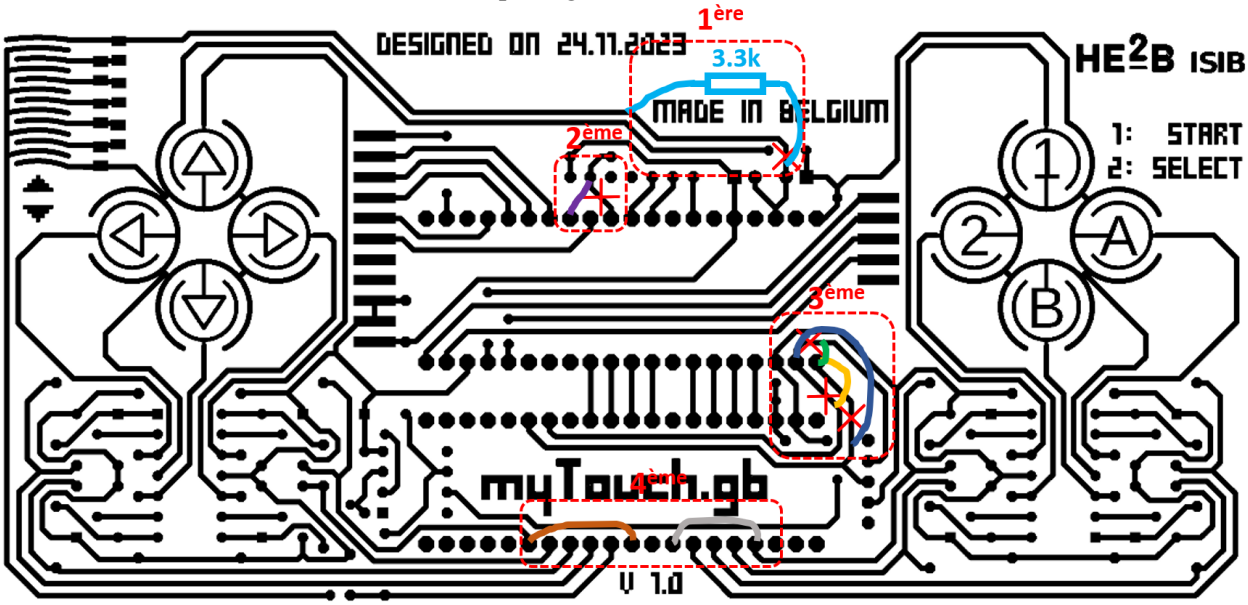


Figure 17 : Circuit électronique sur PCB avec corrections

Quatre erreurs ont été corrigés. La première erreur engendrait une mauvaise lecture sur le *SLIDER*. L'ajout d'une résistance de 3.3 kΩ en amont et en série des 6 résistances de 1 kΩ SMD (R19 à R24) a corrigé le problème. Initialement, lorsque le doigt est tout en haut du *SLIDER*, il est polarisé à 0V. En descendant, la polarisation imposée augmente graduellement jusqu'à 5V. Cependant, avec les offsets générés par les *AOP* (1.2V), toute la zone du dessus du *SLIDER* était aveugle pour le STM32. L'ajout de la résistance de 3.3 kΩ a permis de fixer la polarisation la plus petite (zone du dessus du *SLIDER*) à 1.8V à la place de 0V.

La deuxième erreur a été une erreur de distraction lors de la conception. Sans correction, le *Raspberry Pi Pico* effectuait un *reboot* en continu. La broche *GND* de la carte son était censée être connectée au *GND* du *Raspberry Pi Pico*, mais à la place, elle était connectée sur la broche *RUN* du microcontrôleur.

La troisième erreur a également été une erreur de distraction lors de la conception. Cela concerne le bus *UART* se composant d'un *TX* et *RX* respectivement à chaque microcontrôleur. Les connexions étaient inversées (*TX* vers *TX* ; *RX* vers *RX*), provoquant une communication impossible. L'inversion des branchements a résolu le problème (*RX* vers *TX* ; *TX* vers *RX*).

La dernière erreur a été le résultat d'une confusion dans l'attribution des entrées analogiques dans le logiciel *STM32CubeIDE*. Dans l'onglet "*Pinout & Configuration*", une image interactive de la puce *STM32* est présente, permettant entre autres d'assigner directement une broche à une interface telle que *GPIO_INPUT*, *GPIO_OUTPUT*, *GPIO_ANALOG*, etc. Une interface *GPIO_ANALOG* est proposée dans le menu déroulant de la quasi-totalité des broches. Des broches ont été choisies inconsciemment pour le circuit électronique analogique. Cependant, les convertisseurs *ADC* (*ADC1* & *ADC2*) du *STM32* sont liés à seulement 10 broches parmi les 32 disponibles. Parmi les 9 broches sélectionnées, 2 d'entre elles n'ont pas été reconnues par les convertisseurs *ADC*. En ce qui concerne les 7 autres, 4 avaient déjà été programmées et étaient donc correctes, tandis que les 3 restantes étaient heureusement également correctes malgré leur choix inconscient. En résumé, sur le PCB, des pontages ont donc été réalisés pour migrer les 2 broches non analogiques (*PB10* (*UP*) et *PC15* (*START*)) vers 2 autres réellement analogiques (*PA5* et *PA3*).

5.3 Microcontrôleurs & programmes

Le prototype utilise deux microcontrôleurs, à savoir un *Raspberry Pi Pico* et un *STM32F103C8T6*. Cette configuration pourrait prêter à confusion, car un seul microcontrôleur aurait pu être suffisant. Cependant, ce choix est justifié.

Raison du choix :

Pour rappel, le projet doit respecter le budget stipulé par le cahier des charges et être conçu sur un *PCB* monocouche. Ces contraintes limitent considérablement la complexité du *PCB* ainsi que le nombre de composants électroniques, ce qui a influencé le choix des microcontrôleurs.

Si l'émulation de jeux vidéo n'avait pas été un élément essentiel dans la console, un microcontrôleur 8 bits aurait largement suffi, tel qu'un *Arduino*. Il aurait été en mesure d'afficher les données ADC acquises sur l'écran et de créer une scène minimaliste en termes de puissance de calcul, telle qu'un mini-*dashboard* ou un mini-jeu vidéo.

Cependant, après quelques années d'apprentissage avec *Arduino*, ses limitations en termes de puissance de calcul deviennent rapidement apparentes, surtout lorsque les projets deviennent plus complexes. L'idée de rendre les projets plus complexes devient alors importante, instructive, et surtout, divertissante lors de la conception. La volonté de progresser a conduit au choix d'un microcontrôleur plus puissant pour ce projet.

Bien que le côté émulateur de jeu n'ait pas été spécifiquement mentionné dans le cahier des charges, l'aspect *rétro-gaming* de la console justifie la forme physique du prototype et apporte une dimension ludique au projet. Intégrer un émulateur de *ROM* dans cette console a été la complexité choisie.

Le microcontrôleur *Raspberry Pi Pico* a été choisi pour émuler les jeux, avec l'aide d'une bibliothèque *open source* [2]. Cependant, il était nécessaire que le microcontrôleur dispose également d'un nombre suffisant d'entrées analogiques pour lire les touches tactiles analogiques (9 entrées), ce qui n'était pas le cas.

Le défi consistait à trouver un microcontrôleur présentant suffisamment d'entrées analogiques, une précision adéquate, à un coût abordable, tout en offrant un émulateur de *ROM* en open source. Après de nombreuses recherches, la solution optimale a été d'ajouter au système embarqué un module agissant comme passerelle entre l'interface physique (boutons tactiles) et l'interface visuelle (gestion de l'écran). Cela offrirait également l'avantage de répartir les tâches sur deux ou plusieurs composants, limitant ainsi les latences du système embarqué.

Des convertisseurs analogique-numérique auraient pu être suffisants, mais cela aurait entraîné une architecture plus complexe à tracer sur le *PCB* en raison du brochage prédéfini des modules, contrairement à un microcontrôleur où les entrées *ADC* peuvent être assignées selon les besoins. De plus, ces modules *ADC* auraient entraîné un coût plus élevé par rapport à l'utilisation d'un microcontrôleur. Actuellement, le marché des microcontrôleurs est si concurrentiel que ces composants sont vendus à des prix très abordables malgré leur complexité.

De plus, un microcontrôleur offre plus de possibilités, telles que la gestion locale de divers autres capteurs, notamment la lecture *ADC* d'un indicateur de niveau de charge d'une éventuelle batterie, la surveillance de la consommation totale du circuit, la gestion d'indicateurs *LED*, etc. Un *STM32F103C8T6* a donc été choisi, d'autant plus que sa puce est vendue à moins d'1€.

Algorithmes installés :

Les deux microcontrôleurs communiquent entre eux via *UART*. Une extension *USB* vers *TTL* (*CP2102*) utilisée pour programmer le *STM32* utilise également ce même bus. En revanche, le *Raspberry Pi Pico* est programmé directement depuis son port micro-*USB* via le transfert d'un fichier compilé au format *uf2*.

Le *STM32* est programmé dans l'environnement de développement de *STMicroelectronics* appelé *STM32CubeIDE*. Le *Raspberry Pi Pico*, quant à lui, est programmé dans l'éditeur *Visual Studio Code* de *Microsoft*. Pour ce dernier, une manipulation assez minutieuse [3] a dû être effectuée pour compiler un code C compatible avec le processeur *ARM* du *Raspberry Pi Pico*, étant donné qu'un *PC* sous *Windows 10* a été utilisé plutôt qu'un *PC* sous *Linux*.

Pour la version 1.0 du prototype, le *STM32* se limite à la lecture des données *ADC* sur 10 bits et à leur envoi en *UART* vers le *Raspberry Pi Pico*. Cette acquisition de données a nécessité quelques adaptations logicielles avant l'émission de la charge utile. Le programme du *STM32* prend en compte les offsets générés par les bruits des *AOP*, d'environ 1.2V sur les 3.3V totaux. Une mise à l'échelle a donc été effectuée, mais pas seulement.

Afin que le *Raspberry Pi Pico* puisse lire en toute tranquillité les trames, le *STM32* cadence les charges utiles en fonction des interactions homme-machine. À chaque appui, c'est-à-dire à chaque variation analogique d'une électrode, une trame est envoyée. Cependant, étant donné que l'électrode d'une touche électrostatique est très sensible, il y a toujours une variation analogique, même sans appui. Un algorithme a été instauré pour filtrer les petites variations et se concentrer sur les variations assez conséquentes, mais pas trop, afin d'être réactif à l'appui d'une touche électrostatique. Le programme se compose de moins de 100 lignes de code.

