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

Console à touches électrostatiques

Cours de **Microcontrôleurs – 4EN0302**

myTouch.gb

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

Professeure :
Mme A. DEGEEST

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

1 Introduction

Ce rapport expose les résultats d'une étude réalisée dans le cadre du cours de Microcontrôleurs du premier quadrimestre du master en ingénierie industrielle, option électronique, à l'Institut Supérieur Industriel de Bruxelles (*ISIB*). L'objectif de cette investigation est la programmation d'un prototype basé sur des microcontrôleurs, baptisé '*myTouch.gb*', qui s'inspire de l'esthétique et des fonctionnalités des consoles *rétro-game*, en particulier la *Game Boy*. Ce prototype a été conçu dans le cadre d'un autre cours associé : Prototypage Industriel.

Au fil de ce document seront explorés les diverses phases du processus de développement, mettant en lumière la programmation des microcontrôleurs, décrivant les défis spécifiques rencontrés dans le cadre du développement logiciel.

Le rapport offre également un regard approfondi sur les composants clés des microcontrôleurs, les stratégies d'optimisation adoptées, et les enseignements tirés des tests pratiques, notamment lors de l'événement *Brotaru* où le prototype a été soumis à un public diversifié. L'accent est mis sur la démarche technique, mettant en avant les principes fondamentaux de l'électronique, de la conception de circuits à la programmation des microcontrôleurs.

2 Description du projet

Le projet, intitulé "*myTouch.gb*", s'inspire de l'esprit d'une console de jeu rétro, arborant une interface similaire à celle de la *Game Boy*. Il a été présenté lors du *Brotaru*, un événement axé sur les jeux vidéo, organisé par l'établissement scolaire le 4 décembre 2023.

2.1 Cahier des charges

Le projet vise la création d'un banc de test doté d'interfaces tactiles sur un *PCB* monocouche. Il est impératif de respecter un budget prédéfini, mettant ainsi l'accent sur la minimisation des coûts totaux du projet. Ces contraintes imposent des limitations à la complexité du *PCB* et au nombre de composants électroniques. L'optimisation doit être intégrée dès le début du processus de conception pour atteindre cet objectif financier. La direction spécifique vers laquelle le projet évoluera reste flexible, pourvu que les directives établies soient scrupuleusement respectées.

2.2 Architecture électronique proposée (en bref)

Des touches tactiles électrostatiques ont été dimensionnées pour répondre aux exigences tactiles du cahier des charges. Il était essentiel de prendre en considération le comportement électrostatique d'un doigt humain et de trouver une solution compatible avec la diversité humaine. En raison de divers facteurs physiologiques et environnementaux, chaque individu peut présenter un potentiel électrique cutané différent.

Étant donné que la densité de charges électriques dans un doigt humain varie considérablement d'une personne à l'autre, une solution a été expérimentée en laboratoire et appliquée au projet pour neutraliser ces variations.

Une touche tactile devra nécessiter deux électrodes : une de polarisation et une de réception. En polarisant le doigt à une tension connue (+5V), une référence de potentiel commune pour les utilisateurs est établie. Cette méthode permet de normaliser le potentiel électrique du doigt à une valeur connue, facilitant ainsi la mesure du côté réception en offrant une base de référence constante et prévisible pour tout utilisateur. Selon la pression exercée sur une touche tactile, il y aura une conduction plus ou moins forte. Une lecture ADC sera envisageable suivant la pression.

Après la conception de l'architecture d'une touche tactile, des montages électroniques d'amplification ont été dimensionnés pour acquérir les données.

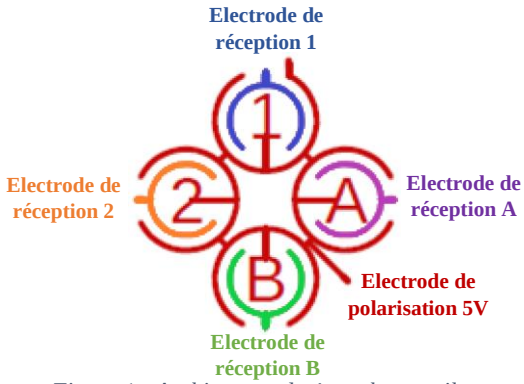


Figure 1 : Architecture de 4 touches tactiles

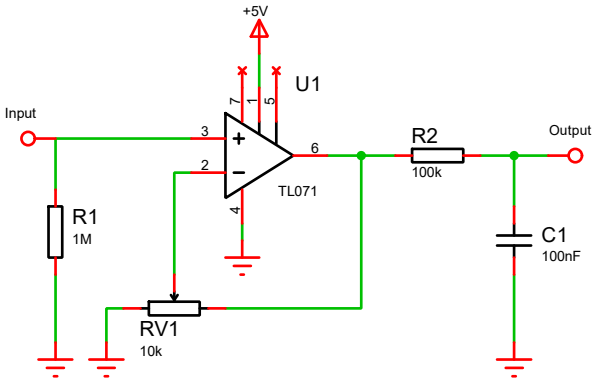


Figure 2 : Montage électronique par touche tactile

En *Input* du schéma électronique présent dans la figure 2 est connectée une électrode de réception d'une touche tactile (figure 1). La *R1* a été dimensionnée pour négliger le spectre sinusoïdal de 50Hz rayonné par le bâtiment tout en tenant compte de l'impédance moyenne d'un doigt, située entre 10k et 100kΩ. La *RV1* corrige le gain effectué par l'*AOP* pour régler la sensibilité à l'appui. Un filtre passe-bas avec une fréquence de coupure de 15Hz a été installé en sortie par *R2* et *C1* pour apporter une plus grande fluidité dans le signal. Une lecture *ADC* peut être effectuée en Output.

2.3 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 le *PCB* et 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*.

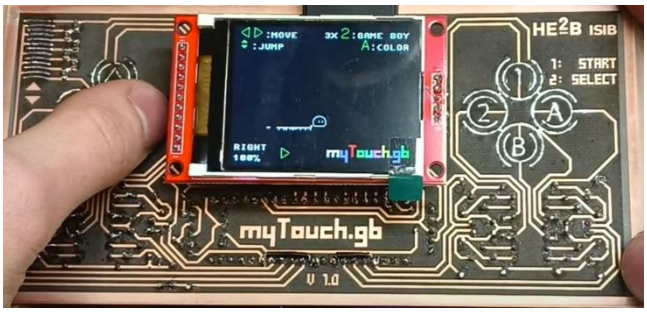


Figure 3 : Démonstration du déplacement

Voici une deuxième démonstration (saut) :