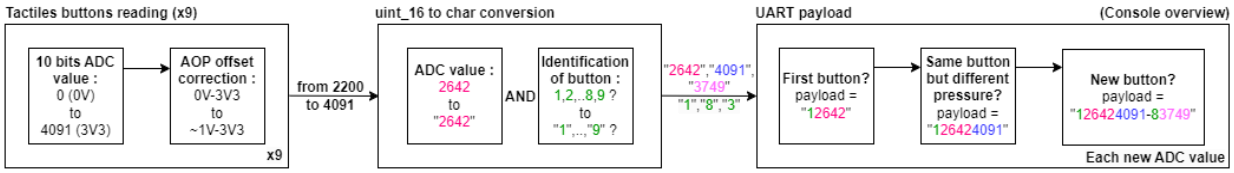


Figure 18 : Schéma bloc du code du STM32

Le *Raspberry Pi Pico*, quant à lui, assume des tâches plus lourdes. Il contrôle l'écran, la carte *SD* et le son. La bibliothèque [2] utilisée pour l'émulation est installée dans ce microcontrôleur. Ce dernier est overclocké à 266MHz pour permettre un fonctionnement fluide des jeux. Il émule la toute première console *Game Boy* sortie en 1989. Les entrées des boutons proposées par la bibliothèque ne permettent qu'une lecture tout-ou-rien, ce qui est tout à fait normal car les consoles n'ont que des boutons poussoirs. La bibliothèque a donc été modifiée pour pouvoir interpréter des données autres que tout-ou-rien. Cependant, éditer les *ROM* de jeux tels que *Tetris* ou *Pokémon* pour qu'ils puissent interagir de façon modulée avec les données analogiques demanderait trop de temps. C'est alors que le professeur a proposé des idées de conceptions logicielles. Une interface avant la sélection de *ROM* de jeu vidéo a donc été conçue. Ainsi, l'utilisateur peut découvrir pleinement le potentiel électrostatique du produit tout en ayant la possibilité de jouer à des jeux vidéo.

Cette interface tire parti de la bibliothèque émulatrice pour réutiliser certaines de ses fonctions, en particulier celles permettant l'affichage sur l'écran. L'ajout de ressources a également été effectué, notamment celle permettant une communication *UART*. Il a donc été nécessaire de comprendre pleinement les grandes lignes de la bibliothèque afin d'y introduire une toute nouvelle fonctionnalité, à savoir la nouvelle interface proposée.

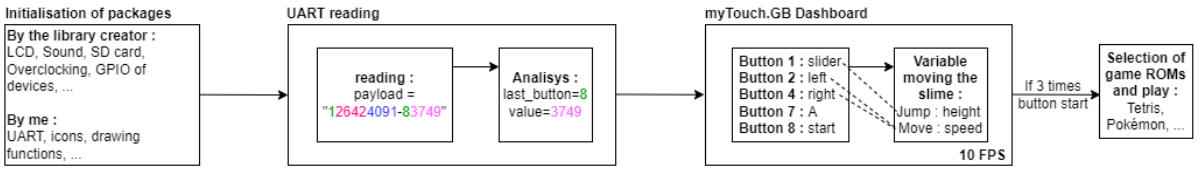


Figure 19 : Schéma bloc du code du Raspberry Pi Pico

Les inquiétudes concernant le temps nécessaire à la mise en place de toute cette programmation étaient présentes, étant donné que ce type de développement, en particulier le débogage, peut généralement prendre beaucoup de temps. Ce choix d'émulateur aurait pu mettre en péril le respect de la deadline du projet *Brotaru*. Cependant, un weekend, du vendredi 17 au dimanche 19 novembre 2023, a suffi pour établir la liaison UART avec les charges utiles et développer le mini-jeu exploitant l'aspect analogique tactile, le tout en un total de 35 heures. Un ajout de 500 lignes de code a été effectué dans la bibliothèque.

5.4 Organigramme

Pour clore l'aspect technique du prototype, un organigramme a été élaboré afin de résumer les différents aspects embarqués. Le voici à la figure 20 :

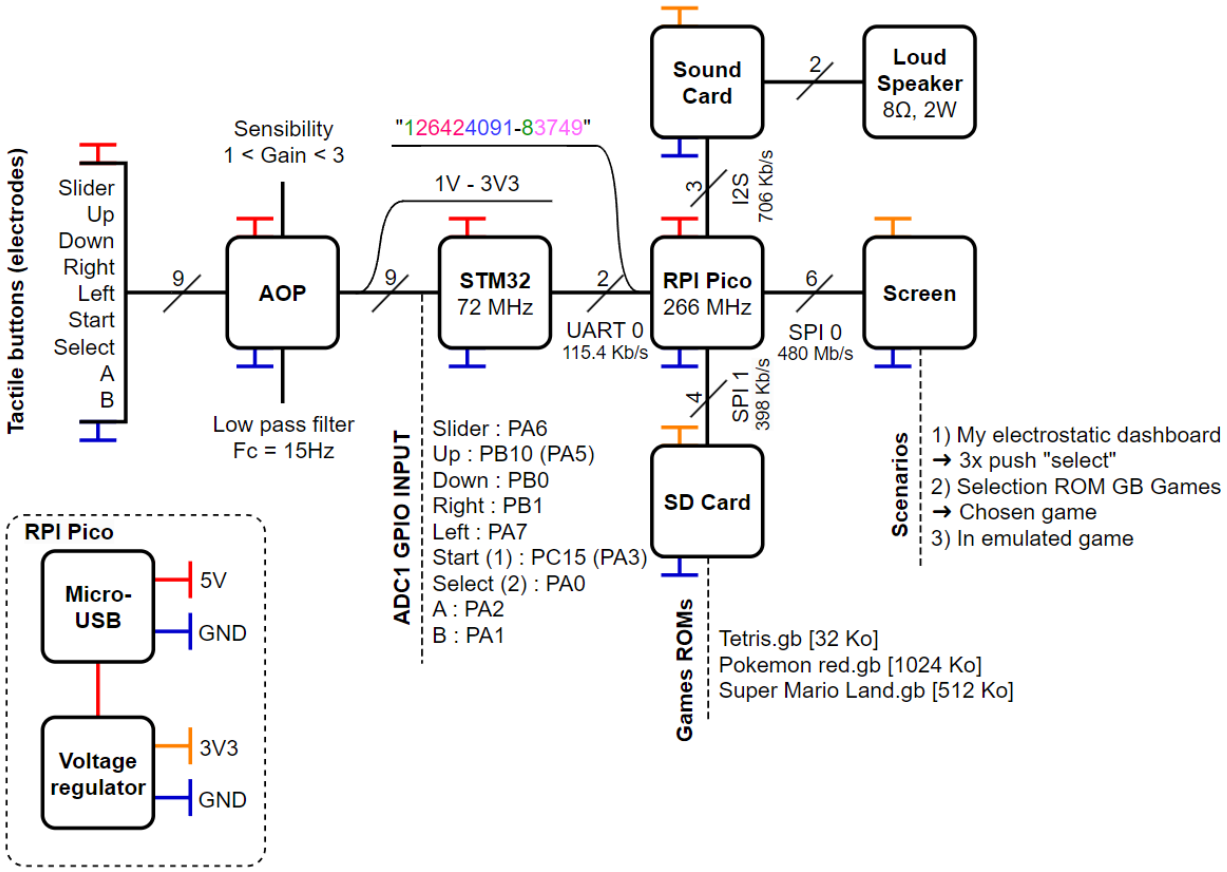


Figure 20 : Organigramme du prototype myTouch.gb

Cet organigramme renferme des informations supplémentaires qui n'ont pas été précisées précédemment.

L'alimentation du circuit s'effectue en connectant le *Raspberry Pi Pico* au port micro-*USB* de l'ordinateur. Les microcontrôleurs, la polarisation des touches tactiles, ainsi que les étages d'amplification avec des filtres, sont alimentés à 5V. En revanche, la carte son, l'écran et la carte *SD* sont alimentés par le régulateur de tension du *Raspberry Pi Pico*, générant une tension de 3V3.

La carte *SD* et l'écran sont commandés selon le protocole *SPI*, mais utilisent deux bus distincts. Unifier ces deux composants sous un même bus n'aurait pas été une solution optimale. En effet, l'écran doit fonctionner à une fréquence considérablement élevée pour générer 30 images par seconde (*FPS*). En revanche, la carte *SD*, étant donné que les fichiers *ROM* ont une taille de l'ordre du kilo-octet, fonctionne à une fréquence plus raisonnable.

Le *STM32* est équipé de deux convertisseurs *ADC* : *ADC1* et *ADC2*. Pour optimiser l'utilisation des ressources, un seul *ADC* (*ADC1*) est employé pour les différentes touches tactiles. Chaque ADC intégré dans le *STM32* peut traiter au maximum 10 entrées analogiques. À mesure que le nombre d'entrées augmente, la bande passante allouée à chaque donnée diminue. Cela se traduit par une réduction de la vitesse de réception des données. Dans le cas présent, 9 entrées analogiques doivent être traitées en séquence avant d'effectuer la première lecture. Un *prescaler* de 6 est utilisé pour cadencer *l'ADC*, correspondant à une fréquence de 12 MHz. En d'autres termes, pour les 9 entrées, chacune bénéficie d'une bande passante d'environ 1,33 MHz lorsqu'elle est échantillonnée par l'*ADC1*. Cette fréquence reste grande, elle n'impacte aucune latence dans le prototype.

6 Analyse du design

Un aspect intéressant consisterait à évaluer le prototype en fonction de critères couramment utilisés en entreprise. Il existe 15 principes pour la conception d'*IHM* [5]. Cette section comprendra une analyse de quelques-uns de ces principes. Un total de 5 principes adaptés au sujet sont évoqués dans cette section. Pour accompagner les analyses portant sur la conception, la figure 21 est présentée ci-dessous :

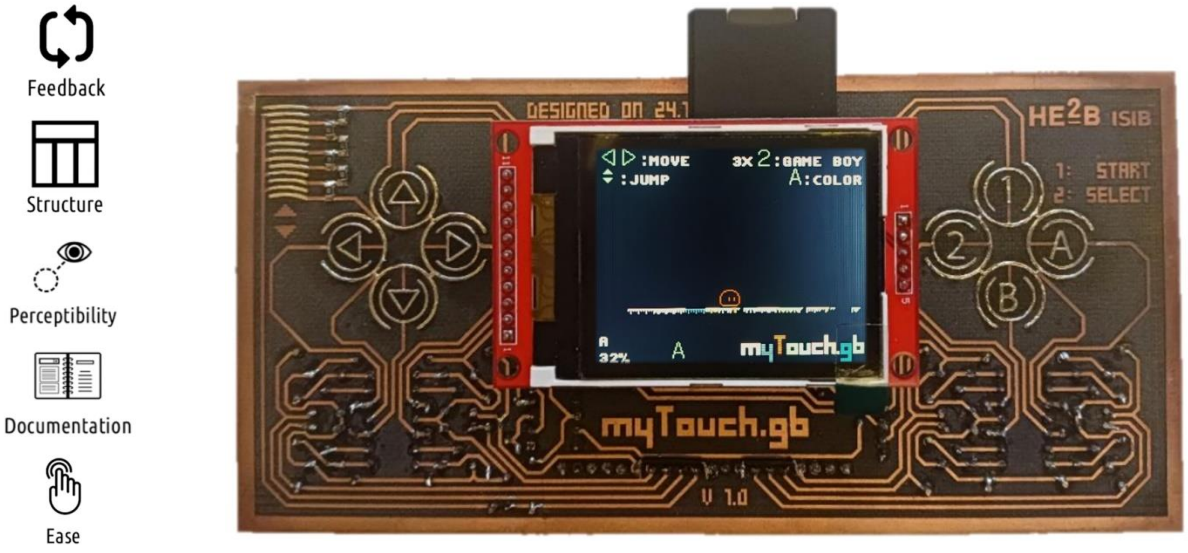


Figure 21 : 5 principes de design du prototype myTouch.gb

***Feedback* :**

Le feedback représente un élément essentiel dans un appareil à interaction homme-machine. Lorsqu'il s'agit d'un bouton poussoir de qualité, le *feedback* s'effectue instinctivement en ressentant le bouton s'enfoncer dans son siège, accompagné d'un bruit caractéristique. Toutefois, du côté des touches tactiles, aucun retour "mécanique local" n'est perceptible. Bien que l'ajout d'un son de *buzzer* lors des appuis puisse être une solution, cela n'est pas toujours confortable lors de l'utilisation d'une console.

La solution adoptée dans ce prototype ne repose ni sur un mécanisme ni sur une indication auditive, mais plutôt sur un *feedback* visuel. En bas à gauche de l'écran, des inscriptions correspondant aux boutons appuyés sont affichées. Dans l'exemple de la figure 21 ci-dessus, le texte "*A*" suivi de "*32%*" est affiché en blanc, indiquant que le bouton *A* est appuyé et que l'utilisateur exerce une pression de 32%. À droite de ces inscriptions se trouve une lettre *A* en vert. Cette lettre apparaît au front montant et disparaît au front descendant. Si la touche est relâchée, le texte en vert disparaît, mais le texte en blanc reste, permettant ainsi d'observer la dernière valeur d'un bouton lue avant l'appui d'un nouveau.

***Structure* :**

La structure du design de l'écran a été maintenue volontairement minimaliste pour ne pas perdre le lecteur. Au centre, l'espace est dédié au personnage, tandis qu'en haut à gauche et à droite se trouvent les touches tactiles qui peuvent interagir avec le scénario. En bas à gauche, le *feedback* de la touche appuyée est affiché, et en bas à droite, le logo du prototype est positionné.

Concernant le *PCB*, sa structure a été conçue pour s'adapter au mieux aux mains de l'utilisateur, lui permettant ainsi d'interagir avec le projet en utilisant uniquement ses deux pouces. La présence de l'électronique en couche inférieure peut initialement déstabiliser l'utilisateur, mais une fois pris en main, cela ne pose plus de problème. Il est arrivé que des utilisateurs touchent des pistes d'électrodes d'autres touches tactiles, entraînant des incompréhensions. La solution aurait pu être l'utilisation d'un vernis tropicalisant sur toute l'empreinte électronique, à l'exception des touches.

***Perceptibility* & *Documentation* :**

En fonction de la perspicacité de la personne, le produit peut sembler très intuitif dès le début pour certaines, mais pas pour toutes. Au *Brotaru*, il est arrivé que certaines personnes minimisent rapidement l'envergure du projet, avant même de l'essayer, en justifiant qu'il s'agit simplement d'une console à touches tactiles tout-ou-rien ordinaire. Une mise en contexte était nécessaire avant l'utilisation du prototype pour que les utilisateurs comprennent les scénarios possibles avec le côté analogique des touches. Pour remédier à cela, une idée pourrait être de développer une petite animation sur l'écran avant l'apparition du *dashboard*. Cette animation inviterait l'utilisateur à appuyer sur les touches à différents niveaux de pression.

Il est également arrivé que les utilisateurs ne remarquent pas la présence du *SLIDER*. Ils essayaient de sauter avec les touches haut et bas. Pour mettre en avant les touches tactiles, elles ont été étamées à l'étain. La confusion peut être liée au symbole utilisé dans le design. Le *SLIDER* est représenté par des flèches pleines haut-bas. L'icone du *SLIDER* doit être retravaillé.

Que ce soit une documentation interactive, comme l'idée évoquée pour une nouvelle animation plongeant l'utilisateur, ou une documentation externe sur le *PCB* ou sur papier, tout peut aider, à condition que cela donne envie d'être lu.

***Ease* :**

Une dernière analyse peut encore être effectuée. Les facilités d'utilisation d'un produit permettent à l'utilisateur de contrôler nativement l'*IHM* plutôt que d'être contraint par l'architecture, laissant derrière du stress et de l'impatience. Le prototype n'est pas encore tout à fait au point sur ce principe. Une difficulté en particulier intervient.

En effet, une situation dans le scénario est limitée par l'algorithme des touches tactiles. À l'appui d'une touche, une pression est affichée en pourcentage et varie en fonction de la pression exercée. Cependant, elle varie brutalement. Par exemple, si l'utilisateur souhaite passer de 32% à 9%, il arrive parfois qu'il y ait des pourcentages intermédiaires, en passant de 32% à 70%, puis à 43%, et enfin à 9%. Entre cette action, 1 seconde s'est passée seulement. Pendant cette seconde, insignifiante pour l'homme mais conséquente pour la machine, beaucoup de choses peuvent se passer. Ces valeurs intermédiaires sont dues à l'imprécision de l'homme, mais le déstabilisent en conséquence. Cela se produit en particulier lorsqu'on essaie de faire sauter le personnage. Le *slime* s'écrase de plus en plus fort, mais parfois il se détend puis revient à sa position compressée. Le capteur est sensible et ne ment pas, c'est un fait. La solution serait d'adapter l'algorithme sans le *Raspberry Pi Pico* de manière à filtrer les pics, afficher les valeurs moyennes entre une donnée précédente et actuelle, et donc rendre le tout plus lisse, malgré l'imprécision de l'humain.

7 Conclusion

Les expériences menées au laboratoire ont permis d'observer des comportements d'amplification remarquables dans un environnement riche en phénomènes électrostatiques. Ces découvertes ont été très enrichissantes, mais elles ont également révélé certaines approches qui restent encore à approfondir, tellement les opportunités sont vastes.

À une sensibilité augmentée, correspondant à un gain proche de 5, les touches tactiles permettent une détection momentanée sans nécessiter une pression significative. En augmentant encore le gain selon l'architecture de la touche, un autre phénomène a été observé : la détection de présence à distance. Cependant, cette détection était limitée à environ 1 mm de distance avec l'architecture de la figure 3. Une hypothèse relie 3 facteurs influant sur ce phénomène sur *PCB* : l'écart isolant (diélectrique) entre les électrodes, la surface cuivrée des électrodes réceptrices et la tension appliquée sur l'électrode de polarisation. Une surface plus grande facilite la détection, à condition que l'écart soit respecté. Quant à la tension de polarisation, elle permet d'influencer directement la portée de détection. Bien que le cahier des charges ne porte pas sur les capteurs de distances, une expérience a été menée sans le vouloir. Cependant, cette hypothèse pourrait mériter d'être plus largement expérimentée et validée à l'avenir.

Par ailleurs, avec l'électronique actuelle, la polarisation d'un doigt n'est détectée que lorsqu'elle est à potentiel positif (+5V), et non lorsqu'elle est négative ou neutre. Suivant la polarisation, le montage de *R1* est tiré vers le haut ou vers le bas respectivement (voir figure 6). Curieusement, seuls les potentiels positifs sont pris en compte, un phénomène difficile à expliquer en lien avec la physiologie organique d'un doigt.

Lors du *Brotaru*, événement axé sur les jeux vidéo organisé par l'établissement scolaire le 4 décembre 2023, la console à *IHM* a été mise à l'épreuve. Des dizaines de personnes ont utilisé le prototype et l'ont apprécié, revivant entre-autre leurs souvenirs d'enfance avec *Tetris*, *Pokémon*, *Mario Land*, etc. Une découverte intéressante a émergé : la sensibilité des touches tactiles variait selon la personne. Les jeunes manifestaient une sensibilité plus élevée, tandis que les personnes plus âgées devaient exercer une pression plus forte pour interagir avec la console. Cette différence s'explique par le fait que l'âge est inversement proportionnel au taux d'humidité dans la peau, affectant la conduction des charges électriques.

Les participants avaient tendance à comparer cette technologie à celle des écrans capacitifs présents aujourd'hui sur les tablettes, *GSM*, et autres. Cependant, ils ont réalisé à quel point ces technologies sont totalement différentes et que les opportunités d'application varient considérablement entre elles. Les capteurs électrostatiques sont peu répandus aujourd'hui, mais ils méritent d'être explorés plus profondément, car cette technologie présente un grand potentiel, ce qui est non négligeable.

Bibliographie

[1] Qvar electrostatic sensor, publié en août 2022, par STMicroelectronics, en ligne, https://www.st.com/resource/en/application_note/an5755-qvar-sensing-channel--stmicroelectronics.pdf, consulté le 15 septembre 2023.