Figure 4 : Démonstration 1 du saut

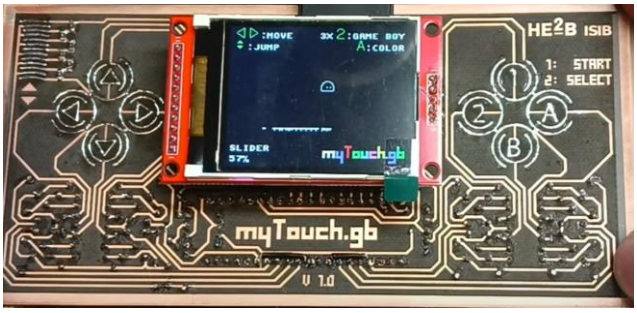


Figure 5 : Démonstration 2 du saut

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) :

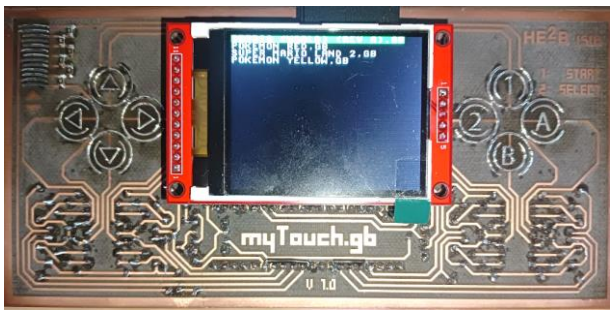


Figure 6 : Démonstration 1 de l'émulateur

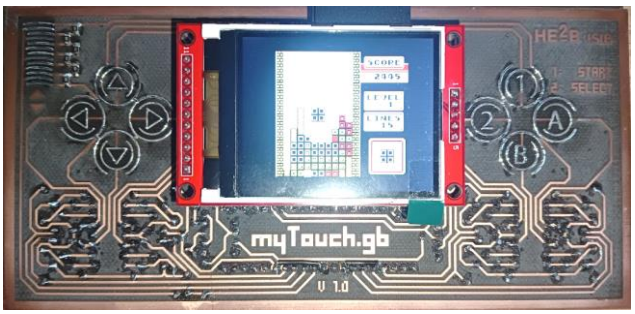


Figure 7 : Démonstration 2 de l'émulateur

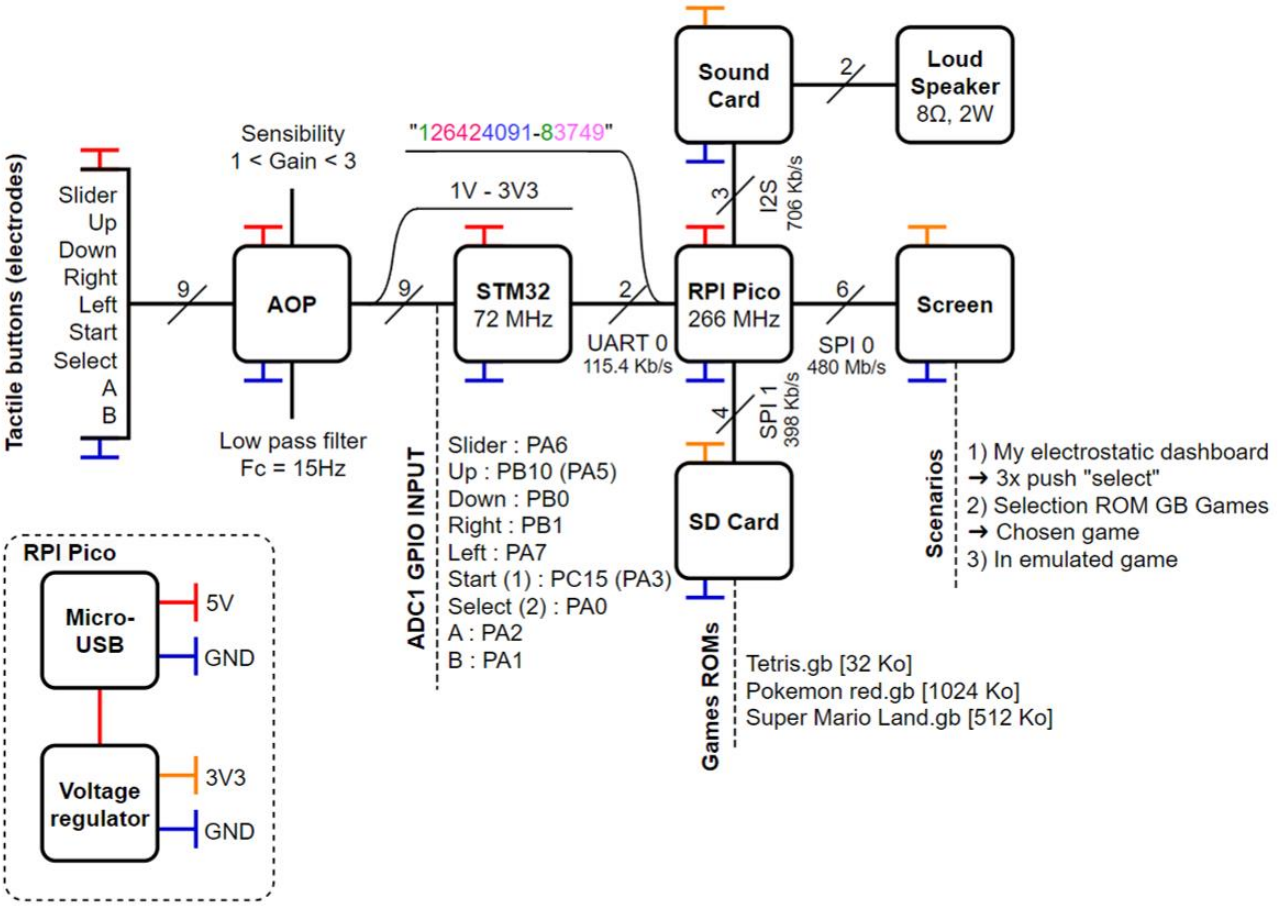
En effectuant trois pressions successives sur la touche *2* (*SELECT*), une nouvelle page se dévoile. Il s'agit du menu où la sélection des jeux vidéo téléchargés dans la carte *SD* s'opère (voir figure 15). Le défilement des jeux s'effectue de haut en bas, et l'entrée dans un jeu particulier se réalise 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).

3 Systèmes embarqués

Les systèmes embarqués dans le prototype ne sont pas nombreux, mais ils ont rempli les exigences du cahier des charges. Chaque système embarqué sera cité dans cette section.

3.1 Organigramme

Cet organigramme résume le circuit électronique de la console (annexes 1 & 2) ainsi que les diverses interfaces logicielles établies :



Le prototype utilise deux microcontrôleurs, à savoir un *Raspberry Pi Pico* et un *STM32*, qui interagissent par le biais de l'*UART*.

La carte *SD* et l'écran sont commandés selon le protocole *SPI*, mais utilisent deux bus distincts. L'unification de ces deux composants sous un même bus n'aurait pas été une solution optimale. En effet, l'écran nécessite une fréquence considérablement élevée pour générer 30 images par seconde (*FPS*), tandis que 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, ce qui 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 12MHz. En d'autres termes, pour les 9 entrées, chacune bénéficie d'une bande passante d'environ 1,33MHz lorsqu'elle est échantillonnée par l'*ADC1*. Cette fréquence reste élevée et n'impacte aucune latence dans le prototype.

3.2 Raspberry Pi Pico

Le *Raspberry Pi Pico* est programmé directement depuis son port micro-*USB* via le transfert d'un fichier compilé au format *uf2*. Ce microcontrôleur est programmé dans l'éditeur *Visual Studio Code* de *Microsoft*. Pour ce dernier, une manipulation assez minutieuse [2] 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*.

Le *Raspberry Pi Pico* assume des tâches lourdes. Il contrôle l'écran, la carte *SD* et le son. La bibliothèque [1] utilisée pour l'émulation de jeux *Game Boy* 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*.

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. 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. Un total de 500 lignes de codes a été ajoutés.

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.

Caractéristiques principales du microcontrôleurs :

Le *Raspberry Pi Pico* est un microcontrôleur développé par la *Fondation Raspberry Pi*. Il est basé sur le processeur *RP2040*, dispose de 2 cœurs *ARM Cortex-M0+* cadencés à 133MHz, offre 264Ko de mémoire *RAM*. Le Raspberry Pi Pico ne possède pas de mémoire *ROM*, il utilise une méthode de démarrage appelée « *USB mass storage boot* » de 2Mo. Ce microcontrôleur possède uniquement 3 entrées *GPIO ADC*.

3.3 STM32

Sur le meme bus UART, une extension *USB* vers *TTL* (*CP2102*) est utilisée pour programmer le *STM32*. Ce microcontrôleur s'agit d'un *STM32F103C8T6.* Ce microcontrôleur est programmé dans l'environnement de développement de *STMicroelectronics* appelé *STM32CubeIDE*.

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 en UART 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 d'environ 100 lignes de code.

Caractéristiques principales du microcontrôleurs :

Le *STM32F103C8T6*, également connu sous le nom de *Blue Pill*, est basé sur le noyau *ARM Cortex-M3*. Il dispose d'une fréquence d'horloge maximale de 72MHz, d'une mémoire Flash de 64Ko pour le stockage du programme, et de 20Ko de *RAM* pour les données. Ce microcontrôleur possède 10 entrées *GPIO ADC*.

4 Analyse des codes

Aux annexes 3 et 4 se trouvent les codes des microcontrôleurs. Il s'agit uniquement des lignes de code personnellement rédigées, en dehors des lignes de code générées par le système *STM32CubeIDE* dans le cas du *STM32* et en dehors de celles rédigées par le développeur de la bibliothèque émulatrice dans le cas du *Raspberry Pi Pico*.

Afin de maintenir la cohérence avec les lignes de code non rédigées par moi, quelques lignes essentielles ont été conservées, tandis d'autres ont été supprimées mais commentées à leur même position.

4.1 Raspberry Pi Pico main.c

Avant d'établir une fonction quelconque, tout commence dans la fonction *main* située à la ligne 505 :

```
505 int main(void)
506 {
507   /* ------------ Peanut-GB emulator ------------
508   setup variable (struct, gb, overclock, GPIO)
509   setup libraries
510   set overclock at 266MHz
511   set pullup gpio
512   set SPI clock at HF
513   initialise Sound
514   */
515
516   while(true)
517   {
518
519     #if ENABLE_LCD               // By Peanut-GB emulator
520       #if ENABLE_SDCARD          // By Peanut-GB emulator
521         mk_ili9225_init();       // By Peanut-GB emulator
522         mk_ili9225_fill(0x0000); // By Peanut-GB emulator
523         native_user_interface(); // Par moi : à cette ligne l'interface avec le slime commence
524         rom_file_selector();     // By Peanut-GB emulator
525       #endif
526     #endif
527
528     /* ------------ Peanut-GB emulator ------------
529     Initialise GB context (error, extension, rom size, title,)
530     get state of buttons
531     in rom file selector
532     ...
533     */
534   }
535
536 }
```

Dans cette fonction se trouve le setup suivi de la boucle infinie de l'émulateur *Game Boy*. Pour maintenir la cohérence dans le projet, les lignes ont été supprimées mais commentées à la place. Dans le setup se trouvaient donc les diverses initialisations, comme celles des librairies *GPIO*, *SPI*, *I2S*, etc. Dedans se trouvait également l'*overclocking* du microcontrôleur à 266MHz. Dans la boucle infinie, l'algorithme commence par vérifier l'état du débogage afin d'exécuter certaines fonctions associées si l'opérateur l'a permis. Si les *flags* ont été mis à 1, alors il y a initialisation de l'écran, puis entrée dans l'interface utilisateur native (ligne 523), puis entrée dans l'interface de sélection de jeu (ligne 524) depuis les *ROMs Game Boy*.

Justement, l'entrée dans l'interface utilisateur native est une ligne que j'ai ajoutée (ligne 523). Elle correspond à l'interface conçue dans le cadre de ce cours, avec le *slime*. En effet, comme précisé dans la section *2.3 Démonstration*, après cette interface se trouve une autre page, celle de sélection de jeu. Bref, après avoir sélectionné la *ROM* dans la liste, il y a initialisation de la *ROM* (à partir de la ligne 528).

Tout le code développé dans le cadre de ce cours se trouve donc dans la fonction *native_user_interface()*, voici sa localisation :

```
166 void native_user_interface(){
167  /* Gère l'interface avec le slime avec les interactions Homme-Machine des touches tactiles
168   * Se compose d'un équivalent "setup" et "loop"
169   */
170   uart_init(UART_ID, BAUD_RATE);                              // Set up our UART with a basic baud rate.
171   gpio_set_function(UART_TX_PIN, GPIO_FUNC_UART);             // Set the TX and RX pins by using the function select on the GPIO
172   gpio_set_function(UART_RX_PIN, GPIO_FUNC_UART);             // Set datasheet for more information on function select
173   int __unused actual = uart_set_baudrate(UART_ID, BAUD_RATE);
174   uart_set_hw_flow(UART_ID, false, false);                    // Set UART flow control CTS/RTS, we don't want these, so turn them off
175   uart_set_format(UART_ID, DATA_BITS, STOP_BITS, PARITY);     // Set our data format
176   uart_set_fifo_enabled(UART_ID, false);                      // Turn off FIFO's - we want to do this character by character
177   int UART_IRQ = UART_ID == uart0 ? UART0_IRQ : UART1_IRQ;    // Set up a RX interrupt.
178   irq_set_exclusive_handler(UART_IRQ, on_uart_rx);            // And set up and enable the interrupt handlers
179   irq_set_enabled(UART_IRQ, true);
180   uart_set_irq_enables(UART_ID, true, false);                 // Now enable the UART to send interrupts - RX only
181   bool NATIVE_UI = 1;                                         // Flag permettant de rester ou sortir de l'interface avec le slime
182   bool previous_clock;                                        // Cadence la loop, et donc les FPS de l'interface
183   bool clock;                                                 // Cadence la loop, et donc les FPS de l'interface
184   int clock_period = 100000;                                  // 100ms de rafraichissement
185   int time;                                                   // Cadence quelques mécaniques dans la loop
```

Cette fonction s'articule autour d'un *setup* et d'une *loop* également. Au *setup* se trouve l'initialisation de l'*UART* avec interruption. À chaque donnée reçue en *UART*, une fonction *on_uart_rx()* est appelée (ligne 178). Quelques autres variables sont initialisées, celles permettant les animations du *slime*, par exemple.