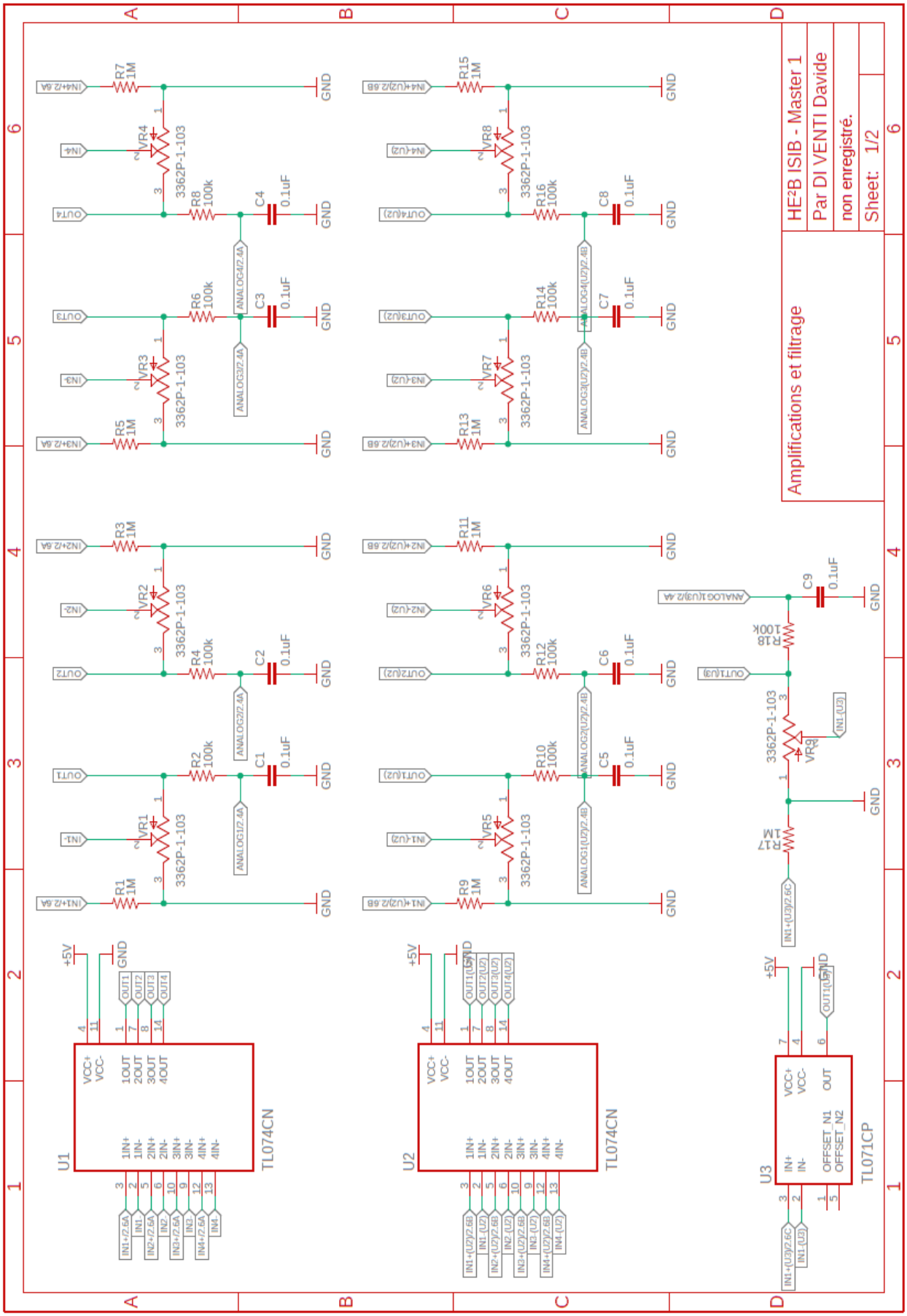
[2] RP2040-GB Game Boy emulator, publié en mai 2022, par YouMakeTech, en ligne, https://github.com/YouMakeTech/Pico-GB, consulté le 11 novembre 2023.

[3] Setup Raspberry pi pico on VS Code, publié en janvier 2021, par Learn Embedded Systems, https://www.youtube.com/watch?v=mUF9xjDtFfY, consulté le 11 novembre 2023.

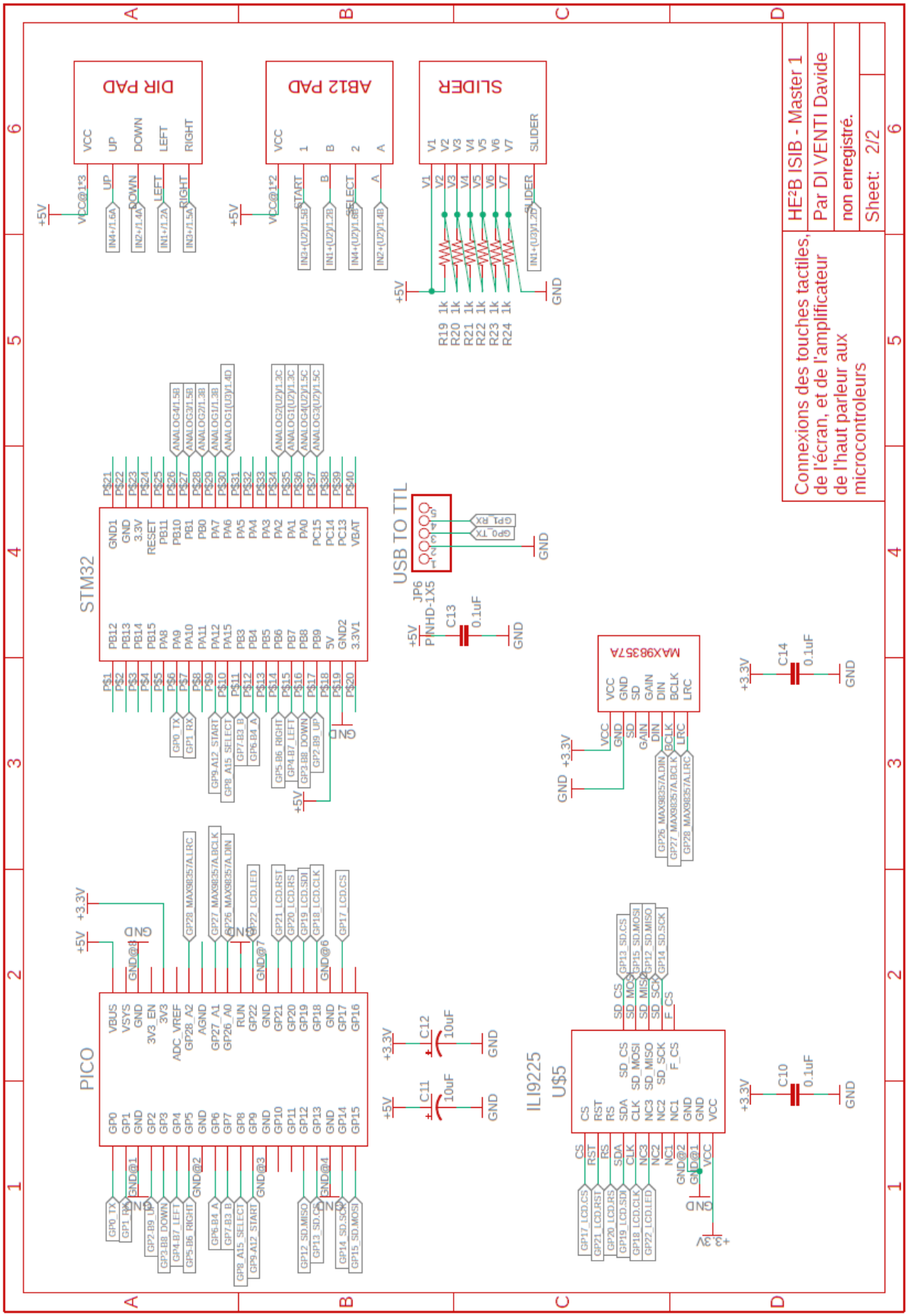
[4] Procédés de Prototypage, publié en 2018, par G. Le Vaillant, en ligne, https://www.isibnet.be/, consulté le 15 septembre 2023.

[5] Interface Homme-Machine, publié en septembre 2022, par G. Le Vaillant, en ligne, https://www.isibnet.be/, consulté le 15 septembre 2023.

Annexe 1 : Schéma électronique [page 1/2]



Annexe 2 : Schéma électronique [page 2/2]