Avant d'aller plus loin, il est intéressant d'introduire l'explication de la fonction *on_uart_rx()*. La voici :

```
47  uint8_t char_d[11] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 61}; // Caractère uint8_t correspondant à 0123456789-
48  char buff[100]= ".";  // Chaine utilisée correspondant au payload UART avec séparation de caractère, et défilement des caractère
49  bool onReading = false; // Flag déterminant si nous sommes en lecture ou pas
50  char payload[10] = "\0"; // Représentant les 4 dernier caractère du buff[100]
51  char touchID = '\0';     // Initialisation de touchID à un caractère nul, pour assurer un tampon vide (bug sinon)
52
53  void on_uart_rx() {
54  onReading = true; // Flag activé, en pleine lecture
55
56    while (uart_is_readable(UART_ID)) {
57      uint8_t ch = uart_getc(UART_ID);
58      char c[2];
59      if (chars_rxed % 2 == 0) {  // Chaque trame reçue a une longueur de 2 caractères (char utile et char de fin)
60        for (int i = 0; i < 11; i++) {
61          if (ch == char d[i]) {
62            if (i <= 9) sprintf(c, "%d", i); // Le caractère de front montant et de pression assigné à un chiffre (0-9).
63                                             // la chaine reçu a un format non ASCII, il a un autre format dans le "char d"
64            if (i == 10) sprintf(c, "%c", '.'); // Le caractère de front montant est assigné à un "."
65            if (chars_rxed / 2 < 20) {
66                        buff[chars_rxed / 2] = c[0];    // Buff est limité à 20 char. exemple : "449.324763421.22676."
67            } else {
68              for (int j = 0; j < 19; j++) {  // Si buff dépasse les 20 char, décalage à gauche pour les nouv. données
69                buff[j] = buff[j + 1];
70              }
71              buff[19] = c[0];
72            }
73          }
74        }
75      }
76      chars_rxed++; // A chaque trame, 2 caractère sont envoyé, dont un inutile (le dernier), donc à chaque trame -> +2
77    }
```

Située à la ligne 53, cette fonction permet de remplir un *buffer* avec les données reçues. Cette trame est composée de l'identifiant de la touche tactile pressée, de la valeur *ADC* correspondante (proportionnelle à la pression exercée) et d'un caractère de séparation. Le *buffer* est traité dans la boucle while.

En sortie de cette boucle, après la ligne 77, le *buffer* est complet (*buff = "449.324763421.22676."*). À chaque nouvelle donnée, toutes les données (caractères du *buff*) se décalent vers la gauche, laissant ainsi de la place aux nouvelles données à droite. Chacune de ces données est ensuite traitée pour en extraire les informations utiles.

Après la ligne 77, la fonction extrait les données utiles du buffer. La première donnée extraite est copiée dans la variable *payload*. Il s'agit des 4 derniers caractères chiffrés du *buffer*. Cela correspond à la valeur *ADC* sur 10 bits (0 < *payload* < 4091). Dans le cas de *buff = "449.324763421.22676."*, *payload* serait 2676.

La deuxième donnée extraite est l'identifiant de la touche tactile appuyée, exprimée sous la variable *touchID*. Dans le cas de *buff = "449.324763421.22676."*, *touchID* serait 2. Il s'agit du premier chiffre après le caractère de séparation '.' et avant un groupe de 4 chiffres correspondant à la valeur *ADC*.

Ces deux données extraites (l'*ID* et l'*ADC*) sont ensuite assignées à des tableaux de variables pour faciliter leur utilisation dans l'interface avec le personnage *slime*. Cette interface en question, située dans la fonction *native_user_interface()* à la ligne 166, est assez conséquente avec les quelques 350 lignes qu'elle compose.

En résumé, pour rappel, dedans se trouve un équivalent *setup* et *loop*. Dans le setup, il y a l'initialisation de l'*UART* comme précédemment expliqué, avec quelques autres variables. Dans la *loop*, il y a toute la mécanique permettant d'animer l'écran en fonction des interactions entre l'Homme et la machine. Cinq statuts du *slime* sont programmés dans la *loop* :

- *Idle* : Ce mode est activé lorsque l'utilisateur n'interagit pas avec la console. Le *slime* ne bouge pas, présentant uniquement une animation passive (animation de respiration du slime).

- *Right/Left* : Ce mode est activé lorsque l'utilisateur appuie sur les touches *RIGHT* et *LEFT* de la console. En plus de l'animation passive, le *slime* se déplace plus ou moins rapidement en fonction de la pression exercée.

- *Jump* : Ce mode est activé lorsque l'utilisateur balaye le *SLIDER* pour écraser le *slime*. Plus le doigt est bas, plus fort le *slime* est comprimé. Une fois relâché, le *slime* effectue un saut. Il n'y a pas d'animation passive dans ce mode.

- *On air* : Ce mode est activé dès que le *SLIDER* est relâché, au moment du saut. En fonction de la position du doigt au moment du relâchement, ce mode calcule la hauteur du saut. Il y a une animation passive en plein air. Durant le saut, le *slime* peut se déplacer à droite et à gauche.

- *Color* : Ce mode est activé en appuyant sur A. Le *slime* ainsi que sa bave changent de couleur.

Dans le *setup* de *native_user_interface()*, se trouve également la création d'icônes à afficher sur l'écran. Ces icônes représentent une matrice de 1 et de 0, correspondant respectivement à des pixels allumés ou éteints. Ces icônes prennent la forme du personnage (*slime*), du logo du projet *myTouch.gb*, des formes des touches tactiles (*A*, *B*, *1*, *2*, *UP*, *DOWN*, …), etc. Pour interpréter et afficher ces dessins matriciels, une fonction a été créée. Grâce à cette fonction, par exemple, le *slime* a une animation passive bouclée de 20 frames, le rendant vivant même à l'arrêt. Voici la fonction en question :

```
134 void mk_ili9225_draw(char *image, int x, int y, int w, int h, uint16_t color, int orientation){
135   /* Dessine sur l'écran les icones personnalisées pixel par pixel
136    * Les icones sont lues sous forme de matrice binaire
137    * Les 0 et 1 sont respectivement des pixel éteint ou allumés.
138    */
139   int x  = x;
140   int y_ = y;
141   int pixels_quantity = w * h;
142   int orientation_x = orientation; // 1 right, -1 left
143   if (orientation_x >=0){
144     for (int pixel = 0; pixel < pixels_quantity; pixel++){
145       if ((pixel % w) == 0){
146         y ++;
147         x  = x;
148       }
149       if (image[pixel] == '1'){
150         mk_ili9225_fill_rect(x_, y_, 1, 1, color);
151       }
152       x_++;
153     }
154   }
155   else if (orientation_x == -1) {
156     for (int row = 0; row < h; row++) {
157       for (int col = w - 1; col >= 0; col--) {
158         if (image[row * w + col] == '1'){
159           mk_ili9225_fill_rect(x_ + (w - 1 - col), y_+1 + row, 1, 1, color);
160         }
161       }
162     }
163   }
164 }
```

La fonction *mk_ili9225_draw()* a 7 arguments. Le premier, *image*, est la chaîne de caractères matricielle. Les deux suivants sont les coordonnées *x* et *y* où l'image doit s'afficher. Les deux suivants sont respectivement la hauteur et la largeur de la chaîne matricielle, *h* et *w*. L'avant-dernier argument est la couleur souhaitée des pixels. Le dernier argument, quant à lui, est l'orientation de l'image, permettant d'effectuer une symétrie orthogonale suivant l'axe de symétrie y. Cet argument permet notamment de retourner le *slime* lorsqu'il va à droite ou à gauche. La fonction parcourt pixel par pixel suivant la longueur et la largeur de la matrice pour afficher en couleur choisie l'icône.

La chaîne matricielle est initialement sous une dimension, il ne s'agit pas d'un tableau à deux dimensions, d'où l'utilisation de la largeur *w* et la hauteur *h* dans la fonction. Il est plus facile de copier-coller une chaîne tout en une vers un code que de devoir le réajuster sous deux dimensions avant de le mettre dans le code.

4.2 STM32 main.c

Le code du *STM32* est plus simple car il n'effectue que des lectures *ADC* suivies d'émissions de trames *UART*. Le fichier *main.c* inclut également un *setup* et une *loop*. Dans le setup, il y a un appel de ressources extérieures et l'initialisation de quelques variables. Dans la *loop*, cadencée toutes les 10ms, il y a en premier lieu la lecture *ADC* :

```
64   while (1)
65   {
66     sConfigPrivate.Rank = ADC_REGULAR_RANK_1;              // Sélectionne le rang de la conversion ADC régulière
67     sConfigPrivate.SamplingTime = ADC_SAMPLETIME_1CYCLE_5;  // Configure le temps d'échantillonnage pour la conversion ADC
68
69     for (int i = 0; i < 9; i++)  // Cette boucle effectue une lecture ADC à la fois pour chacun des 9 canaux
70     {
71       sConfigPrivate.Channel = ADC channels[i];             // Sélectionne le canal ADC à convertir
72       HAL ADC ConfigChannel(&hadc1, &sConfigPrivate);       // Configure le canal ADC
73       HAL_ADC_Start(&hadc1);                                // Démarre la conversion ADC
74       HAL_ADC_PollForConversion(&hadc1, 1000);              // Attends la fin de la conversion ou le dépassement du délai
75       readValues[i] = HAL_ADC_GetValue(&hadc1);             // Lit la valeur convertie
76       HAL_ADC_Stop(&hadc1);                                 // Arrête la conversion ADC
```

À partir de la ligne 71, il y a l'acquisition de données dans un canal *ADC* à la fois. Cette boucle *for* parcourt et traite les données reçues de canal en canal jusqu'à la fin de la boucle *while*. À chaque itération de la boucle *while*, tous les canaux *ADC* ont été parcourus, et les données sont transmises en *UART* au passage si nécessaire.

La transmission *UART* se fait de manière optimisée, afin de ne pas saturer le bus de données ni de déstabiliser le *Raspberry Pi Pico*.

En effet, s'il y a une variation de la donnée lue, et que cette variation suit une tolérance suffisante pour que cela signifie qu'un doigt humain a appuyé, la condition *if* de la ligne 78 est validée :

```
78        if (abs(readValues[i] - previousValues[i]) > ADC toleration)  // Suivant la tolérance ADC, il y a filtrage
79        {
80          switch (ADC channels[i])            // Sélectionne le canal ADC en fonction du cas
81          {
82            case 6: touchID[0] = '1'; break;  // Définit l'identifiant du bouton en fonction (fronts montants + bouton appuyé)
83            case 5: touchID[0] = '9'; break;
84            case 7: touchID[0] = '2'; break;
85            case 8: touchID[0] = '3'; break;
86            case 9: touchID[0] = '4'; break;
87            case 0: touchID[0] = '5'; break;
88            case 1: touchID[0] = '6'; break;
89            case 2: touchID[0] = '7'; break;
90            case 3: touchID[0] = '8'; break;
91          }
92
93          if (readValues[i] < 2200)  // Si la valeur ADC lue est inférieure à 2200 alors cela est considéré comme front descendant
94          {
95            touchID[0] = '-';        // Un caractère de séparation est alors utilisé comme ID
96          }
```

Cette condition va associer la *GPIO* correspondante à l'identifiant *touchID* de la touche tactile en un caractère. Si c'est le *SLIDER*, alors c'est '1', par exemple. À chaque touche son identifiant. À la ligne 93, afin de ne pas tenir compte des offsets générés par les *AOP*, une marge a été ajoutée. Le niveau bas d'une touche est détecté si la valeur *ADC* est inférieure à 2022. Si c'est le cas, alors cela signifie qu'il s'agit d'un front montant.