Annexe 3 : Code main.c simplifié[1] du STM32 (~100 lignes)

```c
// Mon code se trouve à l'endroit spécifique où il est écrit :
//   /* USER CODE BEGIN ... */
//   mycode;
//   /* USER CODE END ... */
// Le reste est généré par le système.

#include "main.h"       // Ressources des constantes, macros et prototypes de fonctions
#include "adc.h"        // Ressources de l'ADC
#include "usart.h"      // Ressources de l'USART
#include "gpio.h"       // Ressources des GPIO

/* USER CODE BEGIN Includes */
#include <string.h>     // Manipulation de chaînes de caractères
#include <stdio.h>      // Utilisé pour sprintf(...)
#include <stdlib.h>     // Utilisé pour abs(...)
/* USER CODE END Includes */

/* USER CODE BEGIN PV */
uint16_t readValues[9];         // Tableau pour stocker les lectures ADC de 9 canaux
uint16_t previousValues[9] = {0};       // Initialisé à 0, tableau pour stocker les lectures ADC précédentes de 9 canaux
uint8_t ADC_channels[9] = {ADC_CHANNEL_9, ADC_CHANNEL_8, ADC_CHANNEL_7, ADC_CHANNEL_6, ADC_CHANNEL_5, ADC_CHANNEL_3, ADC_CHANNEL_2, ADC_CHANNEL_1,
ADC_CHANNEL_0 };
                        // Tableau pour stocker les canaux ADC de 9 entrées tactiles
ADC_ChannelConfTypeDef sConfigPrivate = {0}; // Structure de configuration pour le canal ADC
uint8_t ADC_toleration = 250;   // Valeur de tolérance pour détecter les changements de valeur ADC
char buffer[50];                // Tampon de caractères pour stocker les chaînes formatées
char touchID[2];                // Variable pour stocker l'ID du bouton touché
char previousTouchID[2];                // Variable pour stocker l'ID précédent du bouton touché
/* USER CODE END PV */

void SystemClock_Config(void);

int main(void)
{

  HAL_Init();   // Initialiser le matériel d'abstraction matériel (HAL)
  SystemClock_Config(); // Configurer l'horloge système
  MX_GPIO_Init();       // Initialiser toutes les périphériques configurées pour la GPIO
  MX_USART1_UART_Init();        // Initialiser l'USART1
  MX_ADC1_Init();       // Initialiser l'ADC1

  /* USER CODE BEGIN 2 */
  HAL_GPIO_WritePin(GPIOA,GPIO_PIN_15, 1); // Mise à 1 des touches game boy :
  HAL_GPIO_WritePin(GPIOB,GPIO_PIN_3, 1);  // A, B, Start, Select, Up, Down, Right, & LEFT.
  HAL_GPIO_WritePin(GPIOB,GPIO_PIN_4, 1);
  HAL_GPIO_WritePin(GPIOA,GPIO_PIN_12, 1); // Les état tout-ou-rien sont assignés à des I/O
  HAL_GPIO_WritePin(GPIOB,GPIO_PIN_7, 1);  // en plus d'etre dans l'UART.
  HAL_GPIO_WritePin(GPIOB,GPIO_PIN_6, 1);
  HAL_GPIO_WritePin(GPIOB,GPIO_PIN_9, 1);  // L'UART est utilisé que pour l'interface "analogique",
  HAL_GPIO_WritePin(GPIOB,GPIO_PIN_8, 1);  // les I/O sont untilisés que pour l'émulation

  HAL_GPIO_WritePin(GPIOC,GPIO_PIN_13, 1); // clignotant sur la LED embarquée du STM32
  HAL_Delay(100);                   // de période 200ms permettant d'identifier si
  HAL_GPIO_WritePin(GPIOC,GPIO_PIN_13, 0); // le STM32 est en mode programmation ou
  HAL_Delay(100);               // en mode boot.
  HAL_GPIO_WritePin(GPIOC,GPIO_PIN_13, 1);
  HAL_Delay(100);               // A été très utile lors de la programmation
  HAL_GPIO_WritePin(GPIOC,GPIO_PIN_13, 0); // et du debogage du STM32.
  HAL_Delay(100);
  HAL_GPIO_WritePin(GPIOC,GPIO_PIN_13, 1); // En dehors de ça, ces lignes sont inutiles.
  HAL_Delay(100);
  HAL_GPIO_WritePin(GPIOC,GPIO_PIN_13, 0);
  /* USER CODE END 2 */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    sConfigPrivate.Rank = ADC_REGULAR_RANK_1;               // Sélectionne le rang de la conversion ADC régulière
    sConfigPrivate.SamplingTime = ADC_SAMPLETIME_1CYCLE_5;  // Configure le temps d'échantillonnage pour la conversion ADC

    for (int i = 0; i < 9; i++)  // Cette boucle effectue une lecture ADC à la fois pour chacun des 9 canaux
    {
      sConfigPrivate.Channel = ADC_channels[i];             // Sélectionne le canal ADC à convertir
      HAL_ADC_ConfigChannel(&hadc1, &sConfigPrivate);       // Configure le canal ADC
      HAL_ADC_Start(&hadc1);                                // Démarre la conversion ADC
      HAL_ADC_PollForConversion(&hadc1, 1000);              // Attends la fin de la conversion ou le dépassement du délai
      readValues[i] = HAL_ADC_GetValue(&hadc1);             // Lit la valeur convertie
      HAL_ADC_Stop(&hadc1);                                 // Arrête la conversion ADC

      if (abs(readValues[i] - previousValues[i]) > ADC_toleration)  // Suivant la tolérance ADC, il y a filtrage
      {
        switch (ADC_channels[i])            // Sélectionne le canal ADC en fonction du cas
        {
          case 6: touchID[0] = '1'; break;  // Définit l'identifiant du bouton en fonction (fronts montants + bouton appuyé)
          case 5: touchID[0] = '9'; break;
          case 7: touchID[0] = '2'; break;
          case 8: touchID[0] = '3'; break;
          case 9: touchID[0] = '4'; break;
          case 0: touchID[0] = '5'; break;
          case 1: touchID[0] = '6'; break;
          case 2: touchID[0] = '7'; break;
          case 3: touchID[0] = '8'; break;
        }

        if (readValues[i] < 2200)  // Si la valeur ADC lue est inférieure à 2200 alors cela est considéré comme front descendant
        {
          touchID[0] = '-';         // Un caractère de séparation est alors utilisé comme ID
        }

        if ((touchID[0] != previousTouchID[0])) // Si l'identifiant du bouton a changé depuis la dernière interaction humaine
        {
          sprintf(buffer, "%c", touchID[0]);    // Formatte l'identifiant du bouton en chaîne de caractères
          HAL_UART_Transmit(&huart1, (uint8_t *)buffer, strlen(buffer), 100);// Transmet UART l'ID si != du précédent : "6" ou "-"

          switch (touchID[0])   // En plus de l'UART, mise à 1 ou 0 des I/O correspondant aux touches game boy
          {
            case '9': HAL_GPIO_WritePin(GPIOB, GPIO_PIN_9, 0); break;   // Action associée au bouton 9 (B9 up)
            case '2': HAL_GPIO_WritePin(GPIOB, GPIO_PIN_7, 0); break;   // Action associée au bouton 2 (B7 left)
            case '3': HAL_GPIO_WritePin(GPIOB, GPIO_PIN_8, 0); break;   // Action associée au bouton 3 (B8 down)
            case '4': HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, 0); break;   // Action associée au bouton 4 (B6 right)

            case '5': HAL_GPIO_WritePin(GPIOA, GPIO_PIN_15, 0); break;  // Action associée au bouton 5 (A15 select)
            case '6': HAL_GPIO_WritePin(GPIOB, GPIO_PIN_3, 0); break;   // Action associée au bouton 6 (B3 B)
            case '7': HAL_GPIO_WritePin(GPIOB, GPIO_PIN_4, 0); break;   // Action associée au bouton 7 (B4 A)
            case '8': HAL_GPIO_WritePin(GPIOA, GPIO_PIN_12, 0); break;  // Action associée au bouton 8 (B4 start)
            case '-':                                   // Action associée un front descendant est détecté
              HAL_GPIO_WritePin(GPIOB, GPIO_PIN_7, 1);          // Aucun bouton n'est appuyé, donc tout est à 1
              HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, 1);
              HAL_GPIO_WritePin(GPIOB, GPIO_PIN_9, 1);
              HAL_GPIO_WritePin(GPIOB, GPIO_PIN_8, 1);

              HAL_GPIO_WritePin(GPIOA, GPIO_PIN_15, 1);
```

[1] Sans les parties rédigées (setup, fonctions, …) par le générateur de code de STM32CubeIDE.

```
121                 HAL_GPIO_WritePin(GPIOB, GPIO_PIN_3, 1);
122                 HAL_GPIO_WritePin(GPIOB, GPIO_PIN_4, 1);
123                 HAL_GPIO_WritePin(GPIOA, GPIO_PIN_12, 1);
124                 break;
125               }
126             }
127
128             previousValues[i] = readValues[i];      // Mise à jour des valeurs précédentes avec les valeurs actuelles
129
130             sprintf(buffer, "%d", readValues[i]);   // Convertit la valeur lue en chaîne de caractères
131             char chars[2];
132             for (int c = 0; c < strlen(buffer); c++)
133             {
134               sprintf(chars, "%c", buffer[c]);      // Formatte chaque caractère en chaîne de caractères
135               if (touchID[0] != '-') // Transmet chaque caractère ADC lue : 2654 -> "2", 10ms, "6", 10ms, "5", 10ms, "4", 10ms
136               {
137                 HAL_UART_Transmit(&huart1, (uint8_t *)chars, strlen(chars), 100);
138               }
139               HAL_Delay(10);                // Délai de 10 ms entre chaque transmission
140             }
141
142             previousTouchID[0] = touchID[0];       // Met à jour l'identifiant du bouton précédent avec l'identifiant actuel
143           }
144         }
145         HAL_Delay(10);                          // Délai de 10 ms entre chaque lecture ADC
146         /* USER CODE END WHILE */
147
148       }
149     }
150
151     /*
152     System functions :
153       SystemClock_Config
154       HAL_TIM_PeriodElapsedCallback
155       Error_Handler
156       assert_failed (if USE_FULL_ASSERT)
157     */
158
```

Annexe 4 : Code main.c simplifié[1] du Raspberry Pi Pico (~500 lignes)

```
/**
 * Copyright (C) 2022 by Mahyar Koshkouei <mk@deltabeard.com>
 * {restrictions}
 *
 */

/* ------------ Peanut-GB emulator ------------
#define Enable LCD, SOUND, SDCRD, ...
#define VSYNC, FPS, CLOCK
#include C Headers : stdio, stdlib, string
#include local libraries : GPIO, spi, timer, ...
*/

#include <hardware/uart.h> // inclusion d'une nouvelle bibliothèque pour utiliser l'UART entre Pi Pico et STM32

/* ------------ Peanut-GB emulator ------------
#include Project headers : mk ili9225, I2S, gbcolor, sdcard, ...
#define GPIO pins : up, down, a, b, CS, CLK, SDA, ...
*/

#define UART ID uart0              // Utilisation de l'uart num 0 du pi pico
#define BAUD RATE 115200   // Fréquence uart de 115200 bit/s
#define DATA BITS 8                // Nombre de bits de données par trame (8 bits)
#define STOP BITS 1 // Nombre de bits d'arrêt par trame (1 bit)
#define PARITY    UART PARITY NONE // Pour le setup
#define MAX BUFFER SIZE 100  // Taille maximale du tampon pour les octets reçus
#define UART TX PIN 0 // Pin TX à la GPIO 0
#define UART_RX_PIN 1 // Pin TX à la GPIO 1

int uart_buttons_states[10]; // Etat binaire des 9 boutons dans une liste
int uart buttons adc[10]; // Etat ADC (10 bits) des 9 boutons dans une liste

int chars rxed = 2; // Le le met à 2 pour qu'il prenne en compte le premier "." de buff. sinon TouchID n'est pas détecté
uint8_t rx buffer[MAX BUFFER SIZE]; // Création de la payload qui sera traité en UART

int payload_story = 0;  // longueur de l'historique des valeurs ADC du dernier bouton affiché
// Pour afficher l'ADC tout les 4 caractère, et pas en défilant 1 à la fois, sinon on voit des chiffres au delà de 4095

/* ------------ Peanut-GB emulator ------------
setup (sound, screen, ...)
variables ( struct buttons, rom, ...)
functions ( rom read.write.error, screen spi.cs.delay, postNativeApp screen,...)
load rom, enable
...
*/

uint8_t char_d[11] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 61}; // Caractère uint8_t correspondant à 0123456789-
char buff[100]= ".";  // Chaine utilisée correspondant au payload UART avec séparation de caractère, et défilement des caractère
bool onReading = false; // Flag déterminant si nous sommes en lecture ou pas
char payload[10] = "\0"; // Représentant les 4 dernier caractère du buff[100]
char touchID = '\0';     // Initialisation de touchID à un caractère nul, pour assurer un tampon vide (bug sinon)

void on_uart_rx() {
onReading = true; // Flag activé, en pleine lecture

  while (uart_is_readable(UART_ID)) {
    uint8_t ch = uart_getc(UART_ID);
    char c[2];
    if (chars rxed % 2 == 0) {  // Chaque trame reçue a une longueur de 2 caractères (char utile et char de fin)
      for (int i = 0; i < 11; i++) {
        if (ch == char d[i]) {
          if (i <= 9) sprintf(c, "%d", i); // Le caractère de front montant et de pression assigné à un chiffre (0-9).
                                           // la chaine reçu a un format non ASCII, il a un autre format dans le "char d"
          if (i == 10) sprintf(c, "%c", '.'); // Le caractère de front montant est assigné à un "."
          if (chars_rxed / 2 < 20) {
                      buff[chars rxed / 2] = c[0];    // Buff est limité à 20 char. exemple : "449.324763421.22676."
          } else {
            for (int j = 0; j < 19; j++) {  // Si buff dépasse les 20 char, décalage à gauche pour les nouv. données
              buff[j] = buff[j + 1];
            }
            buff[19] = c[0];
          }
        }
      }
    }
    chars rxed++; // A chaque trame, 2 caractère sont envoyé, dont un inutile (le dernier), donc à chaque trame -> +2
  }

  // A partir de buff = "449.324763421.22676.",payload = 2676
  int unit = 3; // Que buff termine par  "6352." ou "36652", payload copie les 4 derniers chiffres

  for (int i = strlen(buff) - 1; i >= 0 && unit >= 0; i--) {
    if (buff[i] != '.') {
      payload[unit] = buff[i];
     unit--;
    }
  }
  // A partir de buff = "449.324763421.22676.", touchID = 2, provenant du premier chiffre du dernier groupe(22676)
  int lastDotIndex = -1;
  int beforeLastDotIndex = -1;
  // Recherche du dernier point dans buff
  for (int i = strlen(buff) - 1; i >= 0; i--) {
    if (buff[i] == '.') {
      if (lastDotIndex == -1) {
        lastDotIndex = i;
      } else {
        beforeLastDotIndex = i;
        break;
      }
    }
  }

  // Recherche du premier chiffre après le dernier point s'il existe
  if (lastDotIndex != -1 && lastDotIndex < strlen(buff) - 1) {
    char afterLastDot = buff[lastDotIndex + 1];
    if (afterLastDot >= '0' && afterLastDot <= '9') {
      touchID = afterLastDot; // Le premier chiffre après le dernier point est affecté à touchID
    }
  }

  // Si aucun chiffre trouvé après le dernier point, chercher le chiffre à côté de l'avant-dernier point
  if (touchID == '\0' && beforeLastDotIndex != -1 && beforeLastDotIndex < strlen(buff) - 1) {
    char beforeLastDot = buff[beforeLastDotIndex + 1];
    if (beforeLastDot >= '0' && beforeLastDot <= '9') {
      touchID = beforeLastDot; // Le chiffre à côté de l'avant-dernier point est affecté à touchID
    }
  }

  // Mémorisation des données UART payload et touchID dans une liste (avec détection front montant/descendant)
  for (int i=0; i<10; i++){
    uart_buttons_states[i] = 0;
```