L'identifiant se transforme donc en un caractère de séparation '-'. Entre les lignes 95 et 126, les signaux tout-ou-rien des touches tactiles sont assignés à des *GPIO* du *STM32*. Cette partie du code n'est pas exploitée par l'interface avec le *slime*. Elle a simplement permis de lire physiquement, broche par broche, les différentes sorties. Par exemple, pour pouvoir allumer une *LED* respective à chaque appui d'une touche tactile. Cette idée a été abandonnée lors de la création du *PCB*.

```
98          if ((touchID[0] != previousTouchID[0])) // Si l'identifiant du bouton a changé depuis la dernière interaction humaine
99          {
100           sprintf(buffer, "%c", touchID[0]);    // Formatte l'identifiant du bouton en chaîne de caractères
101           HAL UART Transmit(&huart1, (uint8_t *)buffer, strlen(buffer), 100);// Transmet UART l'ID si != du précédent : "6" ou "-"
102
103           switch (touchID[0])   // En plus de l'UART, mise à 1 ou 0 des I/O correspondant aux touches game boy
104           {
105             case '9': HAL_GPIO_WritePin(GPIOB, GPIO_PIN_9, 0); break;   // Action associée au bouton 9 (B9 up)
106             case '2': HAL_GPIO_WritePin(GPIOB, GPIO_PIN_7, 0); break;   // Action associée au bouton 2 (B7 left)
107             case '3': HAL_GPIO_WritePin(GPIOB, GPIO_PIN_8, 0); break;   // Action associée au bouton 3 (B8 down)
108             case '4': HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, 0); break;   // Action associée au bouton 4 (B6 right)
109
110             case '5': HAL GPIO WritePin(GPIOA, GPIO PIN 15, 0); break;  // Action associée au bouton 5 (A15 select)
111             case '6': HAL GPIO WritePin(GPIOB, GPIO PIN 3, 0); break;   // Action associée au bouton 6 (B3 B)
112             case '7': HAL GPIO WritePin(GPIOB, GPIO PIN 4, 0); break;   // Action associée au bouton 7 (B4 A)
113             case '8': HAL_GPIO_WritePin(GPIOA, GPIO_PIN_12, 0); break;  // Action associée au bouton 8 (B4 start)
114             case '-':                                  // Action associée un front descendant est détecté
115               HAL_GPIO_WritePin(GPIOB, GPIO_PIN_7, 1);          // Aucun bouton n'est appuyé, donc tout est à 1
116               HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, 1);
117               HAL_GPIO_WritePin(GPIOB, GPIO_PIN_9, 1);
118               HAL_GPIO_WritePin(GPIOB, GPIO_PIN_8, 1);
119
120               HAL GPIO WritePin(GPIOA, GPIO PIN 15, 1);
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         }
```

L'émission de données en *UART* se fait en deux parties pour une donnée. À la ligne 101, le premier caractère de la trame *UART* est envoyé. Ce premier caractère est l'identifiant *touchID*. Il est envoyé s'il est différent du précédent. C'est le protocole défini.

La deuxième donnée à envoyer est la valeur *ADC*, à la ligne 137 :

```
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 }
```

Pour ce faire, il y a un formatage des données de *int* en *char* avec la méthode *sprintf()*. Ensuite, chaque caractère de la valeur *ADC* est envoyé un à un. Il y a toujours 4 caractères dans la valeur *ADC*.

Ainsi, si l'utilisateur a appuyé sur le *SLIDER* (*ID='1'*) et qu'il a effectué un balayage rapide de haut (*ADC=2200*) en bas (*ADC=4091*), puis a relâché son doigt (*ID='-'*), le *buffer* envoyé serait : *"122004091-"* .

5 Conclusion

Des inquiétudes quant au temps nécessaire pour mettre en place toute cette programmation étaient présentes, étant donné que ce type de développement, en particulier le débogage, peut généralement prendre beaucoup de temps. Le choix de l'émulateur n'était initialement pas prévu, mais pour ajouter du *fun* et de la complexité, il a été inclus. Cela aurait pu compromettre le respect de la deadline du projet *Brotaru*, événement orienté jeux vidéo organisé le 4 décembre 2023 par l'établissement scolaire. Cependant, un week-end, 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. Plus de 500 lignes de code ont été ajoutées à la bibliothèque émulatrice.

La partie la plus stimulante du processus de codage a été la mise en place de la liaison *UART*. Le protocole élaboré pour la communication *UART* a pris en compte la variabilité des données analogiques aux électrodes des touches tactiles. Il a été optimisé pour économiser des ressources tout en assurant un fonctionnement efficace à l'échelle humaine.

Lors du *Brotaru*, , la console a été soumise à l'épreuve. Des dizaines de personnes ont utilisé le prototype et l'ont apprécié, retrouvant leurs souvenirs d'enfance avec des classiques tels que *Tetris*, *Pokémon*, *Mario Land*, etc. Une observation intéressante a émergé : la sensibilité des touches tactiles variait d'une personne à l'autre. Les jeunes manifestaient une sensibilité plus élevée, tandis que les personnes plus âgées devaient exercer une pression plus soutenue pour interagir avec la console. Cette disparité s'explique par le fait que l'âge est inversement proportionnel au taux d'humidité dans la peau, influençant la conduction des charges électriques.

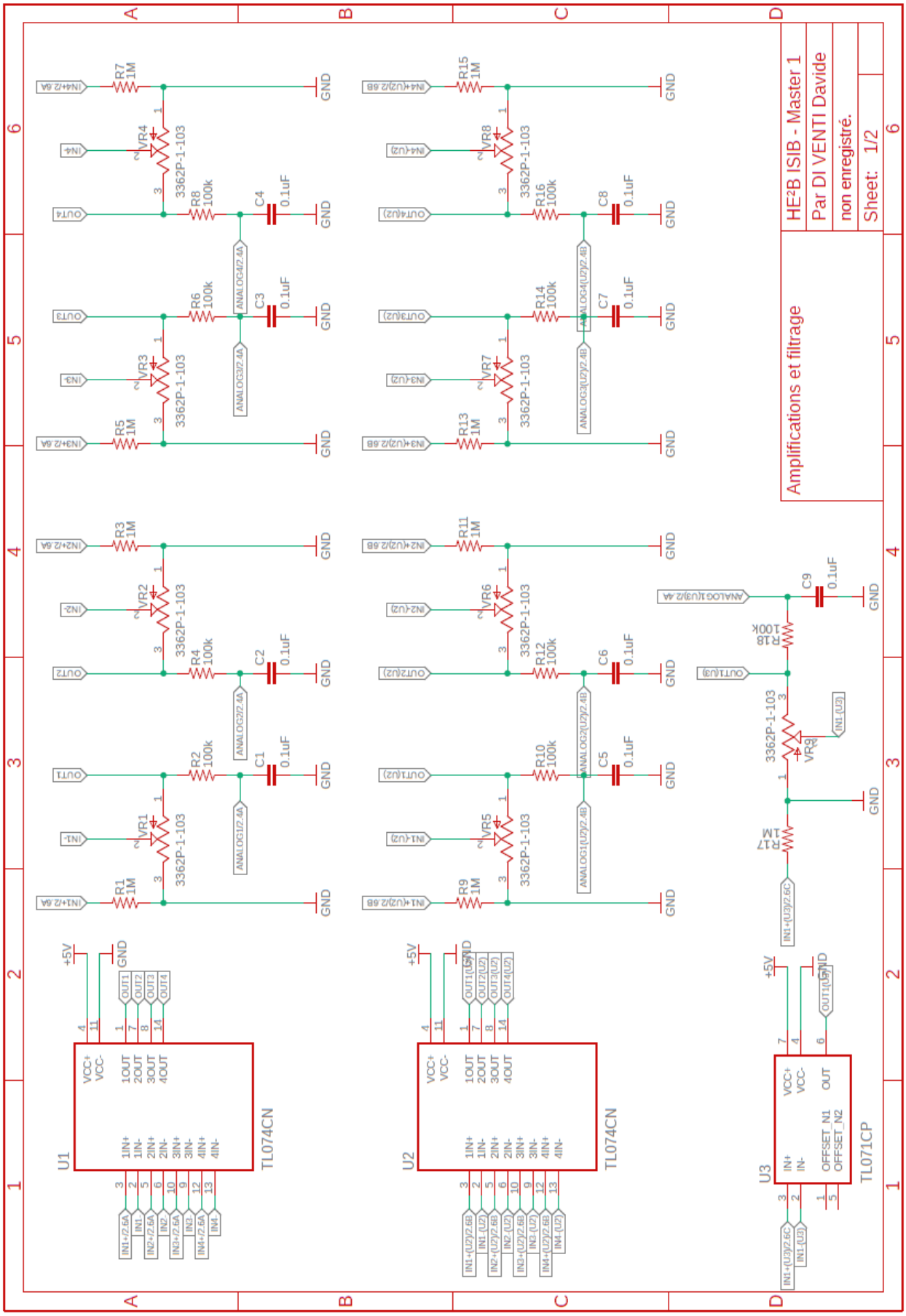
Le prototype "*myTouch.gb*" transcende la simple fonction de banc de test. Le projet a réussi à concilier des aspects techniques, budgétaires et ludiques, offrant une expérience interactive originale, même dans des délais serrés.

Bibliographie

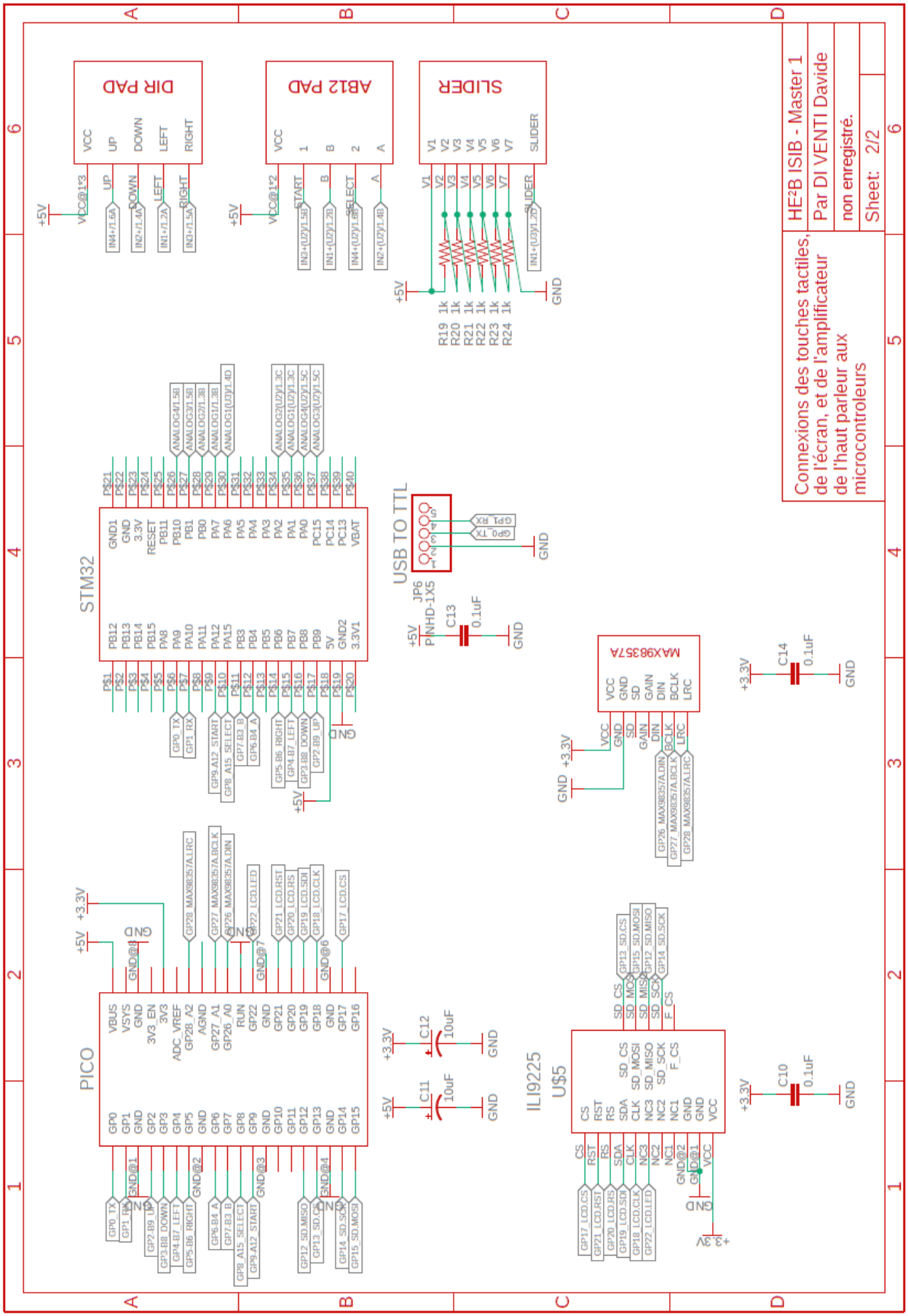
[1] RP2040-GB Game Boy emulator, publié en mai 2022, par YouMakeTech, en ligne, https://github.com/YouMakeTech/Pico-GB, consulté le 11 novembre 2023.

[2] 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.