[1] Sans les parties rédigées (setup, fonctions, …) par le constructeur de la bibliothèque émulatrice.

```
  }
  char ID[2];
  sprintf(ID,"%c",touchID);
  uart_buttons_states[atoi(ID)] = 1;
  uart_buttons_adc[atoi(ID)] = atoi(payload);
  if (buff[strlen(buff)-1]== '.'){
    for (int i=0; i<10; i++){
      uart_buttons_states[i] = 0;
    }
  }
}

void mk_ili9225_draw(char *image, int x, int y, int w, int h, uint16_t color, int orientation){
  /* Dessine sur l'écran les icones personnalisées pixel par pixel
   * Les icones sont lues sous forme de matrice binaire
   * Les 0 et 1 sont respectivement des pixel éteint ou allumés.
   */
  int x_ = x;
  int y_ = y;
  int pixels_quantity = w * h;
  int orientation_x = orientation; // 1 right, -1 left
  if (orientation_x >=0){
    for (int pixel = 0; pixel < pixels_quantity; pixel++){
      if ((pixel % w) == 0){
        y_++;
        x_ = x;
      }
      if (image[pixel] == '1'){
        mk_ili9225_fill_rect(x_, y_, 1, 1, color);
      }
      x_++;
    }
  }
  else if (orientation_x == -1) {
    for (int row = 0; row < h; row++) {
      for (int col = w - 1; col >= 0; col--) {
        if (image[row * w + col] == '1'){
          mk_ili9225_fill_rect(x_ + (w - 1 - col), y_+1 + row, 1, 1, color);
        }
      }
    }
  }
}

void native_user_interface(){
 /* Gère l'interface avec le slime avec les interactions Homme-Machine des touches tactiles
  * Se compose d'un équivalent "setup" et "loop"
  */
  uart_init(UART_ID, BAUD_RATE);                              // Set up our UART with a basic baud rate.
  gpio_set_function(UART_TX_PIN, GPIO_FUNC_UART);             // Set the TX and RX pins by using the function select on the GPIO
  gpio_set_function(UART_RX_PIN, GPIO_FUNC_UART);             // Set datasheet for more information on function select
  int __unused actual = uart_set_baudrate(UART_ID, BAUD_RATE);
  uart_set_hw_flow(UART_ID, false, false);                    // Set UART flow control CTS/RTS, we don't want these, so turn them off
  uart_set_format(UART_ID, DATA_BITS, STOP_BITS, PARITY);     // Set our data format
  uart_set_fifo_enabled(UART_ID, false);                      // Turn off FIFO's - we want to do this character by character
  int UART_IRQ = UART_ID == uart0 ? UART0_IRQ : UART1_IRQ;    // Set up a RX interrupt.
  irq_set_exclusive_handler(UART_IRQ, on_uart_rx);            // And set up and enable the interrupt handlers
  irq_set_enabled(UART_IRQ, true);
  uart_set_irq_enables(UART_ID, true, false);                 // Now enable the UART to send interrupts - RX only
  bool NATIVE_UI = 1;                                         // Flag permettant de rester ou sortir de l'interface avec le slime
  bool previous_clock;                                        // Cadence la loop, et donc les FPS de l'interface
  bool clock;                                                 // Cadence la loop, et donc les FPS de l'interface
  int clock_period = 100000;                                  // 100ms de rafraichissement
  int time;                                                   // Cadence quelques mécaniques dans la loop

  char *slime_idle[] = {
"0000000000000000000000000000000000000000000000000000111111000000000110000011000001000000000100010000000000010010000000000001010000000000000011000000100010001100000010001000110000001000100011000000000000001100000000000000101000000000000010001111111111100",
"0000000000000000000000000000000000000000000000000000111111000000000110000011000001000000000100010000000000010010000000000001010000000000000011000000100010001100000010001000110000000000000011000000000000001100000000000000101000000000000010001111111111100",
"0000000000000000000000000000000000000000000000000000111100000000001100001100000011000000001100010000000000001001000000000000100100000000000001010000001000100011000000100010001100000000000000110000000000000011000000000000001010000000000000100011111111111100",
"0000000000000000000000000000000000000000000000000000111100000000001100001100000011000000001100010000000000001001000000000000100100000000000001010000001000100011000000100010001100000010001000110000000000000011000000000000001010000000000000100111111111111110",
"0000000000000000000000000000000000000000000000000000111100000000001100001100000011000000001100010000000000010001000000000000100100000000000001010000001000100011000000100010001100000010001000110000000000000011000000000000001010000000000000100111111111111110",
"0000000000000000000000000000000000000000000000000000111100000000001100001100000011000000001100010000000000010001000000000000100100000000000001010000000000001100000010001000110000001000100011000000000000001100000000000000110000000000000010111111111111110",
"0000000000000000000000000000000000000000000000000000111100000000001100001100000010000000001000010000000000010001000000000000100100000000000001010000000000001100000010001000110000001000100011000000100010001100000000000000110000000000000010111111111111110",
"0000000000000000000000000000000000000000000000000000110000000000011100111000000010000000001000010000000000010001000000000000100100000000000001001000000000000101000000100010001100000010001000110000001000100011000000000000001100000000000000101111111111111110",
"0000000000000000000000000000000000000000000000000000110000000000011100111000000010000000001000010000000000010001000000000000100100000000000001001000000000000101000000100010001100000010001000110000001000100011000000000000001100000000000000101111111111111110",
"0000000000000000000000000000000000000000000000000000111000000000000110001110000000100000000010000010000000000010001000000000000100100000000000001001000000000000101000000000000000011000000110011000110000000000000011000000000000001100000000000000100111111111111110"};
  char *slime_jump[] = {
"0000000011111100000000000000011000000110000000000001000000000010000000001000000000000010000000001000000000000010000000100010001000100000001000010000100000100000010000100001000010000000100000000000000100000010000000000000001000000100000000000000010000000111111111111110000",
"0000000000000000000000000000111111100000000000000110000000011000000000100000000000001000000000100000000000001000000010001000100010000000100010000100001000000100001000010000100000001000000000000000100000010000000000000001000000100000000000000010000000111111111111110000",
"0000000000000000000000000000000000000000000000001111111000000000000011100000000111000000000100000000000000100000001000000000000001000000100010000100001000000100001000010000100000001000000000000000100000010000000000000001000000100000000000000010000000111111111111110000",
"0000000000000000000000000000000000000000000000000000000000000001111111110000000000110000000000110000000100000000000000100000010000000000000000100000010000001000010000100001000010000100001000010000000000000000100010000000000000000010000011111111111111111000",
"0000000000000000000000000000000000000000000000000000000000000000000000000011111111111000000001100000000000011000000100000000000000100000010000001000010000100001000010000100001000010000000000000000100010000000000000000010000111111111111111111100",
"0000000000000000000000000000000000000000000000000000000000000000000000000000000000000001111111111111000000010000000000000100000100000000000000000100010000000100010000000100100000001000010000001001000000000000000000100011111111111111111100",
"000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000111111111111000000110000000000000000110001000000010001000000010010000000100010000000100100000000000000000010011111111111111111110",
"0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000011111111111111110000110000000000000000011010000000100010000000110000000000000000000001011111111111111111110",
"0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000011111111111111111110100000000100010000000110000000000000000000001111111111111111111111",
"0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000011111111111111111110100000000000000000000000111111111111111111111"};
  char * empty_slime =
"1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111";

  char *circle_31 =
"0000000000000111110000000000000000000000001110000011100000000000000000001100000000000110000000000000001100000000000000011000000000001000000000000000000100000000010000000000000000000001000000010000000000000000000000010000001000000000000000000000001000001000000000000000000000000010000100000000000000000000000100010000000000000000000000000100100000000000000000000000000010010000000000000000000000000000101000000000000000000000000000000110000000000000000000000000000001100000000000000000000000000000011000000000000000000000000000000110000000000000000000000000000010100000000000000000000000000001001000000000000000000000000000010010000000000000000000000000000100010000000000000000000000000001000010000000000000000000000000100000100000000000000000000000010000001000000000000000000000000100000001000000000000000000000001000000000100000000000000000000100000000000110000000000000001100000000000000110000000000000011000000000000011000000000000000000111000001110000000000000000000000000111110000000000000";
  char *a_15 =
"000000000000000000000010000000000000010100000000000001010000000000000101000000000000100010000000000010001000000000001000100000000010000010000000001111111000000001000001000000001000000010000000100000001000000010000000100000000000000000";
  char *b_15 =
"000000000000000000011111100000000001000001000000000100000010000000010000001000000010000000100000000100000001000000000111111000000000100000010000000010000001000000010000000010000000100000001000000010000000111111000000000000000000000000";
```

```c
  char *start_15 =
"000000000000000000001000000000000001100000000000001010000000000001001000000000000001000000000000001000000000000001000000000000001000000000000001000000000000001000000000000001000000000000000100000000000000010000000000011111100000000000000000000";
  char *select_15 =
"0000000000000000000011111000000000001000001000000010000000100000001000000010000000000000010000000000000100000000000001100000000000110000000000000100000000000001000000000000001000000000000001000000000000001111111100000000000000000000";
  char *up_15 =
"0000000000000000000000000000000000000000000000000010000000000000101000000000001000100000000001000100000000010000010000000100000001000000100000001000001000000000100010000000000010011111111111110000000000000000000000000000000";
  char *right_15 =
"0000000000000000110000000000000010100000000000001001100000000001000010000000001000001100000001000000010000001000000001000001000000010000001000001100000001000010000000001001100000000000101000000000000011000000000000000000000000000";
  char *left_15 =
"00000000000000000000000011000000000000101000000000011001000000000100001000000011000001000000100000001000001000000001000000100000001000000011000001000000000100010000000000110010000000000000101000000000000011000000000000000000";
  char *down_15 =
"00000000000000000000000000000011111111111100100000000000100010000000001000001000000010000001000000010000000100000100000000010001000000000001000100000000000010100000000000000100000000000000000000000000000000000000000000000000";
  char *slider_21_2 =  "000011111111111110000111100000000000001111";
  char *arrows_slider_15_15 =
"00000000000000000000000000000000001000000000000011100000000000111110000000001111111000000000000000000000000000000000000000000000000000000011111110000000001111100000000000011100000000000000100000000000000000000000000000000000000000";
  char *delete_15 =
"111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111";

  char *icons[] = { // Tableau d'icônes correspondant aux touches tactiles
    arrows_slider_15_15,
    left_15,
    down_15,
    right_15,
    select_15,
    b_15,
    a_15,
    start_15,
    up_15
  };

  char *m_15_9 =
"11111111111111111111111111111111111111111111111111110001110001111111000111000111111100011100011111110001110001111111000111000111111100011100011 1";
  char *y_9_15 =
"111000111111100011111110001111111000111111100011111110001111111111111111111111111111111110000001110000001110000001110000001110000001110000000111";
  char *T_9_15 =
"11111111111111111111111111111000111000000111000000111000000111000000111000000111000000111000000111000000111000000111000000111000000111000000111000";
  char *o_9_9 =  "111111111111111111111111111111111000111111000111111000111111111111111111111111111111";
  char *u_9_9 =  "111000111111000111111000111111000111111000111111000111111111111111111111111111111111";
  char *c_9_9 =  "111111111111111111111111111111100000011100000011100000011111111111111111111111111111";
  char *h_9_15 =
"111000000111000000111000000111000000111000000111000000111111111111111111111111111111111111000111111000111111000111111000111111000111111000111";
  char *point_2_2 = "1111";
  char *g_9_15 =
"111111111111111111111111111111110001111110001111110001111111111111111111111111111111111100000011100000011100000011111111111111111111111111111111";
  char *b_9_15 =
"111000000111000000111000000111000000111000000111000000111111111111111111111111111111111111000111111000111111000111111111111111111111111111111111";

  int color[] = { 0xFFFF, // White
                  0x001F, // Blue
                  0xF800, // Red
                  0x07E0, // Green
                  0x07FF, // Cyan
                  0xF81F, // Magenta
                  0xFFE0,}; // Yellow
  // Affichage de myTouch.gb
  int logo_x = 117;
  int logo_y = 149;
  mk_ili9225_draw(m_15_9, logo_x, logo_y+6, 15, 9, color[0], 1);
  mk_ili9225_draw(y_9_15, logo_x+21-5, logo_y+6, 9, 15, color[1], 1);
  mk_ili9225_draw(T_9_15, logo_x+31-5, logo_y, 9, 15, color[2], 1);
  mk_ili9225_draw(o_9_9, logo_x+41-5, logo_y+6, 9, 9, color[3], 1);
  mk_ili9225_draw(u_9_9, logo_x+51-5, logo_y+6, 9, 9, color[4], 1);
  mk_ili9225_draw(c_9_9, logo_x+61-5, logo_y+6, 9, 9, color[5], 1);
  mk_ili9225_draw(h_9_15, logo_x+71-5, logo_y, 9, 15, color[6], 1);
  mk_ili9225_draw(point_2_2, logo_x+81-5, logo_y+13, 2, 2, color[0], 1);
  mk_ili9225_draw(g_9_15, logo_x+84-5, logo_y+6, 9, 15, color[1], 1);
  mk_ili9225_draw(b_9_15, logo_x+94-5, logo_y, 9, 15, color[3], 1);
  // Affichage des touches bind en haut
  mk_ili9225_text("    :move", 5,8,0xFFFF,0x0000); //< >
  mk_ili9225_text("  :jump", 5,23,0xFFFF,0x0000); // slider
  mk_ili9225_text("       :color", 110,23,0xFFFF,0x0000); // 1
  mk_ili9225_text("3x  :Game Boy", 110,8,0xFFFF,0x0000); // 2
  // Affichage des icone des touches bind en haut
  mk_ili9225_draw(left_15, 5, 0, 15, 15, 0x07E0, 1);
  mk_ili9225_draw(right_15, 22, 0, 15, 15, 0x07E0, 1);
  mk_ili9225_draw(arrows_slider_15_15, 5, 15, 15, 15, 0x07E0, 1);
  mk_ili9225_draw(a_15, 153, 15, 15, 15, 0x07E0, 1);
  mk_ili9225_draw(select_15, 128, 0, 15, 15, 0x07E0, 1);
  uint8_t current_frame = 0;  // Frame (0-9) de l'animation du slime
  uint8_t previous_frame = 0;  // Frame précédente (0-9) de l'animation du slime
  int direction = 1;             // Direction va et vient : frames 0 1 2 3 4 5 6 7 8 7 6 5 4 3 2 1 2 3 4 ...
  int coo[2] = {50, 110};  // x y du slime
  int previous_coo[2] = {coo[0],  coo[1]}; // x y
  int current_state_slime = 0;  // -1 left, 0, idle, 1 right, 2 jump
  int previous_state_slime = 0; // -1 left, 0, idle, 1 right, 2 jump

  char buffer[MAX_BUFFER_SIZE]; // Buffer UART

  int previous_state_select = 0; // Etat de la touche select pour l'entrée dans la game boy
  int cuontown_before_GB = 3;// Compteur d'appui de select avant l'entrée sur la game boy. dé-incrémente à chaque appui

  int mappedValue = 0;     // Valeurs de la pression émise sur une touche (%)
  int previous_jump_frame = 0; // Correspond aux différentes hauteur possible du jump 0 à 9
  int jump_frame = 0;    // Correspond aux différentes hauteur possible du jump 0 à 9

  int slime_color = 0;    // Correspond aux différentes couleurs, de 0 à 6
  int previous_state_A = 0;    // La touche A fait défiler les couleurs

  int speed_slime=1;     // 1 slow, 2 fast

  int jump_direction = -1;     // -1 vers le haut, +1 vers le bas
  int initial_y = coo[1];    // Position hauteur du slime
  int jump_height = 0;    // Hauteur en pixel pendant le saut
  int acceleration = 0;    // Pixel de rajout/moins simulant l'accélération/décélération à chaque frame

  while (NATIVE_UI){    // Loop de l'interface slime
    clock = (time_us_32()/clock_period)%2;  // Horloge de période clock_period µs, T = 100ms
    if (clock != previous_clock){   // Tous les fronts montants
      time ++;
      if (onReading){       // Si une donnée UART est arrivée
        onReading = false; // Remise à 0 du flag
        char str[30]  = "\0";
        // Si appui sur select, décompte décrémenté de 1
        if ((previous_state_select == 0)&&(uart_buttons_states[5]==1)){ // select front montant
          cuontown_before_GB --;
        }
        previous_state_select = uart_buttons_states[5];
```

```
315
316         // Affichage des icones des boutons appuyés en temps réel
317         mk_ili9225_draw(delete 15, 60, 152, 15, 15, 0x0000, 1);
318         for (int i = 0; i < 9; ++i) {
319           if (uart_buttons_states[i + 1]) {
320             mk_ili9225_draw(icons[i], 60, 152, 15, 15, uart_buttons_states[i + 1] * 0x07E0, 1);
321             if(i + 1 != 5){ // On en profite pour réinitialiser le décompte si ce n'est pas select
322               cuontown_before_GB = 3;
323             }
324             break;
325           }
326         }
327         sprintf(str,"%dx",cuontown_before_GB);       // Formatage de la donnée du compteur de int à char
328         mk_ili9225_text(str, 110,8,0xFFFF,0x0000); // Affichage de décompteur
329
330         int payloadInt = atoi(payload);  // Conversion des 4 dernières valeurs de la trame en entier (ADC) : 2789
331         mappedValue = ((payloadInt - 2200) * 100) / (4095 - 2200); // Mise à l'échelle de la valeur ADC
332
333         if ((mappedValue >= 0)&&(mappedValue <= 100)) {  // Si une valeur ADC est existante
334           switch(touchID){ // Préparation de texte du bouton correspondant pour l'afficher
335             case '1' : sprintf(str, "slider"); break;
336             case '2' : sprintf(str, "left"); break;
337             case '3' : sprintf(str, "down"); break;
338             case '4' : sprintf(str, "right"); break;
339             case '5' : sprintf(str, "select"); break;
340             case '6' : sprintf(str, "B"); break;
341             case '7' : sprintf(str, "A"); break;
342             case '8' : sprintf(str, "start"); break;
343             case '9' : sprintf(str, "up"); break;
344           }
345           // Affichage en bas à gauche du texte de la touche : "start"
346           mk_ili9225_text("     ",5,150,0x0000,0x0000);
347           mk_ili9225_text(str,5,150,0xFFFF,0x0000);
348           // Affichage en bas à gauche de la valeur ADC de la touche : "59%"
349           sprintf(str, "%d%%", mappedValue);
350           mk_ili9225_text("    ",5,163,0x0000,0x0000);
351           mk_ili9225_text(str,5,163,0xFFFF,0x0000);
352         }
353       }
354       // Balayage va et vient des 10 frames du slime -> une période de 20 frames d'animations meme frames
355       current_frame = (current_frame + direction + 10) % 10;
356       if (current_frame == 9 || current_frame == 0) {
357         direction = -direction; // Inversion du sens des frames : 0123456789876543210
358       }
359       // Si on passe d'un état à un autre (jump-idle-left), on nettoie les pixels résiduels par des rectangles pleins
360       if (current_state_slime != previous_state_slime){
361         mk_ili9225_draw(slime idle[previous frame], previous coo[0], previous coo[1], 16, 16, 0x0000, previous state slime); // silouette idle