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[] = {
"000000000000000000000000000000000000000000000000001111110000000011000000110000001000000000001000100000000000001001000000000000001010000000000000001100000
0100010001100000010001000110000001000100011000000000000000110000000000000001010000000000000010001111111111100",
"000000000000000000000000000000000000000000000000001111110000000011000000110000001000000000001000100000000000001001000000000000001010000000000000001100000
0100010001100000010001000110000000000000001100000000000000011000000000000000101000000000000001000111111111110 0",
"000000000000000000000000000000000000000000000000001111000000000011000011000000110000000011000100000000000000100100000000000000100100000000000000101000000
0100010001100000010001000110000000000000001100000000000000011000000000000000101000000000000001000111111111110 0",
"000000000000000000000000000000000000000000000000001111000000000011000011000000110000000011000100000000000000100100000000000000100100000000000000101000000
0100010001100000010001000110000001000100011000000000000000110000000000000001010000000000000010011111111111110",
"000000000000000000000000000000000000000000000000001111000000000011000011000000110000000011000010000000000001000100000000000001001000000000000000101000000
0100010001100000010001000110000001000100011000000000000000110000000000000001010000000000000010011111111111110",
"000000000000000000000000000000000000000000000000001111000000000011000011000000110000000011000010000000000001000100000000000001001000000000000000101000000
0000000001100000010001000110000001000100011000000000000000110000000000000001100000000000000010111111111111110",
"000000000000000000000000000000000000000000000000001111000000000011000011000000010000000010000010000000000001000100000000000001001000000000000000101000000
0000000001100000010001000110000001000100011000000100010001100000000000000011000000000000000010111111111111110",
"000000000000000000000000000000000000000000000000001100000000000011100111000000010000000010000010000000000001000100000000000001001000000000000000100100000
0000000010100000010001000110000001000100011000000100010001100000000000000011000000000000000010111111111111110",
"000000000000000000000000000000000000000000000000001100000000000011100111000000010000000010000010000000000001000100000000000001001000000000000000100100000
0000000010100000010001000110000001000100011000000100010001100000000000000011000000000000000010111111111111110",
"000000000000000000000000000000000000000000000000001110000000000011000111000000010000000010000010000000000001000100000000000001001000000000000000100100000
0000000010100000000000000011000001100110001100000000000000001100000000000000011000000000000000100111111111111110"};
  char *slime_jump[] = {
"00000000111111000000000000000011000000110000000000001000000000010000000001000000000000100000000100000000000000100000001000010000100001000000100001000010000
1000000100001000010000100000010000000000000001000000100000000000000010000001000000000000000100000001111111111110000",
"00000000000000000000000000000011111111000000000000001100000000110000000001000000000000100000000100000000000000100000001000010000100001000000100001000010000
1000000100001000010000100000010000000000000001000000100000000000000010000001000000000000000100000001111111111110000",
"00000000000000000000000000000000000000000000000011111111000000000000111000000000111000000001000000000000010000000100000000000000010000001000010000100001000
1000000100001000010000100000010000000000000001000000100000000000000010000001000000000000000100000001111111111110000",
"0000000000000000000000000000000000000000000000000000000000000000111111111000000000001100000000001100000001000000000000001000000100000000000000000
10000010000010000100000100001000001000010000010000100000000000000001000010000000000000000100000111111111111111000",
"000000000000000000000000000000000000000000000000000000000000000000000000011111111111000000001100000000000110000001000000000000000
10000010000010000100000100001000001000010000010000100000000000000001000010000000000000000100001111111111111111100",
"00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000011111111111110000000100000000000000
10000010000000000000001000100000010000100000010010000001000010000001001000000000000000001000111111111111111111100",
"00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000011111111111111
0000001100000000000000011000100000010000100000010010000001000010000001001000000000000000001001111111111111111111110",
"00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000111111111111111100001100000000000000000011010000000100001000000001100000000000000000000101111111111111111111110",
"00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000001111111111111111111101000000001000010000000011000000000000000000001111111111111111111111",
"00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000011111111111111111110100000000000000000000111111111111111111111"};
  char * empty_slime =
"111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111";

  char *circle_31 =
"00000000000011111000000000000000000000000111000000111000000000000000000001100000000000110000000000000011000000000000000110000000000010000000000000000001
0000000001000000000000000000000001000000010000000000000000000000100000010000000000000000000000010000010000000000000000000000001000010000000000000000000
00000010001000000000000000000000000010010000000000000000000000000100100000000000000000000000000101000000000000000000000000000011000000000000000000
000000000011000000000000000000000000000011000000000000000000000000000011000000000000000000000000000101000000000000000000000000000100100000000000
0000000000000100100000000000000000000000000010001000000000000000000000000100001000000000000000000000000010000010000000000000000000000100000010000000
0000000000000001000000010000000000000000000001000000000100000000000000000010000000000000110000000000000011000000000000000110000000000011000000000000
00001110000011100000000000000000000000001111100000000000000";
  char *a_15 =
"000000000000000000000100000000000001010000000000001010000000000000101000000000001000100000000001000100000000000100010000000001000000100000000011111110000
0000100000010000000010000000100000001000000010000000100000000100000000000000000";
  char *b_15 =
"00000000000000000111111000000000000100000010000000001000000100000000100000010000000010000001000000001000000100000000011111100000000010000001000000001000
000100000000010000000100000000100000001000000010000000111111100000000000000000000";
```

```c
  char *start_15 =
"000000000000000000000010000000000000011000000000000010100000000000001001000000000000001000000000000001000000000000001000000000000001000000000000001000000000000000100000000000000100000000000000100000000000011111100000000000000000000000";
  char *select_15 =
"000000000000000000000111110000000000100000100000001000000010000001000000010000000000000010000000000000100000000000001100000000000110000000000000100000000000100000000000000100000000000000010000000000000111111111000000000000000000000";
  char *up_15 =
"000000000000000000000000000000000000000000000000000100000000000001010000000000001000100000000000100010000000001000001000000001000000010000001000000010000000100000000001000100000000000001001111111111110000000000000000000000000000000000";
  char *right_15 =
"000000000000000011000000000000001010000000000000100110000000000010000100000000001000001100000001000000010000001000000001000001000000001000000100000110000000100000100000000010011000000000001010000000000000110000000000000000000000000000";
  char *left_15 =
"000000000000000000000000000110000000000000101000000000001100100000000001000010000000110000010000000100000001000001000000001000000010000000010000000110000010000000000100001000000000011001000000000000010100000000000000110000000000000000000";
  char *down_15 =
"00000000000000000000000000000011111111111100100000000000100010000000001000001000000010000001000000010000000100000100000000010001000000000001000100000000000010100000000000000100000000000000000000000000000000000000000000000000000000000";
  char *slider_21_2 =  "000011111111111110000111100000000000001111";
  char *arrows_slider_15_15 =
"00000000000000000000000000000000000010000000000000111000000000000111110000000000111111100000000000000000000000000000000000000000000000000000000011111110000000001111100000000000001110000000000000100000000000000000000000000000000000000000";
  char *delete_15 =
"111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111";

  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 =
"11111111111111111111111111111111111111111111111111100011100011111100011100011111100011100011111100011100011111100011100011111100011100011111100011100111";
  char *y_9_15 =
"11100011111100011111100011111100011111100011111100011111111111111111111111111111111111000000111000000111000000111000000111000000111000000111";
  char *T_9_15 =
"111111111111111111111111111100011100000011100000011100000011100000011100000011100000011100000011100000011100000011100000011100000011100000011100000111000";
  char *o_9_9 =  "111111111111111111111111111111100011111100011111100011111111111111111111111111111";
  char *u_9_9 =  "111000111111000111111000111111000111111000111111000111111111111111111111111111111";
  char *c_9_9 =  "111111111111111111111111111111000000111000000111000000111111111111111111111111111";
  char *h_9_15 =
"111000000111000000111000000111000000111000000111000000111111111111111111111111111111111111100011111100011111100011111100011111100011111100011111100011111000111";
  char *point_2_2 = "1111";
  char *g_9_15 =
"111111111111111111111111111111100011111100011111100011111111111111111111111111111111111111000000111000000111000000111111111111111111111111111111111";
  char *b_9_15 =
"111000000111000000111000000111000000111000000111000000111111111111111111111111111111111111100011111100011111100011111111111111111111111111111111111";

  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
    ...
    */
  }

}
```