362         mk_ili9225_fill_rect(previous coo[0]-3, previous coo[1]+16, 3, 1, 0x0000); // bave à gauche
363         mk_ili9225_fill_rect(previous coo[0]+17, previous coo[1]+16, 3, 1, 0x0000); // bave à droite
364         mk_ili9225_draw(slime jump[previous frame], previous coo[0]-3, previous coo[1]+4, 22, 12, 0x0000, previous state slime); // silouette jump
365
366       }
367       // Affichage des frames selon l'état du slime
368       switch(current_state_slime){
369         case 0 : // Idle
370           mk_ili9225_draw(slime idle[previous frame], previous coo[0], previous coo[1], 16, 16, 0x0000, -1);
371           mk_ili9225_draw(slime idle[previous frame], previous coo[0], previous coo[1], 16, 16, 0x0000, 1);
372           mk_ili9225_draw(slime idle[previous frame], previous coo[0], previous coo[1], 16, 16, 0x0000, previous state slime);
373           mk_ili9225_draw(slime idle[current frame], coo[0], coo[1], 16, 16, color[slime color], current state slime);
374           break;
375         case 1 : // Si droite, déplacement + bave au sol
376           coo[0] += current_state_slime*speed_slime;
377           mk_ili9225_draw(slime idle[previous frame], previous coo[0], previous coo[1], 16, 16, 0x0000, previous state slime);
378           mk_ili9225_draw(slime idle[current frame], coo[0], coo[1], 16, 16, color[slime color], current state slime);
379           mk_ili9225_fill_rect(previous coo[0]-3, previous coo[1]+16, 3, 1, 0x0000);
380           mk_ili9225_fill_rect(coo[0]-3, coo[1]+16, 3, 1, color[slime color]);
381           mk_ili9225_fill_rect(coo[0]-3, coo[1]+18, 2, 1, color[slime color]);
382           if(coo[0]%3 == 0){mk_ili9225_fill_rect(coo[0]-3, coo[1]+18, 1, 4, color[slime color]);}
383           else if(coo[0]%2 == 0){mk_ili9225_fill_rect(coo[0]-3, coo[1]+18, 1, 3, color[slime_color]);}
384           else{mk_ili9225_fill_rect(coo[0]-3, coo[1]+18, 1, 2, color[slime_color]);}
385           break;
386         case -1 : // Si gauche, déplacement + bave au sol
387           coo[0] += current state slime*speed slime;
388           mk_ili9225_draw(slime idle[previous frame], previous coo[0], previous coo[1], 16, 16, 0x0000, previous state slime);
389           mk_ili9225_draw(slime idle[current frame], coo[0], coo[1], 16, 16, color[slime color], current state slime);
390           mk_ili9225_fill_rect(previous coo[0]+17, previous coo[1]+16, 3, 1, 0x0000);
391           mk_ili9225_fill_rect(coo[0]+17, coo[1]+16, 3, 1, color[slime_color]);
392           mk_ili9225_fill_rect(coo[0]+19, coo[1]+18, 2, 1, color[slime_color]); //tracé horizontale
393           if(coo[0]%3 == 0){mk_ili9225_fill_rect(coo[0]+19, coo[1]+18, 1, 4, color[slime_color]);}
394           else if(coo[0]%2 == 0){mk_ili9225_fill_rect(coo[0]+19, coo[1]+18, 1, 3, color[slime_color]);}
395           else{mk_ili9225_fill_rect(coo[0]+19, coo[1]+18, 1, 2, color[slime_color]);;}
396           break;
397         case 2 : // Jump
398           mk_ili9225_draw(slime jump[previous jump frame], previous coo[0]-3, previous coo[1]+4, 22, 12, 0x0000, previous state slime);
399           mk_ili9225_draw(slime jump[jump frame], coo[0]-3, coo[1]+4, 22, 12, color[slime color], current state slime);
400           break;
401         case 3 : // Color
402           slime_color = (slime_color + 1) % 7;
403           break;
404         case 4 : // On sky after jump
405           coo[1] += (1 * jump_direction) + (acceleration * jump_direction); // actualisation de la coo y
406           // Tous les 10 pixels, il y a accélération / décélération
407           if ((coo[1]-(initial y-jump height))%10 <= 10){
408             acceleration += 1 * jump direction;
409           }
410           // Si arrivé au sommet ou au sol
411           if (coo[1] <= initial_y-jump_height || coo[1] >= initial_y) {
412             if (coo[1] >= initial_y) { // Si arrivé au sol, le saut est terminer, on passe en mode idle
413               current_state_slime = 0;
414               coo[1]=initial y;
415               jump direction = 1;
416             }
417             jump direction *= -1; // Inversion de la direction
418           }
419           // On peut bouger en plein air
420           if (uart_buttons_states[4]){ // Right
421             coo[0] += 1*speed_slime;
422             mk_ili9225_draw(slime idle[previous frame], previous coo[0], previous coo[1], 16, 16, 0x0000, -1);
423             mk_ili9225_draw(slime idle[previous frame], previous coo[0], previous coo[1], 16, 16, 0x0000, 1);
424             mk_ili9225_draw(slime idle[current frame], coo[0], coo[1], 16, 16, color[slime color], 1);
425             previous state slime = 1;
426           }
427           else if (uart_buttons_states[2]){ // Left
428             coo[0] -= 1*speed_slime;
429             mk_ili9225_draw(slime_idle[previous_frame], previous_coo[0], previous_coo[1], 16, 16, 0x0000, -1);
430             mk_ili9225_draw(slime_idle[previous_frame], previous_coo[0], previous_coo[1], 16, 16, 0x0000, 1);
431             mk_ili9225_draw(slime_idle[current_frame], coo[0], coo[1], 16, 16, color[slime_color], -1);
432           }
433           else { // Sinon le saut est vertical
434             mk_ili9225_draw(slime idle[previous frame], previous coo[0], previous coo[1], 16, 16, 0x0000, -1);
435             mk_ili9225_draw(slime idle[previous frame], previous coo[0], previous coo[1], 16, 16, 0x0000, 1);
436             mk_ili9225_draw(slime_idle[current_frame], coo[0], coo[1], 16, 16, color[slime_color], current_state_slime);
437           }
438           break;
439       }
```

```
      // Si le slider a été relaché alors on saute
      if ((previous_state_slime==2)&&(current_state_slime !=2)){
        mk_ili9225_draw(empty_slime, previous_coo[0]-3, previous_coo[1]+4, 22, 12, 0x0000, previous_state_slime);
        current_state_slime = 4; //on sky
      }
      // A l'atterrissage, on efface les résidus
      if ((previous_state_slime==4)&&(current_state_slime !=4)){
        mk_ili9225_draw(empty_slime, coo[0], coo[1], 22, 12, 0x0000, previous_state_slime);
        mk_ili9225_draw(slime_idle[previous_frame],coo[0], coo[1], 16, 16, 0x0000, -1);
        mk_ili9225_draw(slime_idle[previous_frame], coo[0], coo[1], 16, 16, 0x0000, 1);
      }
      // Fin de la loop tous les fronts montants, mémorisations de quelques états
      previous_frame = current_frame;
      previous_jump_frame = jump_frame;
      previous_coo[0] = coo[0];
      previous_coo[1] = coo[1];
      previous_state_slime = current_state_slime;
    }
    // Nous voilà dans la loop, hors des FPS toutes les 100ms
    // Si la valeur ADC en % est en dessous de 50% alors vitesse lente au mouvement horizontale
    if(mappedValue<51){speed_slime = 1;}
    // Si la valeur ADC en % est en dessous de 100% alors vitesse rapide au mouvement horizontale
    else if(mappedValue<101){speed_slime = 2;}
    // Si le slime n'est pas en l'air suite au jump, des touches sont bind
    if (current_state_slime !=4){
      if (cuontown_before_GB == 0){   // Si select 3x, alors entrée dans le menu de jeux
        NATIVE_UI = 0;  // Sortie de l'interface avec le slime (while(NATIVE_UI))
        irq_set_enabled(UART_IRQ, false);   // Désactivation de l'UART
        uart_set_irq_enables(UART_ID, false, false); // Now disable the UART to send interrupts - RX only
      }
      if (uart_buttons_states[4]){      // Right
        current_state_slime = 1;
      }
      else if (uart_buttons_states[2]){ // Left
        current_state_slime = -1;
      }
      else if (uart_buttons_states[1]){ // Jump
        current_state_slime = 2;
        if(mappedValue<11){jump_frame = 0;}
        else if(mappedValue<21){jump_frame = 1; jump_height = 10; acceleration = jump_height/10;}
        else if(mappedValue<31){jump_frame = 2; jump_height = 20; acceleration = jump_height/10;}
        else if(mappedValue<41){jump_frame = 3; jump_height = 30; acceleration = jump_height/10;}
        else if(mappedValue<51){jump_frame = 4; jump_height = 40; acceleration = jump_height/10;}
        else if(mappedValue<61){jump_frame = 5; jump_height = 50; acceleration = jump_height/10;}
        else if(mappedValue<71){jump_frame = 6; jump_height = 60; acceleration = jump_height/10;}
        else if(mappedValue<81){jump_frame = 7; jump_height = 70; acceleration = jump_height/10;}
        else if(mappedValue<91){jump_frame = 8; jump_height = 80; acceleration = jump_height/10;}
        else if(mappedValue<101){jump_frame = 9; jump_height = 90; acceleration = jump_height/10;}
      }
      else if ((uart_buttons_states[7]) && (time%5 == 0)){ // A (color)
        current_state_slime = 3;
      }
      else { // Idle
        current_state_slime = 0;
      }
    }
    // Mémorisation de certaines données à la fin de la loop
    previous_clock = clock;
    tight_loop_contents();
    if(time>1000){time=0;} // Pour éviter te saturer le tampon
  }
}



int main(void)
{
  /* ------------ Peanut-GB emulator ------------
  setup variable (struct, gb, overclock, GPIO)
  setup libraries
  set overclock at 266MHz
  set pullup gpio
  set SPI clock at HF
  initialise Sound
  */

  while(true)
  {

    #if ENABLE_LCD               // By Peanut-GB emulator
      #if ENABLE_SDCARD          // By Peanut-GB emulator
        mk_ili9225_init();       // By Peanut-GB emulator
        mk_ili9225_fill(0x0000); // By Peanut-GB emulator
        native_user_interface(); // Par moi : à cette ligne l'interface avec le slime commence
        rom_file_selector();     // By Peanut-GB emulator
      #endif
    #endif

    /* ------------ Peanut-GB emulator ------------
    Initialise GB context (error, extension, rom size, title,)
    get state of buttons
    in rom file selector
    ...
    */
  }

}
```